UNIVERSIDADE FEDERAL DE SÃO CARLOS
CENTRO DE CIÊNCIAS EXATAS E DE TECNOLOGIA
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

Método RSCT

Reengenharia de Software Orientada a Componentes usando Transformações

Orientador: Prof. Dr Antonio Francisco do Prado
Aluno: Adriano Aleixo Bossonaro

São Carlos
Agosto/04

UNIVERSIDADE FEDERAL DE SÃO CARLOS
CENTRO DE CIÊNCIAS EXATAS E DE TECNOLOGIA
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

Método RSCT
Reengenharia de Software Orientada a Componentes usando Transformações

Adriano Aleixo Bossonaro

Dissertação 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.

São Carlos
Agosto/04

Ficha catalográfica elaborada pelo DePT da
Biblioteca Comunitária da UFSCar

B745mr

Bossonaro, Adriano Aleixo.
Método RSCT reengenharia de software orientada a componentes usando transformações / Adriano Aleixo Bossonaro. -- São Carlos : UFSCar, 2005.
124 p.

Dissertação (Mestrado) -- Universidade Federal de São Carlos, 2004.

1. Engenharia de software. 2. Reengenharia orientada a objetos. 3. Componentes de software. 4. Padrões de projeto. I. Título.

CDD: 005.1 (20[a])

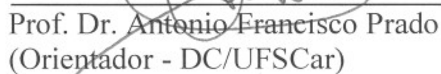
Universidade Federal de São Carlos
Centro de Ciências Exatas e de Tecnologia
Programa de Pós-Graduação em Ciência da Computação

"Método RSCT: Reengenharia de Software Orientada a Componentes usando Transformações"

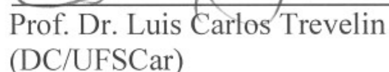
ADRIANO ALEIXO BOSSONARO

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.

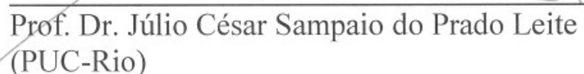
Membros da Banca:



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



Prof. Dr. Luis Carlos Trevelin
(DC/UFSCar)



Prof. Dr. Júlio César Sampaio do Prado Leite
(PUC-Rio)

São Carlos
Agosto/2004

Agradecimentos

A Deus pela oportunidade e capacidade de aprender.

Aos meus pais e a minha namorada que não pouparam esforços para me incentivar, e cuja dedicação e confiança serviram de inspiração para o desenvolvimento deste trabalho.

Ao meu orientador Antonio Francisco do Prado pela orientação, pela amizade, paciência, perseverança, dedicação e incentivo durante todo o decorrer deste trabalho.

Aos amigos do DC, em especial ao grupo de trabalho do Prof. Prado pelo suporte, apoio técnico e pela troca de experiências.

A todos docentes e funcionários do DC que colaboraram pela a realização deste trabalho.

Finalmente a todos aqueles que colaboraram direta ou indiretamente na realização deste.

Obrigado.

Sumário

1. Introdução... 10
2. Principais Tecnologias 12
2.1. Engenharia de Domínio.. 12
2.2. Componentes de Software.. 15
2.3. Padrões de Software 18
2.3.1. Padrões de Projeto 20
2.4. Reengenharia de Software.. 21
2.5. Sistema de Transformação Draco-PUC 24
2.6. Ferramenta Draco Domain Editor (DDE) 27
2.6.1. Editor de Gramáticas.. 29
2.6.2. Editor de Transformadores.. 30
2.7. Ferramenta MVCASE 32
3. Reengenharia de Software Orientada a Componentes usando Transformações (RSCT) 35
3.1. Fase 1 – Construir Domínios e Transformadores.. 37
3.1.1. Construir Domínio.. 38
3.1.2. Construir Transformadores 41
3.1.2.1. Transformador de Identificação 42
3.1.2.1.1. Identificar Supostas Classes e Supostos Atributos....................................... 46
3.1.2.1.2. Identificar Supostos Métodos.. 49
3.1.2.1.3. Identificar o Fluxo de Execução do Sistema Legado 53
3.1.2.1.4. Identificar Relacionamentos.. 55
3.1.2.1.5. Identificar Cardinalidades 57
3.1.2.2. Transformador de Organização.. 58
3.1.2.2.1. Organizar em Supostas Classes e Supostos Atributos 59
3.1.2.2.2. Organizar Supostos Métodos nas Supostas Classes....................................... 60
3.1.2.3. Transformador de Recuperação 62
3.1.2.3.1. Recuperar Especificações das Classes e dos Atributos.................................. 62
3.1.2.3.2. Recuperar Especificações dos Métodos 63
3.1.2.3.3. Recuperar Especificações dos Casos de Uso 64
3.1.2.3.4. Recuperar Especificações dos Diagramas de Seqüência................................ 65
3.1.2.3.5. Recuperar Especificações dos Relacionamentos e Cardinalidades................. 66
3.2. Fase 2 – Obter Projeto Orientado a Objetos... 68
3.2.1. Identificar Elementos 69
3.2.2. Organizar Código 70
3.2.3. Recuperar Projeto... 72
3.2.4. Refinar Projeto Orientado a Objetos 74
3.2.5. Considerações Finais.. 76
3.3. Fase 3 – Construir Componentes 77
3.3.1. Refinar Projeto Orientado a Componentes.. 78
3.3.1.1. Identificar classes como Entity e Session. 79

3.3.1.2. Reespecificar Orientado a Componentes. 81
3.3.1.3. Identificar Padrões........ 86
3.3.2. Implementar Componentes 94
3.4. Fase 4 – Reconstruir Sistema 98
3.4.1. Reespecificar Orientado a Componentes 98
3.4.2. Reprojetar Orientado a Componentes 101
3.4.3. Reimplementar Orientado a Componentes 104
4. Avaliação........ 106
4.1. Introdução........ 106
4.2. Abordagem........ 106
4.2.1. Planejamento 106
4.2.2. Monitoração 107
4.2.3. Análise dos Resultados........ 110
5. Conclusão 112
5.1. Principais Contribuições 113
5.2. Trabalhos Futuros........ 113
6. Referências Bibliográficas 115
7. Anexo 1 - Contribuições Acadêmicas........ 123

Lista de Figuras e Tabelas

Figuras:

Figura 2.1 - Domínio no Draco 25
Figura 2.2 - Framework para transformações do Sistema de Transformação Draco-PUC 26
Figura 2.3 - Apresentação da interface principal da Draco Domain Editor 28
Figura 2.4 - Ambiente de trabalho da Draco Domain Editor 28
Figura 2.5 - Editor de Gramáticas 29
Figura 2.6 - Editor de Transformadores 31
Figura 2.7 - Parte da gramática MDL 33
Figura 3.1 - Reengenharia de Software Orientada a Componentes usando Transformações .. 35
Figura 3.2 - Fases do Método RSCT 36
Figura 3.3 - Construir Domínios e Transformadores (Fase 1) 38
Figura 3.4 - Passos da Etapa Construir Domínio 39
Figura 3.5 - Regras de produção da gramática Clipper na ferramenta DDE 39
Figura 3.6 - Script makefile para geração do parser e prettyprinter 40
Figura 3.7 - Script de execução para submeter um programa ao domínio Clipper 40
Figura 3.8 - Passos da Etapa Construir Transformador 42
Figura 3.9 - Visão geral do metamodelo 43
Figura 3.10 - Visão Lógica (LogicalView) 43
Figura 3.11 - Visão de Caso de Uso (UseCaseView) 44
Figura 3.12 - Visão de Componentes (ComponentView) 45
Figura 3.13 - Relacionamentos do Metamodelo 46
Figura 3.14 - Transformação para identificação de supostas classes persistentes e atributos . 47
Figura 3.15 - Identificação de uma suposta classe transiente e seus supostos atributos 48
Figura 3.16 - Transformações para a identificação de supostas classes e atributos 48
Figura 3.17 - Transformação para identificação dos supostos métodos 49
Figura 3.18 - Identificação de um suposto método de uma classe transiente 50
Figura 3.19 - Identificação de um suposto método de uma classe persistente 50
Figura 3.20 - Identificação de supostos métodos construtores e observadores 51
Figura 3.21 - Organização dos supostos métodos em suas supostas classes 52
Figura 3.22 - Outras transformações para Identificação de supostos métodos 52
Figura 3.23 - Metaclasse Method do metamodelo Orientado a Objetos 54
Figura 3.24 - Informações coletadas no fato CallCalled 54
Figura 3.25 - Identificação de um caso de uso 55
Figura 3.26 - Transformação para identificação de supostos relacionamentos 56
Figura 3.27 - Transformações para identificação de supostos relacionamentos 57
Figura 3.28 - Transformação para identificação de cardinalidades 58
Figura 3.29 - Organização de supostas classes e supostos atributos 59
Figura 3.30 - Organização dos supostos métodos nas supostas classes organizadas 61
Figura 3.31 - Padrões de reconhecimento e substituição para organização do código 61

Figura 3.32 - Transformação de recuperação das supostas classes e atributos 63
Figura 3.33 - Transformação de recuperação das especificações de supostos métodos 64
Figura 3.34 - Transformação de recuperação das especificações de supostos Casos de Uso .. 65
Figura 3.35 - Recuperação das especificações de um Diagrama de Seqüência em MDL 65
Figura 3.36 - Transformação de recuperação das especificações de um relacionamento........ 66
Figura 3.37 - Script makefile para geração de um transformador.. 67
Figura 3.38 - Script de execução para submeter um programa ao transformador ClipperKB . 67
Figura 3.39 - Obter Projeto Orientado a Objetos (Fase 2) 69
Figura 3.40 - Exemplo de fatos da Base de Conhecimento 70
Figura 3.41 - Organização de uma classe com seus atributos e métodos................................ 71
Figura 3.42 - Especificação MDL de uma classe com seus atributos 72
Figura 3.43 - Especificação MDL de métodos com seus corpos escritos em Java 73
Figura 3.44 - Especificação MDL de um Caso de Uso 74
Figura 3.45 - Especificação MDL de um Diagrama de Seqüência 74
Figura 3.46 - Refinamento de um Modelo de Classes 75
Figura 3.47 - Refinamento de um Modelo de Caso de Uso 76
Figura 3.48 - Construir Componentes (Fase 3) 77
Figura 3.49 - Arquitetura adotada no RSCT 78
Figura 3.50 - Classes Entity do Modelo de Classes do POO 80
Figura 3.51 - Classes Session do Modelo de Classes do POO.. 80
Figura 3.52 - Modelo de Caso de Uso do POO... 81
Figura 3.53 - Reespecificação Orientada a Componentes de classe Entity. 82
Figura 3.54 - Reespecificação de um Modelo de Interação 83
Figura 3.55 - Reespecificação de classe do POC considerando classes do POO.................... 84
Figura 3.56 - Reespecificação Orientada a Componentes de classe Session. 85
Figura 3.57 - Modelo de classes do componente Session CadastrarCliente. 85
Figura 3.58 - Generalização de uma classe 87
Figura 3.59 - Classes envolvidas no Caso de Uso CadastrarCliente....................................... 88
Figura 3.60 - Classes envolvidas no Caso de Uso CadastrarFornecedor 88
Figura 3.61 - Padrão CadastrarPessoa... 89
Figura 3.62 - Classes envolvidas no Caso de Uso CadastrarProduto...................................... 89
Figura 3.63 - Padrão CadastrarProduto 90
Figura 3.64 - Classes envolvidas no Caso de Uso VenderProduto 90
Figura 3.65 - Classes envolvidas no Caso de Uso ComprarProduto....................................... 91
Figura 3.66 - Padrão NegociarProduto.. 92
Figura 3.67 - Classes envolvidas no Caso de Uso AtualizarEstoque...................................... 92
Figura 3.68 - Padrão OperarEstoque 93
Figura 3.69 - Classes envolvidas no pagamento e recebimento de duplicatas........................ 93
Figura 3.70 - Padrão AdministrarFinanças 94
Figura 3.71 - Refinamento de um método no projeto de um componente.............................. 95
Figura 3.72 - Modificação do corpo de um método.. 96
Figura 3.73 - Interação entre os padrões e componentes 96
Figura 3.74 - Geração de código Java na ferramenta MVCASE 97

Figura 3.75 - Reconstruir Sistema (Fase 4) 98
Figura 3.76 - Framework de Modelos do padrão CadastrarPessoa 99
Figura 3.77 - Framework de Modelos do padrão NegociarProduto 99
Figura 3.78 - Modelo de Aplicação de Framework para um cadastro de clientes 100
Figura 3.79 - Aplicação dupla do padrão NegociarProduto 101
Figura 3.80 - Reprojeto Orientado a Componentes 102
Figura 3.81 - Refinamento de um método no reprojeto 103
Figura 3.82 - Reimplementação Orientada a Componentes 105
Figura 4.1 - Antiga e nova interface do sistema Informatiza 108
Figura 4.2 - Antiga e nova interface do sistema Caiçara 109
Figura 4.3 - Antiga e nova interface do sistema Gnome 109

Tabelas:

Tabela 2.1 - Principais fontes de informação para a Engenharia de Domínio14
Tabela 2.2 - Elementos de um Modelo de Componentes 18
Tabela 2.3 - Elementos essenciais de um padrão 19
Tabela 3.1 - Domínios construídos na Etapa Construir Domínio da Fase 1 do RSCT 41
Tabela 3.2 - Relação Chama/Chamado 53
Tabela 3.3 - Transformações para recuperação e conversão do corpo dos métodos 64
Tabela 3.4 - Transformadores construídos na Etapa Construir Transformadores 68
Tabela 3.5 - Casos de Uso comuns nos sistemas Informatiza, Caiçara e Gnome 87
Tabela 4.1 - Artefatos obtidos com a reengenharia 107
Tabela 4.2 - Tempo gasto na aplicação das fases do RSCT 107

Resumo

Este projeto pesquisou um Método de Reengenharia de Software Orientada a Componentes usando Transformações, denominado RSCT. O método pesquisado estende o Método RST [Fon02a, Fon02b, Fon02c, Fon02d, Fon04], adicionando recursos para tratar a reengenharia baseada em componentes. O RSCT tem como objetivo orientar a construção e reuso de componentes de software na reengenharia de sistemas legados.

Na execução do método, o Engenheiro de Software é apoiado por duas ferramentas: o Sistema de Transformação Draco-PUC e a ferramenta CASE (*Computer Aided Software Engineering*) MVCASE, e está dividido em 04 (quatro) fases. Na Fase 1, *Construir Domínios e Transformadores*, obtêm-se os domínios e transformadores de software que são usados na Engenharia Reversa do sistema legado para a obtenção de seu Projeto Orientado a Objetos. Na Fase 2, *Obter Projeto Orientado a Objetos*, com o apoio do Sistema de Transformação Draco-PUC, obtém-se o projeto Orientado a Objetos recuperado do sistema legado, usando os domínios e transformadores construídos na Fase 1. Na Fase 3, *Construir Componentes*, obtêm-se os componentes do domínio do sistema legado, a partir dos projetos Orientados a Objetos recuperados na Fase 2. Com o apoio da ferramenta MVCASE, o Engenheiro de Software analisa e refina cada projeto Orientado a Objetos e utiliza padrões de projeto para construir os componentes, disponibilizando-os em uma biblioteca. Finalmente, na Fase 4, *Reconstruir Sistemas*, são reconstruídos os sistemas legados, a partir dos seus projetos Orientados a Objetos obtidos na Fase 2, fazendo reuso dos componentes disponíveis na biblioteca.

Abstract

This project researched a Component-Oriented Software Reengineering Method using Transformations, named RSCT. The researched method extends the RST Method [Fon02a, Fon02b, Fon02c, Fon02d, Fon04], adding resources to treat the component-based reengineering. The RSCT aims to guide the components construction and reuse in the reengineering of legacy systems.

The method is supported by two tools: a Software Transformation System, named Draco-PUC and a CASE tool, named MVCASE, and is divided in 4 (four) phases. In Phase 1, *Construct Domains and Transformers*, it is obtained the domains and transformers used in the legacy system reverse engineering to obtain of Object-Oriented Design. In Phase 2, *Obtain Object-Oriented Design*, it is obtained the legacy system object-oriented recovered design, using the domains and transformers constructed in Phase 1. In Phase 3, *Construct Components*, it is obtained the components of the legacy system domain, from the object-oriented recovered designs in Phase 2. With the MVCASE support, the Software Engineering analyses and refines each Object-Oriented design and uses design patterns to construct the components, making them available in a library. Finally, in Phase 4, *Reconstruct Systems*, the legacy systems are reconstructed from their object-oriented designs, obtained in Phase 2, reusing the components available in the library.

Capítulo 1

Introdução

Sistema de software é um artefato evolutivo e requer constantes modificações, seja para corrigir erros, melhorar desempenho, adicionar novos requisitos ou mesmo para adaptá-lo para novas plataformas de hardware e software [Fon02b].

A dificuldade em atualizar estes softwares com 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 sua manutenção [Fuk99].

O conhecimento adquirido com estes sistemas antigos, denominados 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, facilitando sua reconstrução, sem perda da semântica.

O reuso é um princípio essencial na área de Reengenharia de Software para garantir a redução de esforços e custos na reconstrução de sistemas, além da redundância de código. Na tecnologia Orientada a Objetos, o reuso de software pode ser facilitado com a adoção de padrões de projeto e de componentes de software já existentes e testados.

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, organizando-as de forma coerente e reutilizável [Fon02b].

Diferentes métodos têm sido propostos para orientar a Reengenharia de Software [Ara88, Pen96, Pra92, Brg99, Ber00]. O grupo de Engenharia de Software do DC/UFSCar também tem pesquisado técnicas, ferramentas e métodos que apóiam o processo de Reengenharia de Software [Jes99, Nog02, Nov02, Fon02a, Fon02b, Fon02c, Fon04].

Recentemente, este pesquisador e seu orientador participaram de um projeto de Reengenharia de Software financiado pelo CNPq/RHAE [Fon04], que resultou na definição de um Método de Reengenharia de Software usando Transformações denominado método RST. A grande vantagem do método RST vem da utilização de um sistema de transformação de software que suporta a reconstrução de sistemas legados de diferentes domínios, com a obtenção de seu projeto Orientado a Objetos. Este método, apesar de suportar a construção de software Orientada a Componentes, não trata a obtenção e reuso dos componentes. Assim,

combinando as idéias do método RST com idéias de componentes de software e de padrões de projeto, este projeto pesquisou um método denominado RSCT (Reengenharia de Software Orientada a Componentes usando Transformações), que estende o Método RST visando tratar da construção e reuso dos componentes na Reengenharia de Software. Os componentes são obtidos com a Engenharia Reversa de diferentes sistemas de um determinado domínio do problema e ficam disponíveis em uma biblioteca para reuso tanto na reconstrução de sistemas legados, como na construção de novos sistemas.

Dentre as vantagens do Método RSCT, destaca-se a obtenção dos componentes, baseada no conhecimento embutido nos códigos legados. Estes componentes, quando presentes em vários sistemas de um mesmo domínio do problema, constituem padrões de projeto para este domínio.

O uso de componentes de software proporciona ao sistema reconstruído mais segurança e facilidade de manutenção para acompanhar a evolução contínua do software.

A apresentação desta dissertação está organizada em 06 (seis) capítulos, além desta introdução. O Capítulo 2 apresenta as Principais Tecnologias envolvidas no desenvolvimento deste trabalho. O Capítulo 3 apresenta o método de Reengenharia de Software Orientada a Componentes usando Transformações. O Capítulo 4 apresenta a Avaliação deste método e finalmente, no Capítulo 5 têm-se a Conclusão deste trabalho.

Capítulo 2

Principais Tecnologias

Neste capítulo, são abordados os principais conceitos e ferramentas combinadas para o desenvolvimento desta pesquisa. Um estudo foi realizado com o objetivo de conhecer o estado da arte sobre Engenharia de Domínio, Componentes de Software, Reengenharia de Software e Padrões de Projeto. Também foram estudados o Sistema de Transformação Draco-PUC e as ferramentas Draco Domain Editor e MVCASE. Segue-se uma apresentação sobre Engenharia de Domínio, considerando que no enfoque do Método RSCT, a obtenção dos componentes baseia-se no domínio dos códigos legados.

2.1 Engenharia de Domínio

Segundo Pressman [Pre01], o objetivo da Engenharia de Domínio (ED) é identificar, construir, catalogar e disseminar um conjunto de componentes de software que possam ser reutilizados dentro de um particular domínio de aplicação. Assim, a ED consiste de atividades que sistematizam a busca e representação de informações do domínio, de forma a facilitar ao máximo o seu reuso [Wer00].

Arango [Pri98], Griss [Gri98], Kang [Kan98], Pressman [Pre01], Prieto Diaz [Pri98], Simos [Sim96] e Werner [Wer00] concordam que a Engenharia de Domínio é composta basicamente por 03 (três) fases: *Análise, Projeto e Implementação do Domínio*.

A *Análise de Domínio* corresponde ao processo pelo qual a informação usada no desenvolvimento do sistema, para um domínio em particular, é identificada, capturada e organizada, com o objetivo de ser reutilizada para a criação de novos sistemas [Pri90, Alm03]. Nesta fase, o Engenheiro de Software deve disponibilizar representações que capturem o contexto e a abrangência no domínio, mostrando seu relacionamento com ou outros domínios. São disponibilizados modelos, que capturem os principais conceitos e funcionalidades do domínio, gerando assim modelos com abstrações do domínio [Sei99a, Alm03].

No *Projeto de Domínio*, os resultados da *Análise do Domínio* são usados para identificar e generalizar soluções para os requisitos comuns, através da especificação de uma arquitetura de software do domínio. As oportunidades de reuso, identificadas na análise do

domínio, são refinadas de forma a especificar as restrições do projeto [Wer00, Alm03]. Nesta fase, o Engenheiro de Software deve disponibilizar modelos que especifiquem a estrutura arquitetural a ser seguida pelas aplicações do domínio. As representações geradas provêm modelos arquiteturais que auxiliem na especificação de arquiteturas específicas para cada aplicação [Sei99b, Wer00, Alm03].

Na *Implementação do Domínio*, as oportunidades de reuso e soluções do projeto são transformadas em um modelo implementacional, onde estão incluídos serviços como: a identificação, reengenharia ou construção, e manutenção dos componentes reutilizáveis, que suportem estes requisitos e soluções de projeto[Wer00, Alm03]. Nesta fase, o Engenheiro de Software disponibiliza componentes implementacionais, que especifiquem as principais funcionalidades encontradas em aplicações do domínio. Estes componentes implementacionais estão em conformidade com o modelo de abstrações da fase de *Análise do Domínio* e com os modelos arquiteturais da fase de *Projeto do Domínio*, de forma que possam cooperar entre si para implementar todas as funcionalidades necessárias ao sistema de software [Sei99b, Alm03].

Segundo Simos [Sim96] e Werner [Wer00], os profissionais que atuam em um processo de ED podem ser classificados em 03 (três) grupos: *Fontes, Produtores* e *Consumidores*.

O grupo de profissionais classificados como *Fontes* é composto por usuários finais que utilizam aplicações já desenvolvidas em um domínio particular e também por especialistas do domínio que fornecem informações sobre conceitos e funcionalidades importantes do domínio [Wer00].

O grupo de profissionais classificados como *Produtores* é composto basicamente pelos analistas e projetistas do domínio. Esses profissionais capturam as informações diretamente dos profissionais que compõem o grupo das *Fontes* e de aplicações existentes, para realizar a análise, projeto e implementação dos componentes do domínio [Wer00].

O grupo de profissionais classificados como *Consumidores* é composto pelos desenvolvedores de aplicações e também pelos usuários finais interessados no entendimento do domínio. Estes profissionais utilizam os modelos gerados nas diversas fases da Engenharia de Domínio tanto para especificar aplicações considerando-se a abrangência deste domínio, quanto para aumentar seu entendimento sobre conceitos e funções inerentes ao domínio.

Existem diversas fontes de informação disponíveis para o desenvolvimento da ED. Cada fonte possui vantagens e desvantagens. A Tabela 2.1, adaptada de Kang [Kan90] por

Werner [Wer00], apresenta as principais fontes de informação disponíveis para a Engenharia de Domínio.

Fontes relacionadas com as fases da ED	Fase da ED	Vantagens	Desvantagens
Livros	•Utilizada na Análise do Domínio e no Projeto do Domínio.	•Fonte segura de conhecimento teórico do domínio.	•Apresenta uma visão ideal do domínio.
Padronizações	•Utilizada na Análise do Domínio.	•Representa uma visão de consenso do domínio. •Geralmente utiliza uma nomenclatura amplamente aceita no domínio.	•Deve ser validada para verificar sua aceitação no domínio.
Aplicações existentes	•Utilizada em todas as fases.	•A modelagem, se existir, é bem realista. •Pode ser utilizada para se determinar os requisitos do usuário. •Pode-se abstrair uma arquitetura do domínio, se houver características arquiteturais comuns entre as aplicações. •Permitir a generalização e o empacotamento de componentes.	•Se o domínio é dinâmico, as aplicações podem não representar mais a realidade do domínio.
Especialistas do Domínio	•Utilizada principalmente na Análise do Domínio.	•Permitir a captura mais consistente da dinâmica e da evolução do domínio	•Pode refletir uma visão muito particular do domínio

Tabela 2.1 - Principais fontes de informação para a Engenharia de Domínio

O principal objetivo da ED é o desenvolvimento de Componentes de Software para um particular domínio de aplicação, que possam ser reutilizados no desenvolvimento de aplicações deste domínio. A seguir, serão apresentados conceitos sobre Componentes de Software.

2.2 Componentes de Software

Dentro da Engenharia de Software Baseada em Componentes (*Component-Based Software Engineering*, CBSE), o Desenvolvimento Baseado em Componentes (DBC) é uma atividade de ocorre em paralelo com a Engenharia de Domínio. No DBC, o conceito exato de componente ainda não está muito bem definido. Cada grupo de pesquisa define componente da forma mais adequada ao seu contexto, resultando na falta de uma definição comum e precisa para este termo na literatura [Alm03].

Segundo *Heineman* [Hei01], pesquisas definem um componente de software como um elemento de software em conformidade com um modelo de componentes que pode ser independentemente implantado e composto sem modificações, conforme um padrão de composição.

Brown [Bro98] apresenta diversas definições componentes de software. Algumas dessas definições são [Alm03]:

- "Um componente como um elemento arquitetural. Este componente deve prover e estar em conformidade com um conjunto de interfaces";
- "Um componente como um elemento implementacional acessado através de interfaces bem documentadas, que podem ser descobertas em tempo de execução";
- "Um componente como um elemento implementacional, mas que faz parte de um contexto arquitetural específico";
- "Um componente de negócio representa a implementação de um conceito autônomo de negócio ou processo. São os artefatos de software necessários para expressar, implementar e executar o conceito como um elemento reutilizável de um grande sistema de negócio".

Segundo *Krutchen* [Kru99], componentes reutilizáveis são artefatos autocontidos, claramente identificáveis, que descrevem ou realizam uma função específica e têm interfaces claras em conformidade com um dado modelo de arquitetura de software, documentação apropriada e um grau de reuso definido.

Sametinger [Sam97] define componente reutilizável como um artefato autocontido, de fácil identificação, que descreve ou realiza uma função específica e tem interfaces claras em conformidade com um dado modelo de arquitetura de software, documentação apropriada e um grau de reuso definido. Ser autocontido significa que a função desempenhada pelo componente deve ser realizada por ele de forma completa [Wer00, Alm03].

As primeiras visões sobre componentes enfatizavam somente o seu código. Atualmente, o enfoque encontrado sobre componentes de software dentro da CBSE engloba todos os níveis de abstração [Hei01, Omm02, Alm03].

Segundo Williams [Wil01], os componentes podem ser classificados basicamente de 03 (três) grupos:

● **Componentes GUI:** São os mais conhecidos tipos de componentes encontrados no mercado. Enquadram-se todos os botões, menus, caixas de edição e outros *widgets* utilizados na criação de interfaces para aplicações [Alm03];

● **Componentes de Serviços:** São aqueles componentes que disponibilizam acessos para serviços comuns como acesso a banco de dados, necessários em muitas aplicações. Uma característica dos componentes de serviços é que estes utilizam uma estrutura adicional ou sistemas para realizar suas funções [Alm03];

● **Componentes de Domínio:** Também chamados de componentes de negócio, são os componentes mais difíceis de se projetar e construir. Esses componentes necessitam de um alto conhecimento do domínio para sua construção, a fim de permitirem um maior reuso no desenvolvimento das aplicações [Alm03].

Szyperski [Szy98] e *Wallnau* [Ics02] apresentam a seguinte classificação de componentes:

● **Componentes de negócio:** São componentes cujo domínio da aplicação a qual estão inseridos consegue reconhecer, tais como os componentes Cliente, Fornecedor e Pedido, para um domínio de vendas;

● **Componentes de infra-estrutura:** São componentes de suporte aos componentes de negócio, tais como segurança, distribuição, persistência em banco de dados, auditoria, dentre outros.

Como resultado da Engenharia de Software Baseada em Componentes, são produzidos componentes podem ser classificados também quanto ao seu uso, conforme apresenta *Pressman* [Pre01]:

- **Componentes Comerciais:** Também chamados de Componentes COTS (*Commercial Off-The-Shelf*) ou Componentes de Prateleira, são componentes comprados de terceiros ou desenvolvidos internamente para um projeto anterior, que estão prontos para serem reutilizados no projeto atual e estão plenamente validados [Pre01];

- **Componentes Qualificados:** São componentes já avaliados pelo Engenheiro de Software. São garantidos não apenas a funcionalidade, mas também o desempenho, a confiabilidade, a usabilidade e outros fatores de qualidade satisfazem os requisitos do produto a ser construído [Pre01];

- **Componentes Adaptados:** São componentes adaptados para modificar suas características não requeridas ou indesejáveis (também chamado de empacotar ou mascarar) [Bro96];

- **Componentes Montados:** São componentes integrados no estilo arquitetural e interconectados com uma infra-estrutura adequada para permitir que sejam coordenados e geridos efetivamente [Pre01];

- **Componentes Atualizados:** São componentes cuja finalidade é substituir o software existente à medida que novas versões de componentes se tornam disponíveis [Pre01].

Weinreich [Wei01] define modelos de componentes como um conjunto de padrões para implementação, nomeação, interoperabilidade, customização, composição e evolução dos componentes.

Atualmente, existem diversos modelos de componentes disponíveis. Dentre eles, destacam-se o OMG CORBA Component Model (CMM) [Omg03] e o Sun Microsystem JavaBeans e Enterprise JavaBeans [Sun04a, Sun04b].

Segundo *Weinreich* [Wei01], os elementos de um modelo de componentes são definidos de forma padronizada. São considerados elementos de um modelo de componentes: *interfaces, nomeação, metadados, customização, composição, evolução* e *implantação*. A Tabela 2.2 adaptada de Almeida [Alm03, Wei01] mostra esses elementos.

Padrões para	Descrição
Interfaces	Especificação do comportamento de componentes e de suas propriedades.
Nomeação	Nomes globais únicos para interfaces e componentes.
Metadados	Informações sobre componentes, interfaces e seus relacionamentos.
Customização	Interfaces e ferramentas para customizar componentes.
Composição	Interfaces e regras para combinar componentes a fim de criar grandes estruturas.
Evolução	Regras e serviços para substituir componentes e interfaces por novas versões.
Implementação	Empacotamento da Implementação e recursos necessários para instalar e configurar o componente.

Tabela 2.2 - Elementos de um Modelo de Componentes

No contexto deste trabalho, nos parece mais adequado definir um componente como um artefato de software com alto grau de reuso, construído de acordo com um determinado modelo de componentes e altamente encapsulado tendo em vista permitir ser acessado somente por meio de suas interfaces.

Na tecnologia orientada a objetos, o reuso de software pode ser assegurado com a adoção de *padrões*. Conceitos sobre padrões são apresentados na próxima seção.

2.3 Padrões de Software

A idéia de padrões aplicados à área de software foi apresentada inicialmente em 1987, no Workshop sobre Especificação e Projeto para Programação Orientada a Objetos da Conferência sobre Programação Orientada a Objetos (OOPSLA), onde *Beck* e *Cunningham* apresentaram um trabalho sobre uma linguagem de padrões para projetar janelas em Smalltalk [Bec87]. A partir de então, trabalhos têm sido publicados abordando o assunto padrões.

Um padrão descreve uma solução bem sucedida para um problema que ocorre com freqüência sob um determinado contexto, durante o desenvolvimento de software, podendo ser considerado como um par *"problema/solução"* [Bus96]. O Engenheiro de Software familiarizado com padrões pode aplicá-los imediatamente a problemas apresentados nas fases do desenvolvimento de um software, sem ter que redescobri-los [Gam95].

Segundo *Gamma et al.* [Gam95], para facilitar seu reuso, um padrão deve ser descrito usando um formato consistente, que basicamente é constituído de 04(quatro) elementos essenciais, descritos na tabela 1:

Elemento	Descrição
Nome	É uma referência usada para descrever um problema de projeto, suas soluções e conseqüências. Dar nome a um padrão aumenta imediatamente o vocabulário de projeto. Isso permite projetar em um nível mais alto de abstração.
Problema	Descreve quando aplicar o padrão. Explica-se o problema e seu contexto. Pode descrever problemas de projetos específicos. Algumas vezes, o problema incluirá uma lista de condições que deve ser satisfeita para que faça sentido aplicar o padrão.
Solução	Descreve os elementos que compõem o projeto, seus relacionamentos, suas responsabilidades e colaborações. A solução não descreve um projeto concreto ou uma implementação em particular porque um padrão é como um gabarito que pode ser aplicado em diferentes situações. O padrão fornece uma descrição abstrata de um problema de projeto e de como um arranjo geral de elementos resolve o mesmo.
Conseqüências	São os resultados e análises das vantagens e desvantagens da aplicação do padrão. Embora as conseqüências sejam raramente mencionadas quando se descreve decisões de projeto, elas são críticas para a avaliação de alternativas de projetos e para a compreensão dos custos e benefícios da aplicação do padrão. As conseqüências também podem abordar aspectos sobre linguagens e implementação

Tabela 2.3 - Elementos essenciais de um padrão

Padrões de software podem fazer referência a diferentes níveis de abstração no desenvolvimento de sistemas. *Buschmann* [Bus96] divide os padrões em três principais categorias que representam diferentes níveis de abstração:

- ***Padrões Arquiteturais*** *(Architectural Patterns):* descrevem um esquema de organização estrutural e fundamental para um sistema de software. Eles fornecem um conjunto de subsistemas pré-definidos, especificando as responsabilidades e definindo regras e diretrizes para organizar os relacionamentos entre eles [Sob02];
- ***Padrões de Projeto*** *(Design Patterns):* fornecem um esquema para refinar os subsistemas, os componentes de sistemas de software ou o relacionamento entre eles. Os mecanismos de cooperação entre componentes são descritos e definidos para encontrar soluções para os problemas de projeto em um contexto específico. *Gamma et al.*[Gam95] define que um padrão de projeto nomeia, abstrai e identifica o essencial da estrutura de um projeto comum, tornando-o útil para criação de um projeto Orientado a Objeto reusável; e
- ***Padrões em nível de Idiomas*** *(Idioms):* especificam como implementar os aspectos particulares dos componentes e seus relacionamentos adequadamente às características da linguagem de programação [Sob02].

Tendo em vista o escopo do nosso trabalho, daremos ênfase aos padrões de projeto, que serão apresentados a seguir.

2.3.1 Padrões de Projeto

Um Padrão de Projeto descreve uma estrutura comum para criar um projeto Orientado a Objetos reutilizáveis, baseados em soluções práticas [Gam95].

O foco principal do uso de padrões de projeto na informática é no campo da Orientação a Objetos. Atualmente, observa-se grande ênfase também na utilização de padrões de projeto no desenvolvimento de software Orientado a Componentes.

Segundo *Gamma et al*. [Gam95], a grande dificuldade no projeto Orientado a Objetos está na decomposição do sistema em objetos. A tarefa é difícil porque muitos fatores devem ser considerados: encapsulamento, granularidade, dependência, flexibilidade, desempenho, evolução, reuso, e assim por diante. Todos influenciam essa decomposição, freqüentemente de formas conflitantes.

Na Engenharia Reversa, quando partimos de sistemas legados, temos a mesma dificuldade apresentada por *Gamma et al.* [Gam95], porém observamos que grande parte dessa dificuldade pode ser amenizada por meio da automatização do processo de recuperação

projeto Orientado a Objetos de sistemas legados. Todo o conhecimento contido no código pode ser recuperado e posteriormente analisado, com a finalidade de obter padrões de projeto para o domínio da aplicação.

Muitas idéias do projeto de pesquisa proposto baseiam-se na Reengenharia de Software, que será apresentada a seguir.

2.4 Reengenharia de Software

Software é um artefato evolutivo que precisa acompanhar as mudanças que ocorrem nas plataformas de hardware e software. Por outro lado, a demanda em relação ao negócio e à tecnologia da informação que o apóia está se modificando num ritmo de enorme pressão competitiva em todos os campos. Tanto o negócio quanto o software que o apóia precisam ser trabalhados para acompanhar esta evolução [Pre01].

A solução normalmente adotada para resolver os problemas acarretados por essa evolução é a Reengenharia. Dependendo do tipo de abordagem adotada, a Reengenharia pode ter diferentes significados [Som95]. São exemplos a Reengenharia Comercial, a Reengenharia de Dados, a Reengenharia de Processo do Negócio e a Reengenharia de Software [Nov02].

A Reengenharia de Software [Pre01], também chamada renovação ou recuperação, pesquisa técnicas modernas para a reconstrução de sistemas de software, objetivando a melhora de sua qualidade global e reduzindo os custos na manutenção. Na reconstrução do sistema de software, o Engenheiro de Software poderá manter as mesmas funcionalidades do sistema ou adicionar novas funcionalidades para atualizar este sistema, melhorando o desempenho global [Nog02].

Jacobson e *Lindström* [Jac91] propõem técnicas de reengenharia de software que podem envolver a mudança completa da implementação sem mudança na funcionalidade, mudança parcial da implementação sem mudança na funcionalidade ou mudança na funcionalidade.

Na Reengenharia de Software com mudança completa da implementação sem mudança na funcionalidade, o Engenheiro de Software prepara um modelo da análise mapeando cada objeto da análise para a implementação do sistema antigo, depois reprojeta o sistema usando uma abordagem de Orientação a Objetos e por fim, implementa o modelo de análise [Nov02].

No caso da Reengenharia de Software com mudança parcial da implementação sem mudança na funcionalidade, o Engenheiro de Software deve identificar as partes do sistema que serão reimplementadas usando técnicas da Orientação a Objetos. Após essa identificação,

é preparado um modelo de análise da parte a ser trocada e seu ambiente e depois mapeado cada objeto para a implementação antiga do sistema. Esse processo poderá ser repetido até que a interface entre a parte a ser trocada e a parte restante do sistema existente seja aceitável. Concomitantemente, o Engenheiro de Software pode projetar o novo subsistema e suas interfaces para o que falta do sistema antigo e modificar o sistema antigo, adicionando uma interface ao novo subsistema e depois integrar e testar o novo subsistema e o sistema antigo modificado [Nov02].

No caso de Reengenharia de Software com mudança na funcionalidade, o Engenheiro de Software deve modificar o modelo de análise de acordo com os novos requisitos e reprojetar o sistema para atender esses novos requisitos. As mudanças na funcionalidade são adicionadas no modelo de análise e posteriormente implementadas.

Além das técnicas de Reengenharia de Software propostas por *Jacobson* e *Lindström,* outras técnicas de Reengenharia foram pesquisadas:

Klösh [Klo96, Nog02] discute uma abordagem para Reengenharia visando a mudança de paradigma de linguagem de programação. Aplica-se primeiramente a Engenharia Reversa para obter o projeto do sistema e, somente após isso, ocorre a mudança de paradigmas, visando obter as especificações do sistema em uma linguagem Orientada a Objetos. *Klösh* estende o trabalho de *Gall* [Gal95], que apresenta o Método COREM para transformação de programas. Este método propõe a reconstrução de sistemas de arquitetura procedimental, tornando-os Orientados a Objetos. A justificativa apresentada para o uso do paradigma da Orientação a Objetos seria a melhora da manutenibilidade futura, devido a conceitos como abstração, encapsulamento e conexão de mensagens. Cabe ressaltar que *Klösh* 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;

Markosian [Mar94, Nog02] ressalta como ponto negativo, a falta de apoio computadorizado para a reengenharia de sistemas, em contraposição à grande proliferação de ferramentas CASE para desenvolvimento de software novo. Para *Markosian*, as ferramentas de transformação atuais são muito limitadas, o que dificulta sua adaptação às particularidades de um projeto. *Markosian* propõe também uma nova tecnologia para Reengenharia, chamada de *Enabling Technology*. Os resultados apresentados na aplicação desta tecnologia são bastante animadores, principalmente quanto à produtividade das aplicações. A *Enabling*

Technology consiste no desenvolvimento de ferramentas para analisar e modificar sistemas legados, visando automatizar tarefas complexas de Reengenharia, que estejam sendo feitas de forma manual;

Wilkening [Wilk95, Nog02] acredita que podemos aproveitar parcialmente a implementação e projeto do sistema para efetuar a Reengenharia. Propõe um processo de reengenharia que se inicia com a reestruturação preliminar do código fonte, para posterior analise e construção das representações do sistema em níveis mais altos de abstração. Com base nessas representações, pode-se prosseguir com os passos de reestruturação, reprojeto, redocumentação e reimplementação do sistema na linguagem destino. Para verificar se as funcionalidades do sistema não foram afetadas, convém a aplicação de testes. A ferramenta RET implementa as idéias do processo de reengenharia proposto por ele.

A aplicação de técnicas de reengenharia em sistemas de software facilita 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.

Conforme o observado nas abordagens sobre técnicas de reengenharia apresentadas, a Reengenharia de Software integra técnicas de Engenharia Reversa e de Engenharia Avante.

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 [Nov02]. O ideal é ter um nível de abstração mais alto possível [Pre01].

Quando os modelos obtidos pela Engenharia Reversa seguem o paradigma da Orientação a Objetos, mais vantagens são oferecidas, principalmente quanto à facilidade de Reengenharia com mudança de paradigma.

A Engenharia Avante visa mapear requisitos em projeto, ou projeto em implementação, preservando a exatidão em um sistema de software. É o tradicional processo de desenvolvimento que parte de um alto nível de abstração e de lógica, passando pela análise de requisitos e projeto até a implementação física do sistema [Nov02].

Assim, combinando as idéias da Engenharia Reversa e Engenharia Avante, tem-se a Reengenharia de Software.

Várias pesquisas têm sido realizadas para melhorar e automatizar o processo de Reengenharia de Software. Dentre as pesquisas realizadas, destacam-se as que utilizam transformação de software [Lei96, Pra98, Fuk99, Jes99, Fon02a, Fon02b, Mor02a, Mor02b, Nog02, Nov02, Bos03, Mor04]. Dentre os sistemas de transformação destaca-se o Sistema de Transformação Draco-PUC, que será apresentado a seguir.

2.5 Sistema de Transformação Draco-PUC

Sistemas de Transformação são ferramentas de apoio que permitem ao Engenheiro de Software a manipulação estrutural e semântica de sistemas.

Os sistemas de transformação manipulam programas ou especificações mudando suas descrições, buscando preservar sua semântica. O objetivo de um sistema de transformação é transformar um programa A em um programa B, aplicando um conjunto de transformações que devem manter a semântica original de A em B. Estas transformações são escritas com base na sintaxe e semântica das linguagens fonte A e alvo B.

Os mais conhecidos sistemas transformacionais são o ANTLR/DLG [Par91, Ant95], ASF+SDF [Ber89, Hee89], CobolTransformer [Blo97, Bra97, Bra97a, Sib97], COSMOS [Hal96, Tec97], DMS [Bax97], Popart [Wile00], Refine [Rea92], RescueWare [Faq98, Rel00], Revolve/2000 [Bil89, Mic00], SES GmbH [Sne97], Tampr [Bir00] e o TXL [Cor91, Cor93] . Dentre eles destaca-se o Sistema de Transformação Draco-PUC.

O Sistema de Transformação Draco-PUC [Pra92] é um sistema transformacional genérico construído com o objetivo de testar, desenvolver e colocar em prática o paradigma Draco para o desenvolvimento de software orientado a domínios, usando transformações que mapeiam a sintaxe e semântica dos comandos de um programa em outro programa, de um mesmo domínio ou outro domínio [Nei84, Pra92, Pra98]. O Sistema de Transformação Draco-PUC é composto por:

- Um Sistema gerador de analisadores léxicos e sintáticos (*parser*) baseado na sintaxe Lex/Yacc [Lei94] utilizando mecanismo de análise LALR (*Look-Ahead Left Right*) com *backtracking*;

- Um Núcleo transformacional totalmente aberto, que suporta o uso de pontos de controle para o disparo de eventos e direção de fluxo de controle;

• Uma linguagem para descrição das transformações, a qual usa transformações locais e globais;

• Mecanismo de casamento de padrões altamente expressivo, fornecendo ao usuário um grande potencial para as especificações dos requisitos de software, utilizando a sintaxe de qualquer linguagem que tenha sido definida no subsistema de *parsers*;

• Mecanismo para a geração automática de código em uma linguagem de programação.

