**Padrões de Software a partir da Engenharia
Reversa de Sistemas Legados**

Rosana Teresinha Vaccare Braga

Orientador: Prof. Dr. Paulo Cesar Masiero

Dissertação apresentada ao Instituto de Ciências Matemáticas e de
Computação - USP, como parte dos requisitos para a obtenção do
título de Mestre em Ciências - Área de Ciências de Computação e
Matemática Computacional

USP - SÃO CARLOS
Novembro de 1998

Dedico este trabalho

a Deus

ao Cirilo, ao Thales e ao Vitor

Agradecimentos

Ao Prof. Dr. Paulo Cesar Masiero pela orientação que me deu, sempre com muita atenção e entusiasmo. Pelo direcionamento que deu à pesquisa, tão essencial para aproveitar ao máximo as minhas potencialidades. Pela oportunidade de voltar à pesquisa e poder associá-la à prática profissional.

Aos professores do ICMC pela acolhida calorosa depois de dez anos de afastamento da Universidade.

Aos Profs. Drs. Norm Kerth e Ralph Johnson pela recepção generosa que me deram na área de pesquisa em padrões de software.

À Prof. Dra. Rosângela Dellosso Penteado pela total disponibilidade na execução de trabalhos conjuntos e por sua amizade e atenção.

Ao Prof. Dr. Antonio Francisco do Prado pela oportunidade de ter contato com a máquina Draco-Puc.

Às alunas de graduação Fabiana Rocha, do ICMC e Tiemi Sakata, da UFSCAR, pelos trabalhos de iniciação científica que contribuíram para as publicações feitas durante a execução deste trabalho.

Aos mestrandos, doutorandos, pesquisadores e bolsistas do Laboratório de Engenharia de Software do ICMC pela amizade, atenção e auxílio na utilização de máquinas e programas.

Aos colegas da Universidade de Franca pelo apoio e incentivo durante todo o programa de mestrado.

Aos colegas do Senac – São Carlos pelo incentivo que me deram para que eu pudesse contribuir efetivamente na atividade de treinamento profissional em que estão empenhados e que são tão importantes para o desenvolvimento do país.

Às secretárias da pós-graduação pela atenção, paciência e disponibilidade com que sempre me atenderam.

Às secretárias, bibliotecárias e funcionários do ICMC que se mostraram sempre prestativos no meu atendimento.

À Fapesp pela bolsa de iniciação científica concedida durante meu curso de graduação que me deu oportunidade de travar os primeiros contatos com a pesquisa e que abriram caminho para as atividades aqui relatadas.

Às empresas que me deram a oportunidade de exercer as minhas atividades profissionais na área de consultoria e desenvolvimento de sistemas.

Aos meus pais Miguel e Maria Teresa pela educação que me deram e pelo suporte aos meus filhos durante minha ausência.

Aos meus irmãos Ailton e Anderson pela disposição e prontidão com que sempre me atenderam nas situações imprevistas.

Ao Cirilo pelo companheirismo durante os momentos difíceis, pela dedicação em me substituir junto a nossos filhos durante muitas vezes neste período, pelo incentivo constante e pelas sugestões na correção da ortografia desta dissertação. Ao Thales e ao Vitor pela compreensão da importância deste trabalho para mim.

Resumo

A execução da engenharia reversa orientada a objetos de um sistema legado desenvolvido com orientação procedimental é usada como base para sua reengenharia, seguindo duas abordagens diferentes. Na primeira, o sistema passa por reengenharia com mudança de orientação, por meio de segmentação e, posteriormente, é transformado para uma linguagem orientada a objetos de forma semi-automática. Na segunda, é feito o reconhecimento de padrões recorrentes de software no modelo de objetos produzido pela engenharia reversa, para depois efetuar a reengenharia utilizando esses padrões. Os resultados obtidos por intermédio dessas duas abordagens podem ser comparados quanto à manutenibilidade, legibilidade e reuso. A versão original do sistema legado escolhido para a realização da experiência está implementado na linguagem Clipper e possui cerca de vinte mil linhas de código. Trata-se de uma oficina auto-elétrica e mecânica de veículos. Para a engenharia reversa foi escolhido o método Fusion/RE, sendo feita uma proposta para sua evolução, adicionando um maior detalhamento da etapa de abstração do modelo de análise do sistema. Para mudança de orientação do paradigma de desenvolvimento, de procedimental para orientado a objetos, são propostas duas etapas adicionais a serem executadas após a aplicação do Fusion/RE: o projeto avante do sistema e a segmentação do programa legado. Indicações sobre como fazer a segmentação são fornecidas. A transformação do código segmentado em Clipper para Java é feita com auxílio da máquina Draco-Puc. Uma estratégia é proposta para o reconhecimento de padrões a partir do modelo de objetos do sistema obtido pela engenharia reversa. Por meio dela, instâncias dos padrões *Type-Object*, *Association-Object*, *State Across a Collection* e *Behaviour Across a Collection* podem ser reconhecidas. Experiências de implementação de alguns desses padrões, em Delphi, são feitas.

Abstract

The object oriented reverse engineering of a legacy system, originally developed using the procedural paradigm, is the basis for two different reengineering approaches. In the first, the reengineering is done to change the implementation paradigm by segmentation, followed by the semi-automatic transformation to an object oriented language. In the second, recurring patterns are first recognized in the object model produced by the reverse engineering, and then the reengineering is done adopting these patterns. Results obtained by these two approaches are compared to assess their maintainability, legibility and reuse. The original version of the legacy system used in this experience has about twenty thousand lines of Clipper code and refers to an electric and mechanic car repair shop. For the reverse engineering phase the Fusion/RE method is used, and a proposal is done for its evolution, adding features to detail its system analysis model abstraction phase. To change the system orientation paradigm, from procedural to object-oriented, two additional phases are proposed to be conducted after the application of the Fusion/RE method: the forward design of the system and the legacy code segmentation. Hints and rationales are supplied to conduct the code segmentaion. The code transformation from segmented Clipper to Java is done with support of the Draco-Puc machine. A strategy is proposed for pattern recognition based on the system object model obtained through reverse engineering. Through it, instances of *Type-Object*, *Association-Object*, *State Across a Collection* and *Behaviour Across a Collection* patterns can be recognized. Delphi implementation experiments of these patterns are done.

Sumário

CAPÍTULO 1 – INTRODUÇÃO	1
1.1 – CONSIDERAÇÕES INICIAIS.....	1
1.2 – MOTIVAÇÃO	2
1.3 – OBJETIVOS	3
1.4 – ORGANIZAÇÃO DA DISSERTAÇÃO.....	4
CAPÍTULO 2 - RESENHA BIBLIOGRÁFICA.....	6
2.1 - CONSIDERAÇÕES INICIAIS	6
2.2 - ENTENDIMENTO DE PROGRAMAS	8
2.2.1 - <i>Introdução</i>	8
2.2.2 - <i>Contribuições Relevantes</i>	11
2.2.3 - <i>Conclusões</i>	16
2.3 - ENGENHARIA REVERSA.....	16
2.3.1 - <i>Introdução</i>	16
2.3.2 - <i>Contribuições Relevantes</i>	18
2.3.3 - <i>Conclusões</i>	23
2.4 - REENGENHARIA	24
2.4.1 - <i>Introdução</i>	24
2.4.2 - <i>Contribuições Relevantes</i>	25
2.4.3 - <i>Conclusões</i>	29
2.5 - PADRÕES.....	29
2.5.1 - <i>Introdução</i>	29
2.5.2 - <i>Contribuições Relevantes</i>	31
2.5.3 - <i>Conclusões</i>	41
2.6 - CONSIDERAÇÕES FINAIS.....	41
CAPÍTULO 3 – ENGENHARIA REVERSA.....	43
3.1 – CONSIDERAÇÕES INICIAIS.....	43
3.2 – UM RESUMO DO FUSION/RE	43
3.3 – SOBRE A ABSTRAÇÃO DO MODELO DE ANÁLISE DO SISTEMA	46
3.4 – ESTUDO DE CASO	47
3.4.1 - <i>Recuperação da arquitetura do sistema legado</i>	49
3.4.2 - <i>Obtenção do Modelo de Análise do Sistema Atual</i>	52
3.4.3 - <i>Abstração do Modelo de Análise do Sistema</i>	58
3.4.4 - <i>Mapeamento MAS/MASA</i>	69
3.5 – CONSIDERAÇÕES FINAIS	71
CAPÍTULO 4 – REENGENHARIA	73
4.1 – CONSIDERAÇÕES INICIAIS.....	73
4.2 – REENGENHARIA COM MUDANÇA DE ORIENTAÇÃO E SEM MUDANÇA DE LINGUAGEM	73
4.2.1 - <i>Abordagem proposta</i>	73
4.2.2 - <i>Estudo de Caso</i>	76
4.3 – REENGENHARIA COM MUDANÇA DE LINGUAGEM.....	80
4.3.1 - <i>Sistema de Transformação utilizado</i>	81
4.3.2 - <i>Estudo de Caso</i>	83
4.4 – CONSIDERAÇÕES FINAIS.....	87
CAPÍTULO 5 - RECONHECIMENTO DE PADRÕES DE SOFTWARE.....	89

5.1 – CONSIDERAÇÕES INICIAIS	89
5.2 – PADRÕES RECORRENTES PARA SISTEMAS DE INFORMAÇÃO	89
5.2.1 – <i>Padrão Type-Object</i>	89
5.2.2 – <i>Padrão Association-Object</i>	91
5.2.3 – <i>Padrão State across a collection</i>	92
5.2.4 – <i>Padrão Behaviour across a collection</i>	92
5.3 – UMA ESTRATÉGIA PARA RECONHECIMENTO DE PADRÕES	94
5.3.1 – <i>Padrão Type-Object</i>	94
5.3.2 – <i>Padrão Association-Object</i>	95
5.3.3 – <i>Padrão State across a Collection</i>	96
5.3.4 – <i>Padrão Behaviour across a collection</i>	96
5.4 – ESTUDO DE CASO	97
5.4.1 – <i>Reconhecimento dos padrões no modelo de objetos do sistema legado</i>	97
5.4.2 – <i>Exemplo de implementação de padrão</i>	101
5.5 – CONSIDERAÇÕES FINAIS	121
 CAPÍTULO 6 – CONCLUSÕES FINAIS E SUGESTÕES PARA NOVAS PESQUISAS	122
6.1 – CONSIDERAÇÕES INICIAIS	122
6.2 – CONCLUSÕES GERAIS	122
6.3 – CONTRIBUIÇÕES DESTE TRABALHO	125
6.4 – SUGESTÕES PARA FUTURAS PESQUISAS	126
 BIBLIOGRAFIA	128