Um domínio no Sistema de Transformação Draco-PUC é definido através de uma *Linguagem.* Esta por sua vez é definida por uma *Gramática*, um *parser* e um *prettyprinter* (ou *Unparser*). Além do domínio, têm-se os transformadores que fazem o refinamento sintático e semântico dos programas de uma linguagem qualquer, por exemplo A, para outra, por exemplo B, conforme mostra a Figura 2.1.

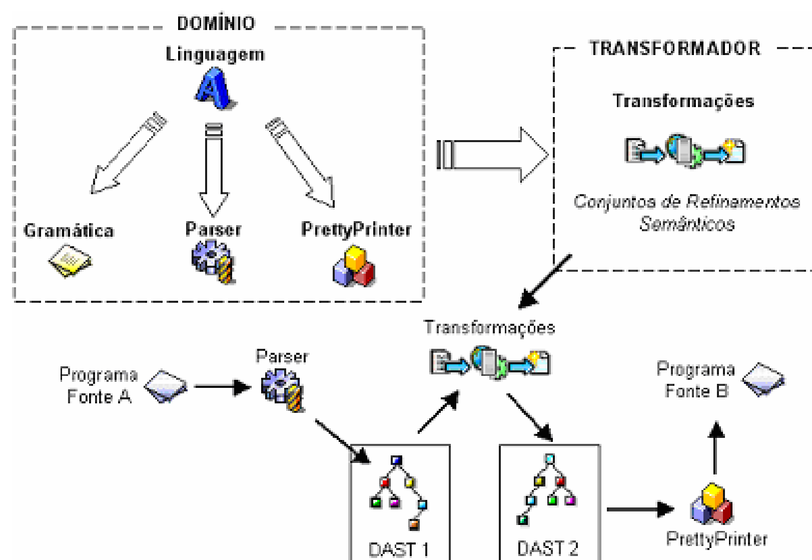


Figura 2.1 - Domínio no Draco

As gramáticas livres de contexto são a base para a geração das Linguagens que definem os domínios no Sistema de Transformação Draco-PUC. A partir da definição das gramáticas o Engenheiro de Software pode gerar o *parser* e o *prettyprinter* no Draco-PUC.

O *parser* da linguagem é responsável por gerar uma representação interna das descrições de uma linguagem. A representação interna das descrições em um domínio, usada pelo Sistema de Transformação Draco-PUC, é feita através de uma *Draco Syntax Abstract Tree* (DAST). Quando um programa escrito em uma linguagem de um domínio Draco é

analisado, o *parser* gera automaticamente a DAST correspondente. Isso permite que o programa possa ser manipulado pelo Sistema de Transformação Draco-PUC. Para auxiliar a criação do *parser* o sistema de Transformação Draco-PUC dispõe de um subsistema gerador de *parser* chamado *pargen.*

O *prettyprinter* ou *Unparser* é responsável por realizar a formatação da DAST, tornando-a novamente textual na linguagem do domínio. Baseado nas definições das gramáticas, o subsistema *ppgen* do Sistema de Transformação Draco-PUC gera, automaticamente, o *prettyprinter* das Linguagens dos domínios no Sistema de Transformação Draco-PUC.

Por exemplo, ao submeter o programa A ao *parser* do domínio A têm-se a DAST 1. Aplicando-se as transformações na DAST 1, obtem-se a DAST 2. Usando o *prettyprinter* do domínio B, obtem-se o programa B a partir da DAST 2.

Cada Transformador (*Transformer*) pode conter seções globais *(Global-Declaration, Global-Initialization e Global-End)* e um ou vários conjuntos de transformações *(Sets of Transforms).* Cada conjunto de transformações (*Set of Transforms*) é formado por várias transformações *(Transforms)* que manipulam a DAST [Fuk99], conforme mostra a Figura 2.2.

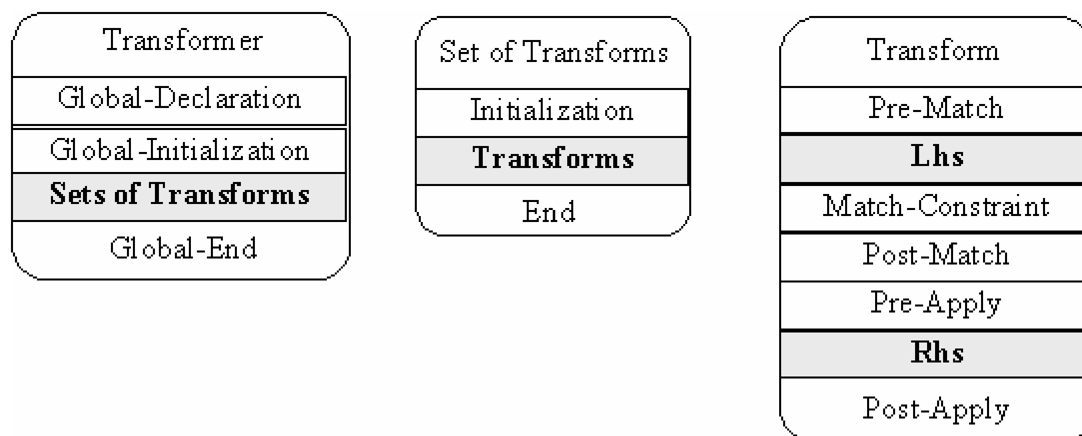


Figura 2.2 - *Framework* para transformações do Sistema de Transformação Draco-PUC

As transformações podem ser Inter-domínios, também conhecidas por transformações verticais, que mapeiam as aplicações descritas em uma linguagem de um domínio para descrições de linguagem de outro domínio ou Intra-Domínio, também conhecidas por transformações horizontais, que mapeiam estruturas de uma linguagem para estruturas na mesma linguagem do domínio. Normalmente executam transformações de otimização ou de preparação para as transformações verticais. Suportam ainda múltiplas visões de uma descrição dentro de um mesmo domínio [Fuk99].

Uma transformação possui os pontos de controle *LHS* e *RHS*. Para que uma regra de transformação seja aplicada, o sistema procura pelo padrão de reconhecimento *(LHS),* e ao encontrá-lo, realiza a reescrita do mesmo de acordo com o padrão de substituição *(RHS)*. As regras podem ainda disparar eventos, ou ainda alterar o fluxo de controle da aplicação das transformações através de outros pontos de controle, aos quais pode-se associar código para o desempenho de tarefas, relacionadas à pré e pós condições da transformação. Os pontos de controle disponíveis no Sistema de Transformação Draco-PUC atendem a todos os requisitos de um sistema transformacional [Fuk99], conforme se segue:

- *Pre-Match*: executado toda vez em que a regra de transformação é testada sobre um trecho da descrição de entrada;
- *Post-Match*: executado após a regra de transformação ser testada sobre um trecho da descrição de entrada;
- *On-Match-Exit*: executado logo após ter sido finalizada uma tentativa de casamento entre o lado esquerdo e um trecho da descrição de entrada;
- *Pre-Apply*: executado imediatamente antes da substituição do trecho de entrada selecionado pelo lado direito; e
- *Post-Apply*: executado após a substituição do trecho de entrada selecionado pelo lado direito.

Para facilitar o desenvolvimento dos domínios e dos transformadores no Sistema de Transformação Draco-PUC foi construída a ferramenta denominada *Draco Domain Editor*. Essa ferramenta suporta a edição textual e gráfica tanto das gramáticas livres de contexto que dão origem aos *parsers* e *unparsers* dos domínios do Sistema de Transformação Draco-PUC, quanto dos transformadores Intra e Inter-Domínios e também dos *scripts* para execução dos transformadores, conforme apresentado a seguir.

2.6 Ferramenta Draco Domain Editor (DDE)

Draco Domain Editor [Gar02, Nog02] é uma ferramenta cujo objetivo é automatizar o processo de criação ou atualização dos domínios e transformadores utilizados no Sistema de Transformação Draco-PUC. A ferramenta Draco Domain Editor possui módulos que permitem a edição dos domínios, transformadores, scripts de execução dos transformadores e scripts de transformação. Será dada ênfase à edição das gramáticas dos domínios e dos

transformadores, conforme as opções do menu da sua interface principal, mostrada na Figura 2.3.

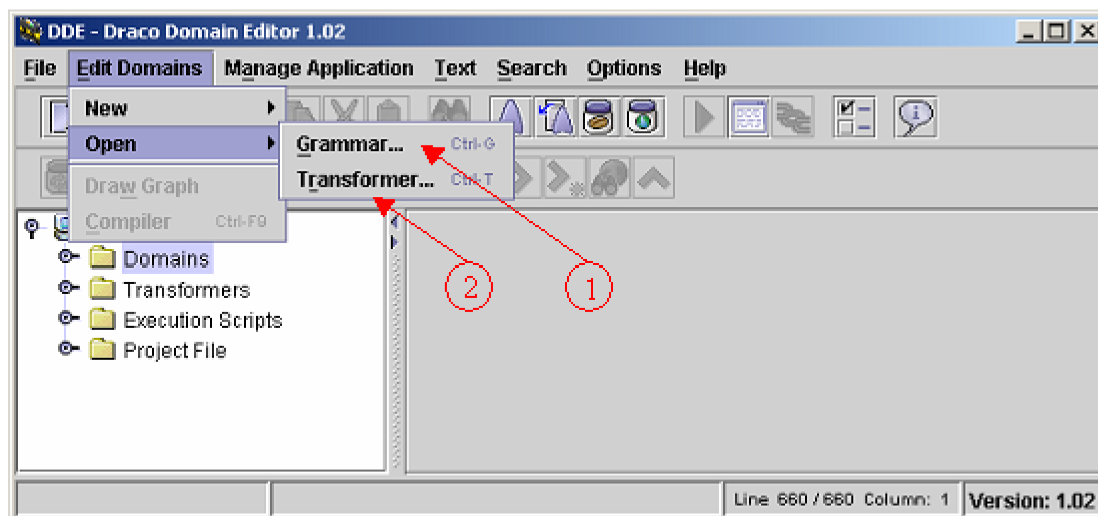


Figura 2.3 - Apresentação da interface principal da Draco Domain Editor

A opção (1) “*Grammar*” suporta a edição da gramática, e a opção (2) “*Transformer*” suporta a edição dos transformadores [Gar02].

O Engenheiro de Software pode editar a gramática do domínio ou transformador a ser utilizado no Sistema de Transformação Draco-PUC, selecionando-os no *browser* e editando-os na forma textual ou gráfica. A Figura 2.4 mostra as áreas para a edição textual e gráfica de uma gramática, para o domínio Clipper.

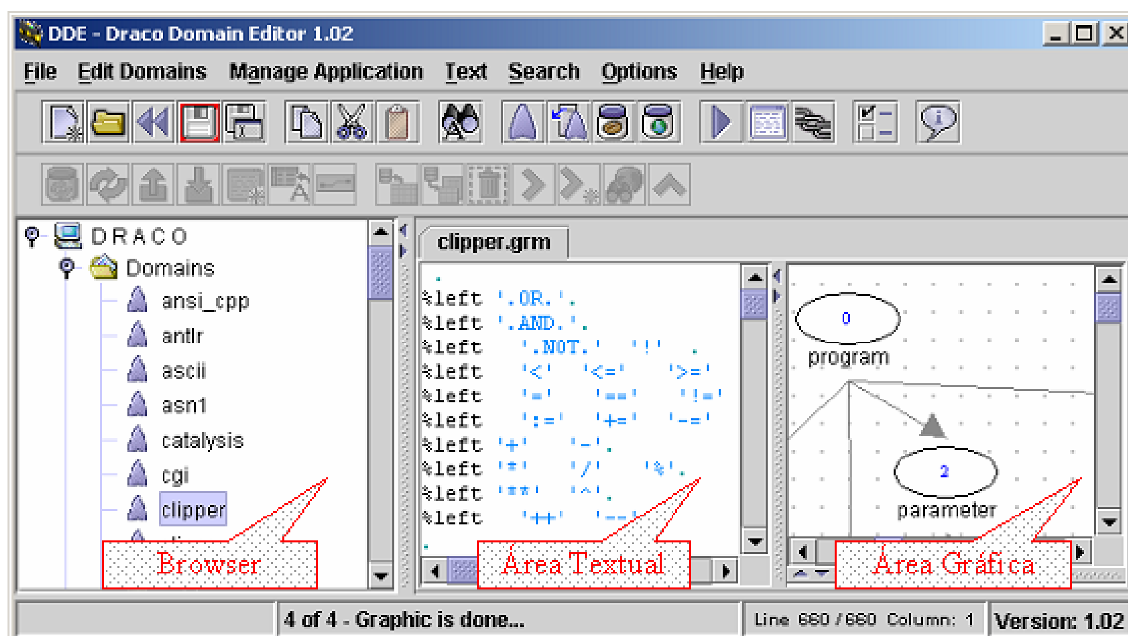


Figura 2.4 - Ambiente de trabalho da Draco Domain Editor

Após editar a gramática da linguagem do domínio, esta é submetida aos subsistemas *pargen* e *ppgen* do Sistema de Transformação Draco-PUC para gerar os respectivos *parser* e o *prettyprinter*. Os transformadores são definidos com base no conhecimento da sintaxe e da semântica das linguagens do domínio de origem e alvo das transformações. Segue-se uma apresentação mais detalhada sobre a edição das gramáticas e das transformações.

2.6.1. Editor de Gramáticas

O Editor de Gramáticas suporta a edição de gramáticas livres de contexto, denotando os símbolos terminais e as regras de produção da gramática por meio de cores. 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 também denotados por meio de cores.

A Figura 2.5 mostra as áreas de edição textual e gráfica, com um trecho da gramática DataFlex Procedural. São denotados os símbolos terminais, os comandos de formatação para o *prettyprinter*, caracteres especiais relevantes para descrição de gramáticas no Sistema de Transformação Draco-PUC e também os comentários.

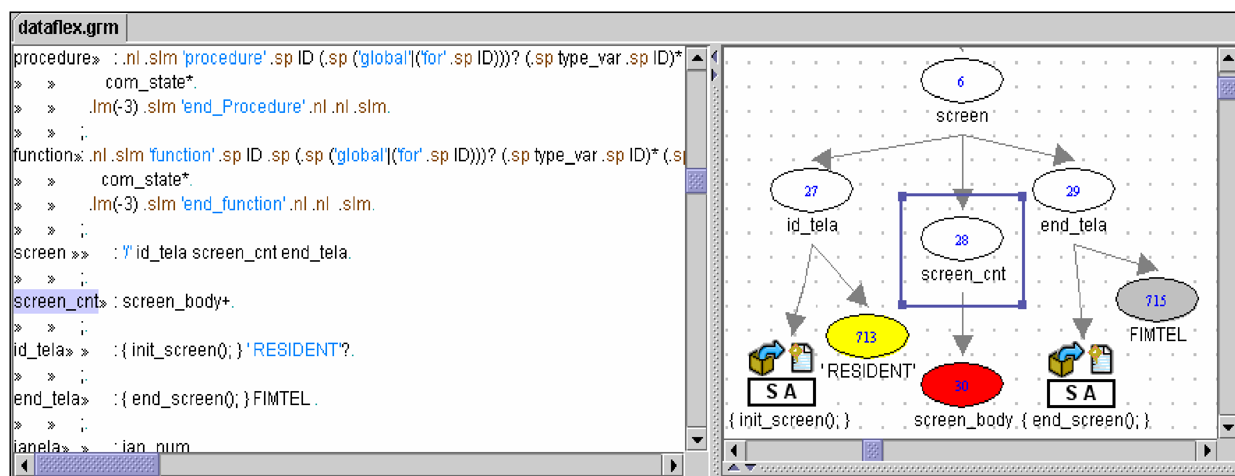


Figura 2.5 - Editor de Gramáticas

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 de seus elementos. Para denotação dos nós na árvore gramatical, utiliza-se a cor branca em nós intermediários quando estes não possuem problemas na derivação, a cor amarela indica nós-folha representando um *token*, a cor cinza indica nós intermediários cuja derivação leva a uma

regra de expressão regular no analisador léxico, a cor vermelha indica nós intermediários cujos filhos não foram definidos e a cor verde indica uma ação semântica associada a uma regra de produção.

Cabe ressaltar que os conflitos do tipo "*shift reduce*" e "*reduce reduce*" não são tratados pela ferramenta Draco Domain Editor. Estes erros são tratados pelo subsistema *pargen* do Draco-PUC.

A ferramenta Draco Domain Editor disponibiliza ao Engenheiro de Software, o recurso de utilizar a área de edição gráfica para editar regras de produção ou a correspondente ação semântica a partir de determinado nó [Nog02].

Depois de editada, a gramática é submetida ao subsistema *pargen* do Draco-PUC para geração do *parser* e ao subsistema *ppgen* para geração do *prettyprinter* [Fre96], concluindo assim, a geração do domínio.

2.6.2 Editor de Transformadores

O Editor de Transformadores suporta a edição dos transformadores. Utiliza-se uma notação adaptada da proposta por Sant´Anna para Circuitos Transformacionais [San99] para gerar a representação gráfica do transformador.

Os transformadores são compostos principalmente por regras de reescrita. Uma regra é composta por um padrão de reconhecimento, chamado de LHS (*left hand side*) e por um padrão de substituição, chamado de RHS (*right hand side*). Estes padrões sintáticos e semânticos são descritos através de parametrização de fragmentos de DASTs, ou trechos de programas nas linguagens dos domínios de origem e de destino. 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 utilizados pelo Engenheiro de Software para controlar o processo de transformação [Nog02].

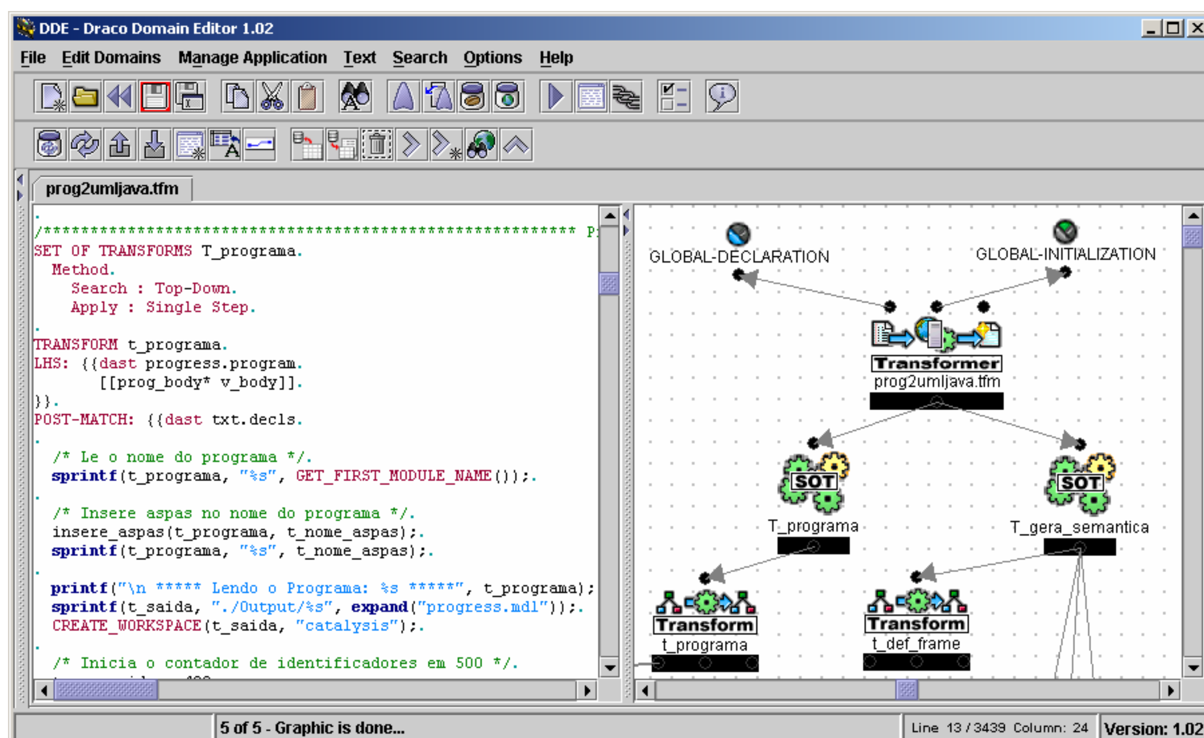


Figura 2.6 - Editor de Transformadores

Para a implementação dos transformadores, é utilizada uma linguagem própria de descrição, baseada em C++. Como mostra a Figura 2.6, o Editor de Transformadores da Draco Domain Editor reconhece os *tokens* da linguagem C++ e os comandos específicos da linguagem de definição dos transformadores. A Figura 2.6 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.

Os transformadores visam mapear cada regra gramatical do domínio original para o domínio alvo, dando o tratamento adequado e gerando outras regras gramaticais [Nog02]. A representação gráfica do transformador gerada pela Draco Domain Editor permite um entendimento mais lógico dos transformadores e facilita a navegação no código, o que auxilia o ES em sua modelagem, implementação e manutenção.

Para a gerar a representação gráfica dos transformadores, são identificadas as estruturas, *"Global Declaration"*, *"Global Inicialization"*, *"Global End"*, *"Set Of Transforms"*, *"Transform"*, *"Template"* e *"KB"*. Estas estruturas são representadas e interligadas conforme os pontos de conexões reconhecidos no código do transformador.

Novos estudos estão sendo realizados para estender a Draco Domain Editor com recursos que incluem a visualização e gerenciamento das transformações associadas às regras de produção das gramáticas dos domínios de origem e alvo das transformações [Gar02].

Outras ferramentas que vêm sendo utilizadas na engenharia e reengenharia de software são as CASE. Uma destas ferramentas é a MVCASE, que será apresentada a seguir.

2.7 Ferramenta MVCASE

As ferramentas CASE (*Computer-Aided Software Engineering*) surgiram com a finalidade de automatizar parcialmente o processo de desenvolvimento de software, visando aumentar a produtividade e qualidade dos softwares. Atualmente podemos notar o largo emprego de ferramentas CASE com sucesso, no desenvolvimento de sistemas.

Dentre as ferramentas CASE, destaca-se a MVCASE [Luc00, Luc01 e Alm02], que suporta a especificação do sistema em uma linguagem de modelagem orientada a objetos e gera esqueleto de código, automaticamente, em uma linguagem de programação orientada a objetos, a partir de especificações em alto nível.

A MVCASE possui interfaces gráficas que facilitam o seu entendimento e utilização em diferentes níveis de abstração. Os modelos gráficos de uma especificação, criados na MVCASE facilitam a comunicação entre os desenvolvedores do sistema e seus usuários.

Na MVCASE, o Engenheiro de Software especifica o sistema em UML, segundo quatro visões: *Visão de Casos de Uso, Visão Lógica, Visão de Componentes e Visão "Deployment"* [Luc03].

A *Visão de Casos de Uso* descreve o sistema como um conjunto de transações do ponto de vista dos atores externos. Esta visão é inicialmente criada na fase de concepção, e refinada nas demais fases do ciclo de vida do sistema, direcionando o processo de desenvolvimento do sistema. As principais técnicas UML disponibilizadas pela MVCASE nesta visão são: Diagramas de Casos de Uso, Diagramas de Colaboração e Diagramas de Seqüência.

A *Visão Lógica* fornece uma visão do sistema estruturada em uma coleção de *packages*, classes e relacionamentos. Esta visão é inicialmente criada na fase de elaboração e refinada na fase de construção do sistema. As principais técnicas UML disponibilizadas pela MVCASE nesta visão são: Diagrama de Classes e Diagrama de Estados.

A *Visão de Componentes* fornece uma visão do sistema estruturada em módulos e subsistemas. Esta visão é inicialmente criada na fase de elaboração e refinada nas fases de

construção e transição. A MVCASE implementa essa visão por meio dos Diagramas de Componentes.

A *Visão "Deployment"* fornece uma visão das partes físicas do sistema e a conexão entre estas partes. Esta visão é normalmente criada na fase de construção e refinada na fase de transição do sistema. A MVCASE implementa essa visão por meio dos Diagramas de *"Deployment"*.

É importante ressaltar que a distribuição das técnicas apresentadas acima, segundo cada uma das quatro visões, é flexível, dependendo do sistema a ser modelado. A MVCASE disponibiliza um conjunto de técnicas da UML, que podem ser combinadas, da melhor forma, para modelar e representar um sistema.

Atualmente, a MVCASE persiste os dados dos modelos dos projetos na linguagem de modelagem denominada MDL. As especificações UML geradas pela ferramenta MVCASE são armazenadas textualmente em um arquivo de extensão ".MDL", baseado na gramática MDL, mostrada em parte na Figura 2.7.

```
mdl        : petal .nl
             design ;
petal       : '(object' 'Petal' .nl version .nl
              written? .nl charSet? ')'
           ;
version     : .slm(+2) 'version' .slm(+15) Number
           ;
written     : .slm(+2) '_written' .slm(+15) STRI
           ;
charSet     : .slm(+2) 'charSet' .slm(+15) Number
           ;
design      : '(object' .sp 'Design' .sp STRI .nl designAttributes ')'
           ;
is_unit     : .nl .slm(+2) 'is_unit' .slm(+15)  boolean
           ;
is_loaded : .nl .slm(+2) 'is_loaded' .slm(+15) boolean
           ;
file_name : .nl .slm(+2) 'file_name' .slm(+15)  STRI
           ;
quid  : .nl .slm(+2) 'quid' .slm(+15) STRI
           ;
designAttributes  :  is_unit .nl is_loaded (.nl file_name)?
                  (.nl quid)? .nl .slm(+2) defaults  .nl
                          root_usecase_package .nl root_category .nl
                          root_subsystem .nl process_structure .nl
                          properties .nl
…
class_Object : '(object' .sp 'Class' .sp STRI .slm .lm(+2)
          classAttributes ')'
        | '(object' .sp 'Class_Utility' .sp STRI .nl .slm(50)
        classAttributes')'
        …
        ;
classAttributes : …
                 | attributes? .nl quid? .nl  classAttributes_Attr
                 ;
…
```

Figura 2.7 - Parte da gramática MDL

A gramática MDL usada pela MVCASE é a mesma usada pela ferramenta Rational Rose [Rat04], para a persistência dos elementos modelados. Dessa forma, o desenvolvedor pode importar na MVCASE, especificações geradas pela Rational Rose e vice-versa, facilitando a portabilidade dos modelos construídos em ambas as ferramentas.

Uma vez especificados e projetados os modelos, a MVCASE disponibiliza ao Engenheiro de Software a opção de gerar seu código em uma linguagem Orientada a Objetos, atualmente a linguagem Java. Este código poderá ser mais completo se o Engenheiro de Software especificar o comportamento dos métodos diretamente em Java, caso contrário, apenas o código das estruturas das classes com seus respectivos atributos, protótipos de métodos e relacionamentos é gerado.

A MVCASE suporta também o Desenvolvimento Baseado em Componentes (DBC), disponibilizando a implementação dos componentes nas tecnologias *Enterprise Java Beans* (EJB) [Sun04a, Sun04b], *Common Object Request Broker Architecture* (CORBA) [Omg04] e Delphi [Del04]. Atualmente, a MVCASE também possui recursos para a geração automática dos artefatos necessários à implementação dos padrões *Business Delegate* [Sun04d] e *Value Object* [Sun04d].

Depois de modelados os componentes, a MVCASE pode também gerar o código da sua implementação. São gerados o código Java e descrições XML [Xml03] que disponibilizam os componentes num servidor EJB, para utilização em diferentes aplicações.

A MVCASE suporta a construção de componentes segundo uma arquitetura multicamadas, separando aspectos de apresentação e interface com o usuário das implementações das regras de negócio e dos serviços de armazenamento em banco de dados ou outro meio de armazenamento.

Baseado nos conceitos, tecnologias e ferramentas apresentadas, e nas experiências com outros projetos de reengenharia, definiu-se o método RSCT, que será apresentado a seguir.

Capítulo 3

Reengenharia de Software Orientada a Componentes usando Transformações (RSCT)

O método RSCT, pesquisado neste trabalho, estende o RST com recursos para tratar a reengenharia baseada em componentes. Esta idéia foi motivada principalmente pelas experiências com o RST. Tanto este pesquisador como seu orientador participaram do desenvolvimento do RST e acreditam que, com a reengenharia de vários sistemas legados de um domínio do problema específico, é possível extrair componentes que constituem padrões de projeto para o referido domínio. Esses componentes podem ser reutilizados tanto na reconstrução de sistemas legados quanto na construção de novos sistemas. O aproveitamento do resultado do projeto RST possibilita dar continuidade à pesquisa de reengenharia que vem sendo realizada no DC-UFSCar [Fuk99, Jes99, Fon02a, Fon04, Mor02, Nog02, Nov02].

Baseado nos estudos apresentados no Capítulo 2 desta tese, pesquisou-se um Método de Reengenharia de Software Orientada a Componentes usando Transformações, cujos objetivos são a construção de componentes a partir dos Projetos Orientados a Objetos recuperados do sistema legado e o reuso destes componentes para reconstruir os sistemas legados Orientados a Componentes e/ou construir novos sistemas deste domínio. O RSCT combina idéias de Engenharia de Domínio e de Reengenharia de Sistema, conforme mostra a Figura 3.1.

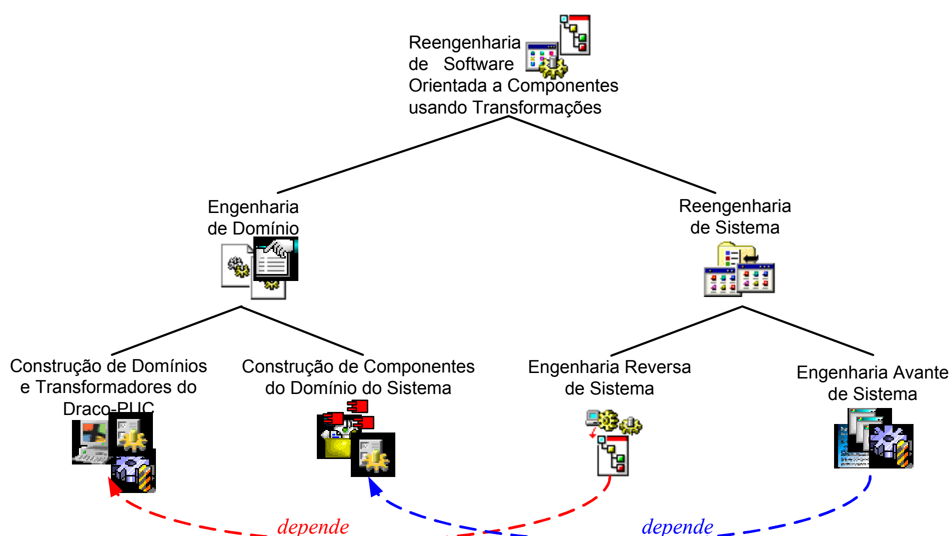


Figura 3.1 - Reengenharia de Software Orientada a Componentes usando Transformações

Para organizar as atividades do RSCT, foram definidas 04 (quatro) fases distintas: *Construir Domínios e Transformadores (Fase 1)*, *Obter Projeto Orientado a Objetos (Fase 2)*, *Construir Componentes (Fase 3)* e *Reconstruir Sistema (Fase 4)*.

A Engenharia de Domínio é realizada nas Fases 1 e 3 e compreende duas atividades: Construção de Domínios e Transformadores do Draco-PUC e Construção de Componentes do Domínio do Sistema. A Reengenharia de Sistema compreende as atividades de Engenharia Reversa de Sistema e de Engenharia Avante de Sistema, e corresponde às Fases 2 e 4 do RSCT.

As Fases 1 e 2 foram desenvolvidas no projeto RST [Fon04], no qual este pesquisador e seu orientador participaram. Como resultado do Projeto RST, definiu-se um método de Reengenharia de Software, denominado RST, que reconstrói sistemas de software legados usando transformações. O Método RST tem uma abordagem voltada para a reconstrução de um sistema legado seguindo as idéias do paradigma Orientado a Objetos. Seu foco principal está na obtenção de um projeto Orientado a Objetos do sistema legado e sua reimplementação em uma linguagem Orientada a Objetos. Embora as Fases 1 e 2 já tivessem sido desenvolvidas, durante as pesquisas deste projeto foram necessárias adaptações para facilitar a obtenção e o reuso dos componentes, conforme objetivos do RSCT. As Fases 3 e 4 do método RSCT constituem as extensões do RST realizada neste projeto de pesquisa.

A Figura 3.2 resume num Diagrama SADT estendido[Ros77], as fases do RSCT. As Fases 1 e 2 estão sombreadas para denotar que já existem no RST. As Fases 3 e 4 são as extensões específicas do RSCT.

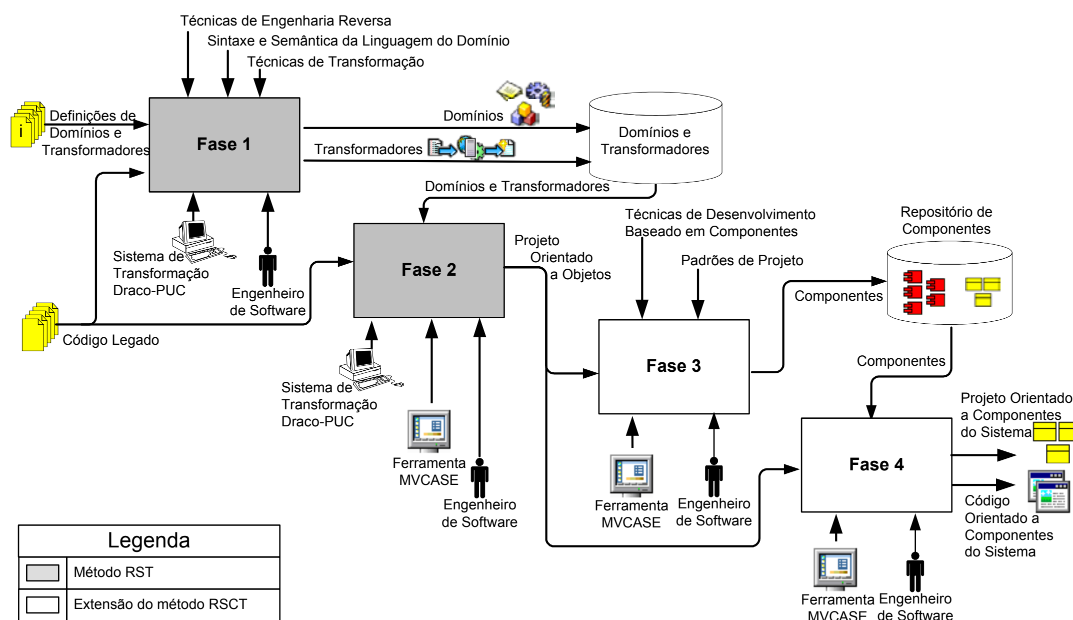


Figura 3.2 - Fases do Método RSCT

Na Fase 1 o Engenheiro de Software, a partir das definições de domínios e transformadores e do código legado do sistema, constrói os domínios e transformadores que serão reutilizados na obtenção dos projetos Orientados a Objetos de cada sistema legado. Na Fase 2, o Engenheiro de Software reutiliza os domínios e transformadores construídos na Fase 1 para obter os Projetos Orientados a Objetos dos sistemas legados, a partir de seus códigos legados. Na Fase 3, o Engenheiro de Software parte dos projetos Orientados a Objetos dos sistemas legados para construir os componentes do domínio do problema usando Padrões de Projeto. No final desta fase têm-se os componentes disponíveis em uma biblioteca para reuso. Na Fase 4, a partir do projeto Orientado a Objetos do sistema legado obtido na Fase 2, o Engenheiro de Software reconstrói o sistema legado, reutilizando os componentes disponíveis na biblioteca. Conforme mostra a Figura 3.2, os principais mecanismos de apoio ao Engenheiro de Software são o sistema de transformação Draco-PUC e a ferramenta MVCASE.

Uma vez tendo obtido os componentes, pode-se reconstruir sistemas antigos ou mesmo construir novos sistemas que reutilizem estes componentes. Dessa forma, tem-se uma redução de código, tempo e recursos tanto na reconstrução quanto na construção de aplicações de um domínio.

Com os objetivos de auxiliar na definição de suas fases e de verificar a viabilidade do RSCT, foram utilizados 03 (três) estudos de caso do domínio de vendas de produtos. Estes estudos de caso serão apresentados como exemplos de aplicação das fases deste método, visando um melhor entendimento do Engenheiro de Software. Dois sistemas legados são escritos em Clipper e serviram para testar todo o processo de reengenharia proposto. Outro sistema escrito em Object Pascal (Delphi) foi desenvolvido segundo o paradigma Orientado a Objetos e tinha sua documentação atualizada, não sendo, portanto, necessário realizar as Fases 1 e 2 para obter seu Projeto Orientado a Objetos. Assim, este sistema foi usado para testar as Fases 3 e 4, servindo principalmente para generalizar os componentes obtidos e facilitar a identificação de padrões de projeto do domínio do problema. Segue uma apresentação de cada uma das fases do RSCT.

3.1. Fase 1 – Construir Domínios e Transformadores

A primeira fase do Método RSCT, *Construir Domínios e Transformadores*, visa construir os domínios e transformadores necessários para a utilização do Sistema de

Transformação Draco-PUC na segunda fase do Método RSCT. Esta fase compreende duas etapas: *Construir Domínio* e *Construir Transformador*, conforme mostra a Figura 3.3.

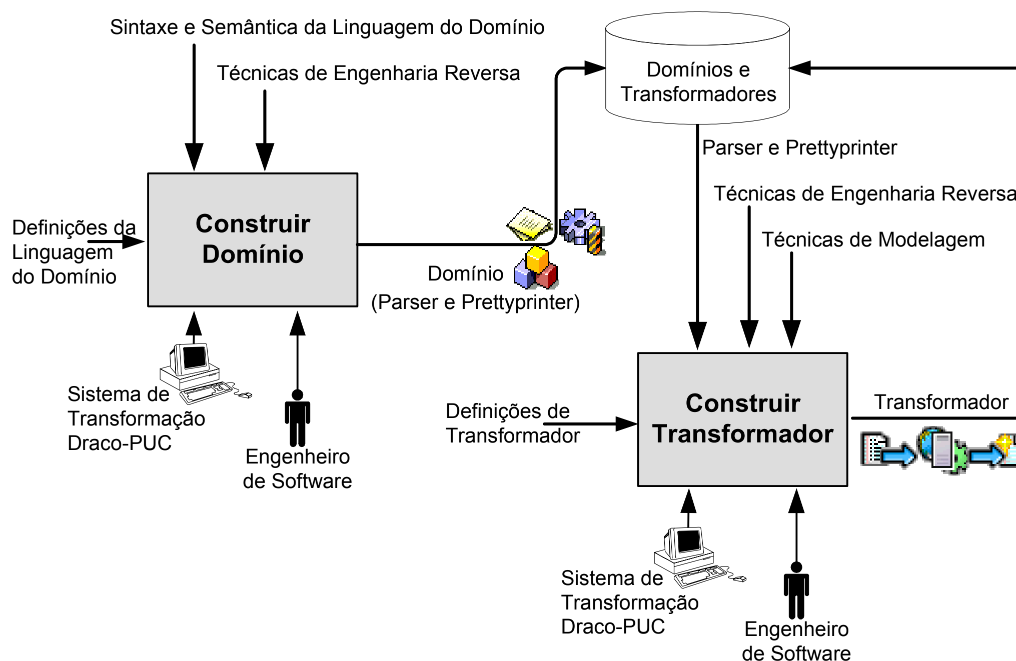


Figura 3.3 - Construir Domínios e Transformadores (Fase 1)

A construção de Domínios e Transformadores é uma atividade que requer do Engenheiro de Software, um bom conhecimento sobre o Sistema de Transformação Draco-PUC. A seguir, serão apresentadas com detalhes as etapas que compõem a Fase 1 do RSCT.

3.1.1. Construir Domínio

Um Domínio no Sistema de Transformação Draco-PUC é definido por uma linguagem. Esta, por sua vez, é definida por uma Gramática, que tem um *parser* e um *prettyprinter*. Para a construção de domínios, o Engenheiro de Software edita a gramática da linguagem e gera o seu *parser* e *prettyprinter*. Testes são realizados e caso sejam verificados erros na gramática, o Engenheiro de Software poderá retornar ao passo Editar Gramática para fazer as correções necessárias e depois gerar novamente seu *parser* e *prettyprinter*, conforme mostra a Figura 3.4. Esse *feed-back* ao passo inicial para a correção de erros baseia-se nas idéias do modelo espiral de desenvolvimento de software, onde a cada ciclo, que compreende a execução dos três passos da etapa *Construir Domínios*, tem-se uma nova versão do domínio.

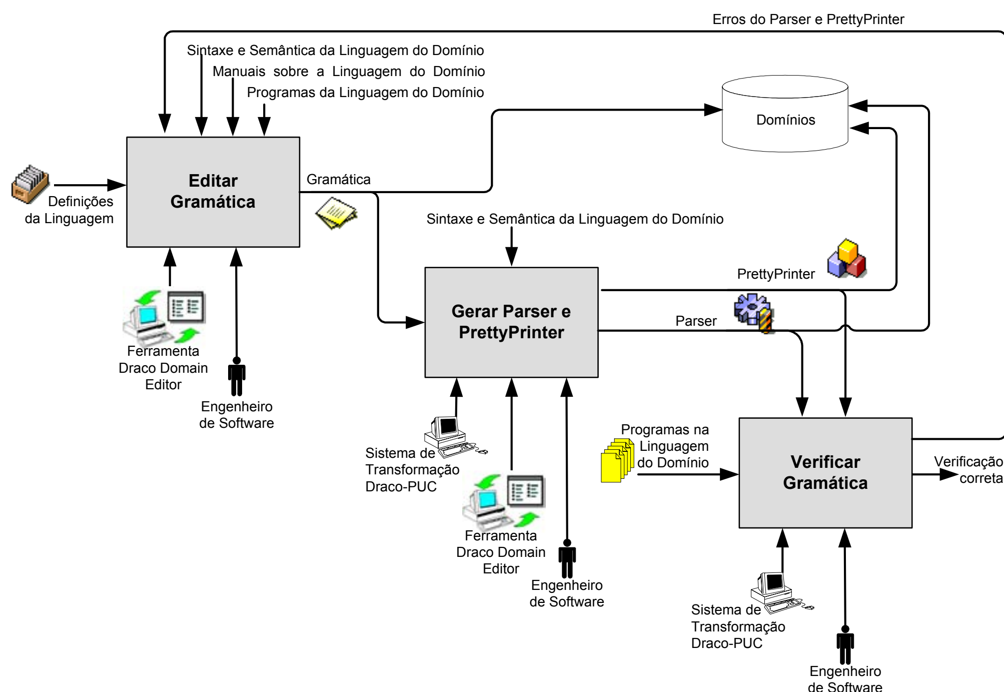


Figura 3.4 - Passos da Etapa Construir Domínio

No passo Editar Gramática, o Engenheiro de Software parte das definições da linguagem para obter a gramática do domínio. A execução deste passo é orientada pela sintaxe e semântica, programas, e manuais sobre a linguagem do domínio. A ferramenta Draco Domain Editor [Gar02] auxilia na edição das gramáticas das linguagens dos domínios.

A Figura 3.5 mostra, por exemplo, a definição de regras de produção da gramática do domínio Clipper, editada pelo Engenheiro de Software utilizando o editor de gramáticas da Ferramenta Draco Domain Editor.

```
clipper.grm
%left '.OR.'.
%left '.AND.'.
%left   '.NOT.'   '!'   .
%left   '<'  '<='   '>='  GT.
%left   '='   '=='   '!='   '<>'   '#'   '$' .
%left   ':='  '+='  '-='   '*='   '/='  '^='   '%='.
%left '+'    '-'.
%left '*'    '/'    '%'.
%left '**'   '^'.
%left   '++'  '--'.
%% .
program :  prog_elements.
        ;.
prog_elements : body_struct ('RETURN' expr?)?.
              | prog_extra+ (prog_diretive+)?.
              ;.
body_struct     : prog_body.
        | var_decl.
        | .sp .sp .sp 'BREAK' .nl.
        | .sp .sp .sp 'LOOP' .nl.
        | .sp .sp .sp 'EXIT' .nl.
              ;.
var_decl        : ('LOCAL' | 'PUBLIC' | 'PRIVATE' | 'STATIC')
                  | 'FIELD' .sp id_list (.sp 'IN' var)? .nl.
```

Figura 3.5 - Regras de produção da gramática Clipper na ferramenta DDE

Depois de editada uma gramática, o Engenheiro de Software, usando o Sistema de Transformação Draco-PUC, gera os respectivos *parser* e *prettyprinter*.