Lista de Figuras

FIGURA 1.1 - ILUSTRAÇÃO DA ABORDAGEM DA DISSERTAÇÃO	4
FIGURA 2.1 - RELACIONAMENTO ENTRE TERMOS [CHI 90].....	6
FIGURA 2.2 - RELACIONAMENTO ENTRE OS TÓPICOS DO CAPÍTULO.....	8
FIGURA 2.3 - SISTEMA RECUPERADOR DE PROJETO BASEADO EM MODELO [BIG 89]	12
FIGURA 2.4 - METAMODELO INTEGRADO [MAY 95].....	15
FIGURA 2.5 - INSTANCIAÇÃO DO PARADIGMA “GOALS/MODELS/TOOLS” [BEN 92]	19
FIGURA 2.6 - DOCUMENTAÇÃO DE OBJETOS [SNE 95]	20
FIGURA 2.7 - PADRÃO “ROLES-PLAYED” (PAPÉIS DESEMPENHADOS) E EXEMPLO [COA 92]	31
FIGURA 2.8 - DIAGRAMA DO PADRÃO “ABSTRACT FACTORY” [GAM 93].....	33
FIGURA 2.9 - EXEMPLO DO PADRÃO “ABSTRACT FACTORY” [GAM 93]	34
FIGURA 2.10 - COREM E TIPOS DE PADRÃO BÁSICOS [GAL 96]	36
FIGURA 2.11 - ARQUITETURA DO SISTEMA “PAT” [KRA 96].....	37
FIGURA 2.12 - DIAGRAMA DE INTERAÇÃO DOS PADRÕES [YOD 98]	40
FIGURA 3.1 - ESQUEMA DO FUSION/RE.....	44
FIGURA 3.2 - PROCESSO DE TRANSIÇÃO DO MASA PARA O MAS	46
FIGURA 3.3 - DIAGRAMA DA BASE DE DADOS.....	49
FIGURA 3.4 - DESCRIÇÃO DE UM DOS ARQUIVOS DA BASE DE DADOS.	50
FIGURA 3.5 - DESCRIÇÃO DE UM DOS MÓDULOS DO SISTEMA, COM A RELAÇÃO CHAMA/CHAMADO POR.	51
FIGURA 3.6 - TRECHO DE UM PROGRAMA CLIPPER DO SISTEMA LEGADO	52
FIGURA 3.7 - PARTE DOS MENUS DO SISTEMA LEGADO.....	52
FIGURA 3.8 - MOSA – MODELO DE OBJETOS DO MASA.....	54
FIGURA 3.9 - PARTE DO MASA E PROCEDIMENTOS DO SISTEMA LEGADO.....	55
FIGURA 3.10 - MCVSA - MODELO DE CICLO DE VIDA DO SISTEMA ATUAL	56
FIGURA 3.11 - ESQUEMA PARA A OPERAÇÃO “LANCAMENTO_PRIM_PARTE” DO MOPSA	57
FIGURA 3.12 - MOS-1 – MODELO DE OBJETOS DO MAS – ETAPA 1	60
FIGURA 3.13 - MOS-2 – MODELO DE OBJETOS DO MAS – ETAPA 2	61
FIGURA 3.14 - MOS-3 – MODELO DE OBJETOS DO MAS – ETAPA 3	62
FIGURA 3.15 - MOS-4 – MODELO DE OBJETOS DO MAS – ETAPA 4	64
FIGURA 3.16 - MOS-5 – MODELO DE OBJETOS DO MAS – ETAPA 5	65
FIGURA 3.17 - MOS-6A – MODELO DE OBJETOS DO MAS – ETAPA 6 – PARTE A	66
FIGURA 3.18 - MOS-6B – MODELO DE OBJETOS DO MAS – ETAPA 6 – PARTE B	67
FIGURA 3.19 - PARTE DO MAS E MÉTODOS CORRESPONDENTES.....	68
FIGURA 3.20 - MCVS - MODELO DE CICLO DE VIDA DO SISTEMA	69
FIGURA 3.21 - ESQUEMA PARA A OPERAÇÃO “ABRE_ORDEM_DE_SERVIÇO” DO MOPS	70
FIGURA 4.1 - ABORDAGEM PROPOSTA PARA REENGENHARIA COM MUDANÇA DE ORIENTAÇÃO.....	74
FIGURA 4.2 - EXEMPLO DE UM GRAFO DE INTERAÇÃO DE OBJETOS.....	76
FIGURA 4.3 - PROGRAMA SEGMENTADO CORRESPONDENTE À OPERAÇÃO ABRE_ORDEM_DE_SERVIÇO.....	78
FIGURA 4.4 - ALGUNS MÉTODOS DO PROGRAMA SEGMENTADO	79
FIGURA 4.5 - ATIVIDADES BÁSICAS PARA A CONSTRUÇÃO DOS DOMÍNIOS DE ORIGEM E DESTINO NO DRACO	82
FIGURA 4.6 - CONSTRUÇÃO DOS DOMÍNIOS CLIPPER E JAVA NO DRACO [PEN 98B].....	85
FIGURA 4.7 - CÓDIGO JAVA RESULTANTE DA TRANSFORMAÇÃO, REF. À OPERAÇÃO ABRE_ORDEM_DE_SERVICO	86
FIGURA 4.8 - ALGUNS MÉTODOS DO PROGRAMA TRANSFORMADO	86
FIGURA 5.1 - PADRÃO TYPE-OBJECT [JOH 98].....	90
FIGURA 5.2 - EXEMPLO DO TYPE-OBJECT [JOH 98].....	90
FIGURA 5.3 - PADRÃO ITEM-DESCRIPTION [COA 92].....	90
FIGURA 5.4 - EXEMPLO DO ITEM-DESCRIPTION [COA 92]	90
FIGURA 5.5 - PADRÃO ASSOCIATION-OBJECT [BOY 98]	91
FIGURA 5.6 - ASSOCIATION-OBJECT [BOY 98]	92
FIGURA 5.7 - PADRÃO STATE ACROSS A COLLECTION [COA 92]	92
FIGURA 5.8 - EXEMPLO DO PADRÃO STATE ACROSS A COLLECTION [COA 92]	93
FIGURA 5.9 - PADRÃO BEHAVIOUR ACROSS A COLLECTION [COA 92]	93
FIGURA 5.10 - EXEMPLO DO PADRÃO BEHAVIOUR ACROSS A COLLECTION [COA 92].....	93
FIGURA 5.11 - NOTAÇÃO USADA PARA EXPRESSAR PADRÕES EM UML.....	98
FIGURA 5.12 - RECONHECIMENTO DE PADRÕES	99

FIGURA 5.13 - MODELO DE OBJETOS DETALHADO DO PADRÃO TYPE-OBJECT	102
FIGURA 5.14 - ESQUELETO DA DECLARAÇÃO DA CLASSE “CLASS” DO PADRÃO TYPE-OBJECT.....	103
FIGURA 5.15 - ESQUELETO DE ALGUNS MÉTODOS DO PADRÃO TYPE-OBJECT	104
FIGURA 5.16 - ESQUELETO DE UM DOS MÓDULOS DE DADOS DO PADRÃO TYPE-OBJECT.....	105
FIGURA 5.17 - CÓDIGO SQL DE ALGUNS COMPONENTES DO MÓDULO DE DADOS “CLASS”.....	105
FIGURA 5.18 - MODELO DE OBJETOS DE UMA INSTÂNCIA DO PADRÃO TYPE-OBJECT.....	106
FIGURA 5.19 - CÓDIGO EM DELPHI REFERENTE À DECLARAÇÃO DA CLASSE PEÇA	107
FIGURA 5.20 - CÓDIGO EM DELPHI REFERENTE À DECLARAÇÃO DA CLASSE FABRICANTE.....	108
FIGURA 5.21 - CÓDIGO EM DELPHI REFERENTE A ALGUNS MÉTODOS DO PADRÃO RECONHECIDO.....	109
FIGURA 5.22 - MÓDULO DE DADOS DA CLASSE PEÇA	109
FIGURA 5.23 - CÓDIGO SQL DE ALGUNS COMPONENTES DO MÓDULO DE DADOS “PEÇA”	110
FIGURA 5.24 - INTERFACE DA OPERAÇÃO DE CADASTRO DE PEÇAS	111
FIGURA 5.25 - TRECHO DE CÓDIGO REFERENTE AO FORMULÁRIO DE CADASTRO DE PEÇAS	111
FIGURA 5.26 - MODELO DE OBJETOS DA OUTRA INSTÂNCIA DO PADRÃO TYPE-OBJECT	112
FIGURA 5.27 - CÓDIGO EM DELPHI DA DECLARAÇÃO DAS CLASSES VEÍCULO E TIPO DE VEÍCULO.....	113
FIGURA 5.28 - CÓDIGO EM DELPHI REFERENTE A ALGUNS MÉTODOS DA SEGUNDA INSTÂNCIA DO PADRÃO	114
FIGURA 5.29 - MODELO DE OBJETOS PARA A OPERAÇÃO “ABRE_ORDEM_DE_SERVIÇO”.....	115
FIGURA 5.30 - DEFINIÇÃO DA CLASSE “ORDEMDeSERVIÇO”	117
FIGURA 5.31 - ALGUNS MÉTODOS DA CLASSE “ORDEMDeSERVIÇO”	118
FIGURA 5.32 - INTERFACE DO SISTEMA MOSTRANDO UM DOS MENUS	119
FIGURA 5.33 - INTERFACE DA OPERAÇÃO DE ABERTURA DA ORDEM DE SERVIÇO.....	119
FIGURA 5.34 - TRECHOS DO CÓDIGO REFERENTE AO FORMULÁRIO DE ABERTURA DA ORDEM DE SERVIÇO	120

Lista de Tabelas

TABELA 3.1 – CORRESPONDÊNCIA ENTRE CLASSE DO MAS E ARQUIVOS DO MASA	71
TABELA 3.2 – CORRESPONDÊNCIA ENTRE CLASSES/ATRIBUTOS DO MAS E OS ARQUIVOS/CAMPOS DO MASA.....	72
TABELA 3.3 – CORRESPONDÊNCIA ENTRE ALGUMAS OPERAÇÕES DO MAS E DO MASA	72
TABELA 6.1 – COMPARAÇÃO DOS RESULTADOS	124

Siglas Utilizadas

DAST – Draco Abstract Syntax Tree
KB – *Knowledge Base* – Base de Conhecimento
MASA – Modelo de Análise do Sistema Atual
MAS – Modelo de Análise do Sistema
MOSA – Modelo de Objetos do MASA
MOS – Modelo de Objetos do MASA
MCVSA – Modelo de Ciclo de Vida do Sistema Atual
MCVS – Modelo de Ciclo de Vida do Sistema
MOpSA – Modelo de Operações do Sistema Atual
MOpS – Modelo de Operações do Sistema

CAPÍTULO 1 – Introdução

1.1 – Considerações Iniciais

O desenvolvimento de sistemas a partir “do zero” é, na maioria das vezes, mais difícil do que quando existe algo pronto para ser usado como base. Por exemplo, em muitos sistemas de informação pode-se aproveitar a estrutura geral de um programa existente, os módulos para montagem de menus, relatórios, consultas e a estrutura de inserção, alteração e eliminação de dados. Esse aproveitamento pode ser maior ou menor de acordo com o tipo de aplicação a ser desenvolvida e de acordo com os sistemas prontos disponíveis.