A geração do *parser* e do *prettyprinter* pode ser executada na própria ferramenta Draco Domain Editor ou então diretamente no Sistema de Transformação Draco-PUC. Na ferramenta Draco Domain Editor, o Engenheiro de Software carrega o domínio e depois utiliza os subsistemas *pargen* e *ppgen* do Sistema de Transformação Draco-PUC para gerar o *parser* e o *prettyprinter*. Caso o Engenheiro de Software deseje gerar o *parser* e o *prettyprinter* diretamente no Sistema de Transformação Draco-PUC, basta utilizar um script do tipo *makefile* com comandos para chamar o *pargen* e o *ppgen,* conforme mostra a Figura 3.6.

```
                        # This makefile uses Draco makefile conventions #
include ../../mkfiles/macros.mk

DOMAIN_NAME    = clipper
CASE_INSENSITIVE_LEX = -i

build          : parser pprinter
include ../../mkfiles/makedomain.mk
                                        # Suffix rules #

include ../../mkfiles/suffix.mk
```

Figura 3.6 - Script *makefile* para geração do *parser* e *prettyprinter*

Erros poderão ocorrer durante a geração do *parser* e *prettyprinter*. Estes erros podem ser tanto de origem léxica quanto de origem sintática e, para ambos os casos, os subsistemas *pargen* e *ppgen* não conseguem gerar o *parser* e *prettyprinter.*Caso ocorram erros desta natureza, o Engenheiro de Software edita novamente a gramática procurando corrigir os erros encontrados e depois gerar novamente seu *parser* e *prettyprinter*.

Após a geração do *parser* e *prettyprinter*, o Engenheiro de Software poderá submeter ao domínio, programas escritos na linguagem do domínio para verificar a existência de erros na gramática. A Figura 3.7 mostra um script de execução por meio do qual se submete um programa ao domínio Clipper. O comando "*load-domain draco dast*" carrega os domínios do Draco-PUC junto com seus subsistemas. O comando "*load-domain clipper prg*" carrega o domínio Clipper associado com uma extensão de arquivo a este domínio, no caso a extensão "**.prg*". O comando "*parse %1 l*" analisa gramaticalmente o programa associado no parâmetro "*%1*". E por fim o comando "*pp l*", chama o *prettyprinter* para mostrar o programa na linguagem do domínio Clipper.

```
(begin
  (load-domain draco dast)
  (load-domain clipper prg)
  (parse %1 l)
  (pp l)
  (exit)
)
```

Figura 3.7 - Script de execução para submeter um programa ao domínio Clipper

Caso sejam encontrados erros, o Engenheiro de Software retorna ao passo Editar Gramática e faz as correções necessárias, para gerar novamente o seu *parser* e *prettyprinter* e assim recursivamente até que se tenha uma versão correta do domínio desejado.

Durante a aplicação da Etapa *Construir Domínio* nos sistemas legados dos estudos de caso, foram construídos os domínios Clipper, MDL e Java apresentados na Tabela 3.1.

Domínio	Finalidade
Clipper	Define a gramática da linguagem Clipper do sistema legado, que dá origem aos seus respectivos *parser* e *prettyprinter.*
MDL	Define a gramática da linguagem MDL de modelagem, que dá origem aos seus respectivos *parser* e *prettyprinter.*
Java	Define a gramática da linguagem Java, que dá origem aos seus respectivos *parser* e *prettyprinter*

Tabela 3.1 - Domínios construídos na Etapa Construir Domínio da Fase 1 do RSCT

Depois de construídos os domínios, o Engenheiro de Software constrói os transformadores que serão utilizados pelo Sistema de Transformação Draco-PUC, na Fase 2 do RSCT.

3.1.2. Construir Transformadores

Orientado por técnicas de Engenharia Reversa e de Modelagem, o Engenheiro de Software parte de definições das transformações para construir os transformadores usados na fase seguinte do RSCT.

Na obtenção do projeto Orientado a Objetos do sistema legado são empregados diversos transformadores. Os transformadores de identificação de supostos elementos Orientados a Objetos no código legado, visam identificar supostas classes com seus supostos atributos, supostos métodos, relacionamentos, dentre outros. Os transformadores de organização visam organizar o código legado segundo os princípios da Orientação a Objetos. Os transformadores de recuperação do projeto visam recuperar o projeto Orientado a Objetos do sistema legado.

Para construir um transformador, o Engenheiro de Software deve: *Editar Regras de Transformações, Gerar Transformador e Verificar Transformador*, conforme mostra a Figura 3.8.

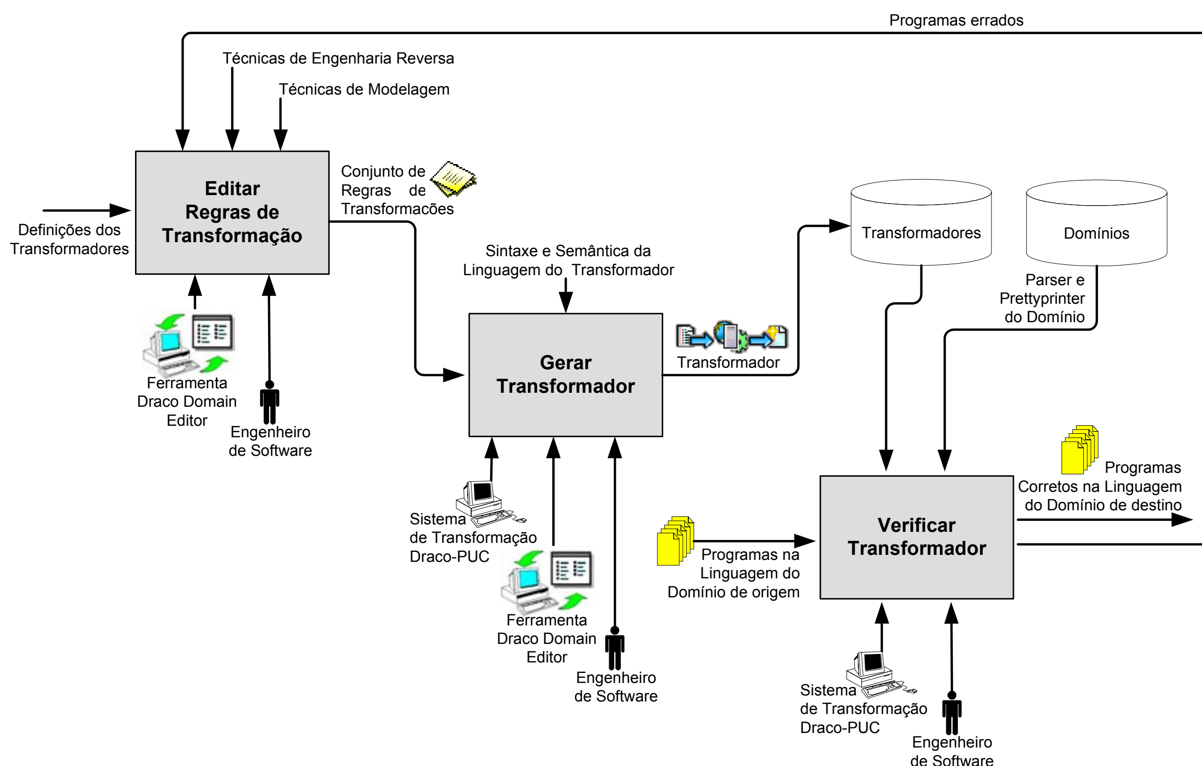


Figura 3.8 - Passos da Etapa Construir Transformador

Os transformadores podem ser editados em qualquer editor de textos. Contudo, para facilitar a edição, o Engenheiro de Software pode utilizar a ferramenta Draco Domain Editor, que dispõe de recursos para a edição e compilação dos transformadores junto ao Sistema de Transformação Draco-PUC.

No passo Editar Regras de Transformação, o Engenheiro de Software edita as regras de transformação, orientado por técnicas de Engenharia Reversa e de Modelagem. Para que se tenha uma idéia das atividades deste passo, conforme apresentado a seguir, para os transformadores usados na Fase 2 do RSCT.

3.1.2.1. Transformador de Identificação

Para orientar a identificação das informações do código do sistema legado e sua organização segundo o paradigma Orientado a Objetos, foi definido um metamodelo, na notação UML [Omg03], cujas instâncias são modelos Orientados a Objetos. O metamodelo com suas metaclasses (denotadas pelo estereótipo << *metaclass*>>), permite representar um projeto segundo diferentes visões, que podem ser: "*Logical*", "*UseCase*" e "*Component*", conforme mostra a Figura 3.9.

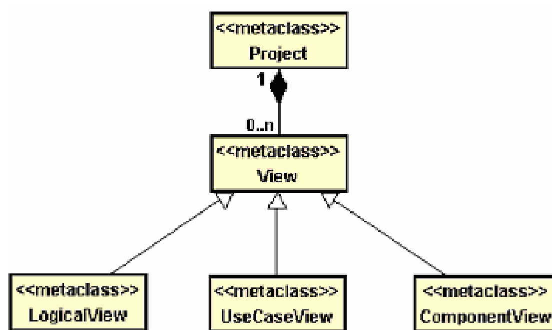


Figura 3.9 – Visão geral do metamodelo

A visão Lógica (*LogicalView*), apresentada na Figura 3.10, possui Modelos de Classes (*ClassModel*) constituídos de classes (*Class*) e seus relacionamentos. Uma classe é composta de atributos (*Attribute*), e métodos (*Method*), e pode estar relacionada com outras classes. Um relacionamento pode ser de dependência (*Dependency*), associação (*Association*), realização (*Realize*) ou de generalização (*Generalization*). Um método pode ter parâmetros (*Parameter*), e ser especializado em construtor (*Constructor*) e observador (*Observer*).

O auto-relacionamento Chama/Chamado (*Call/Called*) da metaclasse Método (*Method*) é usado para representar a hierarquia de chamadas das operações, que identifica o fluxo de execução do código do sistema legado.

Para organização dos diferentes elementos que compõem os modelos tem-se a metaclasse Elemento (*Element*) que se especializa em Modelo (*Model*), que representa a generalização dos Modelos de Classes (*ClassModel*), e Classificador (*Classifier*), que classifica os elementos e seus relacionamentos.

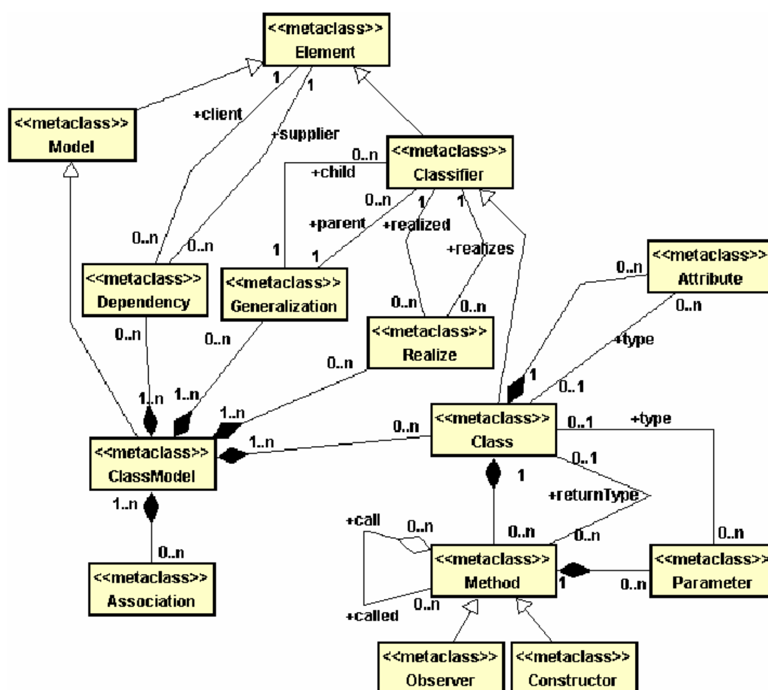


Figura 3.10 - Visão Lógica (*LogicalView*)

A Figura 3.11 mostra a visão de Caso de Uso (*UseCaseView*), que possui modelos de Seqüência (*SequenceModel*) e de Caso de Uso (*UseCaseModel*). Um Modelo de Seqüência representa a troca de mensagens (*Message*) entre objetos (*Object*) ou métodos (*Method*) de uma classe (*Class*).

Um Modelo de Caso de Uso (*UseCaseModel*) possui atores (*Actor*), um tipo especial de classe (*Class*), e Casos de Uso (*UseCase*). Os Casos de Uso podem estender (*extend*) ou incluir (*include*) outros Casos de Uso e possui relacionamentos de associação (*Association*), dependência (*Dependency*), realização (*Realize*) e de generalização (*Generalization*).

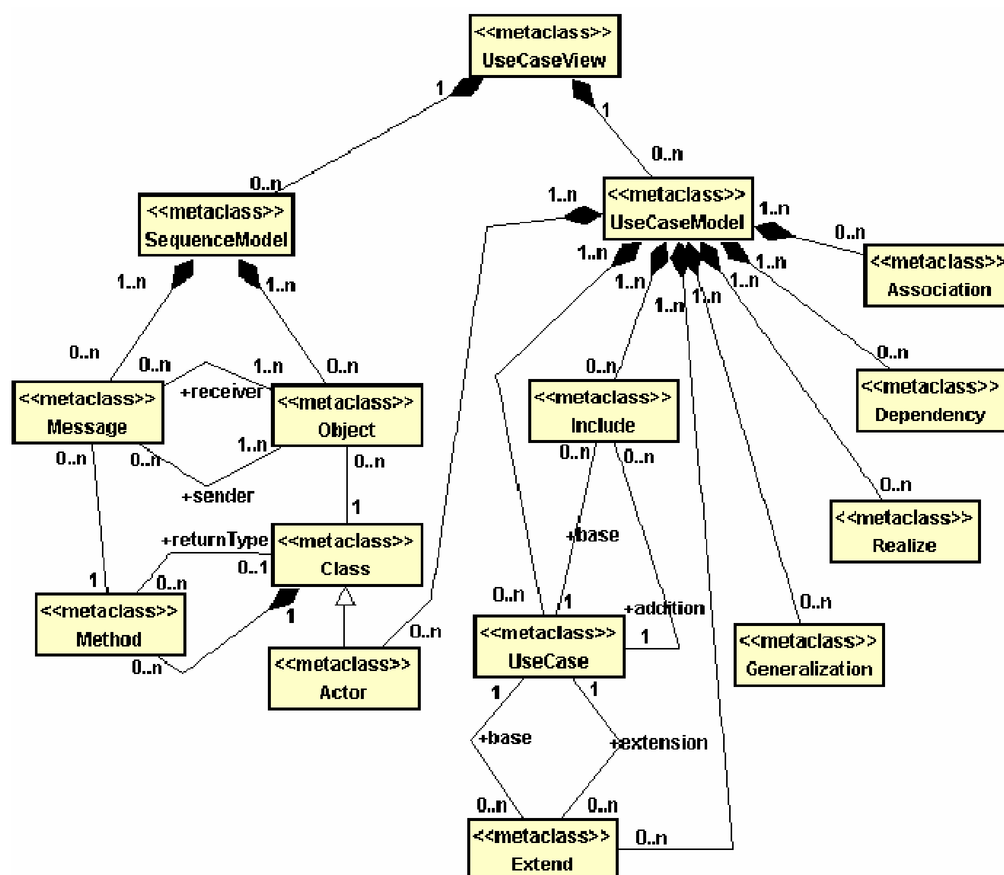


Figura 3.11 - Visão de Caso de Uso (*UseCaseView*)

A visão de Componentes (*ComponentView*), apresentada na Figura 3.12, possui modelos de Componentes (*ComponentModel*) que representam os componentes (*Component*) do sistema, e as dependências (*Dependency*) entre eles. Um componente representa um módulo do sistema. Sendo assim, ele pode agregar zero ou mais elementos, como classes e interfaces, por exemplo.

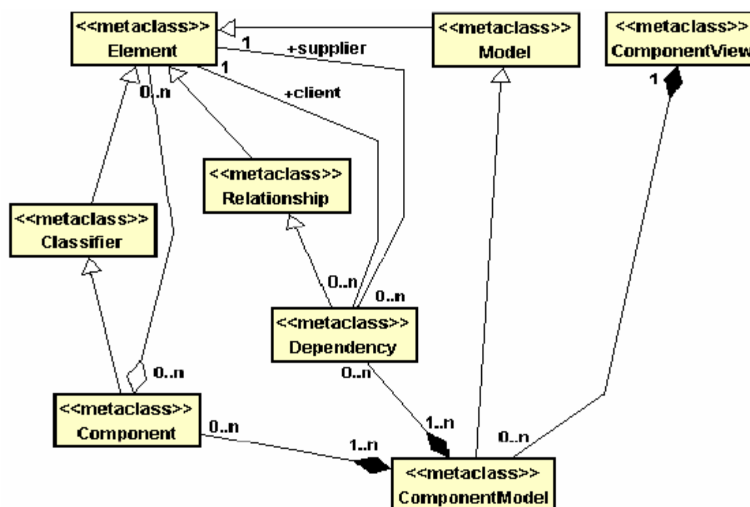


Figura 3.12 Visão de Componentes (*ComponentView*)

A metaclasse *Element* representa os elementos do metamodelo. Conforme mostra a Figura 3.13, um elemento pode ser um modelo (*Model*), um classificador (*Classifier*) ou relacionamento (*Relationship*). Um modelo pode ser um modelo de classes (*ClassModel*), casos de uso (*UseCaseModel*), seqüência (*SequenceModel*) ou de componentes (*ComponentModel*). Um classificador representa os elementos de um modelo que podem ser relacionados, como classes, casos de uso, componentes e atores. São tratados sete tipos de relacionamentos:

• **Dependência** (*Dependency*): representa uma dependência entre dois elementos quaisquer do metamodelo;

• **Realização** (*Realize*): representa a realização de um classificador por outro. Por exemplo, uma classe que realiza um caso de uso, ou um componente que implementa uma interface (tipo especial de classe);

• **Generalização** (*Generalization*): representa uma generalização entre classificadores, por exemplo, herança entre classes ou casos de uso;

• **Associação** (*Association*): representa a associação entre dois classificadores. Por exemplo, um ator associado a um caso de uso ou um relacionamento entre classes. Também é possível representar uma agregação, através do atributo *isAggregate*;

• **Extend**: representa um relacionamento opcional em que um caso de uso pode estender outro;

• **Include**: representa um relacionamento obrigatório em que um caso de uso utiliza outro caso de uso; e

• **Mensagem** (*Message*): representa uma conexão de mensagem entre objetos.

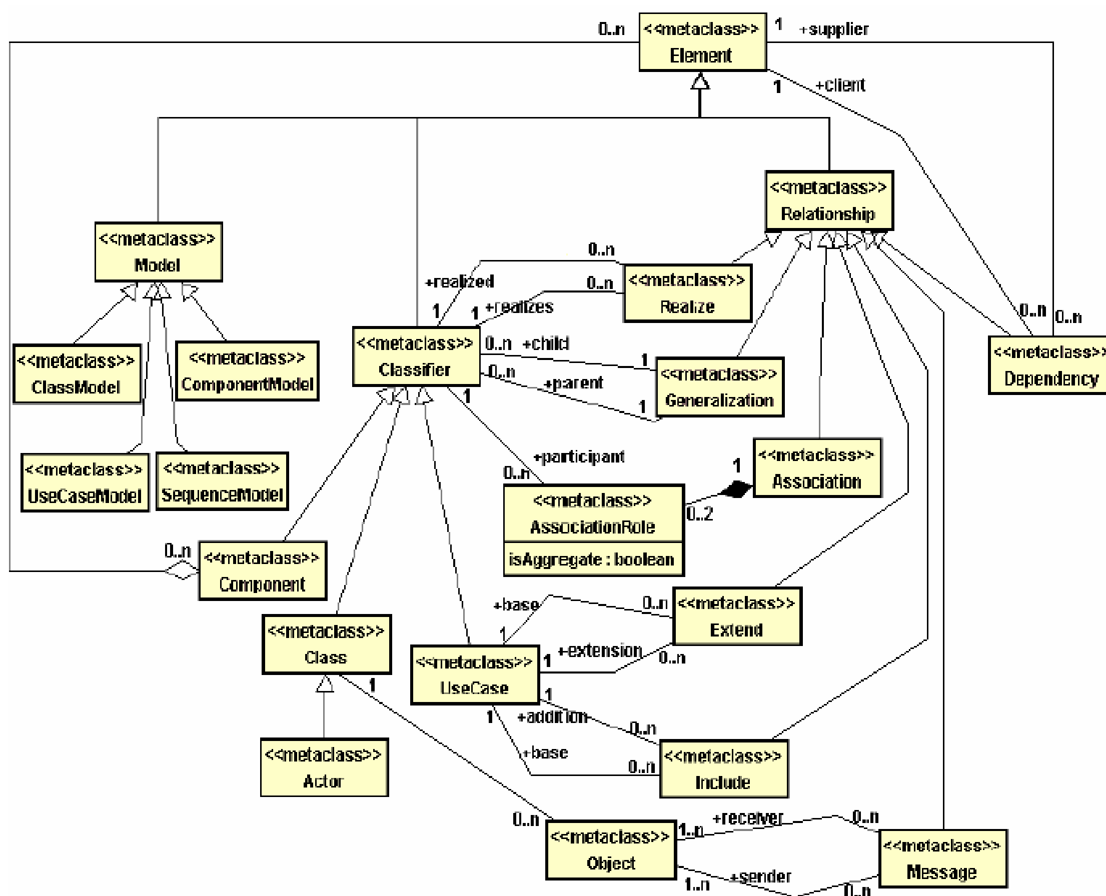


Figura 3.13 - Relacionamentos do Metamodelo

Baseado no metamodelo e nas informações da gramática do domínio são identificados cada elemento que irá compor o modelo Orientado a Objetos do sistema legado a ser recuperado, conforme se segue.

3.1.2.1.1. Identificar Supostas Classes e Supostos Atributos

As supostas classes e seus supostos atributos podem ser identificados com base nos comandos de abertura e acesso a arquivos ou banco de dados, comandos relacionados com as estruturas de dados e com as interações dos atores com as interfaces do sistema legado.

Assim, supostas classes e seus supostos atributos, podem ser obtidos a partir de:

- **Arquivos de dados**: supostas classes são identificadas com base nos comandos de abertura e criação de arquivos de dados. Os supostos atributos são identificados com base nos campos destes arquivos.

- **Banco de Dados**: supostas classes são identificadas com base nos comandos de abertura e criação das tabelas de banco de dados. Os supostos atributos são identificados com base nos campos destas tabelas.

• **Estruturas de Dados**: supostas classes são identificadas com base nos comandos de definição de estruturas de dados. Os supostos atributos são identificados com base nos campos destas estruturas de dados.

• **Unidades de Programas**: supostas classes de interface são identificadas com base nas unidades de programas responsáveis pelas interações com os atores do sistema legado. Os supostos atributos são identificados com base nos elementos de interação das unidades de programas.

As supostas classes são organizadas em transientes e persistentes visando facilitar o reuso e a manutenção do sistema. A Figura 3.14 mostra, por exemplo, uma transformação denominada "*IdentifySupposedClassByUseStatement*" que identifica supostas classes persistentes. Essa transformação possui um padrão de reconhecimento (*LHS*) que identifica comandos de abertura, comandos de acesso aos arquivos ou banco de dados e comandos relacionados com as estruturas de dados, gerando um fato "*SupposedClass*" referente a esse arquivo. No caso, a transformação "*IdentifySupposedClassByUseStatement*" possui o padrão de reconhecimento que identifica o comando de acesso aos arquivos "*USE*", do domínio Clipper. A função "*KBAssertIfNew*" (1) cria um novo fato sobre uma suposta classe na Base de Conhecimento. A função "*IdentifySupposedAttribute*" (2) acessa o arquivo de dados especificado pelo comando "*USE*", para obter informações sobre os campos deste arquivo, identificando-os como supostos atributos de uma Suposta Classe. Finalmente a função "*KBAssertIfNew*" (3) cria fatos na Base de Conhecimento sobre cada suposto atributo.

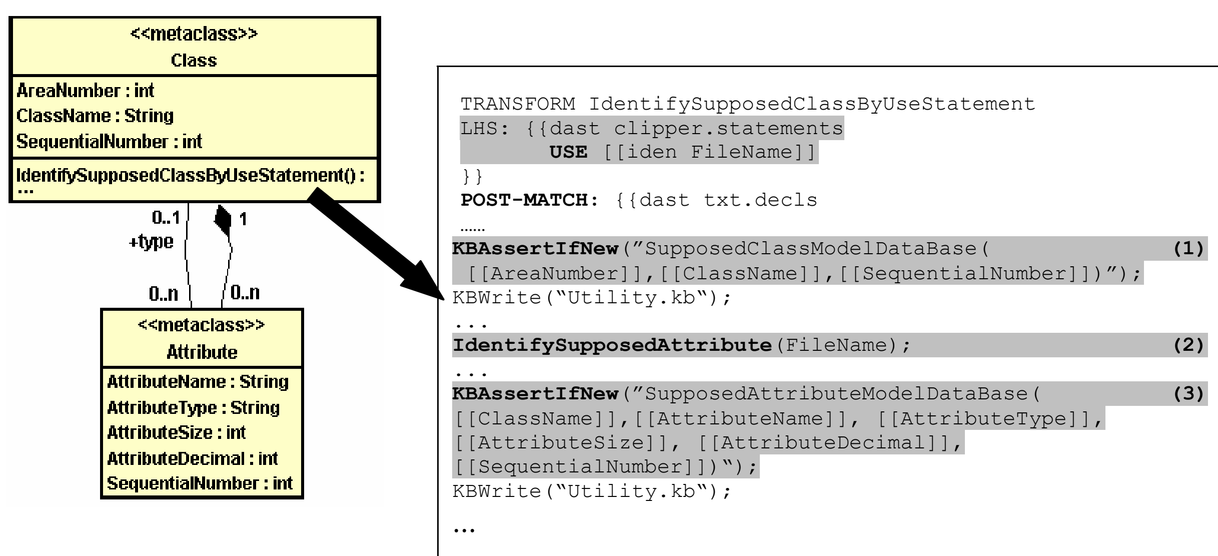


Figura 3.14 - Transformação para identificação de supostas classes persistentes e atributos

A Figura 3.15 mostra, em outro exemplo, uma transformação que identifica uma suposta classe transiente, no caso uma classe de interface, através do comando de interação "*GET PICTURE*" (1) do Clipper. A metavariável "*[[Expr_Get]]*" (1) do comando "*GET PICTURE*", identifica um suposto atributo, do tipo "*TextField*" de uma suposta transiente, identificada pela metavariável "*[[CurrentClass]]*" (2). É identificada também a posição deste suposto atributo através das metavariáveis "*[[Row_Say]]*" e "*[[Col_Say]]*" conforme especificado na função "*KBAssertIfNew*" (3).

```
TRANSFORM IdentifySupposedClassByGetStatement
LHS: {{dast clipper.get_stat
@ [[expr_num Row_Say]],
[[expr_num Col_Say]] GET [[iden Expr_Get]] PICTURE [[expr_num Masc_Get]]          (1)
}}
POST-MATCH: {{dast txt.decls
int cont;
int CurrentClass[100];
char num[100];
int tamanho;
...
KBAssertIfNew("SupposedClassView([[CurrentClass]])");                            (2)
KBAssertIfNew("SupposedAttributeView([[Row_Say]], [[Col_Say]],                   (3)
              "TextField", [[Expr_Get]], [[CurrentClass]])");
...
KBWrite("Utility.kb");
...
```

Figura 3.15 - Identificação de uma suposta classe transiente e seus supostos atributos

Dessa forma, com base nos comandos que possam dar origem a uma suposta classe, são escritas as transformações de identificação de supostas classes e seus supostos atributos. A Figura 3.16 apresenta, por exemplo, outras transformações que identificam supostas classes e supostos atributos do código no sistema legado Clipper.

Comando Clipper	Transformação

TAG ID	`TRANSFORM IdentifySupposedClassByProgramWithTag`
	`LHS: {{ dast clipper.program`
	`[[TAG tag]] [[ID main_proc]] [[prog_elements elements]] }}`

SELECT expression	`TRANSFORM IdentifySupposedClassBySelectStatement`
	`LHS: {{ dast clipper.statements`
	`SELECT [[ID express2]] }}`

USE file_name	`TRANSFORM IdentifySupposedClassUseStatement`
	`LHS: {{ dast clipper.statements`
	`USE [[iden file_name]] }}`

USE file_name param	`TRANSFORM IdentifySupposedClassUseStatementParam`
	`LHS: {{ dast clipper.statements`
	`USE [[iden file_name]] [[use_par param]] }}`

GET identifier	`TRANSFORM IdentifySupposedClassByGet`
	`LHS: {{ dast clipper.get_stat`
	`GET[[iden expr_get]] }}`

Figura 3.16 Transformações para a identificação de supostas classes e atributos do Clipper

3.1.2.1.2. Identificar Supostos Métodos

Os supostos métodos são identificados a partir das unidades de programas que podem ser programas, funções, procedimentos ou blocos de código. Baseado no metamodelo, a Figura 3.17 mostra, à esquerda, um suposto método que pertence a uma suposta classe e tem um nome (*MethodName*), parâmetros (*Parameter*), tipo de retorno (*ReturnType*), modificador (*Modifiers*) e outros atributos operacionais usados para facilitar a reengenharia como, por exemplo, um número seqüencial (*SequentialNumber*) que identifica o método. A direita da Figura 3.17 tem-se a transformação "*IdentifySupposedMethodByProcedureDeclaration*" que identifica um suposto método de uma suposta classe, a partir do bloco de comando "*PROCEDURE / RETURN*" (1). Este suposto método é alocado numa classe através do fato "*SupposedMethodModelBusinessRule*" (2).

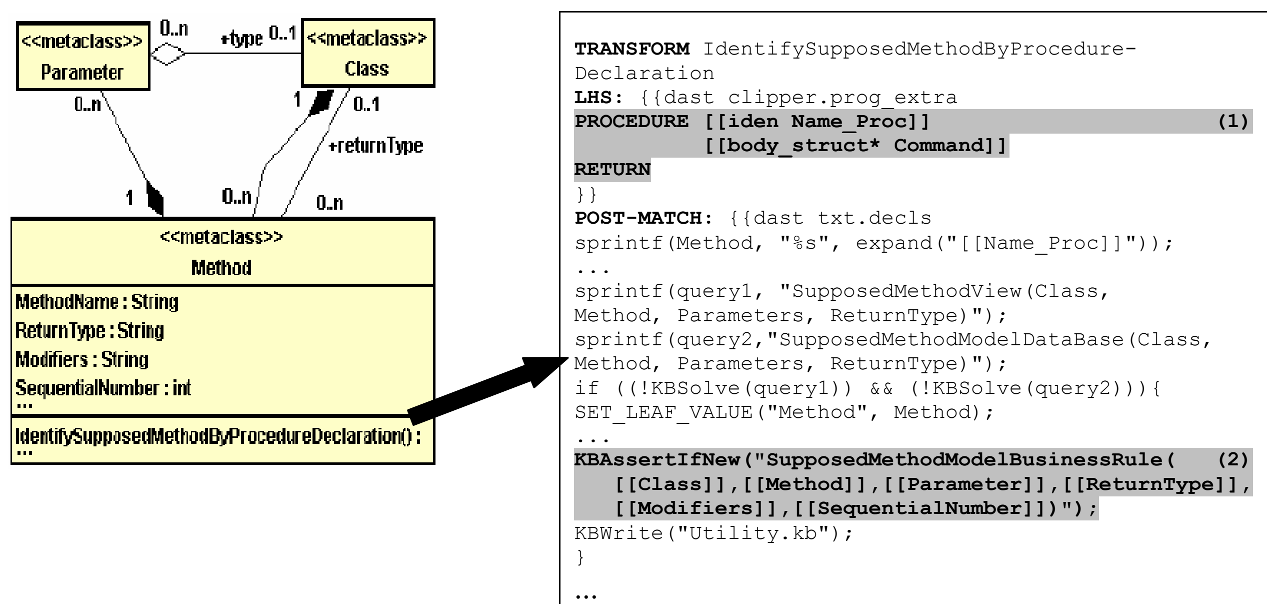


Figura 3.17 - Transformação para identificação dos supostos métodos

A Figura 3.18 mostra a transformação "*IdentifySupposedMethodByStatementBox*" que identifica um suposto método a partir do comando "*@ / TO*" (1). Em (2) são identificadas as informações que definem o protótipo de um suposto método e sua suposta classe cujo nome é guardado na metavariável "*Class*". Finalmente, as informações sobre o suposto método são armazenadas na Base de Conhecimento através do comando "*KBAssertIfNew*" (3).

```
TRANSFORM IdentifySupposedMethodByStatementBox
LHS: {{dast clipper.statements
@ [[expr_num Row_Say]],[[expr_num Col_Say]] TO [[expr_num_list Expr1]]          (1)
}}
POST-MATCH: {{dast txt.decls
...
SET_LEAF_VALUE("Class", CurrentClass);                                          (2)
SET_LEAF_VALUE("Method", CurrentMethod);
SET_LEAF_VALUE("Parameters", CurrentParameters);
SET_LEAF_VALUE("ReturnType", CurrentReturnType);
SET_LEAF_VALUE("Modifiers", CurrentModifiers);
SET_LEAF_VALUE("SequentialNumber", CurrentSequentialNumber);
...
KBAssertIfNew("SupposedMethodView([[Class]],[[Method]],[[Parameters]],          (3)
              [[ReturnType]],[[Modifiers]],[[SequentialNumber]])");
KBWrite(kb);
...
}
```

Figura 3.18 Identificação de um suposto método de uma classe transiente

A Figura 3.19 mostra, por exemplo, outra transformação denominada "*IdentifySupposedMethodByAppendStatement*", que identifica um suposto método de uma classe persistente, com base no comando "*APPEND BLANK*" (1) do Clipper. Da mesma forma que na transformação anterior, em (2) são identificadas as informações do protótipo do suposto método e sua respectiva suposta classe. Finalmente, o comando"*KBAssertIfNew*" (3) armazena as informações sobre o suposto método na Base de Conhecimento.

```
TRANSFORM IdentifySupposedMethodByAppendStatement
LHS: {{dast clipper.statements
APPEND BLANK                                                                    (1)
}}
POST-MATCH: {{dast txt.decls
...
SET_LEAF_VALUE("Class", CurrentClass);                                          (2)
SET_LEAF_VALUE("Method", CurrentMethod);
SET_LEAF_VALUE("Parameters", CurrentParameters);
SET_LEAF_VALUE("ReturnType", CurrentReturnType);
SET_LEAF_VALUE("Modifiers", CurrentModifiers);
SET_LEAF_VALUE("SequentialNumber", CurrentSequentialNumber);
...
KBAssertIfNew("SupposedMethodModelDataBase([[Class]],[[Method]],               (3)
              [[Parameters]],[[ReturnType]],[[Modifiers]],[[SequentialNumber]])");
...
}
```

Figura 3.19 - Identificação de um suposto método de uma classe persistente

Os supostos métodos de uma classe persistente, conforme a forma que manipulam os dados, podem ser classificados em [Pen96]:

a) construtores **(c)**, quando alteram as informações de arquivos ou tabelas de dados;

b) observadores **(o)**, quando somente consultam informações de arquivos ou tabelas de dados; e

c) observadores-construtores **(oc)**, quando alteram e consultam informações de arquivos ou tabelas de dados.

Quando um suposto método construtor **(c)** e/ou observador **(o)** 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)**, é alocado na suposta classe que faz referência como construtor;
- Se observador de uma suposta classe e construtor de várias outras classes **(oc+)**, é alocado na primeira suposta classe que faz referência como construtor;
- Se observador de mais de uma suposta classe e construtor de apenas uma **(o+c)**, é alocado na primeira suposta classe que faz referência como observador.

A Figura 3.20 mostra, por exemplo, a transformação "*IdentifySupposedMethodByFunctionDeclaration*" que identifica supostos métodos construtores de uma suposta classe, com base nos comandos "*delete*" e "*replace*" (1), que alteram dados, e supostos métodos observadores, com base no comando "*Seek*" (2), que apenas consulta dados.

```
TRANSFORM IdentifySupposedMethodByFunctionDeclaration
LHS: {{dast clipper.prog_extra
     FUNCTION [[iden Name_Func]]
          [[body_struct* Command]]
     RETURN [[expr Return]]
}}
POST-MATCH: {{dast txt.decls
...
cont_aux = 1;
...
while ((cont_aux <= cont_fact_rel) && (strcmp(Method_Constructor, "") != 0)){  ┐
   sprintf(query, "CommandMethod(%d, %s, *x, *y, %s)", cont_aux, "delete",      │
                                                   Method_Constructor);          │
   sprintf(query1, "CommandMethod(%d, %s, *x, *y, %s)", cont_aux, "replace",     ├─ (1)
                                                   Method_Constructor);          │
   KBAssert("SupposedMethodConstructor([[Class]], [[Method_Constructor]])");     │
   KBWrite(kb);                                                                  ┘
}  ...
contaux = cont;
while (contaux <= cont_fact_rel){                                                ┐
 sprintf(query, "CommandMethod(%d, *x, *y, *z, %s)", contaux,Method_Observer);   │
 ...                                                                             ├─ (2)
 if (strcmp(type_rel, "Seek") == 0){                                             │
   KBAssertIfNew("SupposedMethodObserver([[Class]], [[Method_Observer]])");      │
   KBWrite(kb); }                                                                ┘
 }
...
}
```

Figura 3.20 - Identificação de supostos métodos construtores e observadores

A Figura 3.21 mostra parte da implementação destes critérios usados para orientar a alocação dos métodos em suas supostas classes. São verificadas as supostas classes que o suposto método referencia como "*construtor*" (1) e "*observador*" (2). As variáveis

"*count_constructor*" e "*count_observer*", armazenam o número de vezes o suposto método referencia as supostas classes na condição de "*construtor*" e/ou de "*observador*". Com base nestas variáveis, determina-se a classe na qual o método é alocado (3).

```
sprintf(Constructor, "SupposedMethodConstructor(*x, %s)", Method);       ┐
 while ((KBSolve(Constructor)) {                                         │
   sprintf(ClassC, "%s", KBRetrieve("*x", Method));                      ├ 1
   sprintf(ClassesC[cont_constructor], ClassC);                          │
   cont_constructor ++;                                                  │
   sprintf(Constructor, "SupposedMethodConstructor(*x, %s)", Method);    ┘
 } ...
 sprintf(Observer, "SupposedMethodObserver(*x, %s)", Method);            ┐
  while ((KBSolve(Observer)) {                                           │
    sprintf(ClassO, "%s", KBRetrieve("*x", Method));                     ├ 2
    sprintf(ClassesO[cont_observer], ClassO);                            │
    cont_observer ++;                                                    │
    sprintf(Observer, "SupposedMethodObserver(*x, %s)", Method);         │
  }                                                                      ┘
 if (cont_constructor >= 1) && (cont_observer == 1){                     ┐
   TEMPLATE("T_Main_Proc")                                               │
     TRANSPORT_VALUE("Constructor[0]");                                  │
     TRANSPORT_VALUE("Method");                                          │
   END_TEMPLATE;                                                         │
 }                                                                       │
 if (cont_constructor == 1) && (cont_observer > 1){                      ├ 3
   TEMPLATE("T_Main_Proc")                                               │
     TRANSPORT_VALUE("Observer[0]");                                     │
     TRANSPORT_VALUE("Method");                                          │
   END_TEMPLATE;                                                         │
 } ...                                                                   ┘
```

Figura 3.21 - Organização dos supostos métodos em suas supostas classes

Da mesma forma, são construídas outras transformações que completam a identificação dos supostos métodos em todo o código legado. A Figura 3.22 mostra outras transformações que identificam supostos métodos no código do sistema legado Clipper.

Comando Clipper	Transformação

PROCEDURE name
     body
RETURN
	TRANSFORM IdentifySupposedMethodByProcedureDeclaration
	LHS: {{ dast clipper.prog_extra
	PROCEDURE [[iden Name_Proc]] ()
	   [[body_struct* Command]]
	RETURN }}

FUNCTION name
     body
RETURN expression
	TRANSFORM IdentifySupposedMethodByFunctionDeclaration
	LHS: {{ dast clipper.prog_extra
	FUNCTION [[iden Name_Func]] ()
	   [[body_struct* Command]]
	RETURN [[expr Return]] }}

REPLACE field_file WITH variable	**TRANSFORM** IdentifyReplaceStatement **LHS**: {{ dast clipper.statements REPLACE [[replacement* rep1]] WITH [[scope scp1]] }}

Figura 3.22 - Outras transformações para Identificação de supostos métodos

3.1.2.1.3. Identificar o Fluxo de Execução do Sistema Legado

O fluxo de execução do sistema legado, normalmente disparados por eventos internos ou externos, são identificados para determinar os casos de uso que representam os diferentes comportamentos do sistema. As informações sobre o fluxo são obtidas com base nas conexões de mensagens entre as unidades de programa, que pode ser um programa (*prg*), uma chamada de função (*fun*), procedimento (*pro*) ou bloco de comandos (*blo*).

As conexões de mensagens entre as unidades de programas são obtidas e mapeadas em uma relação de chamadas. Numa referência cruzada entre as unidades de programa são identificadas as unidades que utilizam outras unidades de programas, denominadas *chamadoras*, e as unidades de programa que por elas são utilizadas, denominadas *chamadas*. Esta relação é denominada Chama/Chamado.

A Tabela 3.2 mostra um exemplo de relação Chama/Chamado, onde a unidade de programa *ProgMenuPrincipal* chama as unidades de programas *CadClientes*, *CadProdutos*, *CadPedidos*, *ConRequisicoes*, *ConMovimentacoes* e *ImpRelatorios* e a unidade de programa *ProgVendas* chama as unidades de programas *CadProdutos* e *LocalizaCliente* que é uma função que está dentro de *CadClientes,* e a unidade de programa *CadClientes* chama as unidades de programas *CadCidades* e *SalvarDados*.

Chama	Chamado
ProgMenuPrincipal	CadClientes (prg), CadProdutos (prg) , CadPedidos (prg), ConRequisicoes (prg), ConMovimentacoes(prg), ImpRelatorios(prg)
ProgVendas	CadProdutos(prg), LocalizaCliente(fun) (CadClientes)
CadClientes	CadCidades(prg), SalvarDados(pro)

Tabela 3.2 - Relação Chama/Chamado

Existem ferramentas que inspecionam o código do sistema legado e geram a relação Chama/Chamado, como por exemplo a Legacy [Pen96] e a DracoKB, apresentada por [Nov02]. Outra forma de determinar a relação Chama/Chamado é usando transformações.

A Figura 3.23 mostra, por exemplo, uma transformação denominada "*IdentifyCallFunctionOrProcedure*" que implementa a identificação de chamadas de uma unidade de programa. Esta transformação tem um padrão de reconhecimento (*LHS*) que identifica chamadas de procedimentos ou funções do domínio Clipper, através do comando "*DO*" (1), e armazena essas informações na Base de Conhecimento através do comando "*KBAssertIfNew*" (2). As informações armazenadas na Base de Conhecimento (2) servem para identificar o protótipo do suposto método.