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

Além desse aproveitamento puramente de código, seria desejável aproveitar soluções de análise e projeto, que exigiram bastante esforço para serem elaboradas e poderiam ser reutilizadas em novos desenvolvimentos. Essas soluções poderiam ser sintetizadas em “padrões”, sejam eles de análise, de projeto ou de código. Para criar esses padrões nada mais natural do que investigar sistemas prontos, em busca de trechos de código que representem soluções para determinados problemas e que possam ser reutilizados no futuro. Deve-se estudar, então, uma forma de documentar esses padrões, disponibilizando-os a quem possa interessar.

Padrões de software podem se referir a diferentes níveis de abstração no desenvolvimento de sistemas orientados a objetos. Assim pode-se ter desde padrões de análise, em que esse nível é bastante alto, passando por padrões de projeto e indo até padrões de código. Coad [Coad 92] apresenta padrões de análise bem gerais, úteis principalmente no desenvolvimento de sistemas de informação na área empresarial. Gamma [Gam 95] apresenta padrões de projeto de utilização bastante ampla que abrange sistemas os mais diversos, para várias áreas de aplicação. Fowler [Fow 97] discute padrões de análise específicos para as áreas de contabilidade, observações e

medidas, finanças empresariais, referenciação de objetos, controle de estoque, planejamento, comercialização de bens, negociação em bolsas de valores e negociação de lotes de papéis, entre outros.

1.2 – Motivação

Minha formação acadêmica enfatizou o uso do desenvolvimento estruturado de sistemas, tanto com base nos dados como nos processos. Recebi uma bolsa de Iniciação Científica da FAPESP, cujo plano de trabalho “Estudo Comparativo de uma Abordagem Convencional com uma Operacional para o Desenvolvimento de Sistemas de Informação” levou à preparação de vários relatórios técnicos, entre os quais [Bra 86a] e [Bra 86b]. No primeiro foi usado o método JSD [Jac 83], que é um método estruturado com base nos dados e no segundo foi usado o método de Gane [Gan 83], que é um método estruturado com base nos procedimentos. Na época, o JSD era considerado uma abordagem operacional, enquanto que o método de Gane era considerado uma abordagem convencional. O trabalho de Masiero [Mas 88], mostra como o JSD pode ser considerado como um método orientado a objetos. Seu estudo motivou-me a saber mais sobre a orientação a objetos, familiarizando-me com a bibliografia sobre esse assunto.

Durante cerca de dez anos de prática profissional utilizei a análise, projeto e programação estruturados para o desenvolvimento de mais de vinte sistemas de informação. Esses sistemas usaram bases de dados relacionais e foram desenvolvidos em linguagens como Dbase, Clipper, FoxPro for Windows, Visual FoxPro e Access, em um total de aproximadamente 400.000 linhas de código.

Muitos desses sistemas foram feitos baseados em outros sistemas anteriormente por mim desenvolvidos. Por exemplo, com base em um sistema para Controle de Estoque e Emissão de Notas Fiscais de Produtos Agrotóxicos, foi feito um sistema para uma Revendedora de Motocicletas e Peças. Mais tarde, algumas partes desse sistema foram adaptadas para uma Oficina Eletrônica de Reparos. A parte financeira do primeiro sistema por mim construído foi usada quase que integralmente em muitos outros sistemas. As bases de dados também possuem diversos arquivos similares, aproveitando-se toda a idéia de projeto.

Nessas atividades de reuso, deparei-me com todos os problemas citados na seção anterior. Alguns sistemas ficaram ineficientes, com características desnecessárias para a aplicação específica; algumas adaptações forçadas dificultaram a manutenção; alguns módulos ficaram

totalmente diferentes daqueles nos quais foram baseados; houve dificuldade em escolher qual o melhor sistema para servir de base, devido à similaridade dos mesmos e à falta de documentação explicando detalhadamente cada um. Pude sentir, também, dificuldades na manutenção desses sistemas. Alguns deles foram feitos em pequenas equipes (analista e um ou dois programadores), que nem sempre mantinham a documentação em ordem. Assim, muitos efeitos colaterais ocorreram devido a pequenas alterações, além da constante dificuldade encontrada para localizar a origem dos problemas que causaram a necessidade de manutenção. Diante disso, surgiu a grande motivação para estudar uma forma de facilitar o reuso, não somente de trechos de código, mas também de conceitos de análise e projeto, bem como de facilitar a manutenção dos sistemas.

Apesar dessa prática profissional ter sido desenvolvida principalmente orientada a procedimentos, foi usada orientação a objetos em outros trabalhos, como um relatório técnico do ICMC [Bra 98a] e dois artigos em congressos internacionais [Pen 98a, Pen 98b] dos quais sou co-autora.

Trata-se principalmente de execução de engenharia reversa orientada a objetos em sistemas legados desenvolvidos com orientação a procedimentos. Essa engenharia reversa foi feita de acordo com um processo geral denominado Fusion/RE [Pen 95, Pen 96, Pen 96a e Pen 96b], que será detalhado mais adiante. Esse processo é baseado no método Fusion [Col 94] para desenvolvimento orientado a objetos.

1.3 – Objetivos

Este trabalho propõe uma abordagem alternativa à usual, que é descartar o código antigo quando se cogita de ampliar a funcionalidade de um sistema existente, com idade avançada e desatualizado. O objetivo dessa abordagem é reconhecer padrões de software, sejam eles de análise, de projeto ou de código, que sejam úteis no reuso, na ampliação de funcionalidade e na manutenção de sistemas. A base para esse reconhecimento é a realização de experimentos de reengenharia de sistemas legados, mudando a orientação procedimental original para orientação a objetos. Dá-se um enfoque maior para o domínio de sistemas de informação. A reengenharia com mudança de linguagem é feita tanto de forma automática como de forma manual, obtendo-se sistemas em linguagens orientadas a objetos. Uma comparação é feita entre os sistemas obtidos por transformação automática de linguagem e por reengenharia manual com uso de padrões.

A figura 1.1 ilustra a abordagem da dissertação. Após a engenharia reversa do sistema legado, obtém-se o Modelo de Análise do Sistema, seguindo a orientação a objetos. Dois caminhos são então percorridos: primeiramente, o projeto avante e a segmentação do sistema legado são realizados, após os quais obtém-se o sistema segmentado, na linguagem procedimental em que foi desenvolvido. Pode-se então usar uma ferramenta para fazer a transformação automática para uma linguagem orientada a objetos. O caminho alternativo, explorado depois, faz o reconhecimento de padrões a partir do Modelo de Análise do Sistema, elaborando a seguir o projeto avante e implementando o sistema, manualmente, em uma linguagem orientada a objetos. Durante esse processo, pode-se fazer uma busca por situações em que esses padrões encontrados possam ser empregados, elaborando-se diretrizes para o reconhecimento e uso de padrões.

Figura 1.1 - Ilustração da Abordagem da Dissertação

1.4 – Organização da Dissertação

Este trabalho está organizado como segue. No capítulo 2 é relatada a pesquisa bibliográfica feita sobre o assunto. No capítulo 3 é descrita a realização da engenharia reversa em um sistema legado de oficina auto-elétrica e mecânica de veículos. No capítulo 4 é descrita a reengenharia desse sistema com mudança de orientação mas sem mudança de linguagem e,

posteriormente, a reengenharia com mudança de linguagem de Clipper para Java. No capítulo 5 é estabelecida uma estratégia para reconhecimento de padrões recorrentes, com base nos modelos de análise do sistema e é feita uma experiência de reengenharia com mudança para linguagem Delphi usando esses padrões. O capítulo 6 apresenta as conclusões finais, sugerindo também novas pesquisas que poderiam ser feitas nesta área.

CAPÍTULO 2 - Resenha Bibliográfica

2.1 - Considerações Iniciais

A terminologia empregada na engenharia de software para referenciar as tecnologias de análise e entendimento de sistemas existentes é apresentada por Chikofsky em [Chi 90], com o objetivo de racionalizar termos que já estão em uso. Os termos definidos e relacionados são: engenharia avante, engenharia reversa, redocumentação, recuperação de projeto, reestruturação e reengenharia.

Figura 2.1 - Relacionamento entre termos [Chi 90].

Engenharia avante é o processo tradicional de partir de um nível de abstração alto, de requisitos do sistema, e chegar ao nível físico, de implementação do sistema. Reestruturação é a transformação de uma forma de representação para outra no mesmo nível relativo de abstração, preservando o comportamento externo do sistema (funcionalidade e semântica). Como exemplos de reestruturação pode-se citar estruturação de programas e normalização de dados. Engenharia reversa, bem como duas sub-áreas amplamente referenciadas, a saber, redocumentação e recuperação de projeto, são definidas na seção 2.3. Reengenharia é definida na seção 2.4. O relacionamento entre os termos (figura 2.1) é feito considerando-se que o ciclo de vida do

software possui três grandes etapas (requisitos, projeto e implementação) com claras diferenças no nível de abstração. Requisitos tratam da especificação do problema, incluindo objetivos, restrições e regras de negócio. Projeto trata da especificação da solução. Implementação trata da codificação, teste e entrega do sistema em operação.

A figura 2.1 mostra claramente a direção seguida pela engenharia avante, do nível mais alto para o nível mais baixo de abstração. Mostra, também, que a engenharia reversa percorre exatamente o caminho inverso, podendo utilizar-se da recuperação de projeto para aumentar o nível de abstração. A reengenharia geralmente inclui alguma forma de engenharia reversa, seguida de alguma forma de engenharia avante ou reestruturação. A redocumentação é uma forma mais fraca de reestruturação, permanecendo no nível de implementação do sistema.