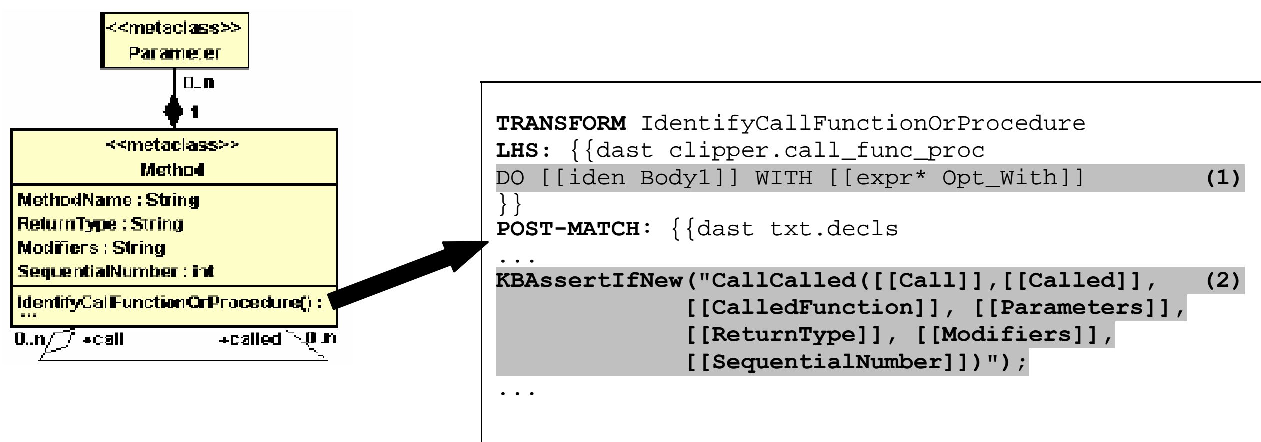


Figura 3.23 - Metaclasse Method do metamodelo Orientado a Objetos

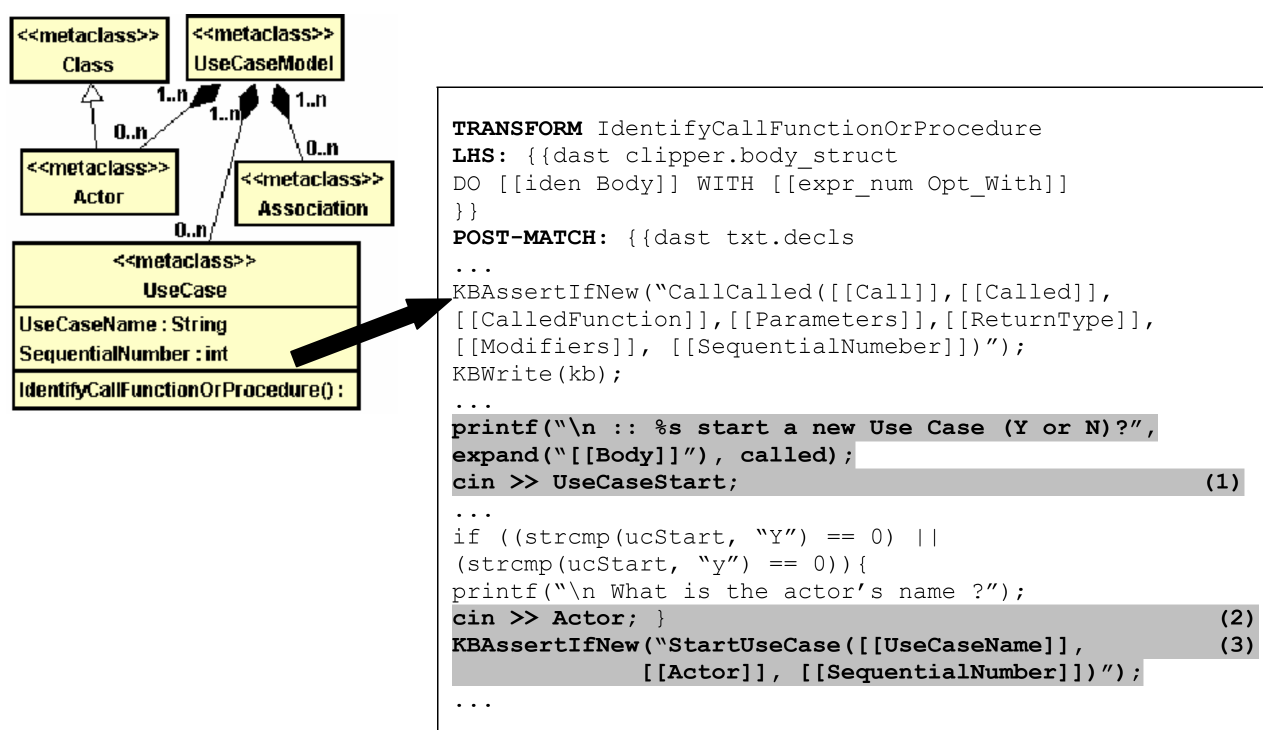
A Figura 3.24 resume as informações coletadas no fato "*CallCalled*", sobre as unidades de programa. Uma vez determinada a relação Chama/Chamado, os casos de uso, que correspondem aos cenários de utilização do sistema legado, podem ser identificados. Um caso de uso descreve um comportamento do sistema e pode ter um curso, denominado normal, correspondente ao comportamento mais comum do cenário, e um ou mais cursos alternativos, que correspondem aos seus comportamentos alternativos.

Informação	Descrição
[[Call]]	unidade chama
[[Called]]	unidade chamada
[[CalledFunction]]	suposto método da unidade chamada
[[Parameters]]	parâmetros do suposto método
[[ReturnType]]	retorno do suposto método chamado
[[Modifiers]]	modificadores do suposto método
[[SequentialNumber]]	número sequencial do fato

Figura 3.24 Informações coletadas no fato *CallCalled*

O Engenheiro de Software interage com o Sistema de Transformação Draco-PUC, durante a identificação dos casos de uso para especificar o ator e o nome de cada caso de uso. A determinação dos Casos de Uso baseia-se nos eventos que estimulam o sistema. Em sistemas baseados em menus, um caso de uso é determinado para cada opção do menu, seu comportamento é o mesmo do fluxo do código de execução que implementa esta opção. Casos de Uso sinalizados por uma condição interna do sistema são iniciados pela unidade chamada quando ocorre a condição que dispara este evento e tem o próprio sistema como Ator.

A Figura 3.25 mostra uma transformação que identifica casos de uso e interage com o Engenheiro de Software para especificar seus atores. No caso da transformação "*IdentifyCallFunctionOrProcedure*", ao identificar um evento externo que sinaliza uma chamada de uma unidade de programa, o Engenheiro de Software é consultado (1) para informar se a unidade inicia ou não um caso de uso. Caso a unidade inicie um caso de uso, é solicitado o nome do Ator, através do comando "*cin>>Actor*" (2). Estas informações são armazenadas na Base de Conhecimento, através do fato "*StartUseCase*" (3). Com base na unidade de programa que inicia um caso de uso e nas informações sobre o fluxo de execução do sistema legado (armazenadas na relação Chama/Chamado), determinam se as demais conexões de mensagens, considerando os fluxos normais e alternativos das execuções.




```
TRANSFORM IdentifyCallFunctionOrProcedure
LHS: {{dast clipper.body_struct
DO [[iden Body]] WITH [[expr_num Opt_With]]
}}
POST-MATCH: {{dast txt.decls
...
KBAssertIfNew("CallCalled([[Call]],[[Called]],
[[CalledFunction]],[[Parameters]],[[ReturnType]],
[[Modifiers]], [[SequentialNumeber]])");
KBWrite(kb);
...
printf("\n :: %s start a new Use Case (Y or N)?",
expand("[[Body]]"), called);
cin >> UseCaseStart;                                (1)
...
if ((strcmp(ucStart, "Y") == 0) ||
(strcmp(ucStart, "y") == 0)){
printf("\n What is the actor's name ?");
cin >> Actor; }                                     (2)
KBAssertIfNew("StartUseCase([[UseCaseName]],        (3)
              [[Actor]], [[SequentialNumber]])");
...
```

Figura 3.25 Identificação de um caso de uso

3.1.2.1.4. Identificar Relacionamentos

Na identificação dos supostos relacionamentos entre as supostas classes do sistema legado, as transformações limitam-se a verificação da existência dos supostos relacionamentos e as supostas classes envolvidas. A definição do tipo do suposto relacionamento (associação ou agregação por valor e referência) é feita com interação do Engenheiro de Software.

Dada a dificuldade de se determinar outros tipos de supostos relacionamentos, como generalização, dependência e realização, estes são tratados na ferramenta CASE, após a recuperação do Projeto Orientado a Objetos do sistema legado na Fase 2 do RSCT.

Os supostos relacionamentos, são identificados através de variáveis e comandos que manipulam dados de um arquivo ou de uma tabela de um Banco de Dados. É verificado se um campo de um arquivo ou tabela, identificado como chave, tem seus valores consultados e atribuídos a uma variável, e se posteriormente o valor desta variável é armazenado em campos de outro arquivo ou tabela, ou é utilizado para fazer busca em registros de outros arquivos ou tabelas. Durante a transformação, ao identificar um suposto relacionamento, o Sistema de Transformação Draco-PUC interage com o Engenheiro de Software, que informa o tipo do suposto relacionamento (associação ou agregação por valor ou referência).

Conforme mostra o metamodelo à esquerda da Figura 3.26, um suposto relacionamento tem um nome (*Name*), os nomes das supostas classes de origem (*participantOrigin*) e de destino (*participantDestination*), e seu tipo (*IsAggregate*). No caso do suposto relacionamento de agregação (*Type*) indica se o mesmo é por valor ou referência. À direita da Figura 3.26, a transformação "*IdentifySupposedRelationshipBySeek*", identifica um suposto relacionamento a partir do comando "*SEEK*" do Clipper (1). Durante a execução da transformação, o Engenheiro de Software é consultado sobre o tipo do suposto relacionamento a ser adotado através do comando "*cin >> Decision*" (2). No caso da agregação é informado em "*cin >> Type*" (3), se é por valor ou referência. Finalmente, em (4) as informações sobre o suposto relacionamento são armazenadas na Base de Conhecimento.

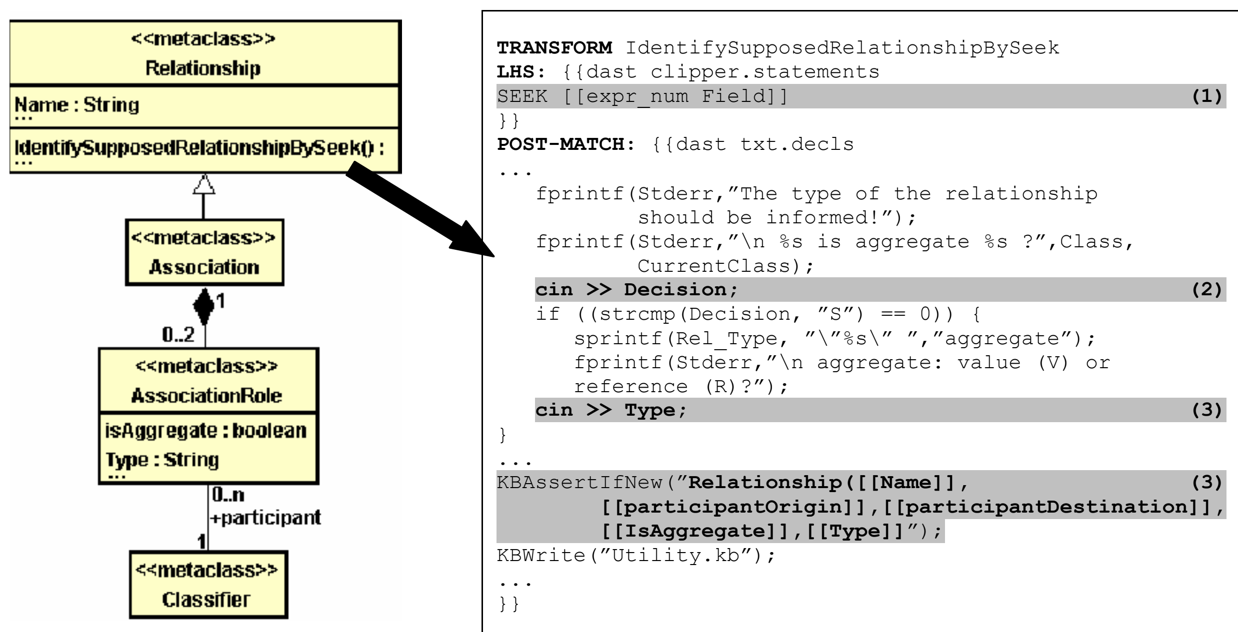


Figura 3.26 - Transformação para identificação de supostos relacionamentos

Assim, são construídas as transformações que identificam os supostos relacionamentos. A Figura 3.27 mostra outras transformações que identificam relacionamentos a partir dos comandos da linguagem Clipper.

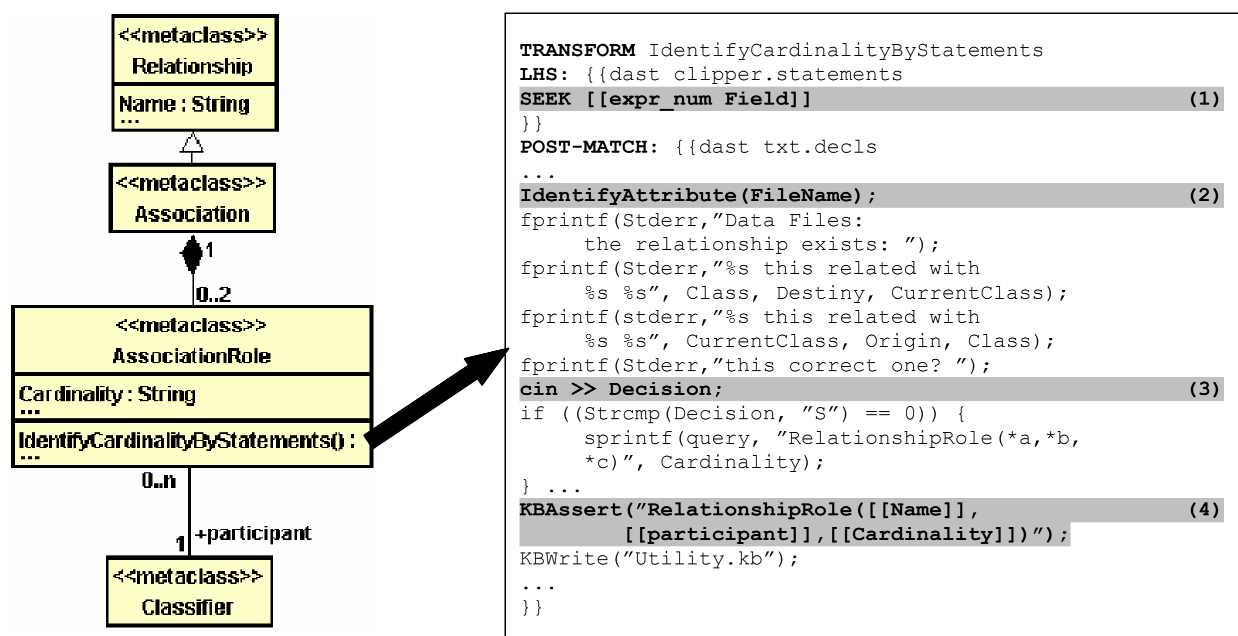
Comando Clipper	Transformação
DELETE	TRANSFORM IdentifySupposedRelationshipByDeleteStatement LHS: {{dast clipper.statements DELETE }}
identifier WITH variable	TRANSFORM IdentifySupposedRelationshipByReplaces LHS: {{dast clipper.replacement [[iden Iden1]] WITH [[expr_num Varia]] }}
file_name -> (functions)	TRANSFORM IdentifySupposedRelationshipByAliasFunctions LHS: {{dast clipper.functions [[iden File_Name]] -> ([[functions Func]]) }}
variable = expression	TRANSFORM IdentifySupposedRelationshipByAssignment LHS: {{dast clipper.body_struct [[var Var1]] = [[expr_rel Expression]] }}
GET identifier WHEN expression	TRANSFORM IdentifySupposedRelationshipByGetStatement LHS: {{dast clipper.get_stat GET [[iden Expr_Get]] WHEN [[expr Expression1]] }}
ID name	TRANSFORM IdentifySupposedRelationshipByExpressionNumber LHS: {{dast clipper.expr_num [[ID Name]] }}

Figura 3.27 - Transformações para identificação de supostos relacionamentos

3.1.2.1.5. Identificar Cardinalidades

Cardinalidade representa o número de ocorrências de um objeto em relação a outro objeto [Til93]. No caso de supostas classes obtidas a partir de arquivos ou banco de dados, consultam-se as informações dos campos destes arquivos ou tabelas para determinar as cardinalidades dos relacionamentos entre estas supostas classes. Percorrendo cada registro da tabela, pode-se identificar o número de ocorrências de um determinado campo de relacionamento. Um dado valor no campo de um arquivo ou tabela, que ocorre repetidas vezes, sugere que a cardinalidade máxima do arquivo ou tabela é "*n*". Neste caso, mais de um registro deste arquivo ou tabela está relacionado com um registro de outro arquivo ou tabela que contém o mesmo valor. No entanto, as informações obtidas dos arquivos ou tabelas podem não ser suficientes para definir as reais cardinalidades de um relacionamento, necessitando da interação do Engenheiro de Software. No método RSCT esta interação poderá ser feita durante a execução das transformações de identificação do relacionamento, ou com o apoio de uma ferramenta CASE, após a recuperação do Projeto Orientado a Objetos do sistema legado.

A Figura 3.28 mostra a transformação "*IdentifyCardinalityByStatements*" que identifica um suposto relacionamento através do comando de pesquisa "*SEEK*" (1) do Clipper, e utiliza a função "*IdentifyAttribute*" que percorre um arquivo de dados, para determinar as ocorrências de um atributo (2). Após a identificação do suposto relacionamento, há uma interação com o Engenheiro de Software, através do comando "*cin >> Decision*" (3) para confirmar o valor da cardinalidade. A cardinalidade do suposto relacionamento é armazenada no fato "*RelationshipRole*" (4). No caso, o fato guarda o nome do relacionamento "*Name*", a classe da cardinalidade "*participant*" e a cardinalidade"*Cardinality*".




```
TRANSFORM IdentifyCardinalityByStatements
LHS: {{dast clipper.statements
SEEK [[expr_num Field]]                                   (1)
}}
POST-MATCH: {{dast txt.decls
...
IdentifyAttribute(FileName);                              (2)
fprintf(Stderr,"Data Files:
     the relationship exists: ");
fprintf(Stderr,"%s this related with
     %s %s", Class, Destiny, CurrentClass);
fprintf(stderr,"%s this related with
     %s %s", CurrentClass, Origin, Class);
fprintf(Stderr,"this correct one? ");
cin >> Decision;                                          (3)
if ((Strcmp(Decision, "S") == 0)) {
     sprintf(query, "RelationshipRole(*a,*b,
     *c)", Cardinality);
} ...
KBAssert("RelationshipRole([[Name]],                      (4)
        [[participant]],[[Cardinality]])");
KBWrite("Utility.kb");
...
}}
```

Figura 3.28 - Transformação para identificação de cardinalidades

Transformadores de Organização também são utilizados durante a obtenção do projeto Orientado a Objetos do sistema legado e a edição de suas regras de transformação são realizadas conforme se segue.

3.1.2.2. Transformador de Organização

O código legado 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 [Nov02].

Esta organização pode ser automatizada, construindo-se transformadores intradomínio. Para isso, o Engenheiro de Software edita regras de transformação que utiliza nos pontos de controle de reconhecimento (*LHS*) e de substituição (*RHS*), os padrões sintáticos da mesma linguagem. 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.

Um transformador de organização baseia-se nas informações identificadas no código do sistema legado, segundo o metamodelo Orientado a Objetos, organizando-o em supostas classes, supostos atributos e supostos métodos e pode ser construído com base nos passos apresentados a seguir.

3.1.2.2.1. Organizar em Supostas Classes e Supostos Atributos

As supostas classes identificadas no passo anterior darão origem a classes. Para cada classe deve-se criar um arquivo com o mesmo nome da classe, onde serão alocados todos os seus atributos e métodos.

Os supostos atributos identificados no passo anterior originam atributos, portanto é necessária a interação do Engenheiro de Software para informar se cada um destes supostos atributos será realmente um atributo da classe.

Esta tarefa é realizada visando garantir a integridade nas informações pertinentes aos atributos, recuperadas automaticamente do sistema legado e diminuir a possibilidade de criação automática de atributos que não são necessários. Os fatos de supostos atributos normalmente são oriundos dos campos das tabelas, de arquivos de dados ou de bancos de dados do código legado. Existem recursos de implementação como, por exemplo, contadores, que podem ser descartados pelo Engenheiro de Software durante sua interação.

A Figura 3.29 mostra uma transformação que cria uma suposta classe e aloca nela seus supostos atributos. Para conter a suposta classe e seus supostos atributos no diretório"*./Output/Model/DataBase*"(1), a *template* "*T_Main_Proc*"(2) cria um arquivo de extensão "*.prg*". Seus supostos atributos são passados por parâmetros no comando *MOVE* (3).

```
TRANSFORM OrganizeSupposedClassByProgramWithTag
LHS: {{dast clipper.program
[[TAG Tag]] [[ID Proc_Princ]] [[prog_elements Elemts]]  }}
POST-MATCH: {{dast txt.decls
...
sprintf(Main_Proc, "./Output/Model/DataBase/%s.prg", Class);                (1)
CREATE_WORKSPACE(Main_Proc, "clipper");
...
sprintf(Attribute, "SupposedAttributeModelDataBase(%s,*x,*y,*z,*r,%d)",Class,Cont);
while ((KBSolve(Attribute))) {
     sprintf(Attributes[cont], "%s", Attribute); cont ++;
     sprintf(Attribute, "SupposedAttributeModelDataBase(%s,*x,*y,*z,*r,%d)",Class,Cont);
} ...
TEMPLATE("T_Main_Proc")                                                     (2)
TRANSPORT_VALUE("Class");
MOVE(Attributes, "All_Attributes");                                         (3)
...
PLACE_AT(Main_Proc);
END_TEMPLATE;
...
```

Figura 3.29 - Organização de supostas classes e supostos atributos

3.1.2.2.2. Organizar Supostos Métodos nas Supostas Classes

O objetivo deste passo é organizar nos arquivos das classes, os seus supostos métodos identificados, considerando as informações sobre os protótipos dos métodos, que estão armazenadas na Base de Conhecimento, e as especificações dos corpos dos métodos, que estão no código do sistema legado.

Para atingir este objetivo, o Engenheiro de Software utiliza o Sistema de Transformação Draco-PUC. As transformações deverão consultar a Base de Conhecimento e o código do sistema legado para obter informações sobre os supostos métodos e alocá-los no arquivo da classe a qual esse suposto método pertence. As anomalias encontradas são corrigidas da seguinte forma: Os supostos métodos são classificados em construtores **(c)**, quando alteram a estrutura de dados, e observadores **(o)**, quando somente consultam as estruturas de dados. São classificados como (**i**), os supostos métodos que não se referem a classe alguma, sendo dependentes da implementação ou relacionados à interface. Assim, quando um suposto método refere-se a mais de uma suposta classe ele deve ser alocado usando os seguintes critérios [Pen96]:

• Se construtor de uma suposta classe e observador de outra **(oc)**, será alocado na suposta classe que faz referência como construtor;

• Se construtor de uma suposta classe e também observador dessa mesma suposta classe, este deve ser classificado somente como construtor e alocado na suposta classe que faz referência como construtor;

• Se observador de uma suposta classe e construtor de várias outras classes **(oc+)**, será alocado na primeira suposta classe que faz referência como construtor;

• 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.

As anomalias não solucionadas pelas transformações são resolvidas pelo Engenheiro de Software que pode tomar decisões, interagindo com o Sistema de Transformação Draco-PUC, durante a aplicação das transformações.

A Figura 3.30 mostra, por exemplo, a transformação "*OrganizeSupposedMethodByProgBody*" que, ao reconhecer o corpo de um suposto método no código legado através de seu padrão de reconhecimento LHS (1), consulta a Base de Conhecimento para obter o correspondente protótipo deste suposto método, através do

comando "*sprintf*" (2). Em seguida a transformação chama a *template* "*T_MethodFunction*"(3) que organiza o suposto método em sua suposta classe.

```
TRANSFORM OrganizeSupposedMethodByProgBody
LHS: {{dast clipper.body_struct
     [[prog_body statement]] (1)
}}
POST-MATCH: {{dast txt.decls
...
APPLY("MakeBodyStructs","statement");
     sprintf(name, params, return, (2)
     "SupposedMethodModelDataBase(%s,*x,*y,*z,a,%d)",Class,cont);
     while !(method.equals(name)) {
     sprintf(name, params, return,
     "SupposedMethodModelDataBase(%s,*x,*y,*z,a,%d)",Class,cont);
     cont++
     } ...
TEMPLATE("T_MethodFunction") (3)
    TRANSPORT_VALUE("name");
    MOVE(params_list, "params");
    MOVE(statement, "bodys");
    TRANSPORT_VALUE("returns");
    PLACE_AT(file[cont]);
END_TEMPLATE;
...
}}
```

```
TEMPLATE T_MethodFunction
RHS: {{dast clipper.prog_extra
     FUNCTION [[ID name]] ()
     PARAMETERS [[iden* params]]
     [[body_struct* bodys]]
     [[body_struct* returns]]
     RETURN NIL
}}
```

Figura 3.30 - Organização dos supostos métodos nas supostas classes organizadas

A Figura 3.31 mostra um exemplo de transformação intradomínio usada na organização dos supostos métodos nas supostas classes. Do lado esquerdo da Figura 3.31, tem-se denotado o padrão de reconhecimento do comando "*SEEK*" (1), que procura um registro no banco de dados. Do lado direito tem-se os correspondentes padrões de substituição, as *templates* "*T_Call_Seek*" (2), que faz a chamada a nova função, e "*T_Seek*" (3), que cria a nova função que substitui o código legado organizando-o em uma função. A chamada da função é inserida no corpo do código da classe organizada mantendo assim o mesmo fluxo de execução do código do sistema organizado.

```
TRANSFORM OrganizeSupposedMethodBySeek
LHS: {{dast clipper.statements
SEEK [[expr_num field]]          (1)
}}
POST-MATCH: {{dast txt.decls
...
TEMPLATE("T_Call_Seek")       (2)
TRANSPORT_VALUE("stringclass");
TRANSPORT_VALUE("stringnum");
TRANSPORT_VALUE("field");
PLACE_AT(area);
END_TEMPLATE;
...
TEMPLATE("T_Seek")            (3)
PLACE_AT(clip);
END_TEMPLATE;
...
}}
```

```
TEMPLATE T_Call_Seek
RHS: {{dast clipper.prog_body
DO fSeek WITH [[STRI stringclass]],
[[STRI stringnum]],
[[expr_num field]], pfound
}}
```

```
TEMPLATE T_Seek
RHS: {{dast clipper.decl_func
FUNCTION fSeek ()
PARAMETERS file,index,field,pfound
SELECT &file
SET ORDER TO &index
SEEK field
IF FOUND () pfound = .T.
ELSE pfound = .F.
ENDIF
RETURN NIL }}
```

Figura 3.31 - Padrões de reconhecimento e substituição para organização do código

A organização do código legado, segundo os princípios da Orientação a Objetos, facilita a recuperação do projeto em uma linguagem de modelagem, que é o objetivo dos transformadores de recuperação.

3.1.2.3. Transformador de Recuperação

Um Transformador de Recuperação visa reescrever os trechos de código e as informações já identificadas e organizadas em na linguagem de modelagem MDL, para obter o projeto Orientado a Objetos do sistema legado.

Em MDL as mini-especificações 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 pela recuperação direta do código legado dos corpos dos métodos para a linguagem *Java*. O código *Java* gerado fica embutido no código MDL, na especificação da semântica de cada método de uma classe [Nov02].

Estes transformadores utilizam as informações da Base de Conhecimento e o código do sistema legado organizado e podem ser construídos com base nos passos apresentados a seguir:

3.1.2.3.1. Recuperar Especificações das Classes e dos Atributos

Neste passo são recuperadas as especificações, na linguagem de modelagem MDL, das classes e seus atributos. Cada unidade de programa do sistema legado, organizada como uma classe, dá origem a sua especificação na linguagem de modelagem. O mesmo acontece para cada atributo reconhecido, dando origem a sua respectiva especificação na linguagem de modelagem.

A Figura 3.32 mostra como é criada a especificação de classes e de seus atributos na linguagem de modelagem. O ponto de controle *LHS* (1) faz o reconhecimento da regra de produção "*program*", do domínio Clipper, que reconhece uma unidade de programa. No ponto de controle *POST-MATCH*, a partir do fato "*Class*" (2) armazenado na Base de Conhecimento, é criado um novo fato "*Class*" (3), que dará origem à descrição de uma classe na linguagem de modelagem pela *template "Tclass"*. Posteriormente a Base de Conhecimento é consultada para encontrar todos os fatos "*Attributes*" (4) desta classe. Em seguida, para cada atributo, é criada a sua especificação na linguagem de modelagem através da *template "Tattribute"*.

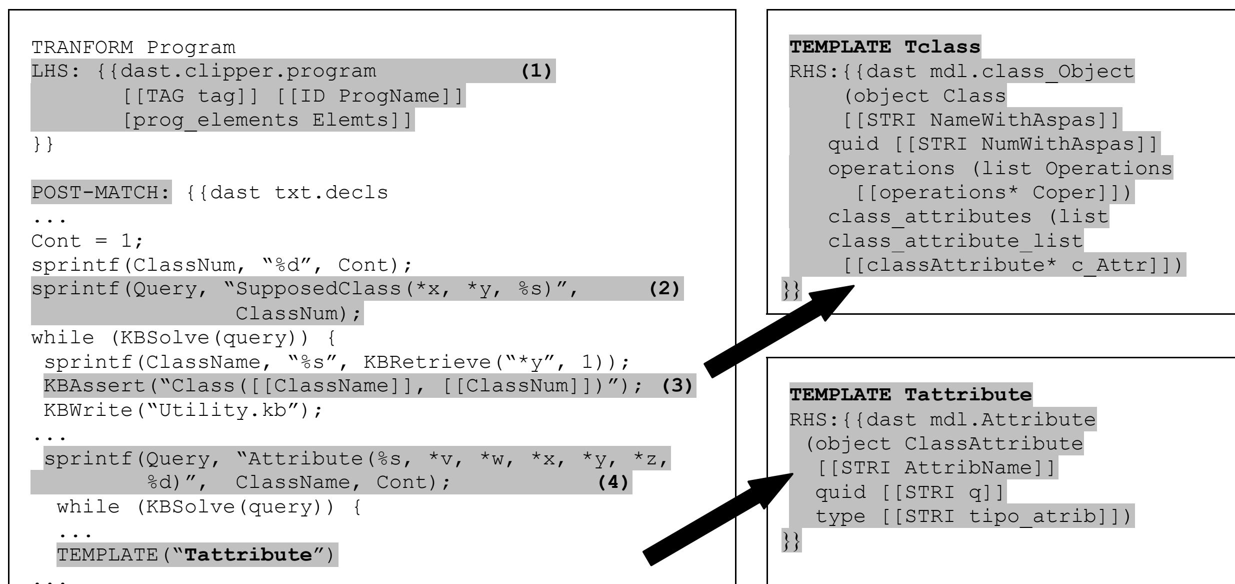


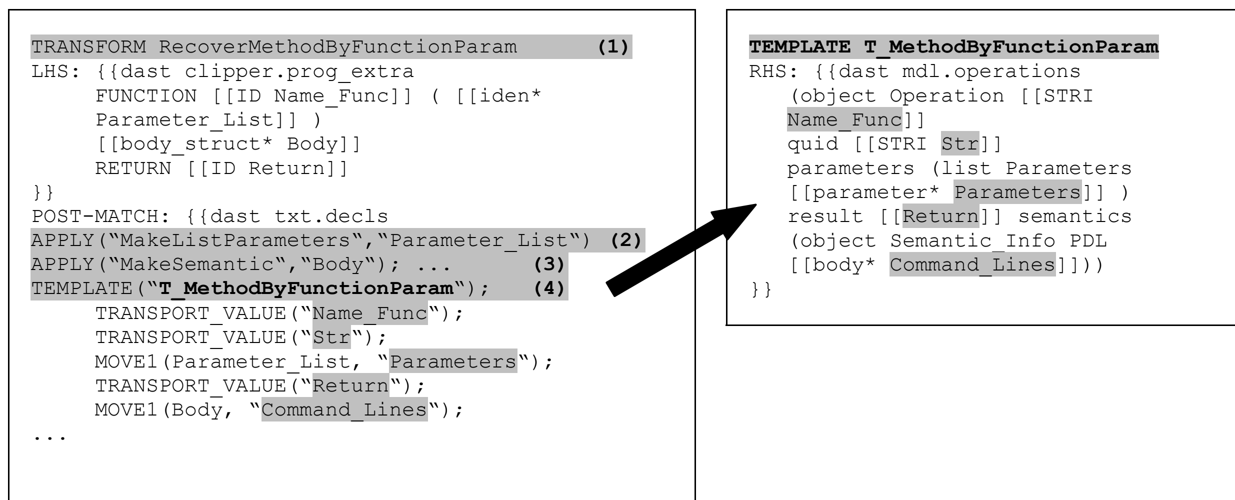
Figura 3.32 - Transformação de recuperação das especificações de supostas classes e atributos

3.1.2.3.2. Recuperar Especificações dos Métodos

Neste passo recuperam-se as especificações das assinaturas dos métodos e transformam-se os corpos dos métodos diretamente para uma linguagem de programação alvo da reimplementação.

Para cada suposto método da suposta classe, cria-se a especificação correspondente ao seu protótipo na linguagem de modelagem e depois é incluída a especificação do corpo do método, já na linguagem alvo da reimplementação, na especificação da linguagem de modelagem.

Assim como é feito para as supostas classes e seus respectivos atributos, a especificação dos métodos também é obtida com o auxílio do Sistema de Transformação Draco-PUC. A Figura 3.33 mostra, por exemplo, a criação de um método com protótipo em MDL e corpo em Java, a partir de seus comandos em Clipper. A transformação "*RecoverMethodByFunctionParam*" (1) recupera métodos a partir de funções do sistema legado organizado Clipper. Ao identificar uma função, a transformação chama outras transformações "*MakeListParameters*" (2) para tratar os parâmetros do método. A Transformação "*MakeSemantic*" (3) trata o corpo da função, representada pela variável "*Body*", para obter seu código em Java. A *template* "*T_MethodByFunctionParam*" (4) define um padrão formatado do método.



```
TRANSFORM RecoverMethodByFunctionParam          (1)
LHS: {{dast clipper.prog_extra
     FUNCTION [[ID Name_Func]] ( [[iden*
     Parameter_List]] )
     [[body_struct* Body]]
     RETURN [[ID Return]]
}}
POST-MATCH: {{dast txt.decls
APPLY("MakeListParameters","Parameter_List")  (2)
APPLY("MakeSemantic","Body"); ...          (3)
TEMPLATE("T_MethodByFunctionParam");       (4)
     TRANSPORT_VALUE("Name_Func");
     TRANSPORT_VALUE("Str");
     MOVE1(Parameter_List, "Parameters");
     TRANSPORT_VALUE("Return");
     MOVE1(Body, "Command_Lines");
...
```

```
TEMPLATE T_MethodByFunctionParam
RHS: {{dast mdl.operations
   (object Operation [[STRI
   Name_Func]]
   quid [[STRI Str]]
   parameters (list Parameters
   [[parameter* Parameters]] )
   result [[Return]] semantics
   (object Semantic_Info PDL
   [[body* Command_Lines]])))
}}
```

Figura 3.33 - Transformação de recuperação das especificações de supostos métodos

Na Tabela 3.3 são apresentadas como exemplo, algumas outras transformações usadas neste passo para recuperar o corpo dos métodos em Java, a partir do código do sistema legado Clipper.

Linguagem de Origem (Clipper)	Linguagem Alvo (Java)
TRANSFORM ConvertIfElse LHS: {{ dast clipper.prog_body IF [[expr COND]] [[body_struct* stmts]] ELSE [[body_struct* stmts1]] ENDIF }}	RHS: {{ dast java.statement if ([[expression COND]]) { [[statement* stmts]] } else{ [[statement* stmts1]] } }}
TRANSFORM ConvertWhile LHS: {{ dast clipper.prog_body DO WHILE [[expr express]] [[body_struct* corpo]] ENDDO }}	RHS: {{ dast java.statement while ([[expression express]]) { [[statement* corpo]] } }}
TRANSFORM ConvertFor LHS: {{ dast clipper.prog_body FOR [[ID ident]] = [[expr_num express]] TO [[expr_num express1]] [[body_struct* corpo]] NEXT }}	RHS: {{ dast java.statement for ([[Identifier ident]] = [[expression express]]; [[Identifier ident]] > [[expression express1]]; [[Identifier ident]]++) [[statement* corpo]] }}

Tabela 3.3 - Transformações para recuperação e conversão do corpo dos métodos

3.1.2.3.3. Recuperar Especificações dos Casos de Uso

Os cenários de Casos de Uso são compostos basicamente por atores, Use Cases e relacionamentos entre eles. Neste passo criam-se as especificações dos cenários de Casos de Uso a partir dos fatos recuperados da Base de Conhecimento. A Figura 3.34 mostra, como exemplo, a recuperação das especificações de um Use Case através da *template "T_UseCase"* (2), a partir de um fato "*StartUseCase*" da Base de Conhecimento (1).

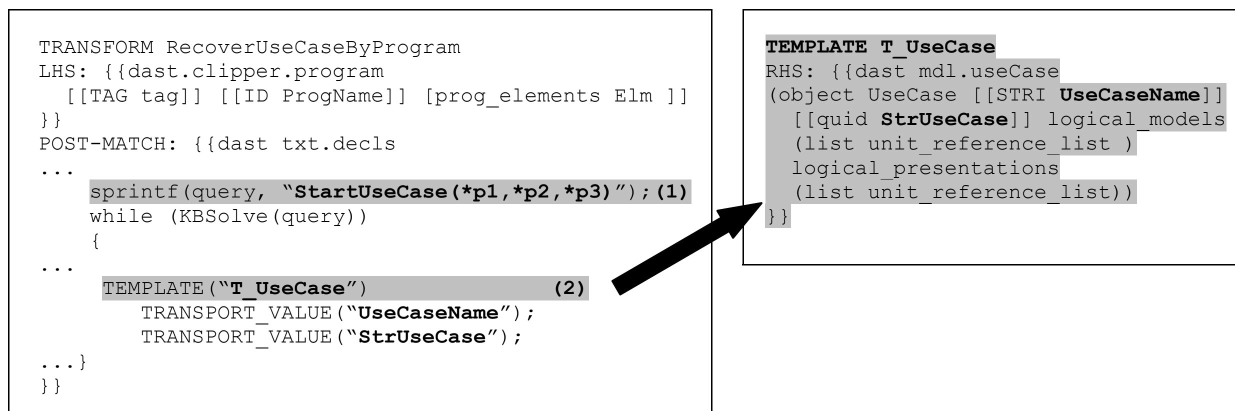


Figura 3.34 - Transformação de recuperação das especificações de supostos Casos de Uso

3.1.2.3.4. Recuperar Especificações dos Diagramas de Seqüência

Neste passo criam-se diagramas de seqüência para as seqüências de chamada de cada cenário de caso de uso. Cada Diagrama de Seqüência corresponde a um curso, normal ou alternativo, do cenário de caso de uso. O fluxo de execução do código legado organizado, definido pela seqüência de chamadas das unidades de programas, é mantido através das conexões de mensagens entre os objetos.

A Figura 3.35 mostra do lado esquerdo, as chamadas às *templates* responsáveis pela criação das especificações MDL dos modelos de seqüências. A *template* "*T_SequenceDiagram*" (1) é responsável pelos objetos que participam da execução do caso de uso atual. A *template* "*T_InteractionDiagram*" (2) é responsável pela representação gráfica desses objetos quando essas especificações são importadas em uma ferramenta CASE.

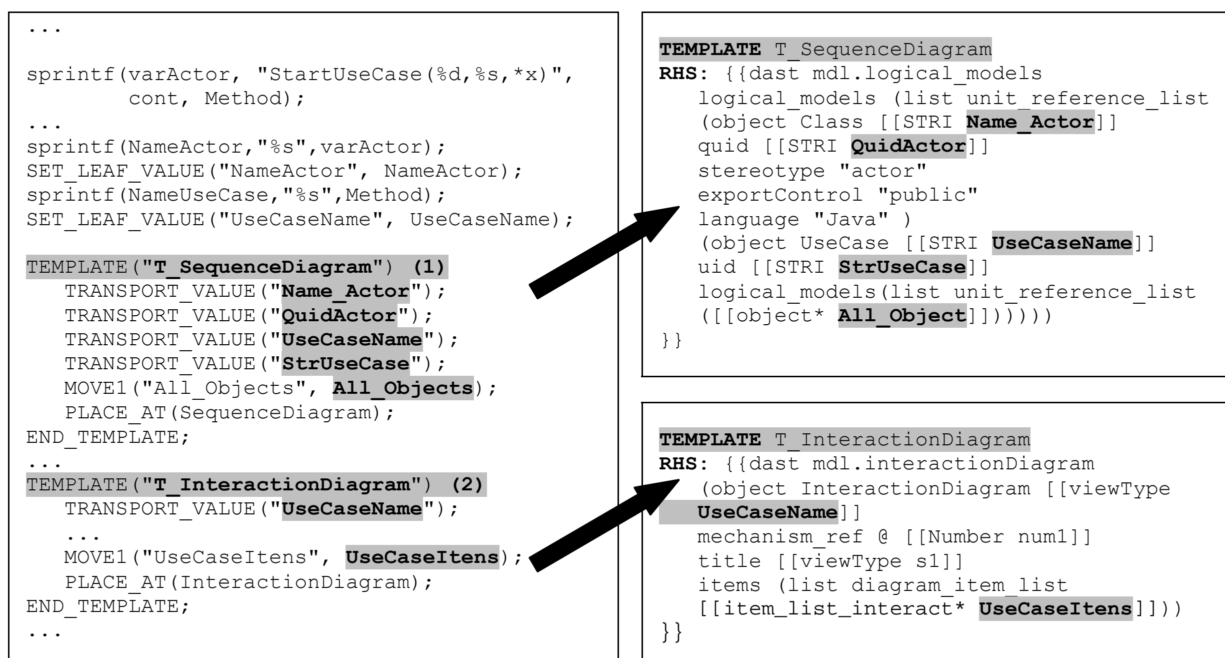


Figura 3.35 - Recuperação das especificações de um Diagrama de Seqüência em MDL

3.1.2.3.5. Recuperar Especificações dos Relacionamentos e Cardinalidades

Neste passo recuperam-se as especificações dos relacionamentos com suas respectivas cardinalidades, baseado nos fatos armazenados na Base de Conhecimento.

Para cada suposto relacionamento já identificado, deve-se gerar especificações correspondentes na linguagem de modelagem, acoplando-se suas respectivas cardinalidades também já identificadas.

A Figura 3.36 mostra a transformação "*RecoverRelationshipByProgram*" que, a partir dos fatos sobre relacionamento "*Relationship*" (1) e suas cardinalidades "*RelationshipRole*" (2), recupera a especificação de um relacionamento de associação em MDL através da *template* "*T_Association*" (3). Para cada relacionamento de associação "*NameAssociation*" (4) têm-se a classe de origem "*participant1*" e sua cardinalidade "*Cardinality1*" (5), e a classe de destino "*participant2*" e sua cardinalidade "*Cardinality2*"(6).

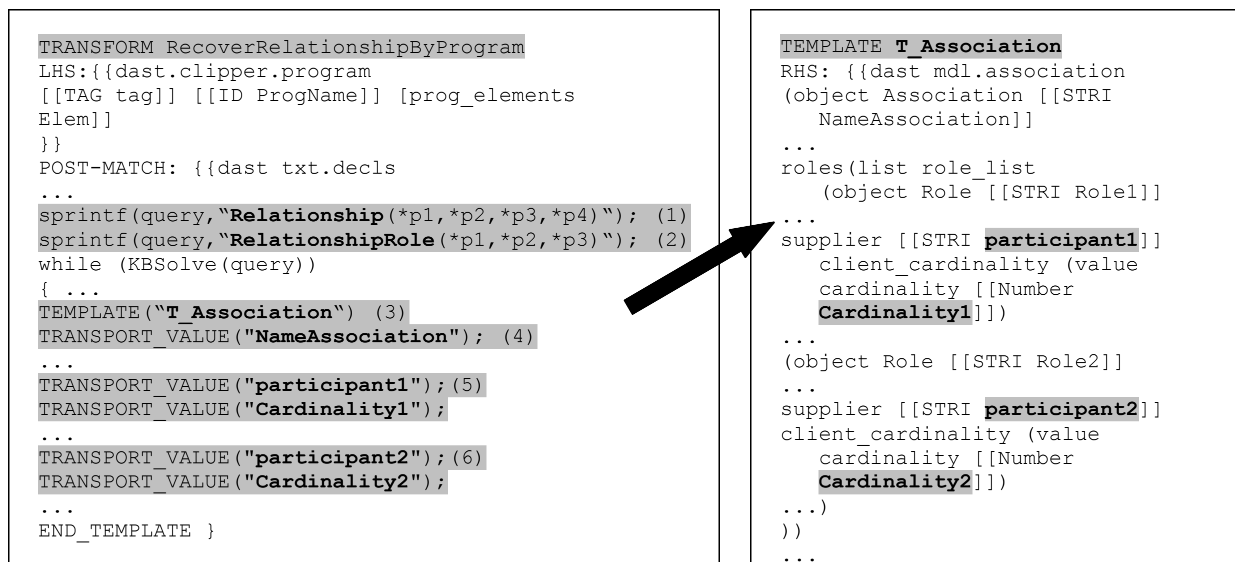


Figura 3.36 - Transformação de recuperação das especificações de um relacionamento

Assim, são criadas pra cada relacionamento, as especificações na linguagem de modelagem contendo informações da classe de origem e classe de destino, com suas cardinalidades.

Depois de editado o transformador, o Engenheiro de Software pode gerá-lo, usando o Sistema de Transformação Draco-PUC. A geração do transformador pode ser executada na própria ferramenta Draco Domain Editor ou então diretamente no Sistema de Transformação Draco-PUC. Na Draco Domain Editor, o Engenheiro de Software carrega o transformador e depois utiliza o subsistema *tfmgen* do Sistema de Transformação Draco-PUC para a geração

do transformador. Caso o Engenheiro de Software deseje gerar o transformador diretamente no Sistema de Transformação Draco-PUC, basta utilizar um script do tipo *makefile* com comandos para chamar o *tfmgen*.