O entendimento de programas é essencial para a manutenção e reuso de software. Qualquer que seja a manutenção a ser feita no software haverá a necessidade de primeiramente entender o programa, para depois efetuar as alterações. O mesmo ocorre no reuso. Um programa, para ser reutilizado, deve ser primeiramente entendido a fim de verificar se há necessidade de adaptá-lo antes do reuso.

A engenharia reversa orientada a objetos pode ser feita em sistemas legados desenvolvidos com orientação a procedimentos. Nela são recuperadas informações com maior nível de abstração do que o código fonte, úteis para o entendimento do sistema, ao mesmo tempo em que o mesmo é modelado segundo a orientação a objetos e preparado para uma futura reengenharia.

A reengenharia com mudança de orientação, que transforma sistemas originalmente procedimentais em sistemas orientados a objetos, deve ser feita após a engenharia reversa citada acima, que obtém modelos de análise e projeto orientados a objetos, mudando também a implementação para uma linguagem orientada a objetos.

Com padrões de software, podem ser preservadas soluções tanto de análise, como de projeto e implementação, úteis em futuros desenvolvimentos. O estabelecimento e uso de padrões de software pode aumentar a produtividade, qualidade e custo do software.

Os grandes tópicos deste capítulo são intimamente inter-relacionados, como mostra a figura 2.2. O entendimento de programas (seção 2.2) é uma fase essencial da engenharia reversa (seção 2.3), que por sua vez é a etapa inicial para se fazer a reengenharia (seção 2.4). Padrões (seção 2.5) são um recurso para reusabilidade, necessitando também da engenharia reversa e

consequentemente do entendimento de programas. Esses tópicos são explorados com maior ou menor intensidade, de acordo com os objetivos pretendidos.

Figura 2.2 - Relacionamento entre os tópicos do capítulo

A importância de fazer uma resenha bibliográfica envolvendo esses quatro tópicos está no fato de que eles estão intimamente ligados com os objetivos propostos no título da dissertação. Padrões devem ser estudados, pois deverão ser reconhecidos facilmente. Engenharia reversa deve ser explorada, pois é durante ela que os padrões serão identificados. O entendimento de programas é importante por ser uma parte essencial da engenharia reversa e a reengenharia por ser um dos objetivos deste trabalho fazer a reengenharia do sistema legado adotando os padrões reconhecidos.

2.2 - Entendimento de Programas

2.2.1 - Introdução

Entender um programa significa poder explicar o programa, sua estrutura, seu comportamento, seus efeitos e seus relacionamentos com o domínio de aplicação [Big 94]. O entendimento de programas (ou, similarmente, a compreensão de programas) é a construção de uma estrutura semântica em diversos níveis de abstração, para representá-los. Para isso, o

programador pode contar com o auxílio de seu conhecimento sintático da linguagem de programação e com seu conhecimento sobre o domínio da aplicação [Shn 79].

O nível de abstração em que um programa é entendido pode ser alto (por exemplo, a descoberta do quê o programa faz) ou baixo (por exemplo, o reconhecimento de seqüências familiares de comandos). Ressalta-se que é possível ter uma compreensão de alto nível mesmo sem ter-se a compreensão de baixo nível, e vice-versa [Shn 79].

A maneira mais elementar de obter conhecimento sobre um sistema é por meio da leitura do código fonte, o que pode ser efetivo mas muito difícil devido à grande quantidade de informação contida no código e à dificuldade de extrair o conhecimento necessário [Rob 91]. Assim, é necessário um instrumental para apoiar o entendimento de programas, que compreende a criação de modelos cognitivos, a construção de ferramentas para ajudar o entendimento e a realização de estudos empíricos sobre entendimento de programas.

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

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

Estudos empíricos são uma forma de obter dados sobre o processo de cognição e validá-lo. Esses estudos podem ser conduzidos de três formas diferentes: observação, correlação e teste de hipóteses. A primeira observa o comportamento da maneira como ocorre no mundo real, por exemplo o comportamento de programadores diante de situações específicas. A segunda assume que uma teoria tenha sido construída a partir dos estudos observacionais e tenta explorar o relacionamento entre as variáveis em questão. A terceira investiga causas e efeitos entre

variáveis, a fim de validar uma teoria existente. Os estudos empíricos levam em consideração, também, o tamanho do código fonte, linguagem de programação e o tipo de programador. Por exemplo, “pequeno” refere-se a programas de menos de 900 linhas, “médio” refere-se a programas entre 900 e 40.000 linhas e “grande” acima de 40.000 linhas. Programadores podem ser “noviços”, “estudantes de graduação” e “programadores profissionais” [May 95] .

O principal motivo que leva à necessidade de entender um programa é que muitas regras de negócios estão embutidas nos programas e não estão documentadas de forma explícita e precisa em nenhum outro lugar. Essas regras de negócios são bens valiosos para as empresas e muito difíceis de serem captadas e redesenvolvidas [Nin 94].

O entendimento de programas tem algumas aplicações de grande importância na Engenharia de Software, entre elas: a manutenção de software, o reuso e a recuperação de projeto.

A manutenção de software é a modificação de um produto de software, depois da entrega, para corrigir falhas, melhorar o desempenho e outros atributos, ou para adaptar o produto a mudanças de ambiente [Chi 90]. A manutenção só é possível após o entendimento do programa, para que possam ser localizados os problemas a serem tratados. Estima-se que 50-90% do tempo de manutenção é gasto com a compreensão de programas. Mesmo quando existe documentação, o tempo gasto para estudo de código é três vezes e meia maior do que o tempo gasto para estudo da documentação [Rob 91].

O reuso, atividade que tem por finalidade reutilizar componentes de software anteriormente desenvolvidos, envolve diretamente o entendimento de programas. Para reutilizar partes de um software existente, é necessário primeiro entender detalhadamente essas partes, para que seja avaliado seu potencial de reuso e para que sejam feitas as adaptações necessárias para torná-las componentes reutilizáveis.

A recuperação de projeto, um subconjunto da engenharia reversa discutida posteriormente na seção 2.3.1, deve reproduzir toda a informação necessária ao entendimento total de: “o quê” um programa faz, “como” o faz e “porquê” o faz. Por intermédio dela são criadas abstrações de projeto do sistema, a partir de uma combinação de código, documentação existente (se disponível), experiência pessoal e conhecimentos gerais sobre os problemas e o domínio de aplicação [Big 89].

A seção seguinte relata contribuições consideradas relevantes para o entendimento de programas.

2.2.2 - Contribuições Relevantes

Shneiderman, em [Shn 79], apresenta um modelo cognitivo do comportamento do programador, dividindo seu conhecimento em semântico e sintático. O conhecimento semântico consiste de conceitos gerais de programação independentes de linguagens específicas, enquanto que o conhecimento sintático é mais preciso, detalhado e fácil de esquecer. O programador cria representações internas na resolução de problemas tais como composição, compreensão, depuração, modificação e aprendizagem do programa. Os componentes da memória humana são utilizados de maneira distinta na resolução de problemas. O conhecimento semântico e sintático do programador é armazenado em sua memória de longo prazo. Esse conhecimento é usado como base na resolução de problemas que envolvem novas informações contidas na memória de curto prazo. São relatados diversos experimentos empíricos, cujas conclusões levam à constatação de que a construção de uma estrutura semântica interna facilita a compreensão de programas.

Soloway, em [Sol 84], trata da compreensão de programas por meio de experimentos empíricos, visando a determinar como esta é afetada pela maior ou menor experiência de programação de quem tenta obtê-la. Quem tem maior experiência possui dois tipos de conhecimento ausentes em quem tem menos: de planos de programas e de regras de raciocínio de programas. Os primeiros são fragmentos de programas que constituem típicas seqüências de comandos que o programador tem experiência de uso em situações anteriores, as segundas são regras de bom senso comumente adotadas na prática de programação. Programadores experientes compreendem mais facilmente programas que tenham sido construídos com um determinado plano e que não infrinjam tais regras. Têm dificuldade de compreender programas que não tenham tais características. Já os novatos não são afetados por essas diferenças.

Biggerstaff, em [Big 89], estabelece um processo básico de recuperação de projeto, composto de três passos. O primeiro ajuda no entendimento de programas, identificando estruturas de grande porte e associando-as com conceitos semânticos informais. O segundo ajuda na alimentação e recuperação de bibliotecas de reuso. O terceiro aplica os resultados obtidos na recuperação de projeto para descobrir candidatos a componentes de um sistema novo. Mostra a importância da utilização de informações informais, tais como comentários e nomes de variáveis mnemônicos, e de um modelo do domínio, para a construção de abstrações conceituais do projeto. Introduz as conexões associativas e os padrões estruturais como um modo de formalizar

parcialmente as abstrações conceituais informais. As conexões associativas fazem a ligação entre um conceito abstrato e o respectivo código. Os padrões estruturais definem os tipos de estrutura de código fonte que representam cada abstração. Apresenta o primeiro protótipo do DESIRE (“*Design Recovery*”), um sistema recuperador de projetos baseado em modelos, que auxilia os engenheiros de software no entendimento de programas, mostrado na figura 2.3. O sistema consiste de três partes principais: um “*parser*”, um conjunto de funções de pós-processamento e um sistema de hipertexto chamado “*PlaneText*”. O *parser* processa programas em linguagem “C” e gera as “*parse trees*”, que são usadas pelos pós-processadores para produzir um dicionário contendo informações e relacionamentos entre funções e itens de dados, além de associá-las a informações informais (comentários, por exemplo). O sistema de hipertexto exibe os relacionamentos por meio de um “*browser*”, permitindo visões específicas em janelas separadas, de acordo com a necessidade do usuário. O sistema oferece, também, “*queries*” pré-definidas em Prolog, que ajudam na busca por padrões.

Figura 2.3 - Sistema recuperador de projeto baseado em modelo [Big 89]

Robson, em [Rob 91], reforça a idéia de que a compreensibilidade de programas é uma parte vital do processo de manutenção. Comenta algumas teorias de compreensão de programas, que modelam o processo de compreensão. Discute a dificuldade de entendimento de código

apenas pela leitura, a qual é diretamente dependente do tamanho e complexidade do programa. Discute ainda outros fatores que afetam o entendimento de programas, tais como: estilo de programação, modularização, nomes de variáveis, tabulação e comentários. Descreve algumas das técnicas empregadas para automatizar a leitura de código, categorizando-as em estratégias estáticas e dinâmicas. Considera-as como auxiliares de baixo nível ao processo de compreensão, pois ajudam na compressão de pequenas unidades do programa, mas oferecem muito pouca ajuda na determinação do projeto geral. As vantagens da automação são óbvias na manutenção de sistemas grandes nos quais nenhum dos mantenedores participou do desenvolvimento. A engenharia inversa ou engenharia reversa é considerada como auxiliar de alto nível, porém ainda precisa de muito trabalho para se tornar efetiva. As técnicas de inteligência artificial são consideradas uma área de pesquisa muito útil à compreensão de programas, porém pouco exploradas.

Biggerstaff, em [Big 94], relaciona o entendimento de programas com o problema de assimilação de conceito. Define como “Problema de assimilação de conceito” a descoberta de conceitos que envolvem um grande conhecimento do domínio de informação e sua associação com um programa ou contexto específico, defendendo a idéia de que não há algoritmo que permita o reconhecimento desses conceitos com plena confiança. É feita uma comparação entre conceitos orientados à programação, que podem ser derivados de forma dedutiva ou algorítmica, e conceitos orientados ao ser humano, que requerem um raciocínio razoável e exigem um conhecimento prévio do domínio em questão. Sugere a utilização de uma ferramenta para recuperação de Projeto, o DESIRE, composto de um assistente “ingênuo”, que oferece serviços simples para apoiar a inteligência humana e um assistente “inteligente”, que oferece uma assistência inteligente, por meio de um motor de inferência baseado em Prolog e um reconhecedor de padrões baseado em conhecimento.

Ning, em [Nin 94], propõe uma abordagem para recuperação de componentes reusáveis, na qual componentes funcionais de sistemas legados são reconhecidos, recuperados, adaptados e reutilizados no desenvolvimento de um novo sistema. Essa recuperação requer profunda análise e entendimento do código antigo. É apresentado um conjunto de ferramentas chamado Cobol/SRE, que ajuda no entendimento de programas por meio da segmentação de programas que, basicamente, divide o programa em partes mais fáceis de serem compreendidas por quem for utilizá-las. O Cobol/SRE oferece recursos para demarcar os segmentos e posteriormente empacotá-los em módulos independentes. Para a demarcação é possível englobar,

automaticamente, as linhas de código afetadas por um “*perform*”, uma condição, o valor de uma entrada ou saída. O empacotamento gera novos módulos de programas, baseando-se nos segmentos criados na demarcação, tornando o código antigo mais modularizado e portanto mais reusável.

Mayrhauser, em [May 95], discute a compreensão de programas durante a manutenção e evolução do software. Subdivide a manutenção em adaptativa, aperfeiçoativa e corretiva, detendo-se também no reuso e na alavancagem de código (“*code leverage*”). Ressalta que, para que o processo de entendimento de programas possa ser otimizado, é necessário que se compreenda como os programadores entendem o código, o que pode ser apoiado pelos modelos de cognição. Identifica os elementos comuns aos modelos de cognição, a saber, o conhecimento, o modelo mental e as características do especialista. Descreve vários modelos de cognição de código, entre os quais: o modelo de Letovsky; o modelo de Shneiderman e Mayer [Shn 79]; o modelo de Brooks; os modelos de fluxo de controle e o funcional, concebidos por Pennington; o modelo “*Top-down*”, concebido por Soloway, Adelson e Ehrlich e o seu próprio modelo “integrado” (figura 2.4). O modelo “integrado” considera três componentes: o modelo do programa, o modelo da situação e os planos de programação. No modelo do programa estão consolidados o conhecimento do domínio do programa, isto é, da estrutura de texto, dos planos e das regras de raciocínio. No modelo da situação estão reunidos os conhecimentos do domínio do problema, ou seja, o conhecimento funcional. Nos planos de programação constam os planos estratégicos, táticos e de implementação, bem como regras de raciocínio. A cada um dos três componentes corresponde um processo de modelagem e um setor integrado na base de conhecimento. No processo de entendimento há alternância entre os três modelos, de modo a conseguir novos conhecimentos que tornam a alimentar a base de conhecimento. Esse modelo permite tanto o entendimento “*top-down*” quanto o entendimento “*bottom-up*”, ou a mistura dos dois.

Mayhauser apresenta também uma análise comparativa quanto aos critérios estáticos: estruturas de conhecimento e representações mentais; dinâmicos e de experimentação, indicando o grau de abstração e dimensão da escala dos experimentos, bem como se o objetivo é ganhar um entendimento geral superficial ou parcial e mais aprofundado. Em [May 96], Mayrhauser reforça a parte experimental do trabalho, comprovando que os processos cognitivos atuam simultaneamente em todos os níveis de abstração, à medida em que os programadores constroem o modelo mental do código.

Figura 2.4 - Metamodelo Integrado [May 95]

2.2.3 - Conclusões

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

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

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

A correspondência entre conceitos orientados ao ser humano e trechos de código nunca será totalmente automatizada, mas é possível apoio por computador, de utilidade para o engenheiro de software. Acrescentando-se elementos por ele fornecidos para completar os aspectos não atingidos pela automação, pode-se acelerar e simplificar de forma significativa o entendimento de programas [Big 94]. Essa idéia é unânime em todos os trabalhos estudados.

2.3 - *Engenharia Reversa*

2.3.1 - Introdução

O termo “engenharia reversa” tem sua origem em análise de hardware, na qual a prática de extrair projetos do produto final é trivial. A Engenharia Reversa é regularmente aplicada para melhorar o produto de uma empresa a partir da análise dos produtos do adversário. “Engenharia Reversa é o processo de desenvolvimento de um conjunto de especificações para um sistema de

hardware complexo por exame de espécimes daquele sistema, de forma ordenada.” [Rek 85]. Esse processo é, em geral, conduzido por outro desenvolvedor, sem o benefício de algum dos desenhos originais, com o propósito de fazer um *clone* do sistema de hardware original.

No software, a engenharia reversa é o processo de análise de um sistema para identificar seus componentes e inter-relacionamentos e criar representações do mesmo em outra forma ou num nível mais alto de abstração [Chi 90]. As informações extraídas do código fonte via engenharia reversa podem estar em diversos níveis de abstração. Por exemplo, num baixo nível de abstração têm-se representações de projeto procedimental, subindo para informações sobre a estrutura de dados e de programa, modelos de fluxo de controle e de dados e chegando a modelos entidade-relacionamento, que constituiriam o nível de abstração mais alto. O ideal seria ter um nível de abstração o mais alto possível [Pre 95].

A engenharia reversa deve produzir, preferencialmente de forma automática, documentos que ajudem a aumentar o conhecimento geral de sistemas de software, facilitando o reuso, manutenção, teste e controle de qualidade de software. Além disso, a engenharia reversa justifica a necessidade de utilizar o conhecimento implementado em software antigo (ou legado) na produção de software novo e ajustar software e documentação existentes a novos padrões e ferramentas introduzidos por novos métodos e tecnologias de produção. Ressalte-se que é necessário adicionar experiência pessoal e conhecimentos gerais a respeito do problema e do domínio de aplicação à informação extraída automaticamente do código [Ben 92].

Existem diversas sub-áreas da engenharia reversa: entendimento de programas (visto na seção 2.1), redocumentação e recuperação de projeto. A redocumentação é a criação de uma representação alternativa de um programa, que seja equivalente e esteja no mesmo nível de abstração. Alguns a consideram uma forma de reestruturação. Seu objetivo é recuperar documentação que existiu ou deveria ter existido de um sistema alvo. A redocumentação pode ser auxiliada por ferramentas automáticas, tais como os “*pretty printers*”, que mostram a listagem do código de uma forma melhor; geradores de diagramas, que criam diagramas para mostrar fluxo de controle a partir do código e geradores de listagens de referência cruzada. A recuperação de projeto usa conhecimentos sobre o domínio de aplicação, informações externas e raciocínio, juntamente com observações sobre o sistema em questão, para identificar abstrações de nível mais alto do que as produzidas diretamente pelo exame do sistema em si [Chi 90].

As duas maiores aplicações da engenharia reversa são o reuso e a manutenção de software. O reuso de software pode ser apoiado pela engenharia reversa tanto na identificação e

composição de componentes a partir de partes reutilizáveis de sistemas existentes quanto na elaboração da documentação dos novos sistemas compostos. A manutenção de software é uma tarefa difícil e por isso deve contar com a ajuda de uma documentação completa de todo o processo de desenvolvimento. Nesse ponto, a engenharia reversa pode fornecer as visões em diversos níveis de abstração, permitindo a localização dos componentes a serem mantidos, além de melhorar a compreensibilidade do software, pela documentação produzida.

As ferramentas de apoio à engenharia reversa executam uma análise pós-desenvolvimento num programa existente. Existem ferramentas estáticas, que usam o código fonte de um programa como entrada e extraem a arquitetura do programa, estrutura de controle, fluxo lógico, estrutura de dados e fluxo de dados; e ferramentas dinâmicas, que monitoram o software em execução e usam as informações para construir um modelo comportamental do programa [Pre 95].

Diversas ferramentas e métodos têm sido propostos para apoiar a engenharia reversa. A seção seguinte sumariza alguns deles.

2.3.2 - Contribuições Relevantes

Benedusi, em [Ben 92], define um paradigma de referência, chamado “*Goals/Models/Tools*”, para estabelecer processos de engenharia reversa capazes de gerar, automaticamente, informações e documentos de projeto, a partir do código fonte. A fase “*Goals*” analisa as razões pelas quais o processo de engenharia reversa deve ser estabelecido e define os documentos a serem produzidos. A fase “*Models*” analisa os documentos a serem produzidos, identifica as informações e relacionamentos que podem ser extraídas diretamente do código e que são direta ou indiretamente necessárias à produção desses documentos, define os modelos de programas que podem representar essas informações e relacionamentos e define os algoritmos de abstração que permitirão que os documentos desejados sejam produzidos a partir desses modelos, via o processo de engenharia reversa. A fase “*Tools*” define, adquire ou constrói todas as ferramentas de software necessárias à produção dos documentos da engenharia reversa. É dado um exemplo de aplicação do paradigma (figura 2.5) a um caso real de reconstrução dos documentos de projeto a partir de código PASCAL. No exemplo, a fase “*Goals*” tem por objetivo a produção de diagramas de estrutura, a fase “*Models*” usa expressões algébricas de programas e técnicas de fluxo de dados global e a fase “*Tools*” é executada por um extrator

Pascal. Em [Ben 96], é mostrado como a qualidade dos modelos obtidos pela engenharia reversa pode ser melhorada quando, além do código fonte, da documentação de uso e da experiência humana, utilizam-se os resultados de casos de teste que ocorrem durante a manutenção. Assim, a análise estática pode ser complementada com a análise dinâmica.