Um exemplo de um script do tipo *makefile* é apresentado na Figura 3.37. Neste exemplo, o script orienta a geração do transformador ClipperKB (1).

```
                        # This makefile uses Draco makefile conventions #
include ../../mkfiles/macros.mk
include ../../mkfiles/suffix.mk

TFMER_NAME     = ClipperKB       (1)

build          : build_$(TFMER_NAME)
include ../../mkfiles/maketfmer.mk

build_$(TFMER_NAME)  : $(DOMAINSDIR)/ppd/$(DOMAINBINDIR)/ppd_pa.$(DLL) \
                       $(DOMAINSDIR)/txt/$(DOMAINBINDIR)/txt_pa.$(DLL) \ make_tfmer
clean_$(TFMER_NAME)  : $(RM) src/*.tmp src/dracotmp.dsf
```

Figura 3.37 - Script *makefile* para geração de um transformador

Após a geração do transformador, o Engenheiro de Software poderá submeter programas escritos na linguagem do domínio de origem para verificar a existência de erros. Caso sejam encontrados erros, o Engenheiro de Software retorna ao passo *Editar Regras de Transformação* e faz as correções necessárias, para gerar novamente o transformador e assim recursivamente até que se tenha uma versão correta.

A Figura 3.38 mostra um script de execução por meio do qual se submete um programa ao transformador *ClipperKB*. O comando "*load-domain draco dast*" carrega os domínios do Draco-PUC junto com seus subsistemas. O comando "*load-domain clipper prg*" carrega o domínio Clipper associado com uma extensão de arquivo a este domínio, no caso a extensão "*.prg*". O comando "*load-tfm ClipperKB*" carrega o transformador chamado *ClipperKB*. O comando "*parse %1 l*" analisa gramaticalmente o programa associado no parâmetro "*%1*". E por fim, o comando "*transform l ClipperKB*" submete o programa associado no parâmetro "*%1*" ao transformador *ClipperKB*.

```
(begin
   (load-domain draco dsf)
   (load-domain clipper prg)
   (load-tfm ClipperKB)
   (parse %1 l)
   (transform l ClipperKB)
(exit)
)
```

Figura 3.38 - Script de execução para submeter um programa ao transformador ClipperKB

Nos estudos de caso, durante a aplicação da Etapa *Construir Transformadores*, foram construídos os transformadores *ClipperCallCalled*, *ClipperKB* e *ClipperMDLJava* apresentados na Tabela 3.4.

Tipo	Transformador	Finalidade
Identificação e Organização	ClipperCallCalled	Identificar o fluxo de execução do sistema legado, obtendo sua hierarquia de chamadas e seus Casos de Uso.
	ClipperKB	Identificar supostos elementos Orientados a Objetos no código legado Clipper e prepará-lo visando facilitar a organização.
Recuperação	ClipperMDLJava	Recuperar o projeto Orientado a Objetos do sistema legado descrito em Clipper.

Tabela 3.4 - Transformadores construídos na Etapa Construir Transformadores

Depois de construídos os domínios e transformadores necessários, o Engenheiro de Software utiliza o Sistema de Transformação Draco-PUC na Engenharia Reversa dos sistemas legados, obtendo assim, os projetos Orientados a Objetos dos sistemas legados do domínio do problema.

3.2. Fase 2 – Obter Projeto Orientado a Objetos

Na segunda fase do Método RSCT, *Obter Projeto Orientado a Objetos*, obtém-se o Projeto Orientado a Objetos do sistema legado, a partir de seu código legado. O Engenheiro de Software, utilizando os domínios e transformadores no Sistema de Transformação Draco-PUC, identifica e organiza os supostos elementos Orientados a Objetos no código legado, seguindo os princípios da Orientação a Objetos. Depois de organizado, o código legado é transformado em especificações MDL, que permite recuperar seu Projeto Orientado a Objetos e refiná-lo conforme as necessidades do Engenheiro de Software.

Técnicas de Engenharia Reversa e técnicas de modelagem são empregadas pelo Engenheiro de Software para orientar a execução dos passos desta fase.

A fase *Obter Projeto Orientado a Objetos* é composta pelos seguintes passos: *Identificar Elementos*, *Organizar Código*, *Recuperar Projeto* e *Refinar Projeto Orientado a Objetos,* conforme mostra a Figura 3.39.

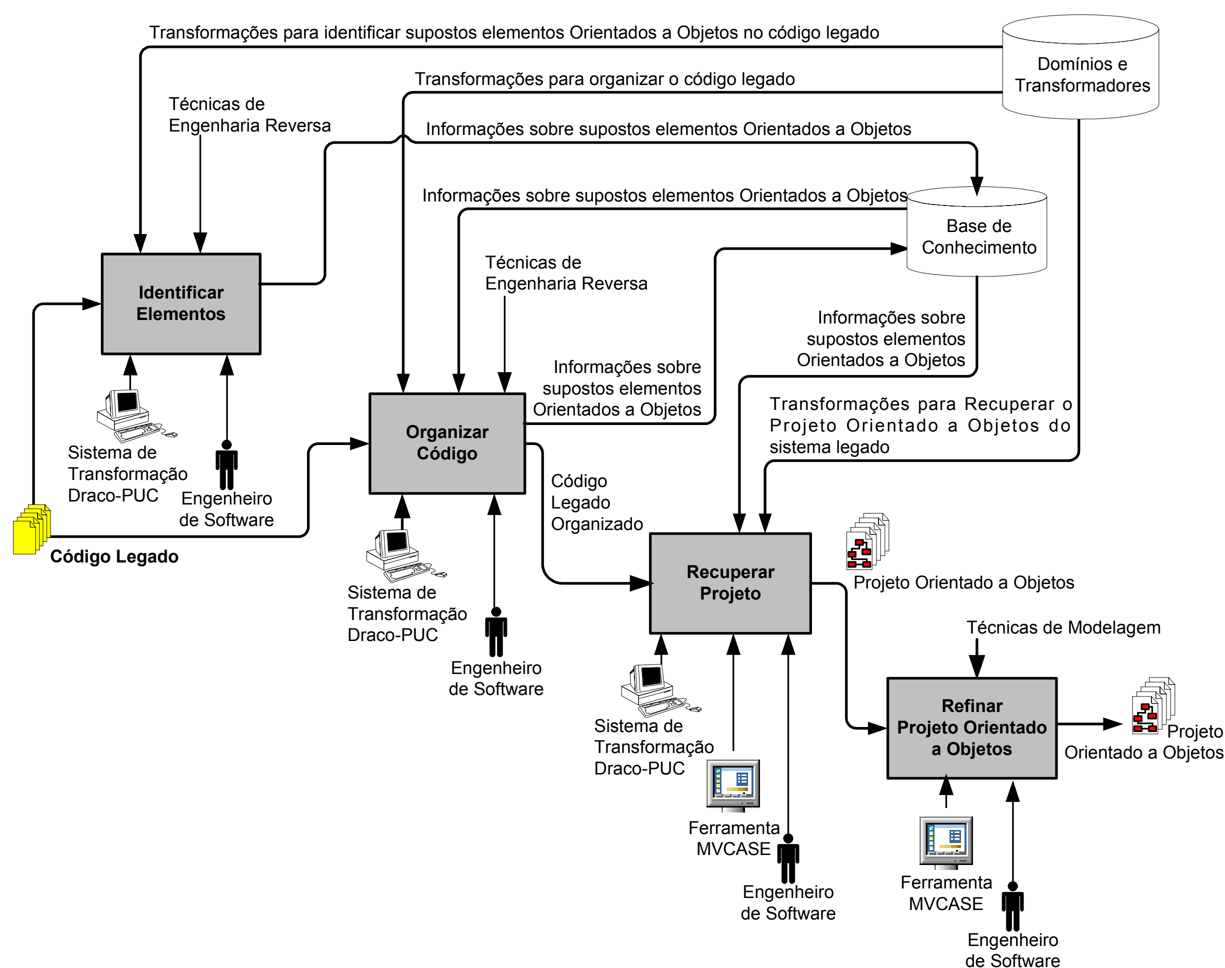


Figura 3.39 - Obter Projeto Orientado a Objetos (Fase 2)

Nesta fase, grande parte das atividades são automatizadas pelos transformadores construídos na Fase 1. Segue-se uma apresentação dos passos desta fase.

3.2.1. Identificar Elementos

O objetivo deste passo é identificar no código legado, os supostos elementos Orientados a Objetos do sistema legado. O Engenheiro de Software, no Sistema de Transformação Draco-PUC, aplica o transformador que identifica supostos elementos Orientados a Objetos no código legado.

Ao submeter todas as unidades de programa do código legado às transformações, o Engenheiro de Software obtém, além da relação Chama/Chamado, os elementos que auxiliam na definição dos Casos de Uso do sistema como supostas classes com seus supostos atributos, supostos métodos, relacionamentos e cardinalidades. Esses elementos, armazenados como fatos numa Base de Conhecimento do Draco-PUC, auxiliarão na organização do código, segundo os princípios da Orientação a Objetos.

A Figura 3.40 mostra, por exemplo, parte da Base de Conhecimento com os fatos dos supostos elementos Orientados a Objetos identificados neste passo, após a aplicação dos transformadores *ClipperCallCalled* e *ClipperKB* no sistema legado. Neste exemplo podemos identificar fatos de supostas classes (1), supostos atributos (2), supostos métodos (3), supostos relacionamentos (4) e também fatos relativos ao fluxo de execução do sistema legado (5).

```
...
SupposedClass(2,Vendas,2).                    (1)
SupposedClass(6,Clientes,3).
...
SupposedAttribute(Vendas,Codven,Numeric,5,0,v2Codven,1).              (2)
SupposedAttribute(Vendas,Dataentra,Date,8,0,v2Dataentra,2).
SupposedAttribute(Vendas,Datavenda,Date,8,0,v2Datavenda,3).
SupposedAttribute(Clientes,Codcliente,Numeric,5,0,v3Codcliente,1).
SupposedAttribute(Clientes,Nome,Character,30,0,v3Nome,2).
...
SupposedMethod(Sistema,Inicial,1).                      (3)
SupposedMethod(Clientes,Limpa,2).
SupposedMethod(Clientes,Insere,3).
SupposedMethod(Vendas,CalculaTotal,4).
...
RelationshipFact(3,seek,codcli,Codcliente,Clientes).            (4)
RelationshipFact(4,atribuicao,codcli,Codcliente, Clientes).
Relationship(Clientes,Codcliente,Cliveic,Codcliente,"0..*","0..1","esta associado a",1).
Relationship(Cliped,Codpedido,Tipoproduto,Codproduto,"0..1","0..*","esta associado a",2).
...
CallCalled(Sistema,Clientes,Inicial).                   (5)
CallCalled(Sistema,Clientes,Logotipo).
CallCalled(Sistema,Vendas,Vend1001).
CallCalled(Sistema,Vendas,Vend2001).
CallCalled(Sistema,Vendas,Vend3000).
CallCalled(Sistema,Vendas,Modos2).
CallCalled(Sistema,Vendas,Elimios).
CallCalled(Sistema,Vendas,Indexa).
...
```

Figura 3.40 - Exemplo de fatos da Base de Conhecimento

Depois de identificados e armazenados na Base de Conhecimento, os supostos elementos Orientados a Objetos do sistema legado, o Engenheiro de Software já possui condições para organizar o código legado, visando facilitar a recuperação do projeto do sistema.

3.2.2. Organizar Código

O objetivo deste passo é organizar o código legado, segundo os princípios da Orientação a Objetos.

Com base nas informações identificadas no passo *Identificar Elementos* e armazenadas na Base de Conhecimento, o Engenheiro de Software submete o código legado ao Sistema de Transformação Draco-PUC, reutilizando os transformadores de organização construídos na Fase 1, para segmentar o código legado, organizando-o em supostas classes, com seus supostos atributos e métodos, produzindo um código intermediário, organizado com característica da Orientação a Objetos, porém ainda na mesma linguagem do código legado.

Para exemplificar a organização de supostas classes com seus supostos atributos e métodos, a Figura 3.41 mostra um fato "*SupposedClass*" que deu origem ao arquivo

Clientes.prg. O fato "*ClassConstructor*" gerou um construtor chamado "*ClientesAbre()*" para a classe *Clientes* através da *template* "*T_Constructor*" (2.1), e o fato "*SupposedMethod*" gerou o método "*ClientesInicializa()*", através da *template* "*T_Initialize*" (2.2), que é o inicializador da classe. Os fatos "*SupposedAttribute*" identificados anteriormente, são localizados na Base de Conhecimento (1) e o Engenheiro de Software, através da interação com o Sistema de Transformação Draco-PUC, informa se esses serão realmente atributos das classes, gerando assim um fato "*Attribute*" que será armazenado na Base de Conhecimento. Em (2) tem-se as *templates* que possuem os padrões de substituição (RHS) para o método construtor e inicializador, utilizados para escrever os métodos "*ClientesAbre()*" e "*ClientesInicializa()*" alocados na classe Clientes(3).

```
        Base de Conhecimento (KB)       (1)

SupposedClass(6,Clientes,3).
...
ClassConstructor(Clientes,ClientesAbre,Indexa,0).
PublicVariable(v3Codcliente,ClientesInicializa,
   Numeric,0).
PublicVariable(v3Nome,ClientesInicializa,
   Character,0).
PublicVariable(v3Endereco,ClientesInicializa,
   Character,0).
PublicVariable(v3Bairro,ClientesInicializa,
   Character,0).
PublicVariable(v3Cidade,ClientesInicializa,
   Character,0).
...
SupposedMethod(Clientes,ClientesInicializa,2).
SupposedAttribute(Clientes,Codcliente,Numeric,5,
   0,v3Codcliente,1).
SupposedAttribute(Clientes,Nome,Character,30,
   0,v3Nome,2).
SupposedAttribute(Clientes,Endereco,Character,30,
   0,v3Endereco,3).
SupposedAttribute(Clientes,Bairro,Character,15,
   0,v3Bairro,4).
...
Attribute(Clientes,Nome,Character,30,0,
   v3Nome,2).
Attribute(Clientes,Endereco,Character,30,0,
   v3Endereco,3).
Attribute(Clientes,Bairro,Character,15,0,
   v3Bairro,4).
Attribute(Clientes,Cidade,Character,15,0,
   v3Cidade,5).
...
```

```
    *$ Nome do programa: CLIENTES.prg

FUNCTION ClientesAbre() (3)
SELECT 10
USE Clientes
GOTO BOTTOM
SKIP
RETURN NIL
FUNCTION ClientesInicializa()
PUBLIC v3Codcliente,v3Nome,
   v3Endereco, v3Bairro,v3Cidade,
   v3Estado, v3Cep,v3Telefone,
   v3Cgccic,v3Rginscest,v3Datanasc,
   v3Tipopes, v3Datamov,v3Nomepai,
   v3Nomemae, v3Endcor,v3Obs,
   v3Entidade, v3Categoria,v3Conspc
STORE 0 TO v3Codcliente
STORE SPACE(30) TO v3Nome
STORE SPACE(30) TO v3Endereco
STORE SPACE(15) TO v3Bairro
STORE SPACE(15) TO v3Cidade
STORE SPACE(2) TO v3Estado
STORE SPACE(9) TO v3Cep
STORE SPACE(8) TO v3Telefone
STORE SPACE(18) TO v3Cgccic
STORE SPACE(16) TO v3Rginscest
STORE CTOD(" / / ") TO v3Datanasc
STORE SPACE(1) TO v3Tipopes
STORE CTOD(" / / ") TO v3Datamov
STORE SPACE(30) TO v3Endcor
STORE SPACE(40) TO v3Obs
STORE SPACE(20) TO v3Entidade
STORE SPACE(1) TO v3Categoria
STORE SPACE(1) TO v3Conspc
RETURN NIL
```

```
TEMPLATE T_Constructor                   (2.1)
RHS: {{dast clipper.decl_func
FUNCTION [[ID CurrentClassAbre]] ()
SELECT [[NUM classnum]]
USE [[ID currentclass]]
GOTO BOTTOM
SKIP
RETURN NIL }}
```

```
TEMPLATE T_Initialize                (2.2)
RHS: {{dast clipper.decl_func
FUNCTION [[ID CurrentClassInicializa]]()
PUBLIC [[decl_atrib* list_var]]
[[body_struct* list_store]]
RETURN NIL
}}
```

Figura 3.41 - Organização de uma classe com seus atributos e métodos

A organização do código legado, segundo os princípios da Orientação a Objetos, facilita a recuperação do projeto Orientado a Objetos do sistema legado, que é apresentada na próxima seção.

3.2.3. Recuperar Projeto

Neste passo, o Engenheiro de Software submete o código legado organizado ao Sistema de Transformação Draco-PUC, reutilizando os transformadores de recuperação construídos na Fase 1, para recuperar as especificações MDL do projeto do sistema legado.

A Figura 3.42 mostra, por exemplo, a especificação MDL recuperada da classe *Clientes* do sistema Caiçara, definida pela declaração "*object Class*", e de seus atributos através da declaração "*object ClassAttribute*".

```
(object Class "Clientes"
   quid "00000000000B"
   exportControl "public"
   operations (list Operations
   (object Operation "Clientes"
      quid "00000000013F"
      result ""
      semantics (object Semantic_Info PDL
      ...// Reservado para especificação dos métodos
      class_attributes (list class_attribute_list
      (object ClassAttribute "CODCLIENTE"
         quid "00000000004B"
         type "int"
         exportControl "private"
      )
      (object ClassAttribute "NOME"
         quid "00000000004C"
         type "String"
         exportControl "private"
      )
      (object ClassAttribute "ENDERECO"
         quid "00000000004D"
         type "String"
         exportControl "private"
      )
      (object ClassAttribute "BAIRRO"
         quid "00000000004E"
         type "String"
         exportControl "private"
      )
      (object ClassAttribute "CIDADE"
         quid "00000000004F"
         type "String"
         exportControl "private"
      )
      (object ClassAttribute "ESTADO"
         quid "000000000050"
         type "String"
         exportControl "private"
      )
...
```

Figura 3.42 - Especificação MDL de uma classe com seus atributos

Na Figura 3.43 tem-se a especificação das assinaturas dos métodos "*ClientesAbre()*" e "*Proc3()*" e também suas especificações implementadas na linguagem Java. A especificação dos métodos é definida pela declaração "*object Operation*" e as especificações implementadas na linguagem Java são armazenadas na parte semântica, definida pela declaração "*semantics*" de cada método.

```
(object Class "Clientes"
   quid "00000000000B"
   exportControl "public"
   operations (list Operations
   (object Operation "ClientesAbre"
      quid "00000000013F"
      result ""
      semantics (object Semantic_Info PDL
         |openDBF("CLIENTES", "NEW");
         |String indexes[] = {"CODCLI","NOMECLI"};
         |indexDBF(indexes, 2);
      )
      opExportControl "public"
      quid 0
   )
   (object Operation "Proc3"
   quid "000000000152"
   parameters (list Parameters
      (object Parameter "codcli"
         quid "000000000153"
         type "int[]"
      )
      (object Parameter "encontrou"
         quid "000000000154"
         type "boolean[]"
      )
   )
   result "void"
   semantics (object Semantic_Info PDL
      |setOrder(1);
      | seek(String.valueOf(codcli[0]));
      | if (isfound()) {
      |encontrou[0] = true;
      | }
      | else{
      | encontrou[0] = false;
      | }
   )
   opExportControl "public"
   quid 0
   )
   class_attributes (list class_attribute_list
      (object ClassAttribute "CODCLIENTE"
         quid "00000000004B"
         type "int"
         exportControl "private"
      )
   (object ClassAttribute "NOME"
...
```

Figura 3.43 - Especificação MDL de métodos com seus respectivos corpos escritos em Java

A Figura 3.44 mostra a especificação criada para um ator "*Cliente*" identificado, através da declaração *stereotype* "*actor*" na declaração "*object Class*" da classe, e para os Casos de Uso "*AtenderPedido*", "*CadastrarCliente*", "*CadastrarProduto*" e "*VerEstoque*", através da declaração "*object UseCase*".

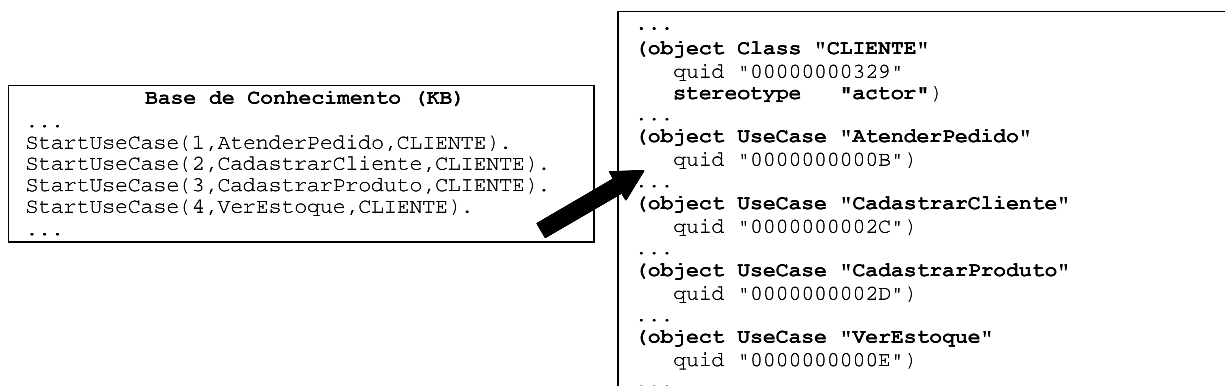


Figura 3.44 - Especificação MDL de um Caso de Uso

Depois de geradas as especificações MDL, estas podem ser importadas pelo Engenheiro de Software na ferramenta MVCASE, obtendo o Projeto Orientado a Objetos recuperado do sistema legado. Este projeto pode conter algumas incorreções que são sanadas pelo Engenheiro de Software durante o refinamento do projeto.

A Figura 3.45 mostra a especificação criada para o Diagrama de Seqüência “Vendas”, onde foi identificado o ator e seus respectivos cenários de relacionamento.

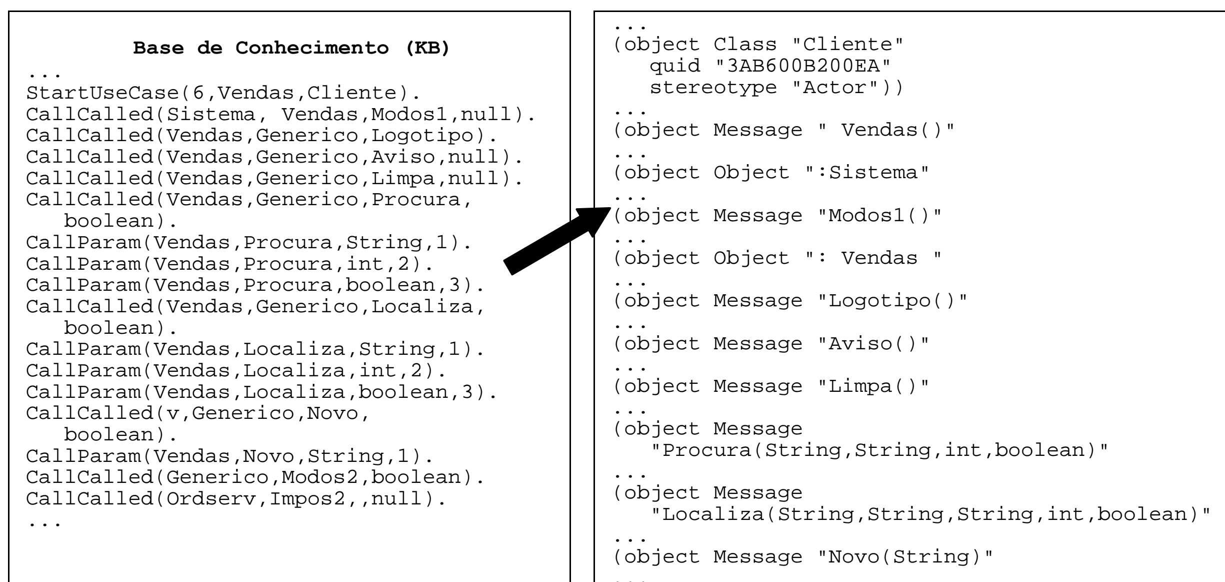


Figura 3.45 - Especificação MDL de um Diagrama de Seqüência

Um refinamento deve ser realizado após a recuperação do projeto Orientado a Objetos do sistema legado, visando aumentar a sua consistência. Este refinamento é feito conforme a próxima seção.

3.2.4. Refinar Projeto Orientado a Objetos

Neste passo, o Engenheiro de Software importa na ferramenta MVCASE o Projeto Orientado a Objetos Recuperado do Sistema Legado e refina os modelos do projeto. Visando aumentar a consistência do Projeto Orientado a Objetos recuperado do sistema legado, o Engenheiro de Software pode refinar os Modelos de Classes, Casos de Uso e de Seqüência

obtidos no passo anterior, alterando os nomes dos Casos de Uso, das classes, atributos e métodos e relacionamentos para nomes mais significativos. Pode-se ainda redefinir os tipos de relacionamentos que não foram tratados pelas transformações, usando generalização, realização e dependência, por exemplo.

A Figura 3.46 mostra um dos refinamentos do projeto Orientado a Objetos do sistema legado Informatiza. O Modelo de Classes recuperado foi importado na MVCASE e foram tratados os nomes das classes modificando-os para nomes mais significativos. O nome da classe "*CadCli*" foi alterado para "*Cliente*" (1). A classe "*ItemPed*" teve seu nome alterado para "*ItemPedido*" (9). A classe "*Funciona*" também teve seu nome alterado para "*Funcionario*" (6). Os nomes dos métodos das classes "*Cliente*" (2), "*Pedido*" (4), "*Funcionário*" (7), "*ItemPedido*" (10) e "*Produto*" (12) também sofreram modificações visando facilitar o entendimento das funcionalidades a que eles se propõe. Nos números (3), (5), (8) e (11) observa-se que os relacionamentos receberam nomes e as cardinalidades dos relacionamentos de Associação (5) e Agregação por valor (8) também foram alteradas pelo Engenheiro de Software.

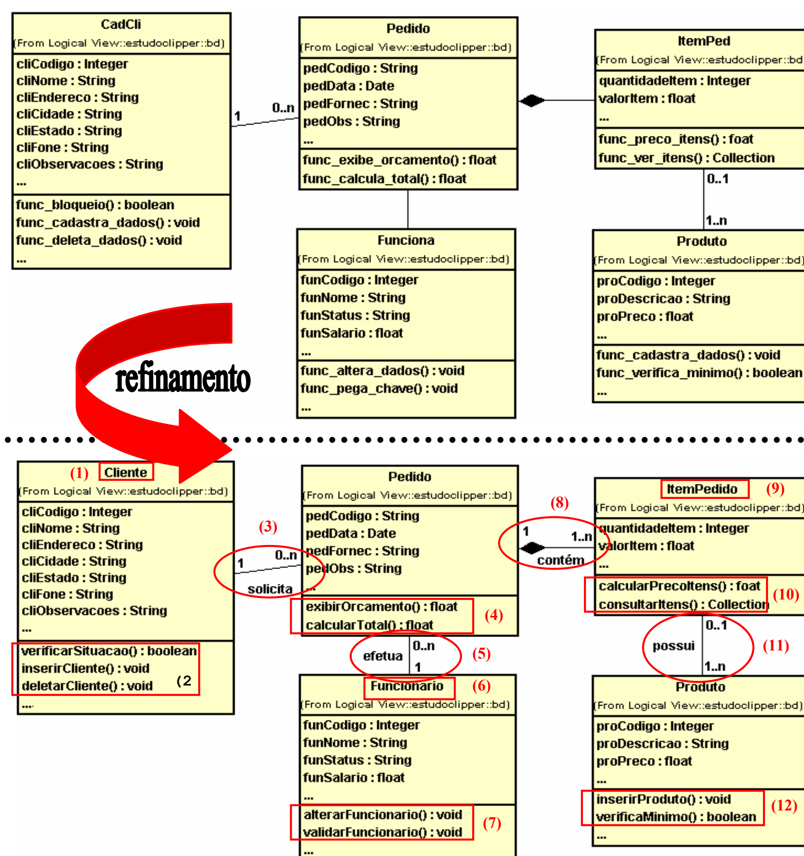


Figura 3.46 - Refinamento de um Modelo de Classes

A Figura 3.47 mostra o refinamento de um Modelo de Caso de Uso do sistema legado Informatiza, no qual foi alterado o nome do caso de uso "*CadCli*" para "*CadastrarCliente*". Também especificados todos relacionamentos deste modelo, facilitando o entendimento do sistema.

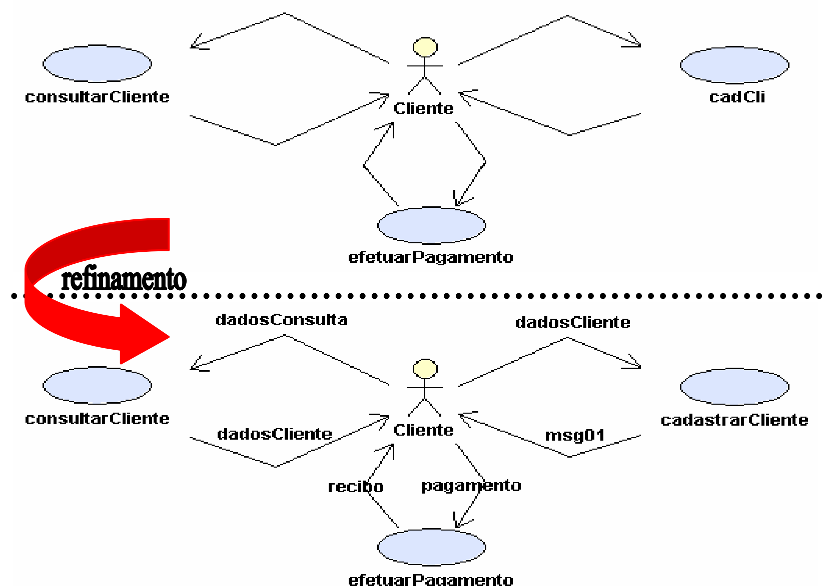


Figura 3.47 - Refinamento de um Modelo de Caso de Uso

Depois de realizados os refinamentos necessários, tem-se os projetos Orientados a Objetos dos sistemas legados, que são o resultado da Fase 2 do RSCT.

3.2.5. Considerações Finais

Uma vez concluída a Fase 2 têm-se os Projetos Orientados a Objetos de diferentes sistemas alvos da reengenharia, dentro de um determinado domínio de problemas. Com a documentação destes projetos recuperou-se o conhecimento sobre o domínio que é importante para construir os componentes que devem atender diferentes aplicações deste domínio.

Grande parte dos sistemas comerciais alvos da reengenharia, normalmente utilizam banco de dados e têm características comuns, que orientaram a definição das próximas fases do RSCT, que visam obter componentes que possam ser utilizados tanto na reconstrução de sistemas legados como no desenvolvimento de novas aplicações de um domínio. Assim, visando diminuir o tempo e o custo de desenvolvimento e facilitar manutenção de sistemas através do reuso nos diferentes níveis de abstração, estendeu-se o Método RST com mais duas fases: Fase 3, denominada *Construir Componentes*, e Fase 4, denominada *Reconstruir Sistema*.

Segue-se a apresentação da próxima fase do RSCT (Fase 3), que visa obter os componentes de um domínio cujos sistemas tiveram seus Projetos Orientados a Objetos obtidos através da Reengenharia, ou já existiam, como no caso do sistema Gnome, escrito em Delphi.

3.3. Fase 3 – Construir Componentes

Nesta Fase, o Engenheiro de Software utiliza a ferramenta MVCASE como principal mecanismo de execução. Técnicas de Desenvolvimento Baseado em Componentes e Padrões de Projeto são utilizados pelo Engenheiro de Software durante o refinamento e implementação dos componentes.

Esta Fase é dividida nos seguintes passos: *Refinar Projeto Orientado a Componentes* e *Implementar Componentes*, conforme mostra a Figura 3.48.

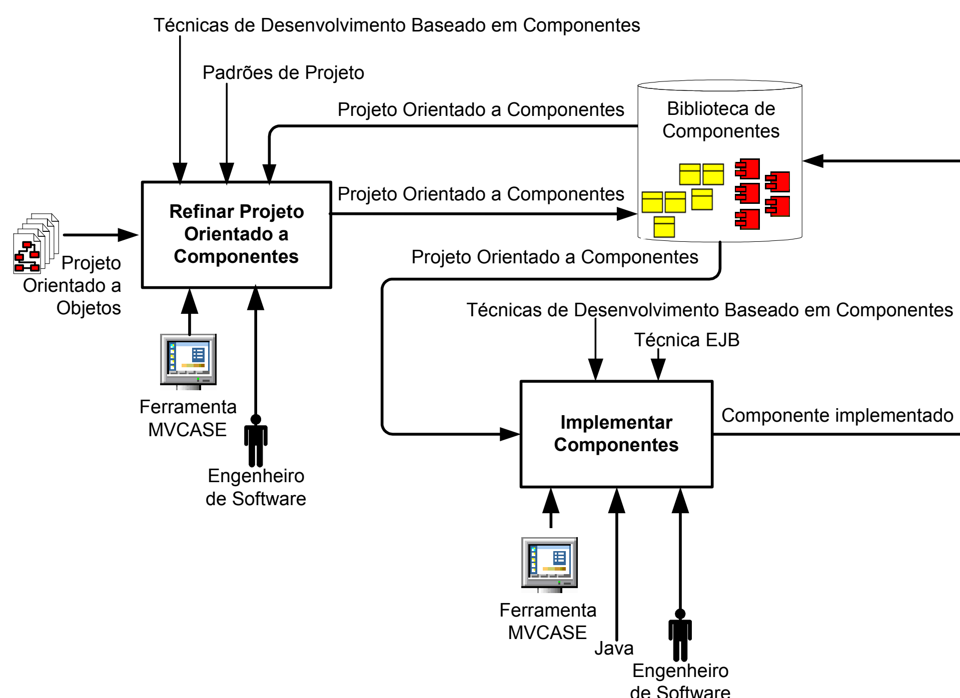


Figura 3.48 - Construir Componentes (Fase 3)

Os componentes obtidos com o método RSCT são organizados segundo o padrão MVC [Mvc03] que possui 03 (três) camadas: *Model, View e Controller*. Este padrão vem sendo comumente utilizado em aplicações comerciais e tem sido apoiado por *frameworks* como o *Structs* [Str04] para orientar o processo de desenvolvimento baseado em componentes.

Na camada *Model* têm-se os componentes persistentes, responsáveis pelo armazenamento e recuperação de dados no banco de dados, e os componentes transientes, responsáveis pela lógica da aplicação. A camada *View* visa à apresentação das informações

pertencentes ao modelo e a captura dos eventos de usuário. Na camada *Controller* têm-se os padrões de controle da aplicação que evitam dependências diretas entre as camadas *View* e *Model*, que possuem tecnologias específicas. Assim, são ocultados da aplicação cliente, todos os detalhes específicos da tecnologia utilizada na construção dos componentes. Caso seja necessária a substituição da tecnologia utilizada para implementar os componentes, as aplicações da camada *View* não são alteradas.

No caso do RSCT, a construção dos componentes é baseada nos padrões *Business Delegate* [Sun04d], *Fast Lane Reader* [Sun04d], *Service Locator* [Sun04d] e *Value Object* [Sun04d]. Estes padrões estão disponíveis na camada *Controller* da arquitetura MVC, conforme mostra a Figura 3.49.

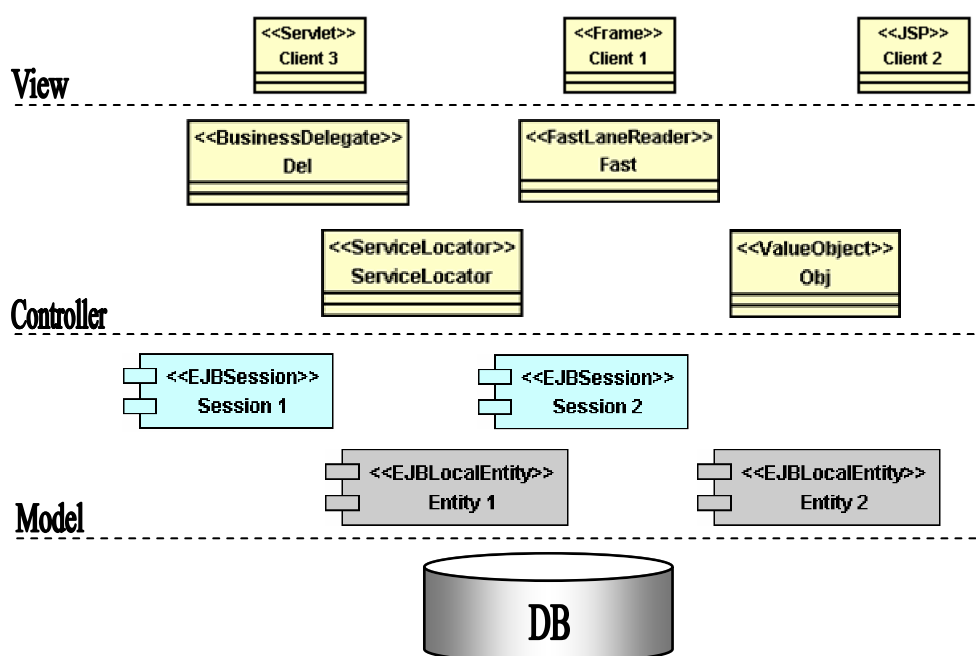


Figura 3.49 - Arquitetura adotada no RSCT

Segue-se uma apresentação de cada um dos passos que compõem a fase *Construir Componentes* do RSCT.

3.3.1. Refinar Projeto Orientado a Componentes

Neste passo, obtêm-se os modelos dos componentes e definem-se seus comportamentos e responsabilidades. Estes modelos constituem o Projeto Orientado a Componentes (POC). O Engenheiro de software, com auxilio da MVCASE, analisa o Projeto Orientado a Objetos (POO) para refiná-lo, reespecificando-o e re-estruturando-o Orientado a Componentes. O processo de refinamento baseia-se em técnicas de Desenvolvimento Baseado em Componentes (DBC) e inicia-se com um primeiro Projeto Orientado a Objetos, que resulta

numa primeira versão do Projeto Orientado a Componentes. Novas versões do POC são obtidas repetindo-se o processo para outros POOs. Técnicas de padrões de projeto são usadas para refinar ainda mais o POC, até obter uma versão com maior generalização que atenda as diferentes aplicações do domínio. Embora esta tarefa de análise, reespecificação e re-estruturação do POO requeira do Engenheiro de Software um bom conhecimento do domínio do problema, a disponibilidade do POO facilita a sua execução.

Dentre as técnicas de modelagem usando a UML [Omg03], no refinamento do POO, foram utilizados: os modelos de casos de Uso e de Interações que ajudam a definir os comportamentos externos dos componentes, e os modelos de classes baseados nos quais se implementam os componentes.

Procurando tirar proveito das características dos sistemas comerciais baseados em banco de dados relacionais e considerando o padrão MVC, as classes dos componentes foram organizadas em Persistentes e Transientes. As persistentes são definidas com base nas classes que se relacionam diretamente com o banco de dados. Estas classes são identificadas pelo *stereotype Entity*. As transientes são obtidas com base nas classes que manipulam os objetos persistentes, porém, não estão relacionadas diretamente com o banco de dados relacional. Estas classes são identificadas pelo *stereotype Session.*

Este passo do RSCT compreende uma seqüência de atividades que serão apresentadas com mais detalhes a seguir.

3.3.1.1. Identificar classes como *Entity* e *Session*.

Inicialmente, cada classe do modelo de classes do POO é analisada para defini-la como *Entity* ou *Session*. Esta classificação baseia-se principalmente na análise dos atributos da classe e nos seus relacionamentos. No caso do POO obtido no processo de Engenharia Reversa, já se tem uma classificação inicial que pode ou não ser confirmada pelo Engenheiro de Software. As classes do tipo *Entity* têm a responsabilidade de guardar os estados dos objetos em um meio persistente, enquanto que as classes do tipo *Session* relacionam-se com o ciclo de vida do componente e com as regras de negócio que o mesmo deve atender. A Figura 3.50 mostra um modelo de classes identificadas como *Entity*. Seus atributos são associados aos campos das tabelas do banco de dados. Os métodos *atualizarCliente(), inserirCliente()* e *deletarCliente()* da classe Cliente, por exemplo, são responsáveis por atualizar, criar e remover os objetos do banco de dados.

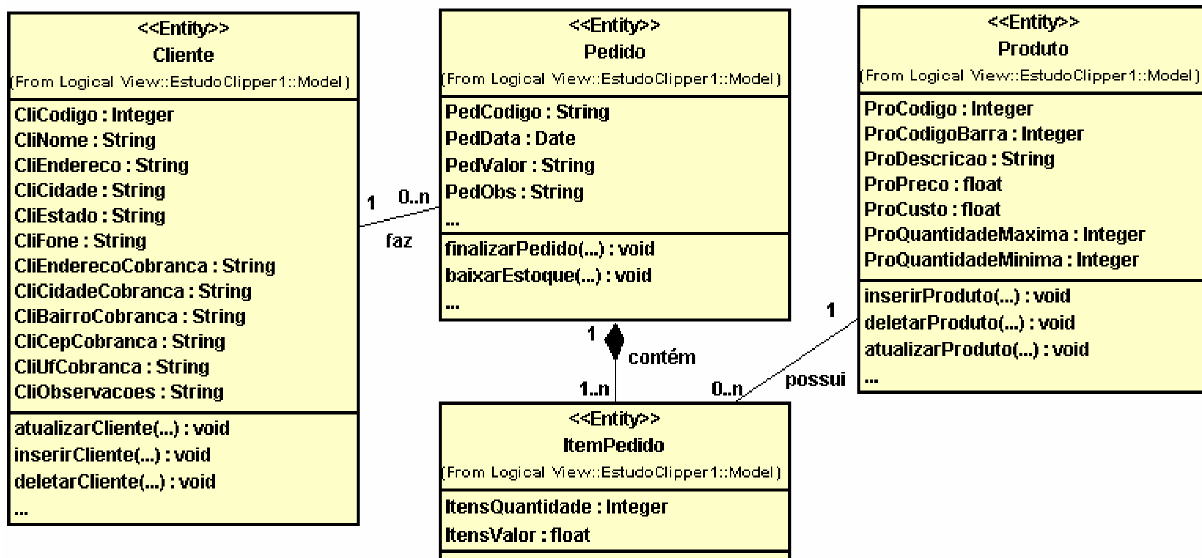


Figura 3.50 - Classes *Entity* do Modelo de Classes do POO

A Figura 3.51 mostra um modelo com as classes *Session CadastrarCliente e VenderProduto* e *CadastrarProduto*. Seus métodos são responsáveis pelas regras de negócio e a persistência dos dados é realizada através das classes *Entity Cliente*, *Pedido*, *ItemPedido* e *Produto*.

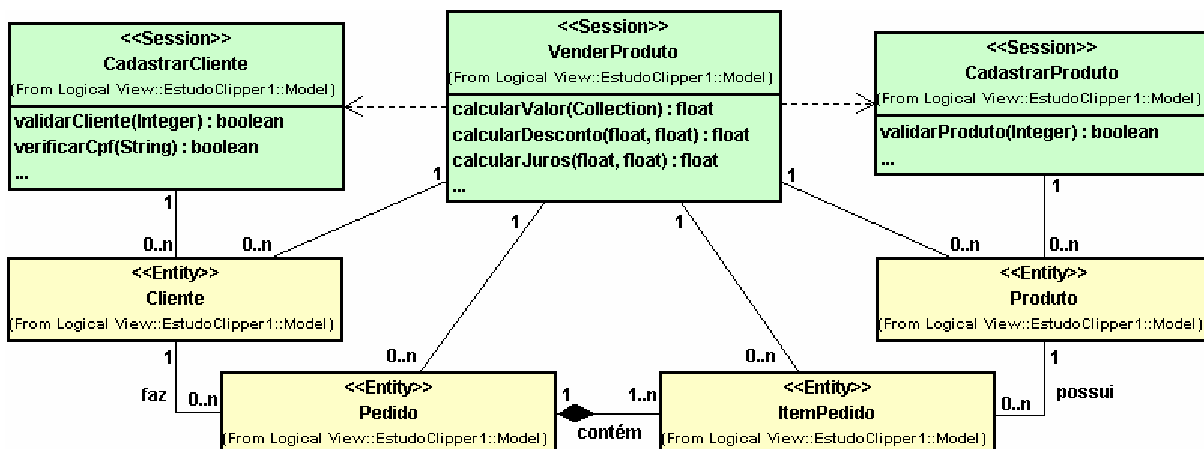


Figura 3.51 - Classes *Session* do Modelo de Classes do POO

Embora não seja uma regra geral, a maioria destas classes difere das anteriores, *Entity*, porque não estão relacionadas diretamente com o banco de dados e utilizam os objetos persistentes para atender os Casos de Uso do sistema. Portanto, uma heurística que orienta a definição destas classes baseia-se nos modelos de Casos de Uso que documentam o comportamento externo do sistema. Pode-se modificar o nome da classe original, agrupar uma ou mais classes, ou separar uma classe em duas, para que no final, cada classe *Session* seja correspondente a um caso de uso. Dessa forma, procura-se definir a classe *Session*, que controla a execução do caso de uso, considerando os objetos que participam da realização

deste caso de uso. A Figura 3.52 mostra o Modelo de Caso de Uso do POO que orientou a identificação das classes *Session* do modelo da Figura 3.50.