Figura 2.5 - Instancição do Paradigma "Goals/Models/Tools" [Ben 92]

Wong, em [Won 95], ressalta que, para sistemas grandes, o entendimento de aspectos estruturais da arquitetura do sistema é mais importante do que o entendimento de um componente isolado qualquer. A redocumentação estrutural é definida como uma engenharia reversa dos aspectos arquitetônicos do software. Apresenta uma ferramenta, chamada "Rigi", para redocumentação estrutural de sistemas, que fornece um método para identificar, construir e documentar hierarquias de subsistemas em camadas. O Rigi possui três componentes: o "Rigireverse", um sistema de *parser*; o "Rigiserver", para armazenar a informação extraída do código fonte e o "Rigiedit", um editor gráfico interativo para manipular representações de programa. Para usar de forma efetiva a informação extraída pelo Rigi deve-se acrescentar o conhecimento sobre o domínio específico de aplicação. Isso é feito por meio de *scripts*, que permitem que o usuário escreva rotinas sob medida para atividades comuns, tais como extração de artifícios, apresentação de gráficos e pesquisa e seleção de objetos. O Rigi foi testado em

sistemas com mais de um milhão de linhas de código, confirmando que as visões produzidas durante a redocumentação estrutural auxiliaram no entendimento desses sistemas legados.

Figura 2.6 - Documentação de objetos [Sne 95]

Sneed, em [Sne 95], apresenta uma abordagem para extrair documentação de projeto orientada a objetos, automaticamente, a partir de programas existentes em COBOL. Comenta que a migração de sistemas legados em operação em "mainframes" para arquiteturas cliente/servidor exige, na maioria das vezes, uma engenharia reversa seguida de reimplementação, sem mudança de funcionalidade, que é a noção de reengenharia proposta por Chikofsky [Chi 90]. Essa migração pode beneficiar-se da reusabilidade, interface gráfica, comunicação inter-programas e outras técnicas modernas de programação proporcionadas pela orientação a objetos. Para isso, Sneed propõe uma abordagem que produz documentos de projeto compatíveis com os vários métodos de análise orientada a objetos, como por exemplo Coad/Yourdon, Shlaer/Mellor ou Rumbaugh. As etapas dessa abordagem, denotadas por retângulos na figura 2.6, são as seguintes:

identificação de objetos, extração de operações, conexão dos objetos, re-especificação da sequência de operação e re-especificação de casos de uso. É apresentada uma ferramenta, chamada OBJECT-REDOC, projetada para analisar código legado em COBOL e produzir documentos de projeto orientado a objetos, tais como: árvores de objetos/atributos, grafos de colaboração, árvores de condição, especificação de casos de teste e listas de referência cruzada.

Penteado, em [Pen 95] e [Pen 96], apresenta um método (posteriormente denominado “Fusion/RE”) para engenharia reversa de sistemas legados, implementados sem usar a tecnologia de orientação a objetos, com o objetivo de produzir o modelo de análise orientado a objetos. Baseia-se no método “Fusion” para desenvolvimento orientado a objetos [Col 94], que reúne diversas técnicas propostas por outros métodos. A engenharia reversa é feita em quatro passos, que serão descritos na seção 3.2. Penteado faz um estudo de caso, aplicando seu método a um ambiente para edição e simulação de statecharts, chamado “Statsim”, desenvolvido no ICMSC/USP. O estudo confirma a viabilidade de derivação de um modelo orientado a objetos com base em uma implementação não orientada a objetos.

Dando continuidade ao trabalho apresentado em [Pen 95], Penteado utiliza, em [Pen 96a] e [Pen 96b], métricas para avaliação da qualidade da implementação atual e do esforço de conversão gastos na eventual reengenharia do sistema. Essas métricas baseiam-se, principalmente, nos procedimentos do sistema atual que contém anomalias e nos procedimentos de implementação. Os procedimentos com anomalias são aqueles que fazem uso de mais de uma classe. Ao serem convertidos, no novo sistema, exigem maior esforço, pois precisam ser desmembrados em diversos métodos de diversas classes. Os procedimentos de implementação são aqueles relacionados à interface, módulo escalonador, etc. O fato deles existirem em grande volume pode indicar grande esforço de conversão, ao passo que se forem poucos pode significar que a interface é muito simples e deveria ser reformulada na reengenharia. São discutidas quatro alternativas a serem seguidas após a engenharia reversa do sistema: uso da documentação produzida para facilitar a manutenção, melhoria da qualidade do sistema atual, desenvolvimento do sistema por reengenharia e conservação da implementação atual, com aperfeiçoamentos futuros realizados usando a orientação a objetos.

Hainaut, em [Hai 96], propõe um processo para elicitação da estrutura na engenharia reversa de bases de dados, que é composto por dois processos principais. O primeiro cuida da extração da estrutura de dados, buscando reconstruir completamente o esquema lógico. O segundo cuida da conceitualização da estrutura de dados, visando a especificar as estruturas

semânticas desse esquema lógico na forma de um esquema conceitual. São relatados alguns problemas encontrados na extração da estrutura de dados, sendo que o maior deles é descobrir e explicitar as estruturas e restrições que estão implementadas de forma implícita. Um exemplo disso é o problema da eliciação de chaves estrangeiras, que é estudado mais profundamente para ilustrar o método proposto. É apresentada uma ferramenta, chamada DB-MAIN [Hai 96a], que é um ambiente CASE programável de propósito geral dedicado à Engenharia de Aplicação de banco de dados. Alguns de seus assistentes, dedicados à eliciação de estruturas implícitas, são também descritos.

Finnigan, em [Fin 97], propõe um ambiente de apoio à migração de software, chamado *Software bookshelf*, que provê meios de captar, organizar e gerenciar informações sobre sistemas legados. O ambiente possui três componentes principais: o construtor, o bibliotecário e o patrocinador, responsáveis por construir, alimentar e utilizar a *bookshelf* (“estante de livros”), respectivamente. O construtor monta a arquitetura da *bookshelf* por meio de mecanismos automáticos de coleta, estruturação e armazenamento de informações que satisfaçam o bibliotecário. Para isso, utiliza-se de *parsers*, analisadores, conversores e visualizadores que permitem que o bibliotecário alimente o repositório a partir de uma grande variedade de fontes de informação. O bibliotecário alimenta a *bookshelf* com informações específicas de um determinado sistema. Utiliza como fontes de informação os códigos fonte e documentação externa (disponível em papel ou de forma eletrônica) tal como dados de teste, registro de defeitos, informação arquitetural, registros de manutenção, etc. Ele deve selecionar os dados que são úteis ou não ao propósito específico de reengenharia, sendo parcialmente auxiliado pelo ambiente. O patrocinador é um usuário final da *bookshelf*, podendo ser um desenvolvedor, um gerente ou qualquer pessoa que precise de mais detalhes para fazer a reengenharia do código legado. Depois que a *bookshelf* estiver alimentada, o patrocinador pode editar o conteúdo existente, adicionar notas e evidenciar pontos chave. Pode também acrescentar novas informações a partir das armazenadas no repositório e da execução de ferramentas de análise e visualização do ambiente.

Armstrong, em [Arm 98], compara cinco ferramentas para recuperação da arquitetura de sistemas legados: *Rigi*, *Dali workbench*, *Software Bookshelf*, *CIA* e *SniFF+*. O *Rigi* e o *Software Bookshelf* são brevemente apresentados nesta mesma seção. O *Dali workbench* é uma ferramenta que ajuda na interpretação de dados extraídos a partir de informações arquiteturais, combinando-os com outros dados já armazenados em um repositório, usando o sistema de visualização do

Rigi. O *CIA* é uma base de dados relacional usada para extrair e armazenar informações sobre código em C, juntamente com uma ferramenta de consulta e visualização. O *SNiFF* é um ambiente de programação extensível e escalonável com recursos de *parsing* e recuperação de informação. Todas as ferramentas comparadas são construídas para a linguagem C. Elas foram analisadas com relação à sua habilidade em efetuar a extração de dados, a classificação e a visualização dos modelos. Três sistemas reais foram utilizados na comparação, sendo que um deles foi especialmente criado para causar dificuldades em *parsers* comuns. A capacidade de extração das ferramentas pôde ser testada com relação à chamada de rotinas de biblioteca escondidas, variáveis globais externas, recursão, compilação condicional, variáveis locais e funções com o mesmo nome, etc. A ferramenta *SNiFF* provou ser a melhor nesse quesito, fornecendo um nível de detalhe suficiente para a análise arquitetural. A capacidade de classificação das ferramentas, isto é, de combinação de diversos fatos de baixo nível em objetos mais abstratos, foi também testada. Nesse quesito as ferramentas *Software Bookshelf* e *Dali* mostraram alguma habilidade de abstração para níveis mais altos. A capacidade de visualização, essencial para o entendimento da arquitetura do sistema, existe em todas as ferramentas comparadas, havendo características espalhadas por todas elas que juntas formariam a ferramenta ideal para visualização.

2.3.3 - Conclusões

A engenharia reversa é de grande importância na manutenção de sistemas, facilitando também sua futura reengenharia. Unindo-se ao entendimento de programas, a engenharia reversa produz excelentes resultados de grande utilidade para o engenheiro de software responsável por alterações, reuso e evolução do sistema. Quando os modelos por ela obtidos seguem o paradigma da orientação a objetos, ainda mais vantagens são oferecidas, principalmente quanto à facilidade de reengenharia com mudança de orientação do sistema.

A engenharia reversa pode ser realizada com base nos diferentes métodos para desenvolvimento de sistemas, sendo facilitada quando o método adotado apresenta transição suave entre as fases de análise e projeto e de projeto e implementação. Por exemplo, se projeto e implementação possuem correspondência natural, fica mais fácil caminhar no sentido contrário, como é necessário fazer na engenharia reversa. No método “Fusion” para desenvolvimento de sistemas orientados a objetos [Col 94], essa transição é bastante suave, por isso o “Fusion/RE”

[Pen 96] é usado com facilidade. Mesmo considerando que o sistema legado não foi desenvolvido orientado a objetos, essa vantagem continua a existir. Deve-se lembrar também que, no “Fusion/RE”, obtém-se os modelos de análise sem passar pelos modelos de projeto.