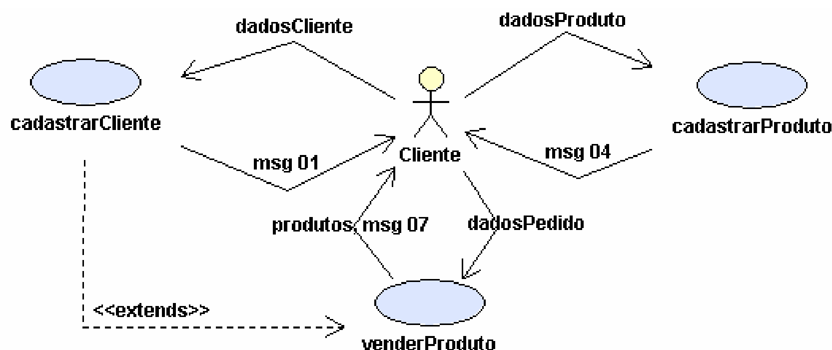


Figura 3.52 - Modelo de Caso de Uso do POO

Uma vez identificada cada classe do POO como *Entity* ou *Session*, pode-se continuar analisando-as para reespecificá-las segundo as idéias da orientação a componentes. Nem todas as classes são classificadas como *Entity* ou *Session.* Classes mais específicas como, por exemplo, de interface e a classe principal do sistema, embora importantes para a compreensão das demais classes, normalmente não geram componentes do domínio do problema.

3.3.1.2. Reespecificar Orientado a Componentes.

O objetivo desta atividade é produzir um projeto, refinado do POO, denominado POC, com características da orientação a componentes.

Técnicas de DBC são utilizadas visando obter o projeto dos componentes, porém ainda sem se preocupar com detalhes de implementação.

Na primeira vez em que esta atividade é realizada trabalha-se com apenas o POO. Nas demais, além do POO, considera-se o POC obtido anteriormente. Dessa forma, na reespecificação do novo POC, levam-se em consideração as decisões do POC já existente, disponível em uma biblioteca.

Para a classe *Entity* do modelo Orientado a Objetos da Figura 3.50 são analisados seu nome, atributos e métodos, para reespecificá-los de forma mais precisa. Dentre as atividades a serem realizadas destacam-se a:

- Reespecificação de cada método, considerando as possíveis entrada e saídas, seus parâmetros e tipo de retorno;
- Reespecificação de cada atributo considerando as responsabilidades do componente;

- Melhoria do modelo re-estruturando métodos e outros aspectos para torná-lo mais claro e fácil de entender. Por exemplo, criando novos métodos e classes mais específicas que evitem duplicações; e
- Verificação das interações, colaborações e dependências de objetos. Deve-se considerar as possíveis interações com os objetos da classe, determinando suas mudanças de estado. Analisam-se como os objetos da classe colaboram com outros objetos para cumprir determinada ação, visando melhor definir o comportamento externo do componente.

A Figura 3.53 mostra, à esquerda, a classe *Entity Cliente*, à direita, tem-se a nova classe *Cliente* reespecificada (1). No caso, os nomes dos atributos foram modificados para facilitar o entendimento e não houve a necessidade de fazer modificações em seus tipos. Os métodos *atualizarCliente()*, *inserirCliente()* e *deletarCliente()* foram realocados em uma classe *Session* mais coerente com suas funcionalidades, no caso a classe *Session CadastrarCliente* (2).

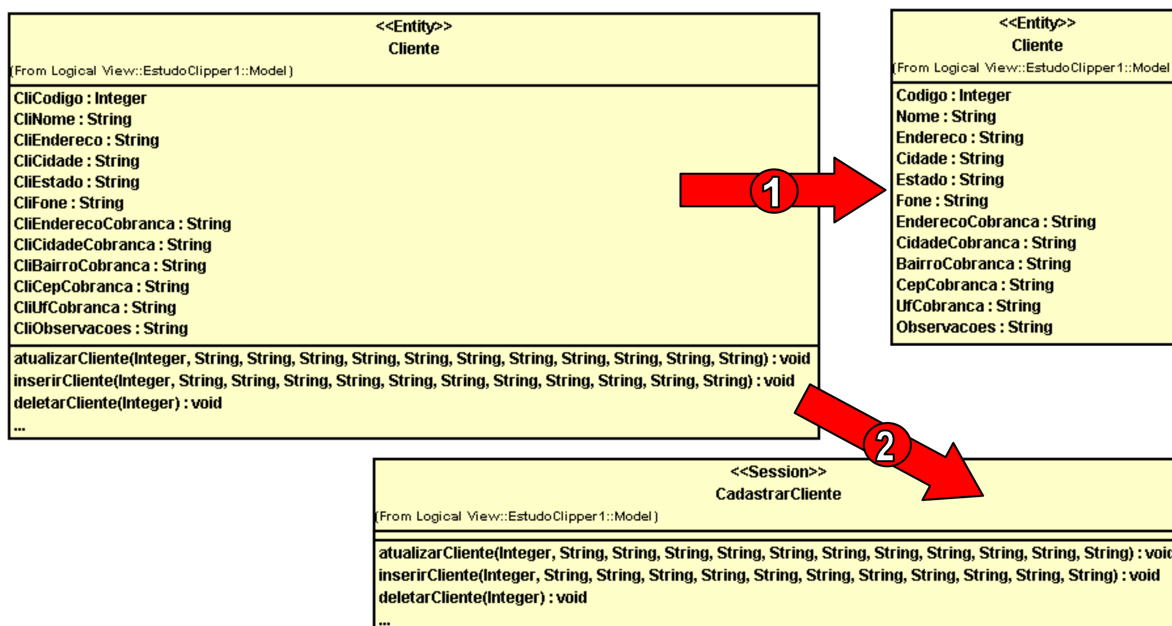


Figura 3.53 – Reespecificação Orientada a Componentes de classe *Entity*.

Outro modelo importante desta etapa é o de Interações, representado em Diagramas de Seqüência, que detalham os comportamentos dos Casos de Uso dos componentes nas diferentes aplicações do domínio do problema. Estes modelos servem para determinar com mais precisão os protótipos dos métodos das classes, conforme os fluxos de execução do caso de uso. A Figura 3.54 mostra, por exemplo, o refinamento de um modelo de interação. Inicialmente tem-se uma seqüência para a venda de produtos, onde o método *fecharVenda()*

da classe de interface *FPedido* utiliza o método *fecharVenda()* da classe *Session VenderProduto* para acessar a classe *Entity Venda*, responsável pela persistência dos dados de uma venda. No refinamento, considerou-se os padrões *Business Delegate*, representado pela classe *DelVenderProduto*, e *Service Locator*, representado pela classe *ServiceLocator*. No novo modelo, tem-se o método *fecharVenda()* da classe de interface *FPedido*, utilizando o método *fecharVenda()* da classe *DelVenderProduto,* que faz a localização da classe *Session VenderProduto*, por meio do método *getHomeJBoss()* da classe *ServiceLocator*. Esta utiliza o método *getHomeJBoss()* da classe *ServiceLocator* para localizar a classe *Entity Venda* que realiza a persistência dos dados de uma venda.

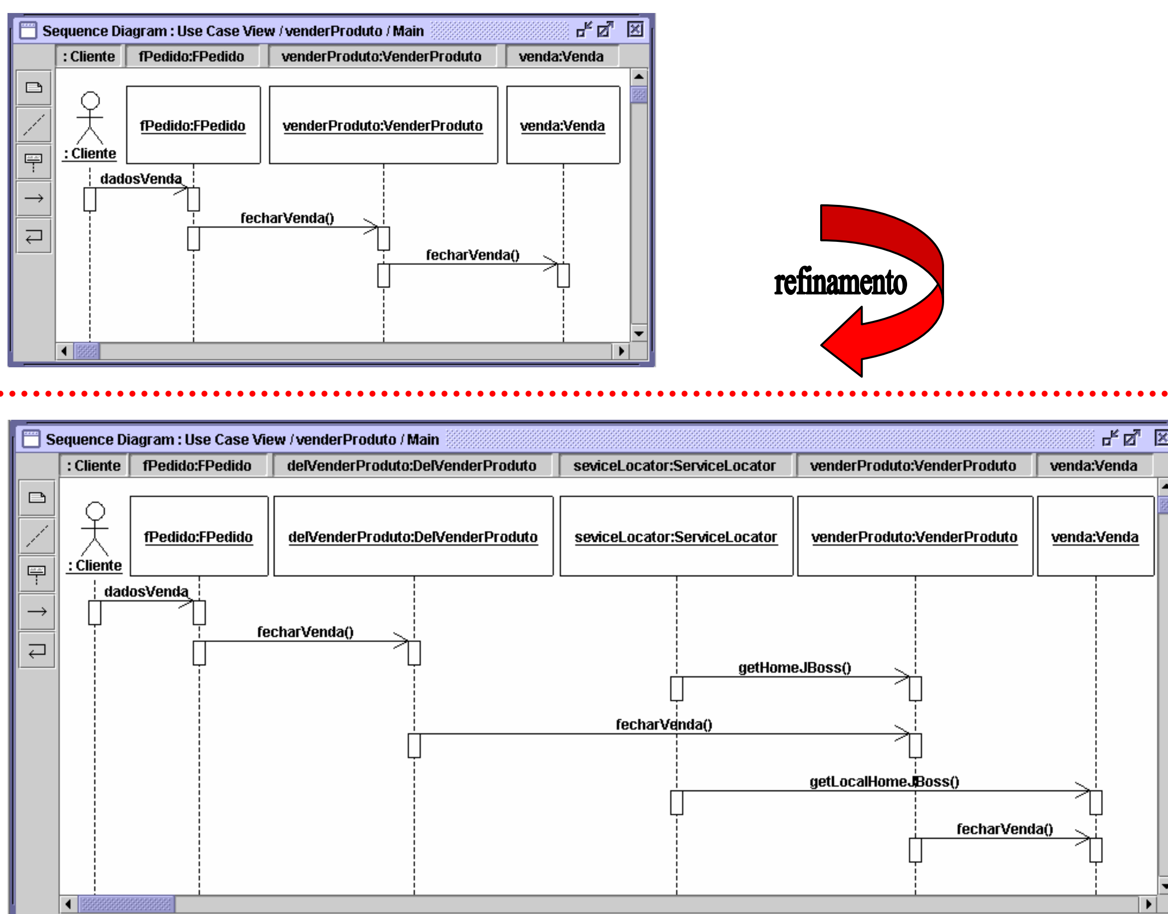


Figura 3.54 – Reespecificação de um Modelo de Interação

Ao concluir a reespecificação, têm-se os novos modelos com características da orientação a componentes, constituindo o Projeto Orientado a Componentes (POC), que fica disponível para novas reespecificações considerando os novos POOs a serem analisados. Quando se repete o passo Reespecificar POO deve-se considerar a existência dos POCs obtidos em análises anteriores. A Figura 3.55 ilustra como é realizada a análise de uma classe do POO considerando outra classe com características semelhantes. A classe *Cliente* do POO

é comparada com a classe *Cliente* do POC. Ao concluir que estas podem ser consolidadas em uma única classe que atenda aos requisitos de ambas, faz-se a reespecificação, fazendo um *merge* de seus atributos e métodos. Pode-se alterar a classe do POC adicionando atributos e métodos específicos da classe do POO, ou pode-se criar uma nova classe POC, para expressar o comportamento conjunto das duas classes inspecionadas. No caso da Figura 3.55, a partir das classes *Cliente* do POO e POC, obteve-se a nova classe *Cliente*, adicionada dos atributos *DataNascimento*, *Naturalidade*, *EstadoCivil* e *Situação*. Na especificação do atributo *NrCliente* da classe Cliente do POO, foi mantido o nome do atributo da classe *Cliente* do POO. Os métodos *cadastra()* e *valida()* da classe *Cliente* do POO foram analisados e verificou-se que já existiam métodos que disponibilizavam os mesmos serviços na classe *CadastrarCliente* do POC, dessa forma, foram mantidas as assinaturas dos métodos da classe *CadastrarCliente* do POC.

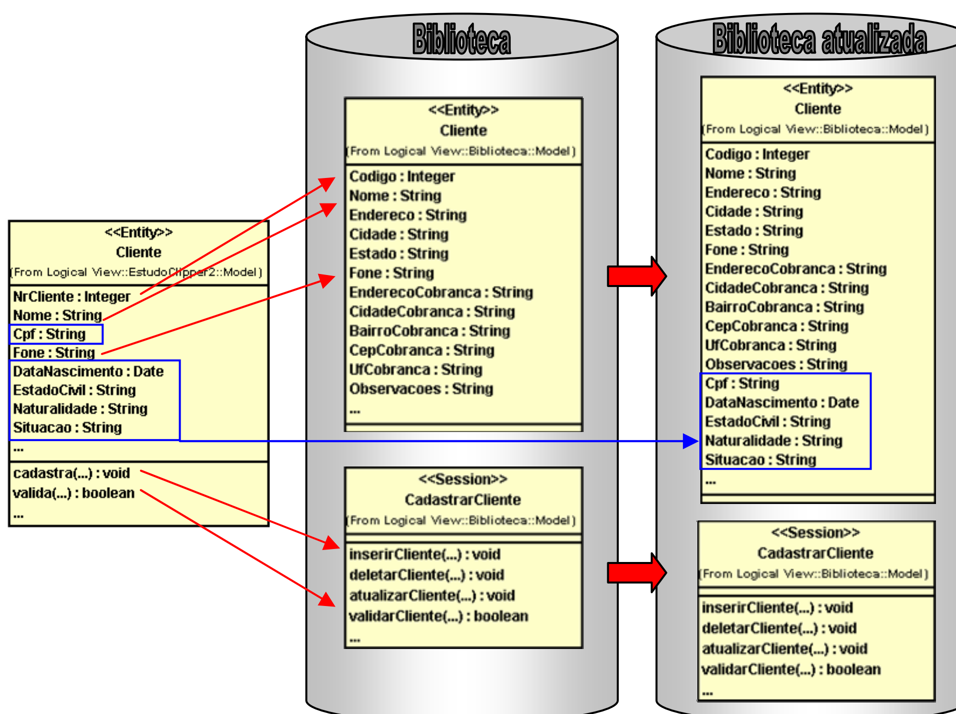


Figura 3.55 - Reespecificação de classe do POC considerando classes do POO.

No caso de classes *Session*, o processo é o mesmo, considerando principalmente os objetos das classes *Entity* que colaboram para cumprir suas responsabilidades. A Figura 3.56 mostra na parte superior, a classe *Session CadastrarCliente* e abaixo, a nova classe reespecificada. No caso, os métodos *inserirCliente()* e *atualizarCliente()* tiveram seus parâmetros alterados para atender as novas especificações da classe *Entity Cliente*, reespecificada conforme a Figura 3.55.

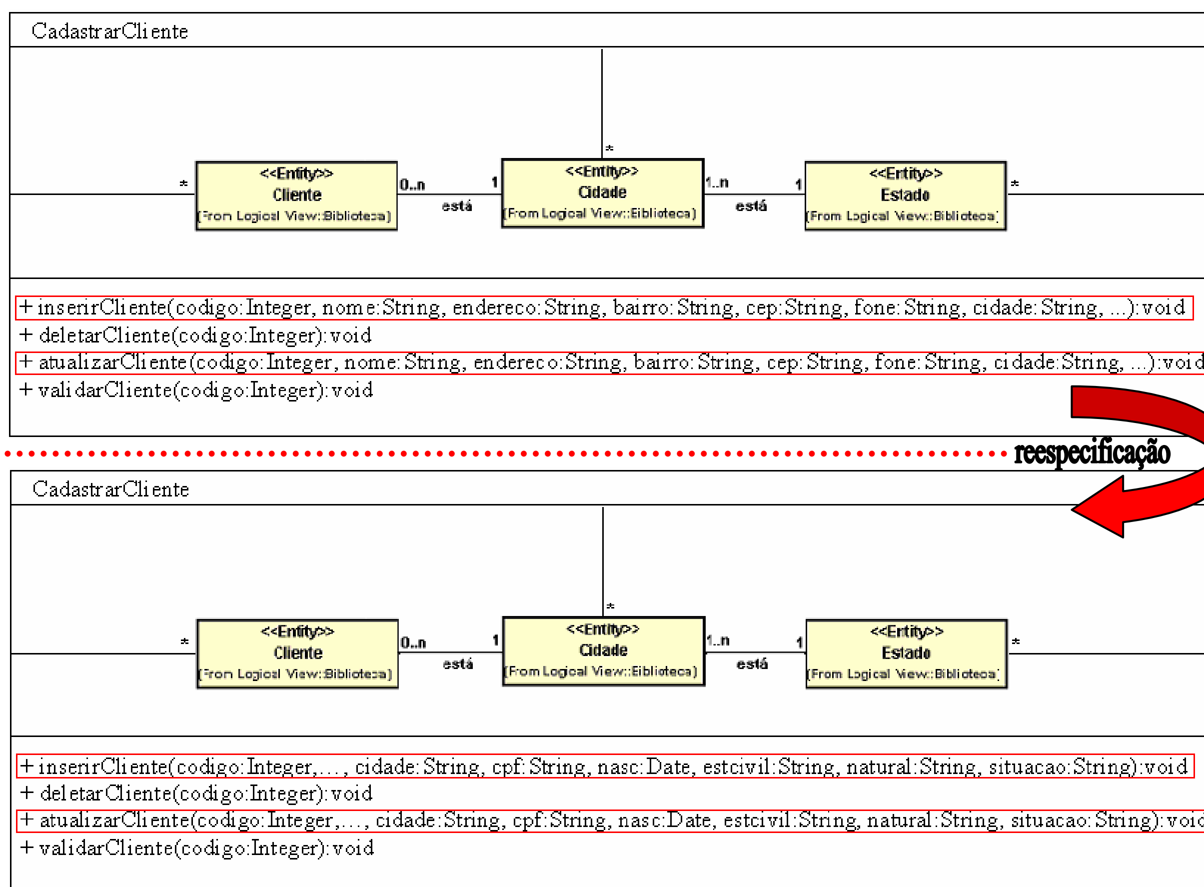


Figura 3.56 - Reespecificação Orientada a Componentes de classe *Session*.

Ao final desta atividade tem-se o POC que contém os modelos de componentes representados pelos diagramas de classes, casos de uso e de seqüências. A Figura 3.57 mostra o modelo de classes de um componente *Session* obtido com a reespecificação Orientada a Componentes. O modelo da Figura 3.57 é equivalente ao da Figura 3.56 e esta notação é a que tem sido usada pelas ferramentas CASE. Conforme mostra a Figura 3.57, observa-se que as classes Entity se relacionam mostrando que um *Cliente* está em uma *Cidade*, a qual está em um *Estado*. Observa-se que a classe Session *CadastrarCliente* tem um papel de gerenciamento das operações de cadastro de um cliente através de seus métodos *inserirCliente()*, *deletarCliente()*, *atualizarCliente()* e *validarCliente()*.

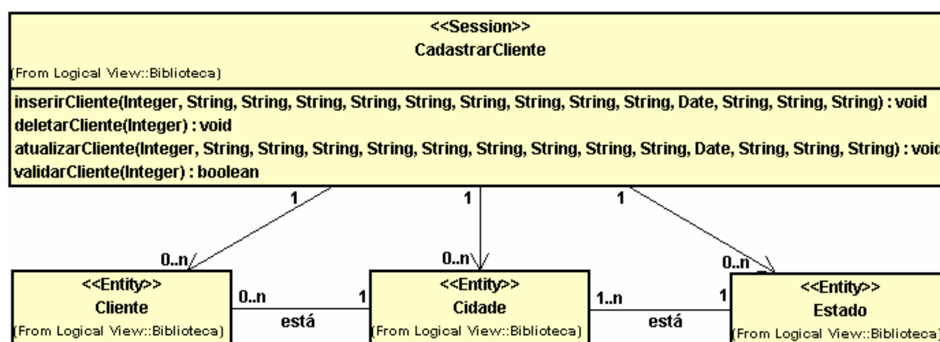


Figura 3.57 - Modelo de classes do componente *Session CadastrarCliente*.

Outra atividade do passo **Refinar Projeto Orientado a Componentes** consiste em identificar possíveis padrões no POC. O uso de padrões pode aumentar o reuso dos componentes e facilitar seu entendimento e manutenção.

3.3.1.3. Identificar Padrões

Componentes construídos a partir de sistemas legados representam soluções previamente testadas de experiências anteriores. O conhecimento neles embutido pode ser utilizado como base para a identificação de padrões de projeto em um domínio. Se um componente é encontrado em diferentes sistemas, este é candidato a compor um padrão dentro do domínio do problema.

Por exemplo, no caso do domínio de Vendas, podem ser identificados alguns padrões recorrentes, isto é, padrões que ocorrem repetidamente no desenvolvimento de sistemas deste domínio.

A primeira atividade que o Engenheiro de Software realiza para identificar um padrão é a generalização dos modelos de classes através da fatoração de suas funcionalidades e atributos [Gam95]. As funcionalidades e atributos que estão fora do contexto da classe são realocados em outras classes, visando melhorar a coesão e diminuir seu acoplamento, facilitando assim o reuso. A execução desta atividade é orientada pelo padrão *Decoupling* apresentado por D'Souza em [Sou99].

A Figura 3.58 mostra, por exemplo, o refinamento da classe *Session VenderProduto*. Esta classe possui o método *baixarEstoque()* cuja responsabilidade é diminuir do estoque a quantidade vendida de determinado produto. Essa funcionalidade pode não ser utilizada em algumas aplicações, e, portanto, deve ser realocada em outra classe mais apropriada, no caso a classe *AtualizarEstoque* (1). A classe *VenderProduto* (2), agora mais genérica, atende a maior parte das aplicações do domínio sem a necessidade de modificações, ou seja, seu reuso foi aumentado e seu refinamento, considerando mais sistemas deste domínio, pode torná-la um padrão dentro do domínio do problema. Caso seja necessária a utilização do método *baixarEstoque(),* o Engenheiro de Software utiliza a classe *AtualizarEstoque* no desenvolvimento de sua aplicação.

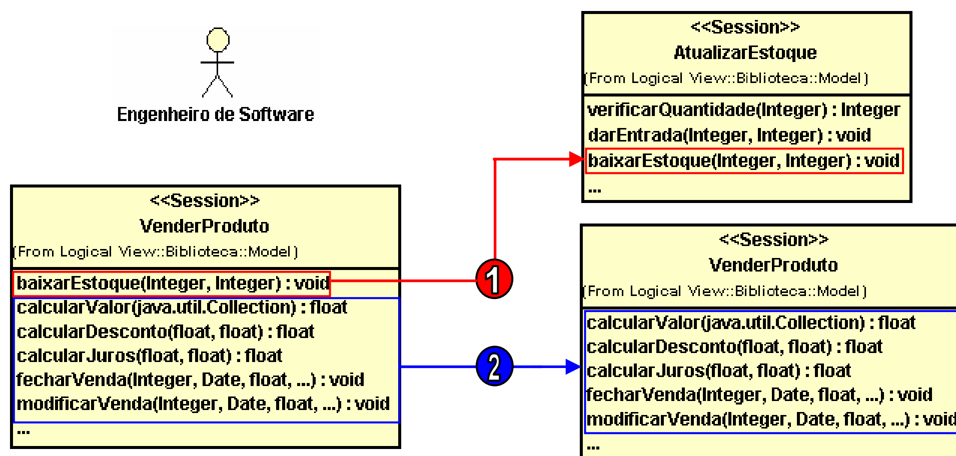


Figura 3.58 - Generalização de uma classe

Outra análise consiste em identificar os Casos de Uso comuns a vários sistemas. Os Casos de Uso já foram utilizados na identificação das classes *Session*, porém estas classes podem ou não participar da composição de um padrão dentro do domínio do problema. Assim, uma nova análise deve ser feita, visando encontrar os Casos de Uso comuns aos sistemas deste domínio.

No domínio de Vendas, foram identificados alguns Casos de Uso comuns aos três sistemas analisados (*Informatiza*, *Caiçara* e *Gnome*), conforme mostra a Tabela 3.5.

Caso de Uso	Descrição
CadastrarCliente	Cadastro e validação de clientes.
CadastrarVendedor	Cadastro e validação de vendedores.
CadastrarFuncionario	Cadastro e validação de funcionários.
CadastrarFornecedor	Cadastro e validação de fornecedores.
CadastrarProduto	Cadastro e validação de produtos.
VenderProduto	Venda de produtos aos clientes.
ComprarProduto	Compra de produtos dos fornecedores.
AtualizarEstoque	Manutenção do estoque de produtos.
PagarDuplicata	Pagamento das duplicatas referentes às compras de produtos dos fornecedores.
ReceberDuplicata	Recebimento dos valores das duplicatas referentes às vendas de produtos aos clientes.

Tabela 3.5 - Casos de Uso comuns nos sistemas Informatiza, Caiçara e Gnome

Estes Casos de Uso servem de partida para a identificação dos padrões. Para cada caso de uso, o Engenheiro de Software analisa o POC, identificando as classes que dele participam. As Figuras 3.59 e 3.60 mostram, por exemplo, as classes *Entity* e *Session* com seus relacionamentos e dependências, responsáveis pelo caso de uso *CadastrarCliente* e *CadastrarFornecedor*, respectivamente.

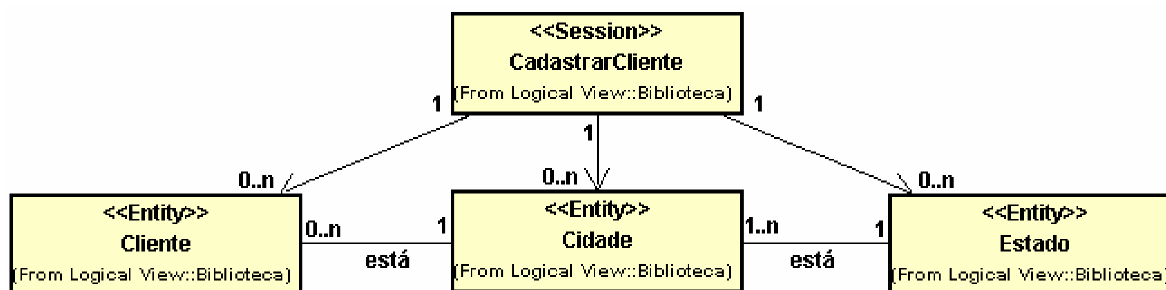


Figura 3.59 - Classes envolvidas no Caso de Uso *CadastrarCliente*

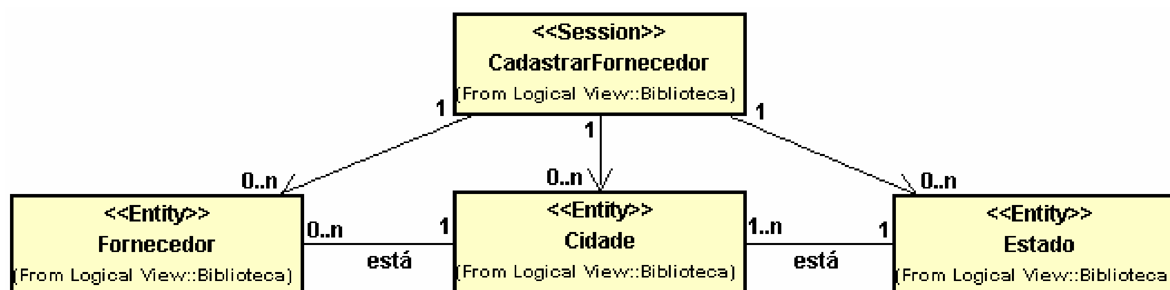


Figura 3.60 - Classes envolvidas no Caso de Uso *CadastrarFornecedor*

Analisando os modelos das classes responsáveis pelos cadastros de clientes e fornecedores, verifica-se que estes têm comportamentos semelhantes, envolvendo os mesmos objetos. Assim pode-se generalizar estes modelos em um *framework* criando um *template package* [Sou99], que pode ser reutilizado pelos diferentes sistemas deste domínio. A Figura 3.61 mostra o padrão *CadastrarPessoa*, identificado com base nos sistemas analisados, que atende tanto o cadastro de clientes, como de fornecedores, vendedores e funcionários. O *stereotype* “*<<framework>>*” é utilizado para identificar o padrão. Os nomes dos tipos genéricos dentro do framework estão escritos entre os caracteres ”< >”, que servem para orientar a instanciação do padrão quando ele for aplicado. As assinaturas dos métodos genéricos do padrão *CadastrarPessoa* são descritas considerando os atributos genéricos do *CadastrarPessoa*.

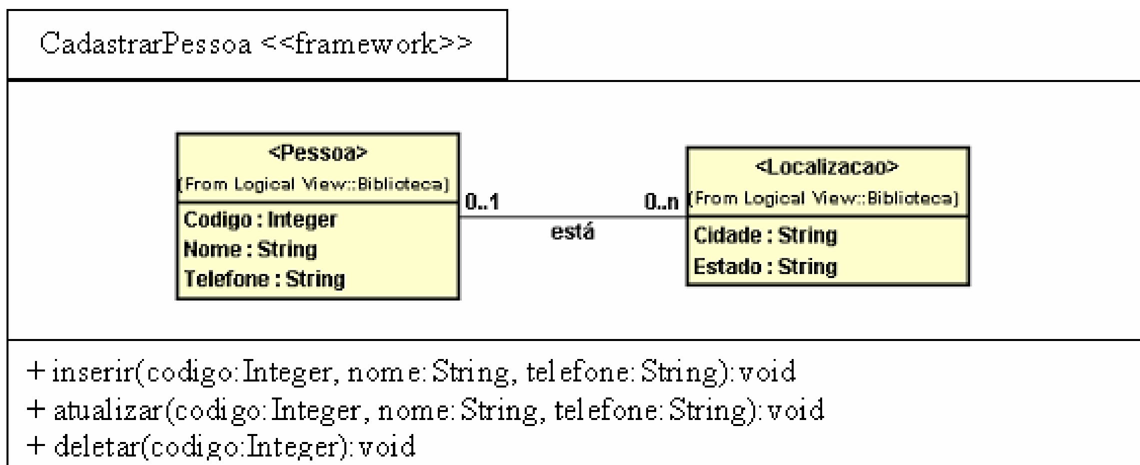


Figura 3.61 - Padrão *CadastrarPessoa*

O tipo genérico <*Pessoa*> representa uma classe genérica responsável pelos objetos *pessoa*. O tipo genérico <*Localizacao*> representa uma ou mais classes responsáveis pelos objetos de localidades que estão associadas à <*Pessoa*>.

Em outro exemplo, a figura 3.62 mostra as classes que atendem o caso de uso *CadastrarProduto*, onde objetos da classe *Session CadastrarProduto* interage com classes *Entity TipoProduto*, *Produto*, *Fornecedor*. Analisando o modelo da Figura 3.63, observou-se que o mesmo pode ser generalizado em um *framework* criando o *template package CadastrarProduto* mostrado na Figura 3.63. Este padrão engloba os tipos genéricos <*Fornecedor*>, <*Produto*> e <*Categoria*> que interagem para atender o cadastro de produtos do sistema.

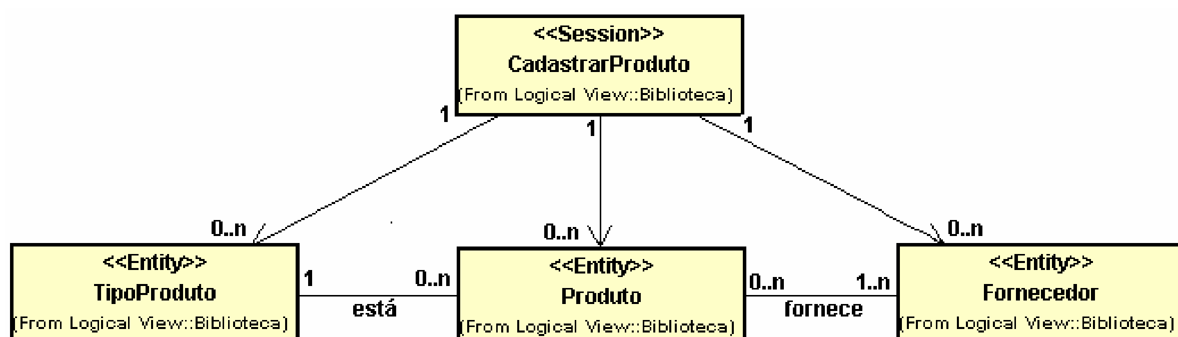


Figura 3.62 - Classes envolvidas no Caso de Uso *CadastrarProduto*

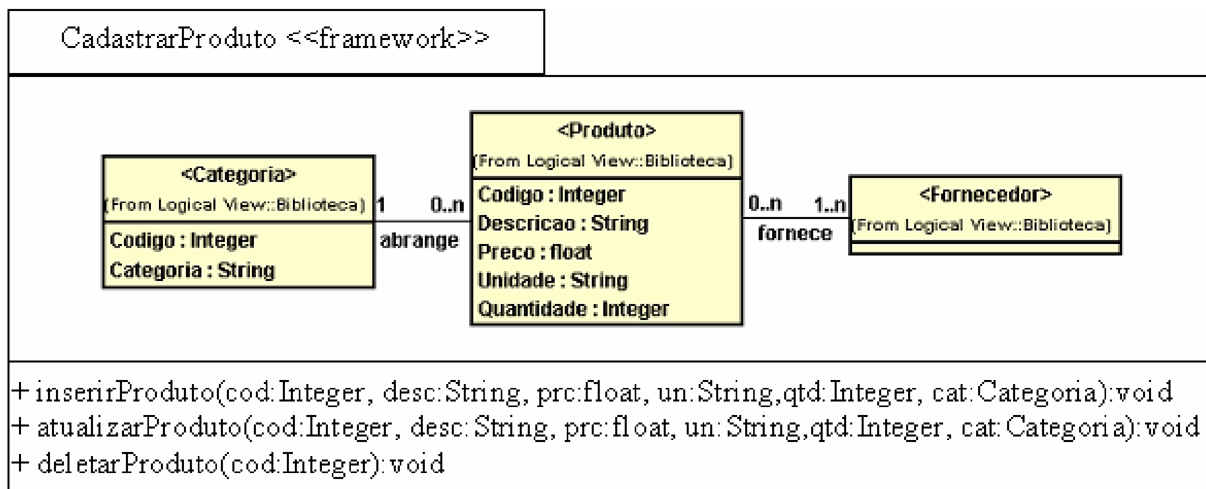


Figura 3.63 - Padrão *CadastrarProduto*

O tipo genérico <*Produto*> representa uma classe genérica responsável pelos objetos *produto*. O tipo genérico <*Categoria*> representa a classe genérica responsável pela persistência de dados referentes à categoria do produto. O tipo genérico <*Fornecedor*> representa a classe responsável pela persistência de dados referentes ao fornecedor do produto.

Considere agora os Casos de Uso *VenderProduto* e *ComprarProduto*. No caso de uso *VenderProduto*, cujo Modelo de Classes é mostrado na Figura 3.64, verifica-se que a classe *Session VenderProduto* interage com as classes de *Entity Cliente*, *Venda*, *ItemVenda*, *Produto* e *Vendedor*. Observa-se que existe uma classe *Session* responsável pelas regras de negócio da venda. Os dados da venda de um produto são persistidos através das classes *Entity Venda* e *ItemVenda*. Uma pessoa solicita a venda, no caso representada pela classe *Cliente* e outra efetua a venda, representada pela classe *Vendedor*.

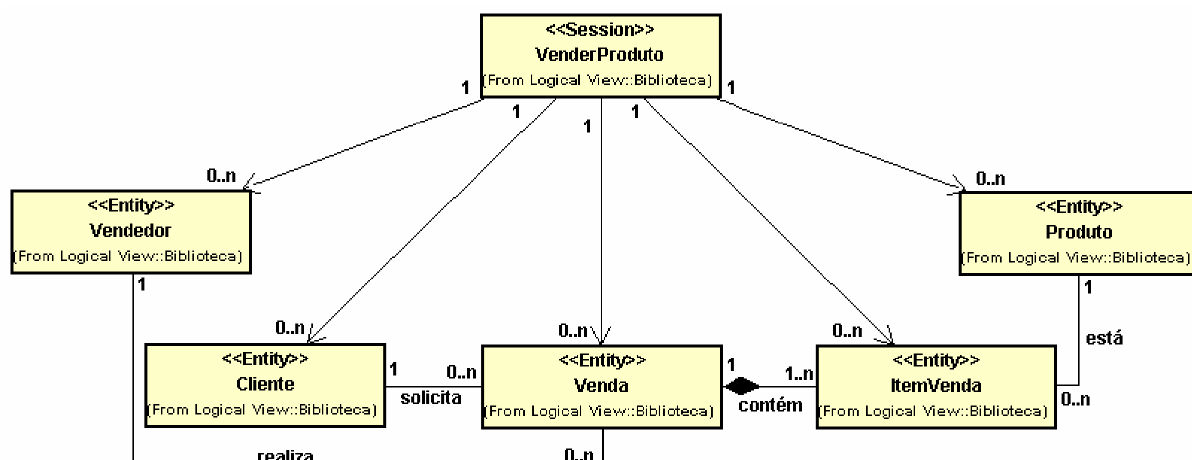


Figura 3.64 - Classes envolvidas no Caso de Uso *VenderProduto*

Observando agora o caso de uso *ComprarProduto*, cujo Modelo de Classes é mostrado na Figura 3.65, verificou-se que as interações ocorrem de forma semelhante ao apresentado no caso de uso *VenderProduto*. No caso, a classe *Session ComprarProduto* é responsável pelas regras de negócio da compra. Esta controla a compra de produtos de um fornecedor, representado pela classe *Fornecedor*. Os dados da compra são persistidos através das classes *Compra* e *ItemCompra*. Existe também um responsável pela compra, representado pela classe *Funcionario*.

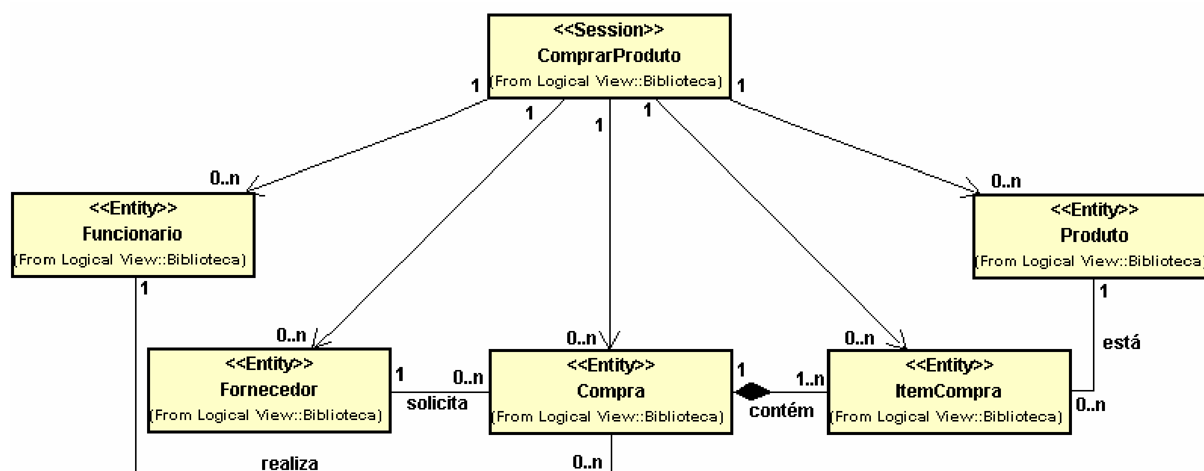


Figura 3.65 - Classes envolvidas no Caso de Uso *ComprarProduto*

Devido à semelhança das interações nos modelos das Figuras 3.64 e 3.65 que representam a venda e compra de produtos, respectivamente, pode-se sugerir um padrão comum às duas situações. Assim, generalizado estes modelos em um *framework,* definiu-se o *template package NegociarProduto*. Este padrão atende os Casos de Uso *ComprarProduto* e *VenderProduto* e engloba os tipos genéricos <*Pessoa*>, <*Responsavel*>, <*Negociacao*>, <*Item*> e <*Produto*> que interagem para realizar a negociação de produtos no sistema, traduzida pela compra ou venda de produtos.

No padrão *NegociarProduto* apresentado na Figura 3.66, o tipo genérico <*Negociacao*> representa uma classe *Session* referente à venda ou compra de um produto e armazena os itens negociados em <*Item*>. O tipo genérico <*Pessoa*> representa um cliente, quando a negociação for uma venda de produtos, ou um fornecedor, caso a negociação seja uma compra de produtos. O tipo genérico <*Responsavel*> representa um vendedor, no caso de uma venda de produtos, ou um funcionário, no caso de uma compra de produtos. O tipo genérico <*Produto*> representa os produtos negociados.

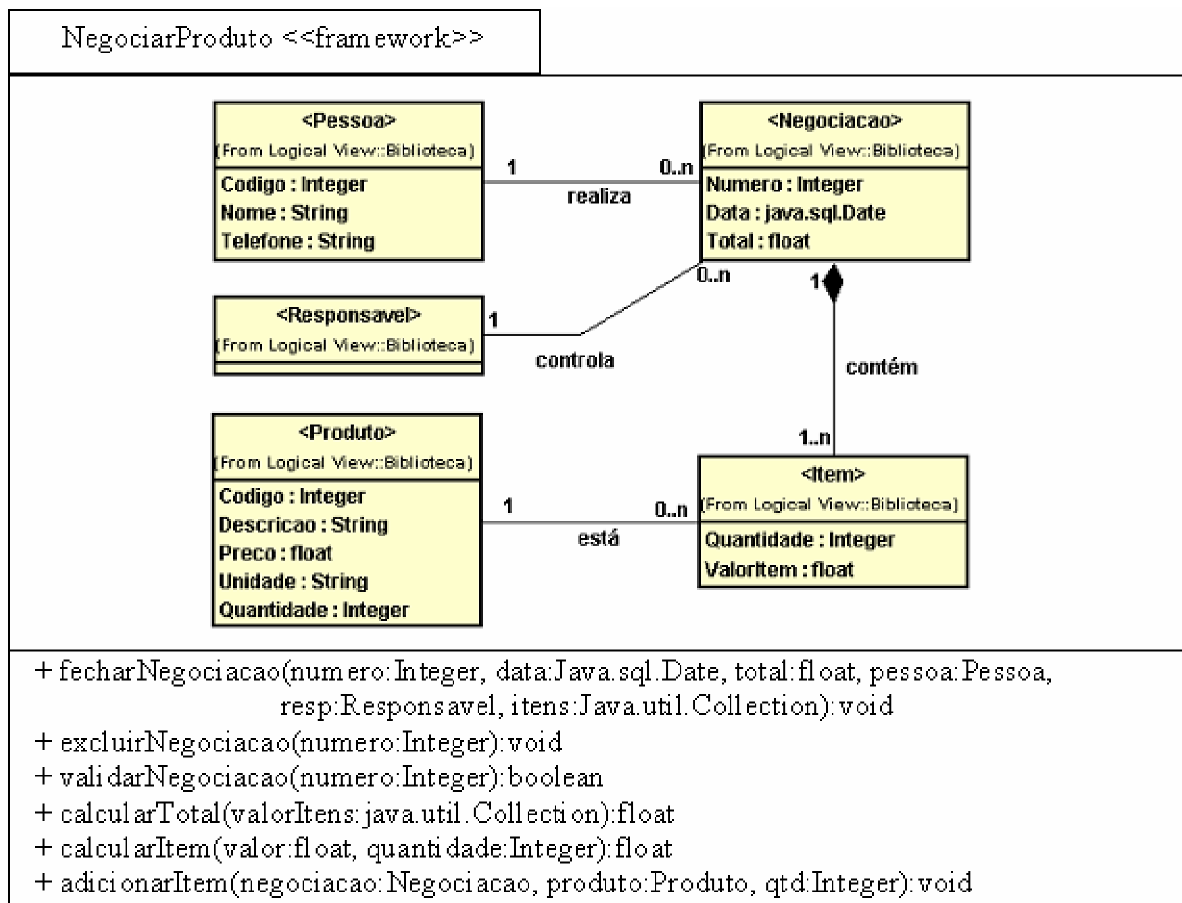


Figura 3.66 - Padrão *NegociarProduto*

No caso de uso *AtualizarEstoque*, cujo Modelo de Classes é mostrado na Figura 3.67, verifica-se uma interação entre as classes *AtualizarEstoque*, *EntradaEstoque*, *EntradaItem*, *Produto*, *SaidaEstoque*, *SaidaItem*. A classe *Session AtualizarEstoque* gerencia as regras de negócio para a atualização do estoque, interagindo com as classes de persistência *EntradaEstoque* e *EntradaItem* responsáveis pela entrada de produtos no estoque, e com as classes *Entity SaidaEstoque*, *SaidaItem*, responsáveis pela saída de produtos do estoque.