2.4 - Reengenharia

2.4.1 - Introdução

A reengenharia tem por finalidade examinar e alterar um sistema existente para reconstituí-lo em uma nova forma e depois implementá-lo na nova forma [Chi 90]. A reengenharia tem como objetivo principal melhorar a qualidade global do sistema, mantendo, em geral, as funções do sistema existente. Mas, ao mesmo tempo, pode-se adicionar novas funções e melhorar o desempenho [Pre 95]. Segundo Jacobson [Jac 91], a reengenharia consiste da engenharia reversa, seguida de mudanças no sistema (que podem ser mudanças de funcionalidade ou mudanças de técnica de implementação) e seguida da engenharia avante. Ou seja, “reengenharia é o processo de criar uma descrição abstrata do sistema, elaborar mudanças em alto nível de abstração e então implementá-las no sistema”. De acordo com Sneed [Sne 95a], a reengenharia engloba a reengenharia do procedimento empresarial, a reengenharia dos dados, a reengenharia do software e a reciclagem (que produz componentes reusáveis, de maneira análoga ao processo de retirada de peças aproveitáveis de um automóvel abandonado).

Uma das grandes motivações para aplicação da reengenharia é a diminuição dos altos custos de manutenção de sistemas, que devem-se a diversos fatores, discutidos por Wilkening em [Wil 95]. A manutenção contínua faz com que a implementação fique inconsistente com o projeto original, o código torne-se difícil de entender e sujeito a erros, além da documentação desatualizada. As linguagens em que esses sistemas foram implementados estão ultrapassadas, não havendo suporte por parte dos fabricantes nem tampouco programadores que as dominem, além de não haver ferramentas “CASE” que as suportem. Esses sistemas foram desenvolvidos sem seguir os preceitos da engenharia de software, tendo, na maioria das vezes, um código desestruturado e difícil de entender. A alta rotatividade de funcionários faz com que muita informação seja perdida, diminuindo o conhecimento existente sobre o sistema.

Muitas empresas possuem programas ou sistemas passíveis de reengenharia, mas deve-se avaliar cautelosamente os prós e contras de efetuar a reengenharia. Programas pouco usados, que não sofrem mudanças, não compensam o custo da reengenharia. Programas que sofrem mudanças constantes podem passar pela reengenharia como forma de manutenção preventiva, para evitar problemas futuros. O custo para a reengenharia de um sistema pode ser compensado, a longo prazo, pela melhoria da manutenibilidade e reuso.

A seção seguinte relata contribuições consideradas relevantes para a reengenharia, preferencialmente aquelas cujo enfoque seja a mudança de paradigma para a orientação a objetos. Nota-se que muitos artigos sobre reengenharia tratam, com mais destaque, ou de entendimento de programas, ou de engenharia reversa ou de padrões. Esses artigos foram discutidos nas seções respectivas desses assuntos, como por exemplo [Nin 94] (seção 2.2.2) e [Gal 96] (seção 2.5.2). Artigos importantes sobre reengenharia, como [Wel 95] e [Sem 95], não são aqui incluídos porque não tratam especificamente da reengenharia com mudança de orientação.

2.4.2 - Contribuições Relevantes

Jacobson, em [Jac 91], apresenta uma técnica para efetuar a reengenharia de sistemas legados, implementados em uma linguagem procedimental como C ou Cobol, obtendo sistemas orientados a objetos. Mostra como fazer essa reengenharia de forma gradual, pois considera impraticável substituir um sistema antigo por um completamente novo (o que exigiria muitos recursos). Considera três cenários diferentes: no primeiro se faz mudança de implementação sem mudança de funcionalidade; no segundo se faz a mudança parcial da implementação sem mudança de funcionalidade; e no terceiro se faz alguma mudança na funcionalidade. Apresenta três estudos de caso: um sistema de peças sobressalentes, um sistema de telecomunicações e um sistema de controle de tráfego. Usa uma ferramenta Case específica para orientação a objetos, o “*ObjectOry*”. Nota-se que o autor não justificou o porquê de passar para a orientação a objetos, tendo apenas usado o termo “modernização do sistema”.

Markosian, em [Mar 94], reclama da falta de apoio computadorizado para a reengenharia de sistemas, em contraposição à grande proliferação de ferramentas CASE para desenvolvimento de software novo. Diz que as ferramentas de transformação atuais são muito limitadas, sendo difícil a adaptação a um projeto em particular. Aborda uma nova tecnologia para reengenharia, que chama de “tecnologia facilitadora” (“*enabling technology*”), relatando resultados práticos

bastante animadores quanto à produtividade da sua aplicação. A “tecnologia facilitadora” consiste no rápido desenvolvimento de ferramentas para analisar e modificar, de forma sistemática, sistemas existentes. Deve ser usada em tarefas complexas de reengenharia, que estejam sendo feitas à mão ou com automatização parcial. Usa como exemplo ilustrativo de aplicação dessa tecnologia o sistema da “Boeing Computer Services”, que mantém muitos sistemas legados grandes, como por exemplo um sistema de folha de pagamento com vinte e dois anos e 650.000 linhas de código Cobol. Explica detalhadamente o processo de modularização desse sistema, para torná-lo compatível com tipos abstratos de dados. Usa para efetuar essa modularização duas ferramentas que incorporam a “tecnologia facilitadora” de reengenharia, a Software RefineryTM e a REFINE/CobolTM, da empresa “Reasoning Systems”. A ferramenta Software Refinery foi usada para estender a ferramenta REFINE/Cobol a fim de executar a modularização. Essas ferramentas oferecem uma base de dados orientada a objetos para modelar o software e suas informações, uma linguagem de especificação executável de muito alto nível, juntamente com um compilador, auxílio para depuração e uma biblioteca “*run-time*” com utilitários independentes de linguagem. Suas saídas são: o grafo de chamadas do programa, análise “*set/use*”, grafo de fluxo de controle e modelo de dados. O trabalho de modularização usando essas ferramentas mostrou-se produtivo, tendo-se detectado menos erros do que na modularização feita à mão.

Sage, em [Sag 95], discute os possíveis tipos de reengenharia que podem ser considerados: *reengenharia de produto, de processo e de gestão de sistemas*. A **reengenharia de produto** é definida como sendo o exame, estudo, captação e modificação dos mecanismos internos ou da funcionalidade de um produto ou sistema existente, para reconstitui-lo em uma nova forma e com novas características, geralmente para desfrutar de vantagens de novas tecnologias emergentes, mas sem maiores mudanças na funcionalidade e propósitos do sistema. A **reengenharia de processo** é o exame, estudo, captação e modificação dos mecanismos internos ou da funcionalidade de um processo existente, ou ciclo de vida de engenharia de sistemas, para reconstitui-lo em uma nova forma e com novas características funcionais e não funcionais, geralmente para desfrutar de capacidades organizacionais ou tecnológicas novas ou desejáveis, mas sem mudar o propósito inerente ao processo em si. A **reengenharia de gestão de sistemas** é o exame, estudo, captação e modificação dos mecanismos internos ou da funcionalidade de processos e práticas de gestão de sistemas existentes em uma organização, para reconstitui-los em uma nova forma e com novas características, geralmente para desfrutar de

requisitos de competitividade organizacional emergentes, mas sem mudar o propósito inerente à organização em si. Segundo Sage, qualquer que seja o tipo de reengenharia, sempre envolve um ciclo de vida de três fases: definição, desenvolvimento e disposição. Na primeira fase, são definidos os objetivos da reengenharia, obtendo-se um conjunto de requisitos para a entidade a passar pela reengenharia. Na segunda fase são determinadas as necessidades arquiteturais e é feito o projeto da entidade a passar pela reengenharia, que é testada e avaliada. Finalmente, na terceira fase, é implementada a reengenharia da entidade, de forma operacional, garantindo-se que a manutenção possa ter continuidade. Sage discute detalhadamente todos os tipos de reengenharia, dando uma visão geral da literatura atual sobre o assunto. Cita os fatores que devem ser levados em conta ao decidir pela reengenharia, os riscos que devem ser tratados durante a mesma e os fatores para aumentar a chance de sucesso na reengenharia.

Wilkening, em [Wil 95], apresenta um processo para efetuar a reengenharia de sistemas legados, aproveitando partes de sua implementação e projeto. Esse processo inicia-se com a reestruturação preliminar do código fonte, para fazer algumas melhorias no mesmo, como remoção de construções não estruturadas, código “morto” e tipos implícitos. A finalidade dessa reestruturação preliminar é produzir um programa fonte mais fácil de analisar, entender e reestruturar. Em seguida, o código fonte produzido é analisado e são construídas representações do mesmo em níveis mais altos de abstração. Com base nisso, pode-se prosseguir com os passos de reestruturação, re-projeto e redocumentação, que são repetidos quantas vezes forem necessárias para se obter o sistema totalmente reestruturado. Pode-se, então, gerar o programa na linguagem destino e dar seqüência aos testes que verificarão se a funcionalidade não foi afetada. Wilkening apresenta a ferramenta RET, que automatiza parcialmente esse processo, sendo portanto uma reengenharia assistida por computador. Essa ferramenta foi instanciada para sistemas cuja linguagem original é FORTRAN, e que após a reengenharia são transformados, automaticamente, para a linguagem ADA. Com a RET, o engenheiro de software não tem que se preocupar com detalhes sintáticos, como por exemplo as diferenças de sintaxe entre a linguagem fonte e destino, podendo dar mais atenção a decisões de projeto e implementação que requerem ajuda humana.

Klösh, em [Klo 96], discute uma abordagem para reengenharia com mudança de orientação, na qual aplica primeiramente a engenharia reversa para depois mudar a linguagem para uma linguagem orientada a objetos. Esse trabalho é uma continuação do trabalho de Gall [Gal 95], no qual é apresentado o método COREM para transformação de programas, que refaz

sistemas de arquitetura procedimental, tornando-os orientados a objetos. Klösh justifica o uso da orientação a objetos como forma de melhorar a manutenibilidade futura, devido a conceitos como abstração, encapsulamento e passagem de mensagens. É contrário à utilização de herança e polimorfismo, por acreditar que esses conceitos complicam potencialmente as operações de manutenção. A abordagem proposta por Klösh possui quatro passos: recuperação de projeto, modelagem da aplicação, mapeamento dos objetos e adaptação do código fonte. No primeiro passo são recuperados modelos do projeto, como por exemplo o diagrama entidade-relacionamento. Esse é transformado em um modelo reverso orientado a objetos da aplicação (RooAM), que é um modelo obtido de forma direta, sem considerar aspectos dinâmicos como serviços e conexão de mensagens. No segundo passo é construído o modelo progressivo orientado a objetos da aplicação (FooAM), usando-se algum dos métodos disponíveis para análise orientada a objetos. Nesse passo há interferência de um especialista humano que seja experiente no domínio de aplicação ou que tenha participado do desenvolvimento do sistema em questão. No terceiro passo é feita a correspondência entre o RooAM e o FooAM, para resolver as possíveis ambigüidades surgidas durante o passo de recuperação de projeto. No quarto passo é feita a adaptação da implementação procedimental para a arquitetura orientada a objetos. Klösh concorda com a automatização de partes do processo de reengenharia, mas acredita que seja melhor uma abordagem híbrida, que use a assistência automática de uma ferramenta computadorizada mas que, quando ela alcançar seus limites, conte com a intervenção da sabedoria humana para superá-los.