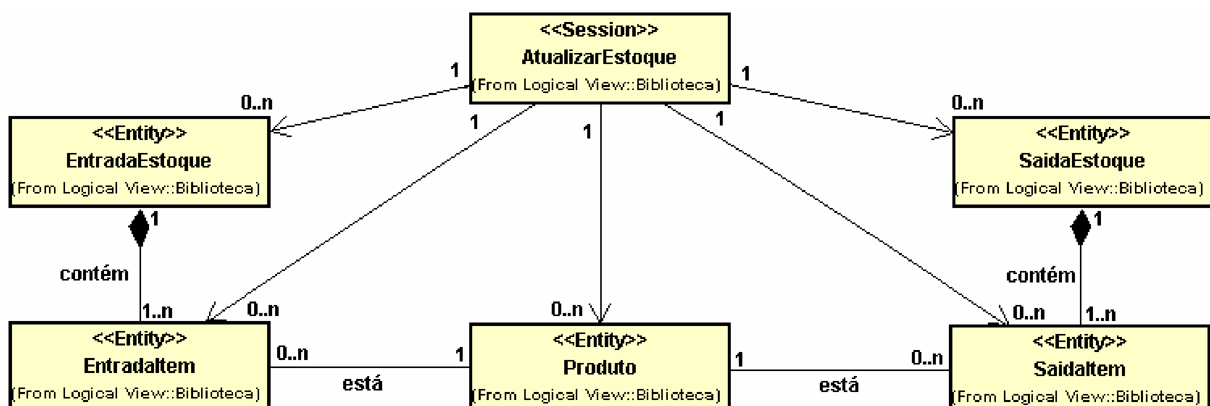


Figura 3.67 - Classes envolvidas no Caso de Uso *AtualizarEstoque*

Para a manutenção do estoque, também se observa um comportamento comum na interação das classes. Este modelo pode ser generalizado em um *framework* criando o *template package OperarEstoque*, com os tipos genéricos *<Negociacao>*, *<Operacao>*, *<Item>* e *<Produto>*, conforme mostra a Figura 3.68.

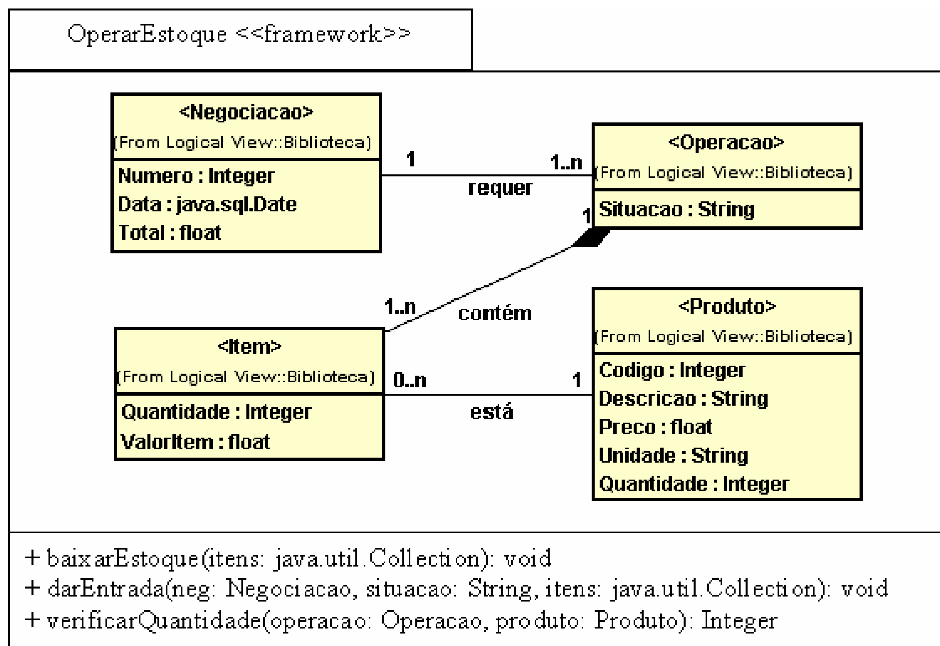


Figura 3.68 - Padrão *OperarEstoque*

Da mesma forma, considerando os Casos de Uso *PagarDuplicata* e *ReceberDuplicata*, foi possível identificar as classes que interagem para realizar o pagamento e recebimento de duplicatas nos sistemas analisados, conforme mostra o modelo da Figura 3.69.

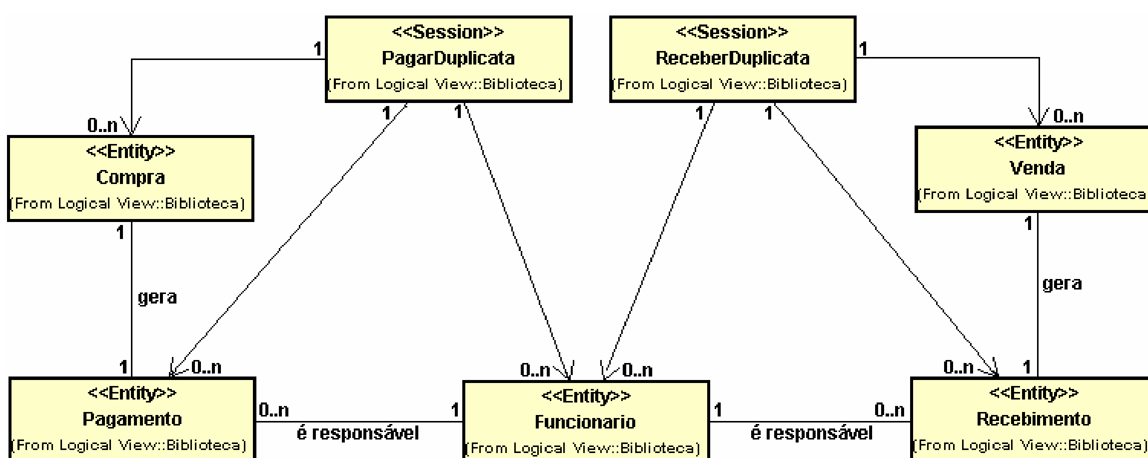


Figura 3.69 - Classes envolvidas no pagamento e recebimento de duplicatas

Estudando as semelhanças no modelo, identificou-se o *template package AdministrarFinanças*, com os tipos genéricos *<Negociacao>*, *<Operacao>*, *<Pessoa>* e *<Responsavel>*, conforme mostra a Figura 3.70.

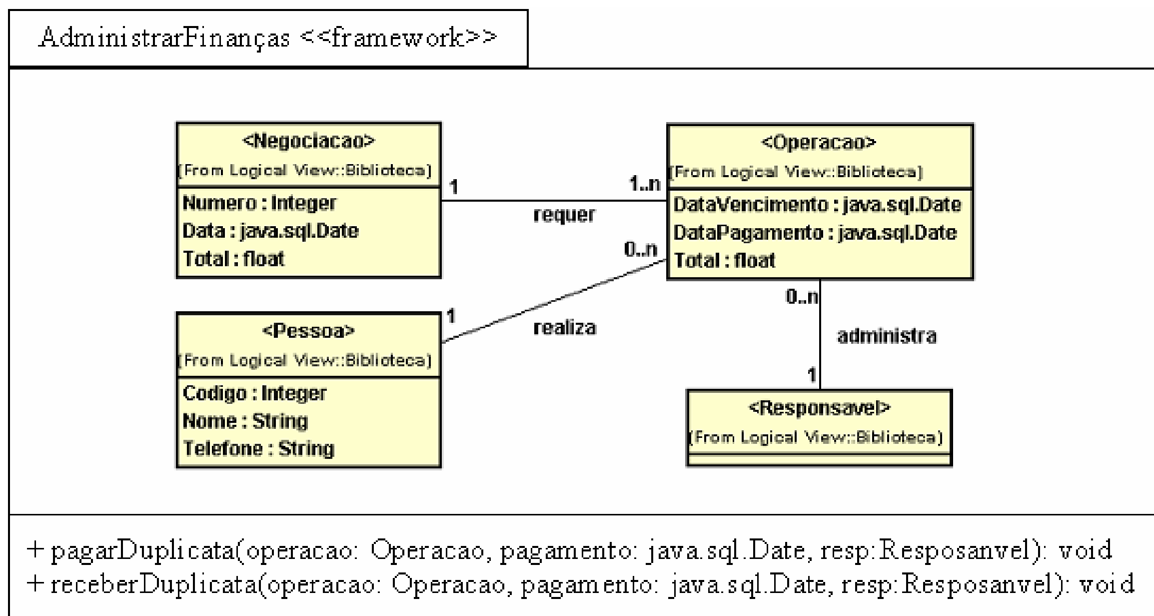


Figura 3.70 - Padrão *AdministrarFinanças*

Dessa forma, foram generalizados os principais Casos de Uso recorrentes nos sistemas analisados, criando-se padrões para o domínio de Vendas.

Em resumo, as atividades para a identificação de padrões são:

- Generalizar as classes, realocando os métodos e funcionalidades em classes mais coerentes com seu contexto, aumentando a coesão e diminuindo o acoplamento;
- Identificar os Casos de Uso que são recorrentes no maior número possível de sistemas do domínio do problema;
- Identificar classes comuns que interagem em cada caso de uso; e
- Abstrair os frameworks a partir dos Casos de Uso, criando os padrões que podem ser reutilizados na reconstrução dos sistemas.

Ao final do passo *Refinar Projeto Orientado a Componentes*, têm-se definidos o comportamento e as responsabilidades de cada classe que representa um componente, porém ainda sem se preocupar com detalhes de implementação considerando a arquitetura e a plataforma de hardware e software a ser adotada para os componentes. Essas atividades são detalhadas no passo *Implementar Componentes*, apresentado a seguir.

3.3.2. Implementar Componentes

Neste passo, faz-se a implementação dos componentes conforme a arquitetura adotada. Diferentes plataformas e arquiteturas podem ser usadas na implementação. No caso do RSCT adotou-se a arquitetura EJB [Sun04b], na plataforma Windows 2000, com banco de

dados relacional MySQL [Mys04]. A decisão de usar EJB facilita a implementação considerando que na fase de projeto, as classes dos componentes já foram organizadas em *Entity* e *Session*.

Na arquitetura EJB um componente tem duas interfaces. A interface *home* define os métodos do ciclo de vida do componente e, quando este componente é de persistência, esta interface também define seus métodos de procura. A interface *remote* define os métodos de negócio específicos para o componente.

Para que os métodos das classes dos componentes possam ser corretamente organizados para a criação das interfaces, estes devem ser refinados antes desta operação. O Engenheiro de Software analisa cada método e define seu *stereotype* utilizando a informação "*BusinessRule*" considerada pela ferramenta MVCASE no projeto do componente. A Figura 3.71 mostra, por exemplo, o refinamento do método de negócio *verificarCPF()* da classe *CadastrarCliente*, cujo protótipo será implementado na interface *remote* do componente.

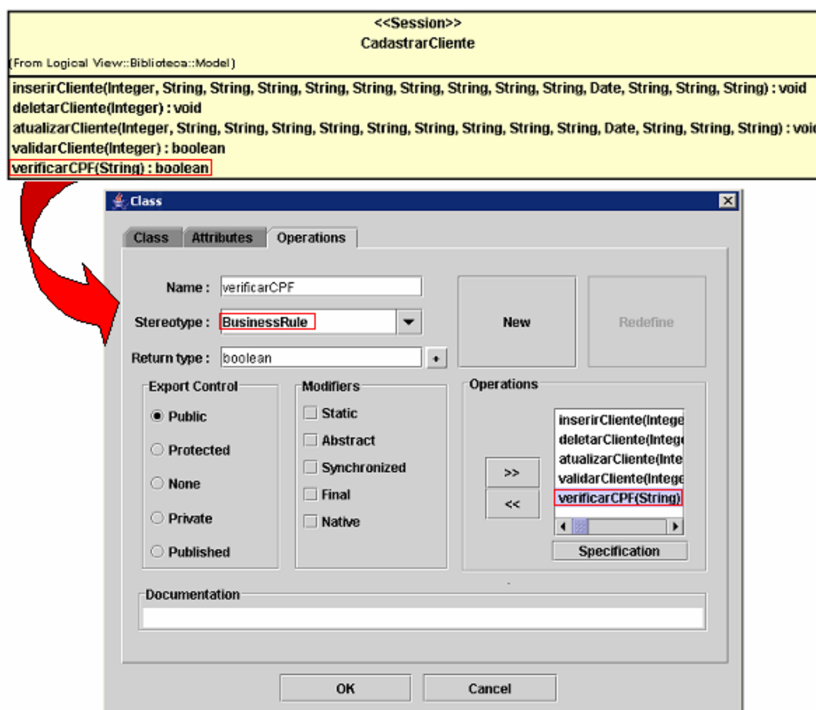


Figura 3.71 - Refinamento de um método no projeto de um componente

As mini-especificações dos métodos devem ser refinadas considerando os detalhes da arquitetura EJB. O Engenheiro de Software baseia-se nos Modelos de Interações para orientar o reprojeto dos métodos. A Figura 3.72 mostra, por exemplo, as mini-especificações do corpo do método *fecharVenda()* do componente Session *VenderProduto*, onde foram efetuadas algumas modificações para interagir com o padrão *ServiceLocator* (1), e com os componentes *Entity VendaBean* (2), *ClienteBean* (3), *VendedorBean* (4) e *FaturaBean* (5).

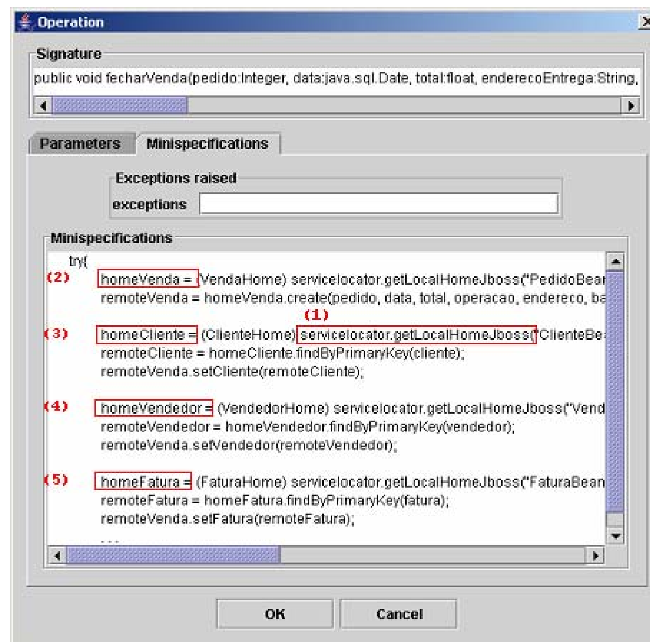


Figura 3.72 - Modificação do corpo de um método

Baseados na arquitetura adotada são gerados os componentes *Entity* e *Session* usando a ferramenta MVCASE. A Figura 3.73 mostra, por exemplo, os modelos de componentes persistentes *ClienteBean*, *VendedorBean* e *VendaBean* e transientes *CadastrarCliente* e *VenderProduto* construídos neste passo. Estes componentes interagem com o padrão *BusinessDelegate*, representado neste exemplo pelos componentes *DelCadastrarCliente* e *DelVenderProdutos*, que controlam as requisições do cliente, fazendo os redirecionamentos necessários para o componente *Session* mais adequado para atender esta requisição. Também está disponível o padrão *ServiceLocator*, responsável pelo controle de transações entre os componentes, como por exemplo, as chamadas via JNDI que reduzem a performance da aplicação, uma vez que muitos recursos são consumidos na localização de um componente.

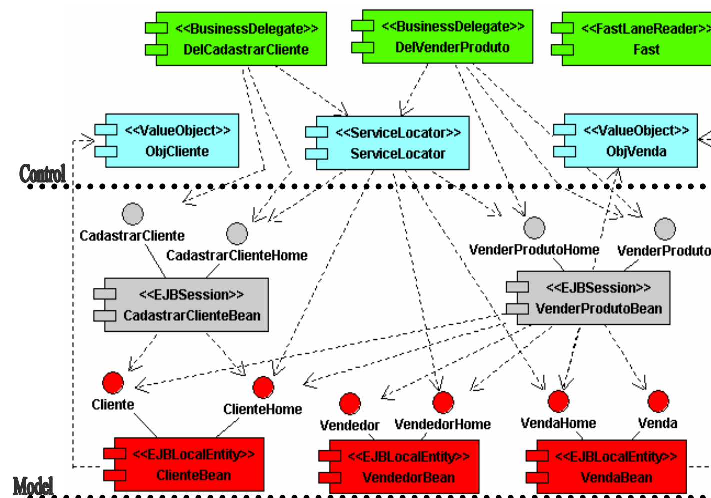


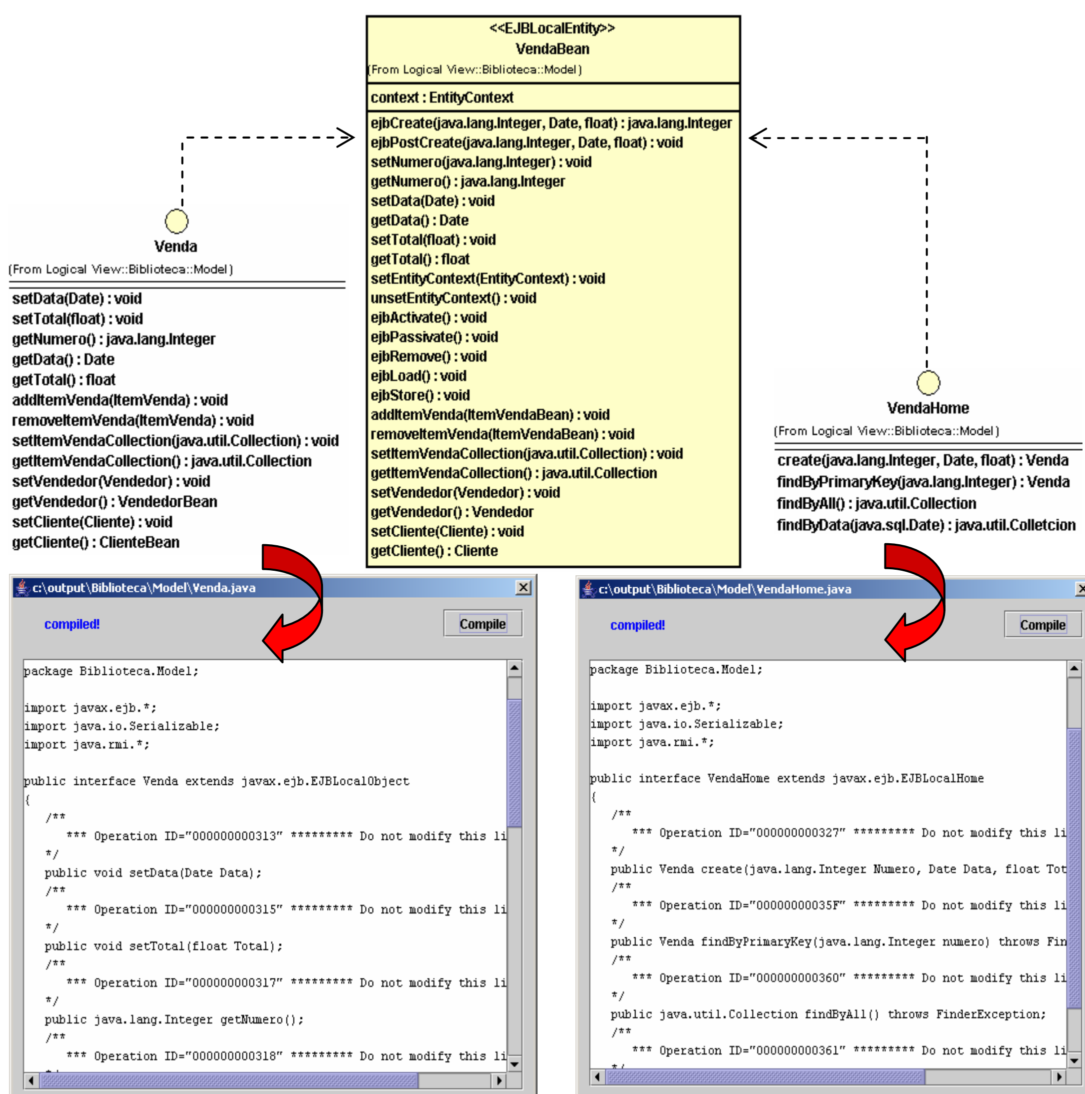
Figura 3.73 - Interação entre os padrões e componentes

Uma primeira versão da implementação pode ser gerada pelo Engenheiro de Software com apoio da ferramenta MVCASE, considerando a arquitetura definida no projeto dos componentes. A ferramenta MVCASE pode gerar uma implementação parcial dos componentes, conforme a arquitetura EJB. Em seguida pode-se refinar o código com mais detalhes de implementação, considerando outros requisitos não funcionais.

A Figura 3.74 mostra, por exemplo, a geração do código Java/EJB do componente *Entity Venda* com suas interfaces, que no caso do EJB, recebem as seguintes convenções:

- <nome>+“Bean*”* – Classe que contém a implementação do componente;
- <nome >+“Home” – Interface *home* do componente;
- <nome > – Interface *remote* do componente;

Na parte inferior da Figura 3.74 tem-se a implementação das interfaces *VendaHome* e



Venda do componente *Entity VendaBean* na ferramenta MVCASE.

Figura 3.74 - Geração de código Java na ferramenta MVCASE

Depois de construídos os componentes, estes ficam disponíveis em uma biblioteca para reuso. A próxima seção apresenta detalhes da reconstrução de sistemas reutilizando os padrões e componentes da biblioteca, última fase do RSCT.

3.4. Fase 4 – Reconstruir Sistema

Nesta Fase do RSCT faz-se a reconstrução dos sistemas, agora com características de Orientação a Componentes. Além de Técnicas de Desenvolvimento Baseado em Componentes, o Engenheiro de Software utiliza Padrões de Projeto visando facilitar o reuso no projeto e implementação do sistema Orientado a Componentes.

A Fase *Reconstruir Sistema* é dividida em: *Reespecificar*, *Reprojetar* e *Reimplementar Orientado a Componentes*, conforme mostra a Figura 3.75.

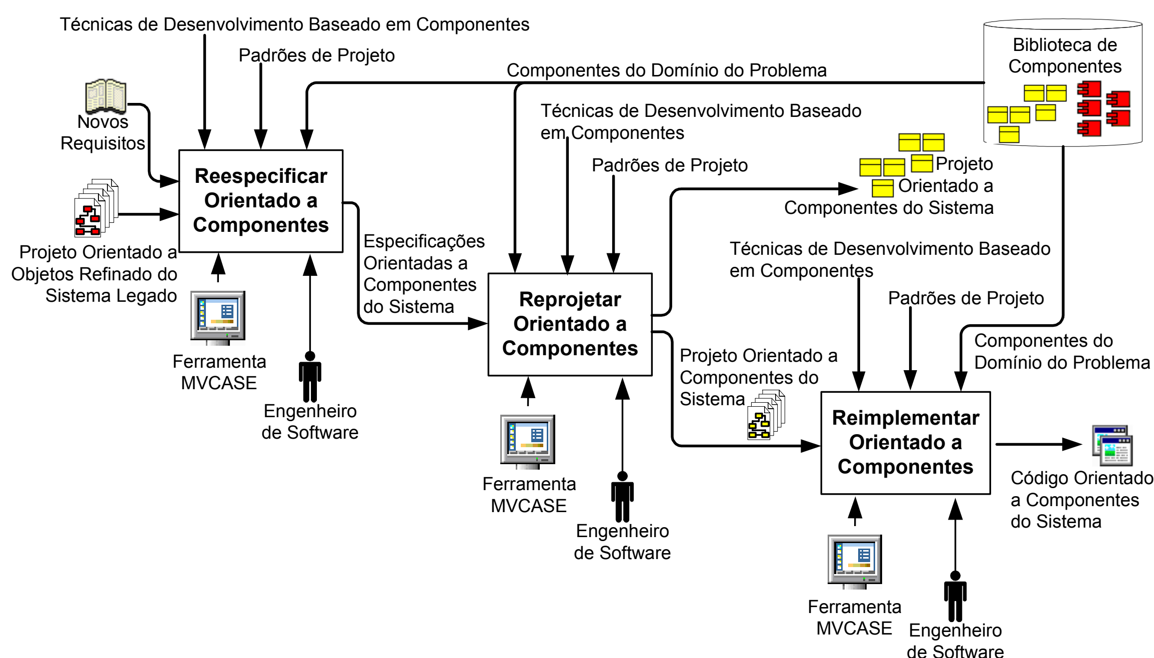


Figura 3.75 - Reconstruir Sistema (Fase 4)

Segue-se uma apresentação mais detalhada dos passos que compõem a Fase 4.

3.4.1. Reespecificar Orientado a Componentes

Nesse passo, reespecifica-se o Projeto Orientado a Objetos do sistema legado recuperado na Fase 2. Apesar da Reengenharia de Software não prever mudanças de funcionalidades na reconstrução de sistemas, neste passo é possível incluir novos requisitos, adicionar novas funcionalidades e aplicar novas tecnologias, visando facilitar o Reprojeto Orientado a Componentes.

A reespecificação baseia-se nos POOs dos sistemas, representados pelos Modelos de Classes e de Interações. Na reespecificação define-se os comportamentos externos das classes, com suas responsabilidades, escopos, operações e interfaces. Pode-se ainda reespecificar os Modelos de Interações para manter suas consistências com os casos de uso que tenham sido alterados ou para um melhor entendimento do comportamento do sistema que está sendo reconstruído.

Uma das técnicas usadas neste passo baseia-se na utilização dos padrões definidos na Fase 3. Estes padrões foram especificados em um alto nível de abstração, disponibilizando modelos genéricos que orientam o Engenheiro de Software no reuso dos componentes. As Figuras 3.76 e 3.77 mostram, respectivamente, os padrões *CadastrarPessoa* e *NegociarProduto*, representados em Frameworks de Modelos [Sou99]. Quando um Framework de Modelos é aplicado, seus tipos genéricos podem ser substituídos por um ou mais componentes que colaboram para realizar o padrão.

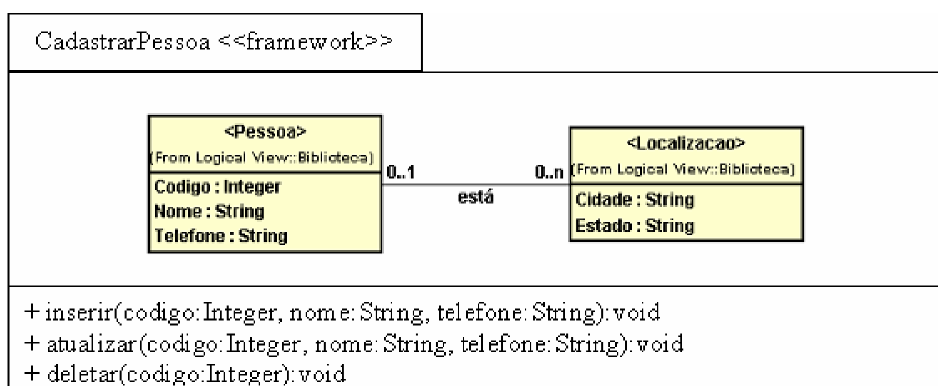


Figura 3.76 - Framework de Modelos do padrão *CadastrarPessoa*

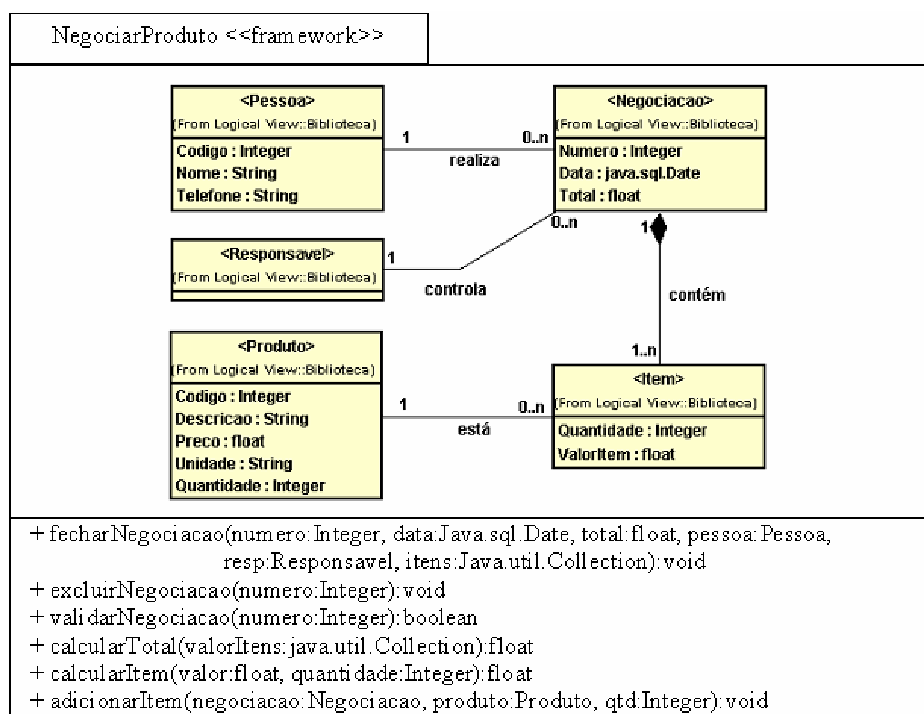


Figura 3.77 - Framework de Modelos do padrão *NegociarProduto*

A aplicação destes padrões na reespecificação do sistema resulta em um Modelo de Aplicação de Framework. Este modelo representa a instanciação dos tipos genéricos do Framework de Modelos reutilizando os componentes. A instanciação de cada tipo é representada por uma seta tracejada rotulada com o nome do tipo instanciado. A Figura 3.78 mostra, por exemplo, o Modelo de Aplicação do Framework de Modelos do padrão *CadastrarPessoa*. O Engenheiro de Software aplica este padrão visando construir um cadastro de clientes no sistema. O tipo genérico *<Pessoa>* foi substituído pelo componente *ClienteBean* e o tipo genérico *<Localizacao>* foi substituído pelos componentes *CidadeBean* e *EstadoBean*. O componente *Session CadastrarClienteBean* controla o cadastro de clientes.

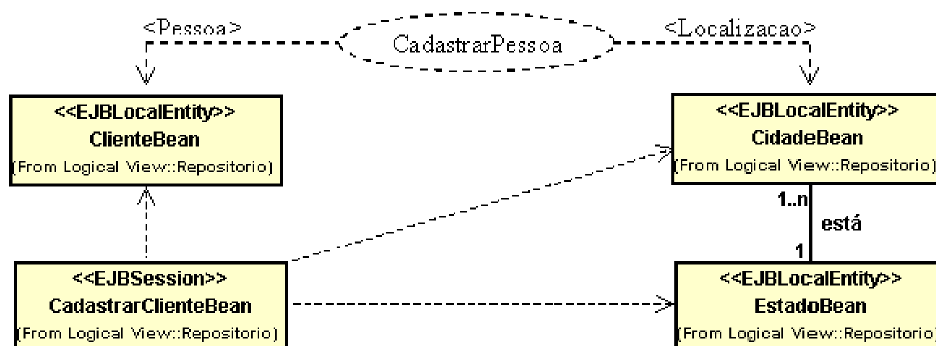


Figura 3.78 - Modelo de Aplicação de Framework para um cadastro de clientes

A Figura 3.79 mostra, por exemplo, uma aplicação dupla do padrão *NegociarProduto* da Figura 3.77, visando realizar a compra e venda de produtos. No lado esquerdo da Figura 3.79 observa-se a instanciação dos tipos genéricos visando a compra de produtos. Os tipos genéricos *<Produto>*, *<Item>*, *<Negociacao>*, *<Pessoa>* e *<Responsavel>* foram instanciados pelos componentes *Entity ProdutoBean*, *ItemCompraBean*, *CompraBean*, *FornecedorBean* e *FuncionarioBean*. O componente *Session ComprarProdutoBean* controla a compra de produtos. No lado direito da Figura 3.79 tem-se a instanciação dos tipos genéricos do padrão *NegociarProduto*, visando a venda de produtos. Os tipos genéricos *<Item>*, *<Negociacao>*, *<Pessoa>* e *<Responsavel>* foram instanciados pelos componentes *Entity ItemVendaBean*, *VendaBean*, *ClienteBean* e *VendedorBean*. O tipo genérico *<Produto>* é comum a ambos os casos. O componente *Session VenderProdutoBean* controla a venda de produtos.

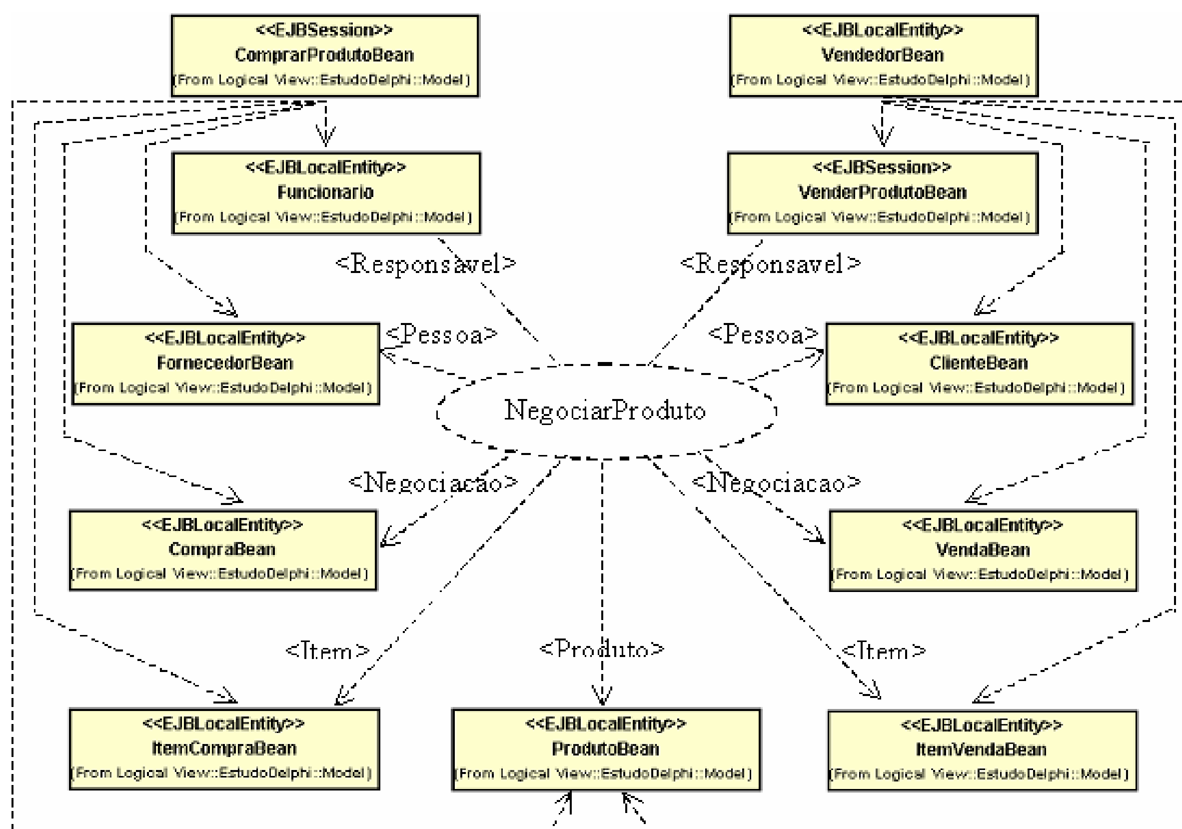


Figura 3.79 - Aplicação dupla do padrão *NegociarProduto*

Assim, pode-se obter as Especificações Orientadas a Componentes do sistema que está sendo reconstruído. Depois de reespecificado, pode-se reprojetar o sistema Orientado a Componentes, conforme apresentar a próxima seção.

3.4.2. Reprojetar Orientado a Componentes

Neste passo, preocupa-se com requisitos não-funcionais, como a distribuição física dos componentes e com detalhes da sua implementação como transação, persistência, linguagem de implementação e a plataforma onde o sistema será executado.

São reprojetados os Modelos de Classes reutilizando padrões e componentes, e também são refinados os Modelos de Componentes, que representam a arquitetura física dos componentes, com suas interfaces para conexão e dependências específicas da aplicação.

A Figura 3.80 mostra, por exemplo, um Modelo de Classes reprojetado reutilizando os componentes segundo a arquitetura MVC. Para a realização de uma venda, a classe de interface *FPedido* da camada *View* interage com o padrão *DelVenderProduto* do tipo *BusinessDelegate*, que utiliza o padrão *ServiceLocator* para localizar as interfaces *EJBHome* dos componentes transientes do tipo *EJBSession* envolvidos na transação. Os componentes transientes também utilizam o padrão *ServiceLocator* para localizar os componentes persistentes do tipo *EJBLocalEntity* através de suas interfaces *EJBHome*, responsáveis pelas transações com o banco de dados. Através ainda do *ServiceLocator,* pode-se interagir com

outros componentes aumentando o reuso e evitando a redundância de código. O padrão *DelVenderProduto* também pode utilizar os recursos do padrão *FastLaneReader,* representado pela classes *Fast*, que permite uma conexão direta com o banco de dados visando agilizar das consultas de dados, considerando que a arquitetura EJB pode tornar a execução da aplicação bastante lenta.

Figura 3.80 - Reprojeto Orientado a Componentes

Considerado os padrões, o Engenheiro de Software modifica o projeto interno dos

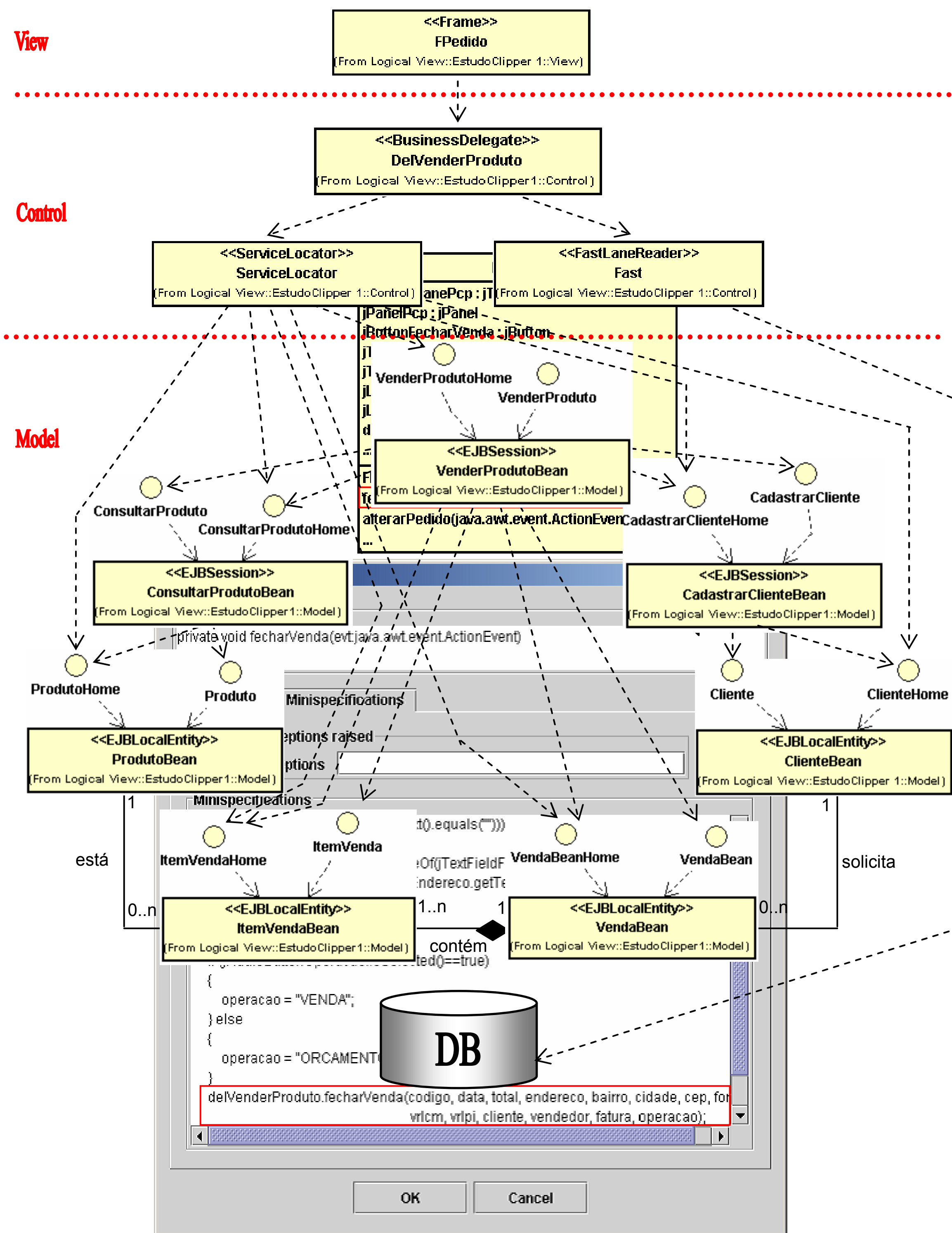


Figura 3.81 - Refinamento de um método no reprojeto

componentes do novo modelo de classes. A Figura 3.81 mostra um desses refinamentos realizados com o apoio da ferramenta MVCASE, onde as mini-especificações do corpo do método *jButtonFecharVenda* da classe *FPedido* foram reprojetadas para atender o padrão *BusinessDelegate*, instanciado pela classe *DelVenderProduto,* responsável pelas interações com os componentes *Session* envolvidos na transação. Dessa forma diminui-se a complexidade da utilização dos componentes nas classes de interface.

Outro modelo deste passos é o Modelo de Componentes. Este modelo mostra a reutilização dos componentes do domínio pelos componentes específicos do domínio da aplicação. Caso o Engenheiro de Software verifique a necessidade da criação de um novo componente que não existe na biblioteca de componentes, deve-se retornar à fase *Construir Componentes* para construí-lo e depois reutilizá-lo a partir da biblioteca. O Engenheiro de Software poderá também utilizar componentes de outros domínios disponíveis na biblioteca que tratam de requisitos não-funcionais, como por exemplo, persistência em banco de dados. Outros padrões como o *Singleton* [Dei03], *Command* [Gam95], *Memento* [Gam95], *Façade* [Gam95], *Observer* [Gam95], *CompositeView* [Sun04d], *Filter* [Sun04d] e *Iterator* [Gam95] podem ser utilizados neste passo, visando facilitar futuras manutenções.

O resultado final deste passo é o projeto Orientado a Componentes do sistema. Considerando que a modelagem é feita numa ferramenta CASE, o projeto fica descrito em uma linguagem de modelagem. No caso da MVCASE, a linguagem de modelagem é a MDL, que tem as mini-especificações dos corpos dos métodos descritas diretamente em Java. As descrições MDL do POC serão utilizadas na reimplementação do sistema, conforme se segue.

3.4.3. Reimplementar Orientado a Componentes

A reimplementação é realizada pela ferramenta MVCASE que gera o código automaticamente, a partir das descrições MDL do POC.

O Engenheiro de Software, orientado por técnicas de Desenvolvimento Baseado em Componentes, importa o projeto Orientado a Componentes do sistema na ferramenta MVCASE e faz a geração do código com base nas descrições MDL dos Modelos de Classes e dos Modelos de Componentes. Os recursos para a geração de código Java da ferramenta MVCASE tratam os métodos das classes, cujos comportamentos já estão especificados em Java e embutidos na descrição MDL. O código Java gerado a partir das especificações MDL é integrado com o código Java das mini-especificações dos corpos dos métodos descritas em Java, para obter a implementação final do sistema. A Figura 3.82 mostra, à esquerda, o código

MDL da classe *FPedido* com as mini-especificações dos corpos dos métodos em Java e, à direita, o código correspondente gerado pela ferramenta MVCASE, na linguagem Java. As descrições MDL "*object Class*" (1), "*object ClassAttribute*"(2) e "*object Operation*"(3) orientam a geração do código Java da classe, de seus atributos e assinaturas dos métodos, respectivamente. A descrição MDL "*semantics*"(4) contém as mini-especificações dos corpos dos métodos em Java que são transportados para o código final, obtendo assim uma primeira versão do sistema reimplementado.

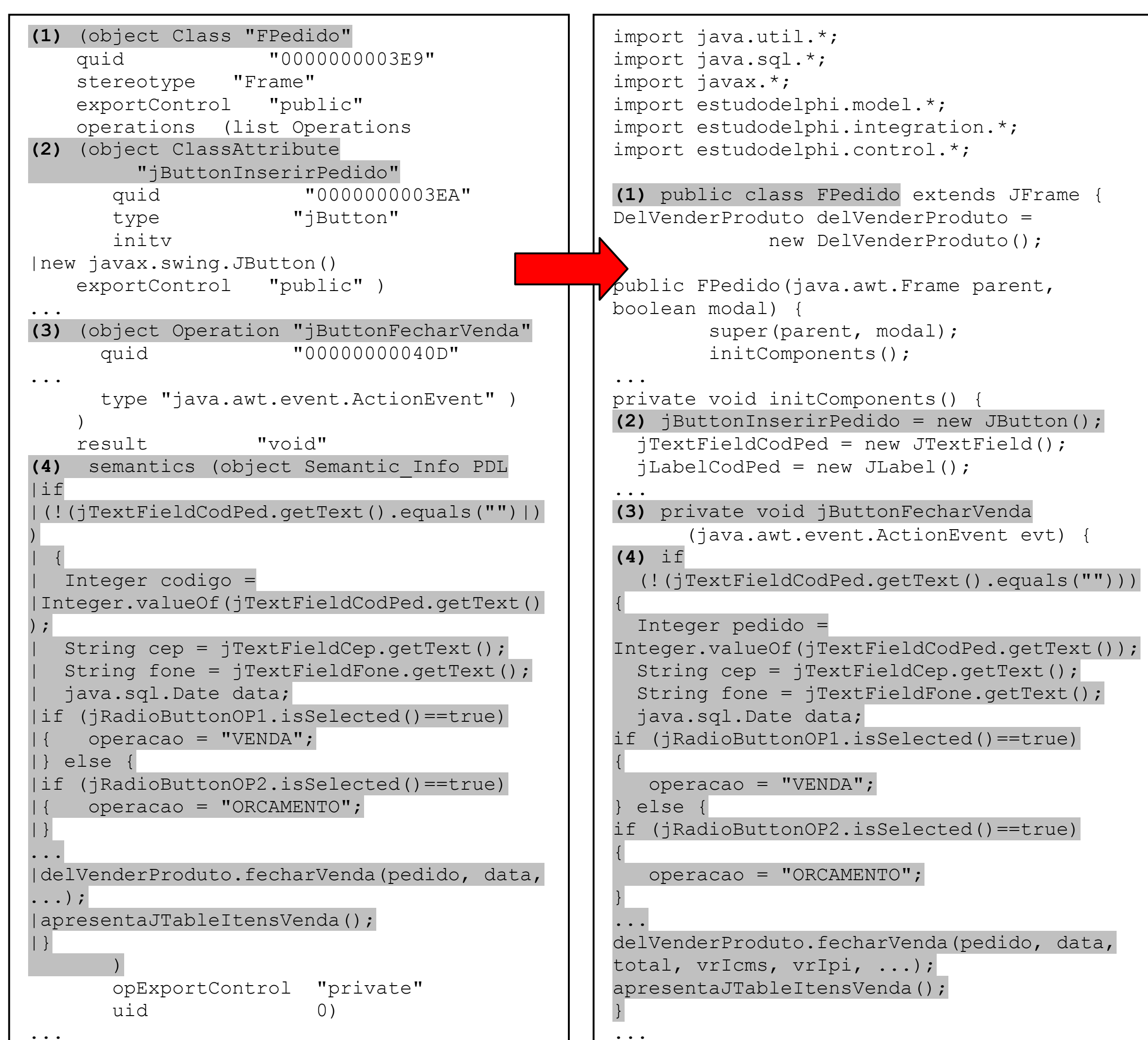


Figura 3.82 - Reimplementação Orientada a Componentes

Uma vez gerado o código, a MVCASE pode, opcionalmente, fazer o *deployment* dos componentes no servidor de aplicações para a execução. Na MVCASE pode-se utilizar os servidores J2EE [Sun04e] e JBoss [Jbo04].

Com o sistema reimplementado pode-se executá-lo e entende-lo melhor. Caso sejam identificados problemas, pode-se retornar aos passos anteriores, importando o Projeto Orientado a Componentes do sistema na ferramenta MVCASE para a correção dos erros e nova geração do seu código na linguagem de programação alvo da reengenharia. Desta forma, o Engenheiro de Software conclui a aplicação do RSCT no seu curso mais completo, abordando suas 04 (quatro) fases.

Capítulo 4

Avaliação

4.1. Introdução

Conforme Kitchenham [Kit95], um bom estudo de caso, apesar de não possuir o rigor dos experimentos formais, pode ser adequado para avaliar os benefícios e dificuldades advindas da utilização de um método ou processo em um determinado contexto.

Assim, com o propósito de testar a viabilidade da utilização do método RSCT na reengenharia de sistemas legados, optou-se por realizar um estudo de caso utilizando 03 (três) sistemas do domínio de Vendas. O sistema legado do primeiro estudo de caso, denominado *Informatiza*, é de propriedade da empresa Informatiza Informática, sediada na cidade de Lins/SP. Este sistema foi escrito em Clipper e tem cerca 45.000 linhas de código. O sistema do segundo estudo de caso, denominado *Caiçara*, também foi escrito em Clipper e tem aproximadamente 12.000 linhas de código. Este sistema é de propriedade da Distribuidora de Produtos Caiçara Ltda, sediada na cidade de Paranaguá/PR. O sistema do terceiro estudo de caso, denominado *Gnome,* é de propriedade da empresa Gnome Informática, sediada na cidade de São Paulo/SP. Este sistema foi escrito na linguagem Object Pascal (Delphi) e possui cerca de 74.000 linhas de código.

4.2. Abordagem

Para a realização deste estudo de caso, seguiu-se a sistemática apresentada por Kitchenham [Kit95], que sugere três fases distintas na elaboração de um estudo de caso.

- **Planejamento**: Consiste na definição do estudo de caso propriamente dito, identificando-se os participantes, as diretrizes e os objetivos a serem alcançados;

- **Monitoração**: Refere-se ao acompanhamento da execução de cada etapa do estudo, registrando-se o tempo de duração e os resultados obtidos;

- **Avaliação dos resultados**: Corresponde à descrição dos resultados gerais obtidos no estudo, utilizando os dados da fase de monitoração.

4.2.1 Planejamento

Um projeto piloto foi estabelecido para a aplicação do método RSCT, com o objetivo de testa-lo na reengenharia de 03 (três) sistemas legados do domínio de Vendas. Estabeleceu-se que todas as fases constantes do método seriam realizadas e, ao final, ter-se-ia uma avaliação, ainda que informal, das vantagens e desvantagens oferecidas com a utilização do método.

Como participante deste processo, contou-se com apenas um Engenheiro de Software, o qual possuía bons conhecimentos em reengenharia de sistemas, além de ano de experiência nas tecnologias mencionadas.

4.2.2 Monitoração

Durante a fase de monitoração, procurou-se registrar o tempo e os resultados obtidos de cada fase do método. Estes registros foram feitos com o objetivo de permitir a avaliação dos resultados deste estudo de caso.

Como resultados intermediários da aplicação do método RSCT nos 03 (três) sistemas, foram obtidos resultados considerados satisfatórios. A Tabela 4.1 resume os principais artefatos obtidos com a reengenharia destes sistemas, usando o método RSCT.

Descrição	Estudo de Caso		
	Informatiza	Caiçara	Gnome
Domínios construídos na Fase 1	3		-
Transformadores construídos na Fase 1	3		-
Casos de Uso obtidos na Fase 2	39	12	42
Classes obtidas na Fase 2	38	22	54
Componentes persistentes construídos na Fase 3	28	11	26
Componentes transientes construídos na Fase 3	39	13	42
Padrões de projeto identificados na Fase 3	05		

Tabela 4.1 – Artefatos obtidos com a reengenharia

O tempo total gasto para a aplicação de cada uma das fases do método RSCT na reengenharia dos 03 (três) sistemas legados do estudo de caso, também foi considerado satisfatório, conforme mostra a Tabela 4.2.

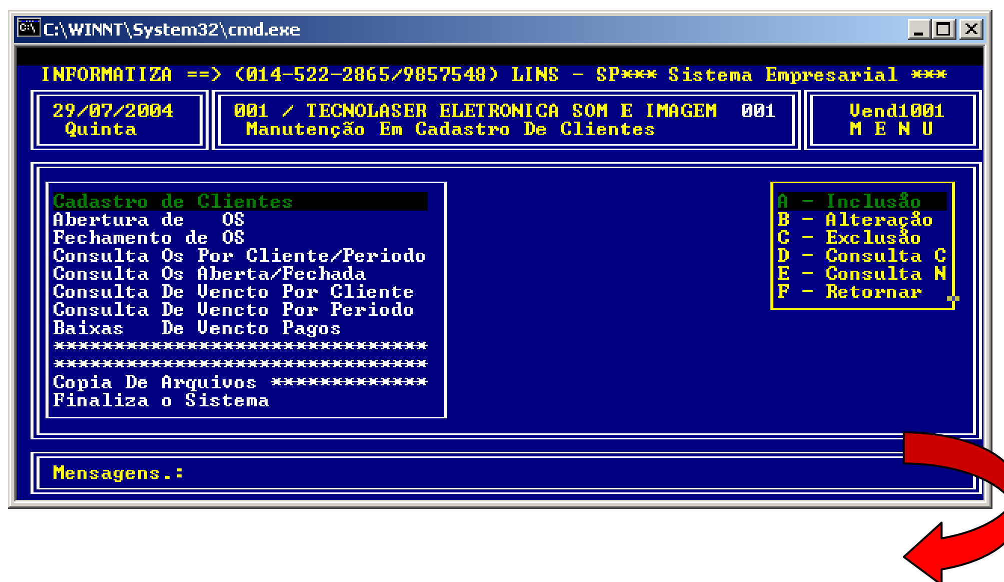
Fase	Descrição	Tempo Gasto
Fase 1	Construir Domínios e Transformadores	562 horas
Fase 2	Obter Projeto Orientado a Objetos	076 horas
Fase 3	Construir Componentes	223 horas
Fase 4	Reconstruir Sistemas	345 horas
	Tempo Total	1206 horas

Tabela 4.2 – Tempo gasto na aplicação das fases do RSCT

Como resultado final da aplicação do RSCT, obtiveram-se os sistemas reconstruídos utilizando padrões e componentes de software, e reimplementados com novas tecnologias.

Os três sistemas foram reimplementados com a tecnologia EJB e rodam no servidor de aplicações JBoss [Jbo04] com o banco de dados MySQL [Mys04]. Utilizou-se as tecnologias Java Servlets [Sun04a], JSP [Sun04a] e a biblioteca javax.swing da linguagem Java para a reimplementação das interfaces dos sistemas *Informatiza*, *Caiçara* e *Gnome*, respectivamente.

A Figura 4.1 mostra, por exemplo, uma das interfaces do sistema *Informatiza* antes e depois da reengenharia. Na parte superior da figura tem-se a antiga interface e na parte inferior tem-se a interface atual do *Informatiza*.



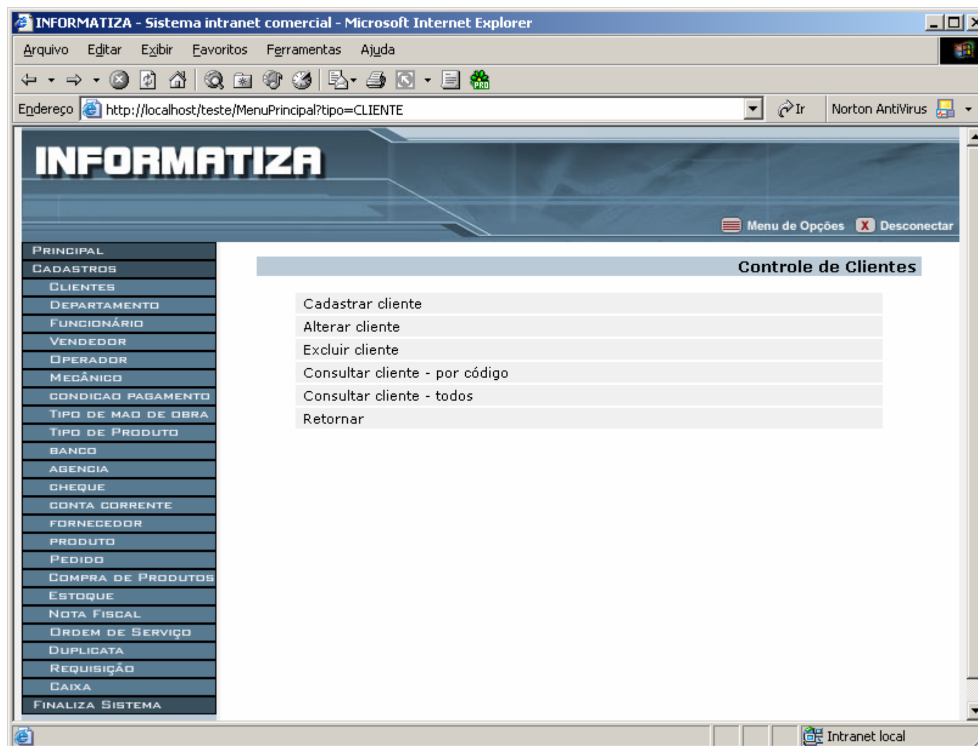
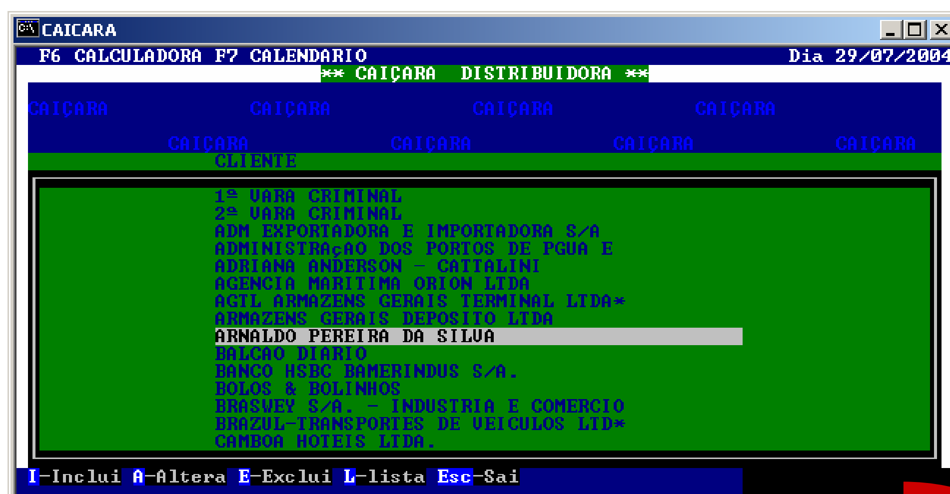


Figura 4.1 - Antiga e nova interface do sistema *Informatiza*

A Figura 4.2 mostra, por exemplo, uma das interfaces do sistema *Caiçara* antes e depois da reengenharia. Na parte superior da figura tem-se a antiga interface e na parte inferior tem-se a interface atual do *Caiçara*.



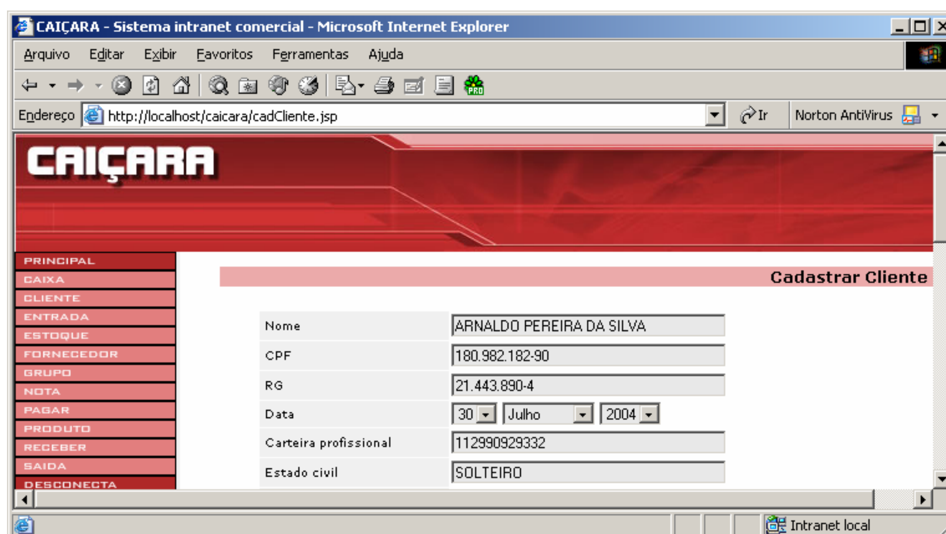


Figura 4.2 - Antiga e nova interface do sistema *Caiçara*

A Figura 4.3 mostra a reconstrução de uma das interfaces do sistema *Gnome*. Na parte superior da figura tem-se a antiga interface e na parte inferior tem-se a sua interface atual.

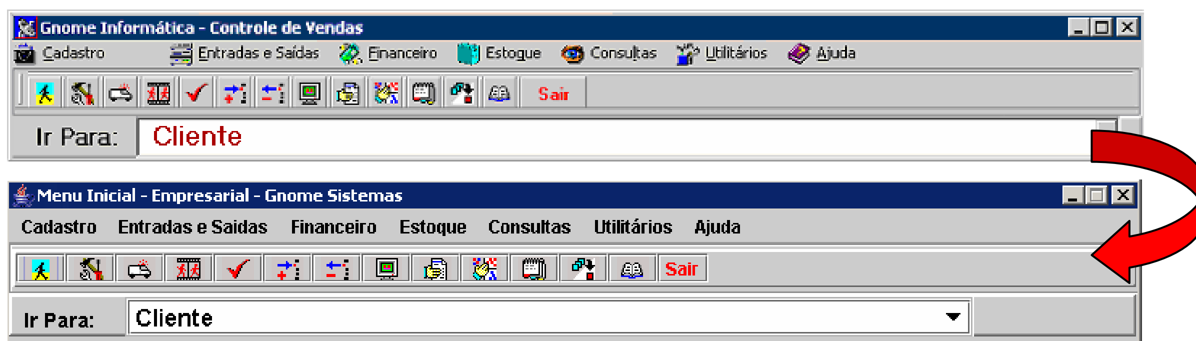


Figura 4.3 - Antiga e nova interface do sistema *Gnome*

4.2.3 Analise dos Resultados

Os resultados obtidos mostraram a viabilidade do RSCT na reengenharia de sistemas legados. Foi possível recuperar grande parte da documentação dos sistemas legados por meio da aplicação da Engenharia Reversa na Fase 2, que pode ser refinada com mais detalhes aumentando sua consistência com o sistema reconstruído.

Os componentes construídos estão disponíveis em uma biblioteca para reuso também no desenvolvimento de novos sistemas. Acredita-se que a reengenharia de novos sistemas do mesmo domínio poderá contribuir para refinar e melhorar ainda mais os componentes construídos.

Além disso, verificou-se que o método RSCT permite separar, sistematicamente, os aspectos de negócio e persistência, evitando o entrelaçamento de código com diferentes propósitos.

Observou-se também que o tempo gasto para desenvolver uma aplicação reutilizando estes componentes é significativamente menor do que se ela fosse desenvolvida sem os componentes, já que não há a necessidade de construir os mesmos componentes diversas vezes.

Outra importante observação é que, após a reengenharia ter sido realizada em diferentes sistemas, um mesmo componente pode vir a sofrer subseqüentes alterações, fazendo com que as aplicações previamente construídas não mais sejam compatíveis com as novas versões dos componentes. Sendo assim, é importante que se realize um controle das diferentes versões de um mesmo componente, de modo que o processo de manutenção e evolução das aplicações possa ser controlado de forma eficiente.

Através da reengenharia foi possível identificar uma primeira versão de alguns padrões no domínio de Vendas. Contudo, a construção de um padrão visando torná-lo genérico o suficiente para atender todas as aplicações do domínio do problema é muito complexa, sendo necessário um bom conhecimento do domínio da aplicação.

O uso de padrões pode aumentar consideravelmente o número de componentes de um sistema, porém aumenta também o nível de organização do sistema e reduz a quantidade de código, facilitando a manutenção.

Além dos aspectos positivos mencionados acima, alguns aspectos negativos também foram identificados. As gramáticas das linguagens dos domínios utilizados no Sistema de Transformação Draco-PUC normalmente não são de domínio público, o que induz o Engenheiro de Software a desenvolve-las a partir de programas descritos na linguagem em questão e da documentação disponível. Desta forma, não se pode garantir que o resultado da Fase 1 (Construir Domínios e Transformadores) do método RSCT esteja correto.

Observou-se também uma grande dificuldade de se mapear todas as regras semânticas de um domínio para outro, durante a construção dos transformadores. Esta operação requer um alto nível de detalhamento dos transformadores e um grande conhecimento dos domínios em questão.

Por fim, é importante ressaltar que o modo como este método foi inicialmente definido, não considera a possibilidade de utilização de componentes adquiridos de terceiros, o que limita consideravelmente a biblioteca de componentes do RSCT.

O capítulo seguinte apresenta uma conclusão sobre este trabalho, suas principais contribuições e algumas perspectivas de trabalhos futuros.

Capítulo 5

Conclusão

No contexto da Reengenharia de Software, o reuso merece destaque nos diferentes níveis de abstração. Além da diminuição da redundância de código, o reuso garante a redução de esforços e custos na reconstrução de sistemas. Componentes de software podem ser reutilizados, visando eliminar a redundância de código e oferecer maior segurança, tendo em vista seu acesso ser disponibilizado por meio de suas interfaces.

Este projeto apresentou um método de Reengenharia de Software Orientado a Componentes usando Transformações denominado método RSCT, que se apóia no reuso dos artefatos produzidos durante suas quatro fases. O RSCT estende a capacidade do método RST, visando tratar a construção e reuso dos componentes na Reengenharia de Software. A extensão proposta visa tratar a construção e o reuso de componentes obtidos com a reengenharia de sistemas legados. Os componentes construídos ficam disponíveis em uma biblioteca para o reuso tanto na reconstrução de sistemas antigos, como na construção de novos sistemas. O refinamento dos componentes da biblioteca, considerando vários sistemas de um mesmo domínio, sugere uma primeira versão de padrões de projeto do domínio da aplicação, que facilitam ainda mais o reuso. O uso destes padrões implica em aumento de produtividade, maior uniformidade na estrutura do software e redução da complexidade, pois blocos construtivos podem ser usados na elaboração de sistemas maiores.

Com a utilização de ferramentas, procura-se automatizar grande parte do processo da Reengenharia. O Sistema de Transformação Draco-PUC, por ser orientado a domínio, possibilita trabalhar com diferentes domínios, como, por exemplo, C/C++, Progress, Dataflex, Visual Dataflex, Clipper, Cobol, Java e Object Pascal (Delphi), dentre outros. A ferramenta MVCASE permite que o Engenheiro de Software realize alterações no projeto em um alto nível de abstração e posteriormente reimplemente-o, de forma semi-automática, em uma linguagem de programação baseada em componentes.

A automatização de grande parte das tarefas do Engenheiro de Software possibilita a redução do tempo e custos de reconstrução de um sistema, através do reuso nos diferentes níveis de abstração, desde os requisitos, análise e projeto, até a implementação.

5.1. Principais Contribuições

A principal contribuição é o método RSCT, que cobre grande parte do ciclo da Reengenharia de um Software. O método, numa primeira etapa, parte de sistemas não Orientados a Objetos para obter componentes do domínio do problema. Numa segunda etapa, realiza-se a reconstrução dos sistemas reusando os componentes. Outras contribuições deste trabalho são:

- A aplicação de Padrões de Projeto nos componentes de software construídos, como o *Business Delegate*, *Fast Lane Reader*, *Service Locator* e *Value Object* [Sun04d], facilitando seu reuso e permitindo melhor organização na estrutura do código;
- A identificação de uma primeira versão de alguns Padrões de Projeto para o domínio de Vendas, que podem ser aperfeiçoados visando atender um maior número de aplicações deste domínio. Embora embrionário o processo usado na identificação dos padrões, pode-se adapta-lo para outros domínios de aplicações.
- A integração e adição de novos recursos na ferramenta MVCASE para automatizar o reuso dos padrões *Business Delgate* [Sun04d] e *ValueObject* [Sun04d] integrados no RSCT. Essas alterações foram necessárias para que a MVCASE pudesse apoiar o processo do RSCT.

5.2. Trabalhos Futuros

Com base nos resultados obtidos, destacam-se algumas idéias que visam dar continuidade a este trabalho:

- A utilização de outras tecnologias, como por exemplo, *CORBA* e *COM+,* visando aumentar o leque de consumidores dos artefatos produzidos;
- O aperfeiçoamento do RSCT para suportar a criação de um *framework* de padrões de um domínio de aplicação, por meio da reengenharia de vários sistemas desse domínio. Após o reconhecimento dos padrões, seu reuso em futuros desenvolvimentos poderia ser mais encorajado se eles ficassem disponíveis na biblioteca, englobando artefatos pré-programados em diversas linguagens, que poderiam ser usados facilmente, instanciando-se o padrão para a aplicação específica;
- Suporte para o controle de versões, permitindo que o gerenciamento das diferentes versões dos componentes construídos facilite o processo de manutenção e evolução das aplicações. O DC/UFSCar já possui trabalhos em andamento [Cun04] visando esse suporte; e
- Por fim, sugere-se a realização de novos estudos de caso visando melhorar o RSCT, aumentando sua confiabilidade.

Capítulo 6

Referências Bibliográficas

[Alm02] Almeida, E. S, Lucrédio, D.; Bianchini, C. P.; Prado, A F.; Trevelin, L.C. **Ferramenta MVCASE – Uma Ferramenta Integradora de Tecnologias para o Desenvolvimento de Componentes Distribuídos.** In: SBES–Sessão de Ferramentas, Pág. 432 – 437. ISBN: 85-88442-31-0, 2002, Gramado/RS – Brasil.

[Alm03] Almeida, E. S. **Uma Abordagem para o Desenvolvimento de Software Baseado em Componentes Distribuídos** .Dissertação de Mestrado, UFSCar – Universidade Federal de São Carlos, 2003.

[Ant95] Antlr: **A Predicated – LL(k)** ***parser*** **Generator** – Software—Practice and Experience, 25(7):789-810, 1995.

[Ara88] Arango, G. **Domain Engineering For Software Reuse**. Technical Report UCI-ICS 88-27. Universidade da Califórnia, 1988.

[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.

[Bec87] Beck, Kent; Cunningham, Ward. **Using Pattern Languages for Object-Oriented Programs**, Technical Report nº CR-87-43, disponível em: url: http://c2.com/doc/oopsla87.html, 1987.

[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 – Pág 1-66. The ACM Press in co-operation with Addison-Wesley, 1989.

[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, Pág 143-151, 1989.

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

[Blo97] Blomi J. – **Metamorphosen Datumsfeldes** – Micro Focus GmbH, München, Germany,url: http://www.microfocus.de/ , 1997.

[Bos03] Bossonaro, Adriano A, Moraes, João L. C., Fontanette, Valdirene, Garcia, Vinicius C., Prado, Antonio F. **Implementações de Frameworks de Componentes, dirigidas por Modelos do Método Catalysis.** Fourth Congress of Logic Applied to Technology (LAPTEC'2003), Marília-SP, 2003.

[Bra97] Brand, M. G. J. VanDen, Sellink M. P. A., and Verhoef C. – **Control Flow Normalization for COBOL/CICS Legacy Systems**. – Technical Report P9714, University of Amsterdam, Programming Research Group, url: http://adam.wins.uva.nl/~x/cfn/cfn.html , 1997.

[Bra97a] Brand, M. G. J. VanDen, Klint P., and Verhoef C. – **Re-Engineering Needs Generic Programming Language Technology**. ACM SIGPLAN Notices, 32(2):54-61, 1997.

[Brg98] Braga, Rosana T. V. **Padrões de Software a partir da Engenharia Reversa de Sistemas Legados**. Dissertação de Mestrado, USP, São Carlos-SP,1998.

[Brg99] Braga, Rosana T. V. and Masiero, Paulo. **Legacy Systems Reengineering Using Software Patterns**, Proceedings of the XIX International Conference of the Chilean Computer Science Society (SCCC'99), IEEE Computer Society Press, p. 160-169, Talca, Chile, November 1999.

[Bro96] Brown, A. W. and K. C. Wallnau, **Engineering of Component Based Systems**, Component-Based Software Engineering, IEEE Computer Society Press, 1996.

[Bro98] Brown, A., Wallnau, K. **The Current State of CBSE**, IEEE Software, Oct 1998.

[Bus96] Buschmann, F. et al. **A System of Patterns**, John Wiley & Sons, 1996.

[Cat01] **Catalysis Enterprise Components with UML**. url: http://www.catalysis.org., acessado em 10/06/2004.

[Chi90] Chikofsky, Elliot J.; Cross, James H. **Reverse Engineering and Design Recovery: a Taxonomy**. IEEE Software, Pág. 13-17, 1990.

[Cor91] Cordy, J. R., Halpern, Hamu C. D., and 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. 1993.

[Cun04] Cunha, J. R. D. D. **Uma Ferramenta de Apoio ao Processo de Gerenciamento de Configuração de Software.** Dissertação de mestrado (em andamento). Universidade Federal de São Carlos, SP, 2004.

[Dei03] Deitel, H. M., Deitel, J. P. **Java – Como Programar** – 4ª ed. Bookman, 2003.

[Del04] **Borland Software Corporation – Delphi**. url: http://www.borland.com/delphi/ acessado em 28/04/2004.

[Faq98] FAQs – **RescueWare**. url: http://www.relativity.com/

[Fon02a] Fontanette, V., Garcia, V.C., Perez, A B., Bossonaro, Adriano A, Prado, A F. **Estratégia de Reengenharia de Software Baseada em Componentes Distribuídos.** 2º Workshop de Desenvolvimento Baseado em Componentes,

Sessão Técnica (5), Itaipava – RJ, 2002.

[Fon02b] Fontanette, V., Garcia, V. C., Perez, Angela B., Bossonaro, Adriano A, Prado, Antonio F. **Reengenharia de Software usando Transformações (RST).** In: The Second Ibero-American Symposium on Software Engineering and Knowledge Engineering (JIISIC/2002), Sessão Técnica (4), Artigo nº3, Salvador – BA, 2002.

[Fon02c] Fontanette, V., Garcia, Vinicius C., Perez, Angela B., Bossonaro, Adriano A, Prado, Antonio F. **Reengenharia de Sistemas Legados Baseada em Componentes usando Transformações.** II Workshop Chileno de Ingeniería de Software, Copiapó-Chile, 2002.

[Fon02d] Fontanette, V., Garcia, V. C., Perez, A B., Bossonaro, Adriano A, Prado, A F. **Reprojeto de Sistemas Legados Baseado em Componentes de Software** In: XXVIII Conferência Latino-Americana de Informática (InfoUYclei), Montevidéu,Uruguai. Anais. Montevidéu: Mastergraf SRL, 2002. 177p. CL89. ISBN: 9974-7704-1-6, 2002.

[Fon04] Fontanette, V., Garcia, V. C., Perez, A B., Prado, Antonio F., Sant'Anna, M.; RHAE/CNPQ – **Projeto: Reengenharia de Software Usando Transformações (RST)**. Processo número: 610.069/01-2, UFSCar – Universidade Federal de São Carlos, 2004.

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

[Gal95] Gall, Harald C., Klösh, René R.; Mittermeir, Roland T. **Architectural Transformation of Legacy Systems**. International Conference on Software Engineering, 11, April 1995.

[Gam95] Gamma, E. Et al. **Design Patterns: Elements of Reusable Object-Oriented Software**. Ed. Addison-Wesley. USA.1995.

[Gar02] Garcia, V. C., Perez, A B, Fontanette, V., Bossonaro, A A, Prado, A F. **DDE – Draco Domain Editor.** In: Simpósio Brasileiro de Engenharia de Software (SBES) – Sessão de Ferramentas, Pág 378 – 383. ISBN: 85-88442-31-0, Gramado-RS, 2002.

[Gri98] Griss, M., Favaro, J., D'Alessandro, M. **Integrating Feature Modeling with RSEB**, 5th International Conference on Software Reuse (ICSR-5), ACM/IEEE, Vitoria – Canada, 1998.

[Hal96] Hall B. – **Year 2000 Tools and Services**. – Symposium/Itxpo, The IT revolution continues: managing diversity in the 21st century, Pág 14. 1996.

[Hee89] Heering J., Hendriks P. R. H., Klint P., Rekers J. – **The Syntax Definition Formalism SDF** – Reference Manual. Sigplan Notices, 24(11):43-75, 1989.

[Hei01] Heineman, G., T., Council, W., T. **Component Based Software Engineering: Putting the Pieces Together**, Addison-Wesley. 2001.

[Ics02] 5th ICSE Workshop on Component-Based Software Engineering: **Benchmarks for Predictable Assembly**, In conjunction with the 24th International Conference on Software Engineering (ICSE), May 2002.

[Ics96] **ICSR4 Tutorial: Transformation Systems**, url: http://vtopus.cs.vt.edu/~edwards/icsr5/icsr4/tut-baxter.html , 1996.

[Jac91] Jacobson, I., Lindstrom, F. **Re-engineering of Old Systems to NA Object-Oriented Architecture**. Object-Oriented Programming Systems, Languages, and Applications – OOPSLA'91. ACM Press. In Proceedings, pp.340-350. 1991.

[Jbo04] **JBoss Application Server**, url: http://www.jboss.org/index.html , em 02/07/2004

[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, 1999.

[Kan90] Kang, K., Cohen, S., Hess, J. et al, **Feature-Oriented Domain Analysis (FODA)** – Feasibility Study, SEI Technical Report CMU/SEI-90-TR-21, 1990.

[Kan98] Kang, K., Kim, S., Lee, J. et al, **FORM: A Feature-Oriented Reuse Method with Domain-Specific Reference Architectures**, SEI Technical Report, 1998.

[Kit95] Kitchenham, B., Pickard, P., Pfleeger, S. L. **Case Estudies for Method and Tool Evaluation**, IEEE Software, vol. 11, no. 4, pp. 52-62, 1995.

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

[Kru99] Krutchen, H.. **Modeling Component systems with the Unified Modeling Language**, International Workshop on Component-Based Software Engineering, Los Angeles – EUA, 1999.

[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, Pág. 94-100. Rio de Janeiro-RJ, 1994.

[Lei96] Leite, J.C.S., Sant'anna, M., Prado, A.F. **Porting COBOL Programs Using a Transformacional Approach**. Journal of Software Maintenance: Reseach and Practice, vol 9, 3-31 , Out 1996 John Wiley&Sons Ltd.

[Luc00] Lucrédio, D; Prado, Antonio F. **MVCASE: Ferramenta CASE Orientada a Objetos** – Sessão de Ferramentas do XIV SBES'2000. Pág 363 – 365. CDU 681.31:519.683.2. João Pessoa-PB, Brasil. 4 – 6 de Outubro, 2000.

[Luc01] Lucrédio, D., Prado, A F. **Ferramenta MVCASE – Estágio Atual: Especificação, Projeto e Construção de Componentes.** Sessão de Ferramentas – XV SBES, Rio de Janeiro-RJ, Pág. 368-373, 2001.

[Luc03] Lucrédio, Daniel. **MVCASE**. url: http://www.recope.dc.ufscar.br/mvcase/ , 2003.

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

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

[Mor02a] Moraes, J. L. Cardoso**. Reutilização de Componentes do Framework do Domínio de Cardiologia (FrameCardio)** – XVI Simpósio Brasileiro de Engenharia de Software, 2002.

[Mor02b] Moraes, J. L. C., Bossonaro, Adriano A, Prado, Antonio F. do. **Desenvolvimento de um Framework, Baseado em Componentes, do Domínio de Cardiologia.** VIII Congresso Brasileiro de Informática em Saúde, Natal-RN, 2002.

[Mor04] Moraes, João L. C., Bossonaro, Adriano A, Garcia, Vinicius C., Fontanette, Valdirene, Lucrédio, Daniel, Prado, Antonio Francisco do. **An Approach For Construction and Reuse of Software Components Frameworks Implemented In Delphi.** ISSADS 2004 – Third IEEE International Symposium and School on Advance Distributed Systems, Guadalajara, México, 2004.

[Mys04] **MySQL Database Server**, url: http://www.mysql.org , acessado em 02/07/2004.

[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.

[Nov02] Novais, E. R. A **Reengenharia de Software Orientada a Componentes Distribuídos**, - Dissertação de Mestrado, Ciências da Computação, UFSCar – Universidade Federal de São Carlos, 2002.

[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.

[Nog02] Nogueira, A R. **Transformação de DataFlex Procedural para Visual DataFlex Orientado a Objetos reusando um Framework.** São Carlos-SP, Dissertação de Mestrado. UFScar-Universidade Federal de São Carlos, 2002.

[Omg03] Object Management Group. **Unified Modeling Language (UML)**.
url: http://www.omg.org/cgi-bin/doc?formal/02-06-65 , acessado em 03/12/2003.

[Omg04] Object Management Group. **Corba Component Model (CCM) 3.0**. url: http://www.omg.org/cgi-bin/doc?formal/02-06-65 , 03/05/2004.

[Omm02] Ommering, R., V. **Building Product Populations with Software Components**, In 24th International Conference on Software Engineering (ICSE). Orlando-Flórida. 2002.

[Par91] Parr T. J., Dietz H. G., Cohen W. E. – **PCCTS Reference Manual**, 1ª ed., 1991.

[Pen96] Penteado, R. D. **Um Método para Engenharia Reversa Orientada a Objetos**. Tese de Doutorado. USP, São Carlos-SP, 1996.

[Pra92] Prado, A F. **Estratégia de Engenharia de Software Orientada a Domínios.**. Tese de Doutorado. Pontifica Universidade Católica, Rio de Janeiro, 1992.

[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. Pág. 383-394, 19-23 de Outubro, 1998.

[Pre01] Pressman, Roger S. **Engenharia de Software** – 5 Ed – Rio de Janeiro: McGraw-Hill, 2001.

[Pri90] Prieto-Diaz, R., Arango, G. **Domain Analysis and Software System Modeling**. IEEE Computer Society, Press Tutorial, 1990.

[Rat97] **Rational Software Corporation et al. UML Metamodel**, version 1.1, September 1997. http://www.rational.com/uml

[Rat04] **Rational Software Corporation et al.** url: http://www.rational.com , 2004.

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

[Rel00] **Relativity Technologies**, http://www.relativity.com/ , 2000.

[Ros77] Ross, Douglas T., **Structured Analysis (SA): A Language for Communicating Ideas**, IEEE Transaction on Software Engineering, 1977.

[Sam97] Sametinger, J., **Software Engineering with Reusable Components**, Springer, 1997.

[San93] Sant'anna, M. Lavoisier: **Uma Abordagem Prática do Paradigma Transformacional.** Monografia de Graduação. Rio de Janeiro. PUC-Rio – Pontifica Universidade Católica do Rio de Janeiro, 1993.

[San99] Sant´anna, M. **Circuitos Transformacionais**. Tese de Doutorado – Departamento de Informática, PUC, Rio de Janeiro-RJ, 1999.

[Sei99a] Software Engineering Institute (SEI). **Domain Analysis**. Disponível site Software Engineering Practices, 1999, url: Iwww.sei.cmu.edu/domain-

engineering/domain_anal.html , consultado em 15/04/2003.

[Sei99b] Software Engineering Institute (SEI). **Domain Design**. Disponível site Software Engineering Practices, 1999, url: Iwww.sei.cmu.edu/domain-engineering/domain_design.html. consultado em 15/04/2003.

[Sib97] Siber Systems Inc. – **CobolTransformer—Peek Under the Hood**: Technical White Paper, url: http://www.siber.com/ , 1997.

[Sim96] Simos, M. **Organization Domain Modeling (ODM): Domain Engineering as a Co-Methodology to Object-Oriented Techniques**. Fusion Newsletter, v.4, Hewlett-Packard Laboratories, Pág 13-16, 1996.

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

[Sob02] Sobral, Diogo F. ***Framework* para Comércio Eletrônico, via Internet Móvel, mediado por Agentes de Software.** Dissertação de Mestrado, UFSCar, 2002.

[Som95] Sommerville, I. **Software Engineering. Fifth Edition**. Addison-Wesley, 1995.

[Sou99] D'Souza, D., F., Wills, A., C., 1999. **Objects, Components, and Frameworks with UML, The Catalysis Approach**, Addison-Wesley. USA.

[Str04] **Struts** – Apache Software Foundation URL: http://struts.apache.org , acessado em 18/03/2004.

[Sun04a] **Java Technologies**. Sun Microsystems, url: http://java.sun.com/products/ , acessado em 17/04/2004.

[Sun04b] **Enterprise JavaBeans (EJB).** Sun Microsystems, url: http://java.sun.com/products/ejb/index.html **,** acessado em 16/05/2004.

[Sun04c] **Model View Controller Architecture.** Sun Microsystems, url: http://java.sun.com/blueprints/patterns/MVC-detailed.html, em 24/05/2004.

[Sun04d] **Core J2EE Patterns: Best Practices and Design Strategies**. Sun Microsystems, url: http://java.sun.com/blueprints/corej2eepatterns/index.html , em 06/06/2004.

[Sun04e] **Java 2 Platform, Enterprise Edition (J2EE)**. Sun Microsystems, url: http://java.sun.com/j2ee , em 06/06/2004.

[Szy98] Szyperski, C. **Component Software - Beyond Object-Oriented Programming**. Addison-Welsley and ACM Press, 1998.

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

[Til93] Tillman, G., **A Practical Guide to Logical Data Modeling**, McGraw-Hill, 1993.

[Wei01] Weinreich, R., Sametinger, J. **Component Models and Component Services:**

Concepts and Principles, in Component-Based Software Engineering: Putting the Pieces Together, Addison-Wesley, 2001

[Wer00] Werner, C., M., L., Braga, R., M. **Desenvolvimento Baseado em Componentes**, XIV Simpósio Brasileiro de Engenharia de Software, Minicursos e Tutoriais, João Pessoa-PB, 2000.

[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.

[Xml03] **Extensible Markap Language - XML**. url: http://www.xml.org , acessado em 26/03/2003.

Anexo 1 - Contribuições Acadêmicas

- Fontanette, V., Garcia, V. C., Perez, A B., Bossonaro, A A, Prado, A F. **Estratégia de Reengenharia de Software Baseada em Componentes Distribuídos.** 2º Workshop de Desenvolvimento Baseado em Componentes (WDBC'2002), Itaipava/RJ, 2002, Sessão Técnica (5), Artigo Nr 12.

- Garcia, V. C., Peres, A B., Fontanette, V., Bossonaro, A A, Prado, A F. **DDE – Draco Domain Editor.** Simpósio Brasileiro de Engenharia de Software (SBES), Gramado/RS, 2002, Sessão de Ferramentas, Anais Pág 378 – 383. ISBN: 85-88442-31-0.

- Fontanette, V., Garcia, V. C., Perez, A B., Bossonaro, A A, Prado, A F. **Reengenharia de Software usando Transformações (RST).** The Second Ibero-American Symposium on Software Engineering and Knowledge Engineering (JIISIC), Salvador/BA, 2002, Sessão Técnica(4), Artigo Nr03.

- Moraes, J. L. C., Bossonaro, A A, Prado, A F. **Desenvolvimento de um Framework baseado em componentes, do domínio de Cardiologia.** VIII Congresso Brasileiro de Informática em Saúde, Natal/RN, 2002, Anais Pág. 1 – 11.

- Fontanette, V., Garcia, V. C., Perez, A B., Bossonaro, A A, Prado, A F. **Reengenharia de Sistemas Legados Baseada em Componentes usando Transformações.** II Workshop Chileno de Ingeniería de Software (JCC), Copiapó/Chile, 2002, Anais Pág. 1-10.

- Fontanette, V., Garcia, V. C., Perez, A B., Bossonaro, A A, Prado, A F. **Reprojeto de Sistemas Legados Baseado em Componentes de Software.** XXVIII Conferencia Latinoamericana de Informática (InfoUYclei), Montevideo/Uruguai, 2002, Anais. Pág 177, ISBN: 9974-7704-1-6.

- Bossonaro, A A, Moraes, J. L. C., Fontanette, V., Garcia, V. C., Prado, A F. **Implementações de Frameworks de Componentes, dirigidas por Modelos do Método Catalysis.** The Fourth Congress of Logic Applied to Technology (LAPTEC), Marília/SP, 2003.

- Moraes, J. L. C., Bossonaro, A. A. , Garcia, V. C., Fontanette, V., Lucrédio, D., Prado, A. F. **An Approach for Construction and Reuse of Software Components *Frameworks***

Implemented in *Delphi.* Third IEEE International Symposium and School on Advance Distributed Systems(ISSADS), Guadalajara/México, 2004.

- Neto, R. M. S., Lucrédio, D., Bossonaro, A A, Cunha, J. R. D. D., Catarino, I. C. S., Souza, A M., Prado A F. **Component-Based Software Development Environment.** 6th International Conference on Enterprise Information Systems (ICEIS), Porto/Portugal, 2004.