Sneed, em [Sne 96], descreve um processo de reengenharia apoiado por uma ferramenta para extrair objetos a partir de programas existentes em COBOL. Ressalta a predominância da tecnologia de objetos nos dias de hoje, principalmente em aplicações distribuídas com interfaces gráficas, questionando a necessidade de migração de sistemas legados para essa nova tecnologia. Identifica alguns obstáculos à reengenharia orientada a objetos, como por exemplo a identificação dos objetos, a natureza procedimental da maioria dos sistemas legados, que leva a blocos de código processando muitos objetos de dados, a existência de código redundante e a utilização arbitrária de nomes. Assume como pré-requisitos para a reengenharia orientada a objetos a estruturação e modularização dos programas, além da existência de uma árvore de chamadas do sistema. É apresentado um processo de reengenharia orientada a objetos composto de cinco passos: seleção de objetos, extração de operações, herança de características, eliminação de redundâncias e conversão de sintaxe. A seleção de objetos é feita com apoio da ferramenta,

mas o responsável pela determinação dos objetos é o engenheiro de software. A extração de operações particiona o programa em subrotinas, removendo os segmentos referentes a um determinado objeto e substituindo-os por envio de mensagens ao objeto no qual os dados locais estiverem encapsulados.

2.4.3 - Conclusões

Fazer ou não reengenharia em um sistema legado é uma decisão que deve ser tomada com muito cuidado. Sistemas que estão operando de forma satisfatória não devem ser necessariamente substituídos, mesmo que os novos sistemas prometam ser melhores. Na verdade, a melhora teria que ser significativamente grande para compensar o risco introduzido [Sne 96].

A transformação de programas procedimentais em programas orientados a objetos não é trivial, requerendo intervenção humana para escolher os objetos [Sne 96]. A quantidade dessa intervenção pode diminuir se houver integração da reengenharia com padrões predefinidos em vários níveis de abstração, tais como padrões do domínio de aplicação, padrões de projeto ou padrões de linguagem [Klo 96].

Ferramentas de apoio à reengenharia ajudam a diminuir o trabalho tedioso de transformação de linguagens, deixando o engenheiro de software com tempo livre para tarefas mais nobres.

2.5 - Padrões

2.5.1 - Introdução

Segundo o dicionário Aurélio [Fer 95], padrão é aquilo que serve de base ou norma para a avaliação de qualidade ou quantidade; ou qualquer objeto que serve de modelo à feitura de outro; ou modelo, exemplo, protótipo, arquétipo. Essas definições aplicam-se a diversos campos, como por exemplo na música, na literatura, na arte, na psicologia, na costura, na decoração, no jogo de xadrez e na aviação [Coa 92]. Na engenharia de software, desenvolvedores têm tentado encontrar padrões que venham ao encontro dos planos, algoritmos, estruturas de dados e idiomas

aprendidos no passado [Gam 95]. Padrões têm sido usados para descrever soluções para um problema repetido de projeto [Vli 95], ou princípios de projeto que se mostraram úteis no desenvolvimento de software [Gal 96]. Estudos mostram que quando especialistas trabalham em um problema particular é raro que inventem uma nova solução para atacá-lo, completamente diferente das já existentes. Diversas soluções de projeto são por eles conhecidas, de acordo com a própria experiência ou a de outros profissionais. Quando confrontam-se com novos problemas, freqüentemente lembram-se de outros similares e reusam a solução antiga, pensando em pares “problema/solução”. Esses pares podem ser agrupados em famílias de problemas e soluções similares, sendo que cada família exibe um padrão tanto de problema quanto de solução [Bus 97]. Projetistas familiarizados com certos padrões podem aplicá-los imediatamente a problemas de projeto, sem ter que redescobri-los [Gam 95].

Os padrões têm pelo menos dois propósitos: primeiro, fornecem exemplos a serem seguidos e artifícios a serem copiados e posteriormente refinados ou estendidos. Segundo, eles garantem uniformidade na estrutura do software. Há, portanto, um considerável aumento de produtividade no desenvolvimento e manutenção do software [Gal 96].

“Padrões de projeto” são definidos como descrições de objetos e classes que se comunicam, os quais são ajustados para resolver um problema genérico de projeto, em um contexto particular. Um padrão de projeto identifica as classes e instâncias participantes, juntamente com seus papéis, colaborações e distribuição de responsabilidade [Gam 95]. Padrões de projeto podem ser considerados micro-arquiteturas reutilizáveis que contribuem para a arquitetura geral de um sistema [Gam 93]. Padrões de projeto são definidos como “*cliches*” adicionados de sugestões para seu devido uso na construção de software; eles embutem o conhecimento do especialista, representando uma solução para um problema comum de projeto e podem ser reusados freqüentemente e facilmente [Kra 96].

O processo de desenvolvimento de software orientado a objetos pode ser facilitado pelo uso de padrões de projeto. Eles proporcionam um vocabulário comum para a comunicação entre projetistas, criando abstrações num nível superior ao de classes e instâncias. A construção de software reutilizável é incentivada, bem como o reuso do conhecimento de projeto obtido por projetistas experientes. O tempo gasto para aprendizado de uma biblioteca de padrões é reduzido, ajudando um noviço a agir como um especialista [Gam 93].

2.5.2 - Contribuições Relevantes

Rich, em [Ric 90], define “*clichês*” como estruturas de programação ou algoritmos comumente usados no desenvolvimento de sistemas. Aponta como vantagens do reconhecimento automático de clichês o aumento da facilidade de manutenção, documentação, evolução, otimização e depuração de software. Além disso, ajuda no estudo de como representar e usar o conhecimento e experiência de programação. Apresenta o protótipo do “Recognizer”, uma ferramenta que encontra, automaticamente, todas as ocorrências de determinados tipos de clichês em um programa e com base nisso, constrói uma descrição hierárquica do programa. O “Recognizer” possui um módulo dependente da linguagem, que traduz o código fonte para uma representação gráfica chamada “Plan Calculus”, produzindo uma árvore de projeto. Aponta algumas características dos clichês que dificultam seu reconhecimento automático, como a variação sintática e de implementação, a não contiguidade e a sobreposição de implementações. Mostra, depois, por meio de exemplos escritos na linguagem LISP, como o “Recognizer” supera essas dificuldades.

Figura 2.7 - Padrão “Roles-played” (papéis desempenhados) e exemplo [Coa 92]

Coad, em [Coa 92], aborda o uso de padrões na análise orientada a objetos e no projeto orientado a objetos. Define padrões e dá algumas dicas de como encontrá-los, especialmente para análise e projeto orientados a objetos. Lembra que os métodos orientados a objetos já destacam

certos padrões de relacionamentos, como a generalização-especialização, agregações, associação e mensagens. Diferencia padrões de “*frameworks*” de aplicação, que são esqueletos de classes, objetos e relacionamentos agrupados para construir aplicações específicas. Define um padrão orientado a objetos como uma abstração de uma dupla, tripla ou pequeno agrupamento de classes que seja útil muitas vezes no desenvolvimento orientado a objetos. Apresenta sete desses padrões, explicando-os de forma textual e por meio de uma figura, bem como dando um exemplo ilustrativo e normas para utilizá-los. No final, dá um exemplo de um modelo maior que usa seis padrões.

A figura 2.7 mostra um dos padrões apresentados por Coad, o “*roles-played*” (papéis desempenhados). Outros de seus padrões são mostrados na seção 5.2. Um objeto “ator” possui atributos e serviços que valem sempre, podendo, às vezes, desempenhar um ou mais papéis. Para cada papel pode haver especializações, de acordo com os atributos e serviços necessários para desempenhá-lo. No exemplo, “pessoa” tem os atributos nome e telefone, mas pode, em um certo instante, desempenhar o papel de cliente ou empregado. No caso de cliente, possui um atributo extra, “desconto”, bem como um serviço extra, “calcular valor desconto”. No caso de empregado, possui um atributo extra, “número”.

Coplien, em [Cop 92], define *idiomas* e *estilos de linguagem* como recurso para aumentar a expressividade de programas em C++ e dar ao software uma estrutura melhor e mais eficiente. Segundo ele, muitas tarefas de programação tornaram-se idiomáticas e devem ser incorporadas naturalmente à funcionalidade do C++, dando a ilusão de que fazem parte da linguagem. Características do C++ para obtenção de blocos construtivos, como classes e funções-membro, são largamente utilizadas na definição dos idiomas. A “forma canônica ortodoxa” é um idioma que define princípios e padrões envolvidos no funcionamento de um objeto. Ela faz parte dos idiomas que fazem com que uma classe se torne um tipo completo, sendo um padrão a ser seguido na implementação de qualquer classe. Um classe nessa forma contém sempre: um construtor *default*, que trata da construção da classe quando um objeto é definido sem parâmetros; um construtor *copy*, que cria um objeto baseado em uma cópia do parâmetro de entrada fornecido na chamada; um operador de atribuição, para transferir o conteúdo de um objeto para outro objeto; e um destrutor, para liberar os recursos adquiridos pelo objeto durante sua construção. Esse idioma deve ser usado sempre que objetos de uma classe precisarem ser atribuídos ou passados como parâmetros (por valor) de uma função e o objeto contiver ponteiros

para objetos contados por referência, ou o destrutor da classe precisar executar um “*delete*” em um objeto.

O idioma “função operador de membro” é usado para converter um tipo declarado pelo usuário para um tipo nativo do “C”. O idioma “classe manuseador/corpo” (*handle/body class*) é usado para decompor uma abstração complexa em classes menores e portanto mais fáceis de manusear. O idioma “contagem referencial” (*reference counting*) é um caso particular desse último idioma, no qual a classe “corpo” contém um contador referencial manipulado pela classe “manuseador”. Ele é útil quando são feitas cópias freqüentes de um objeto grande ou complexo, pois permite a cópia lógica, que é menos dispendiosa. Outros idiomas são definidos por Coplien, todos específicos da linguagem C++ e num nível de abstração baixo como os citados acima.

Figura 2.8 - Diagrama do padrão “Abstract Factory” [Gam 93]

Gamma, em [Gam 93, Gam 95], propõe o uso de padrões de projeto como um novo mecanismo para registrar a experiência conseguida com projetos anteriores. Descreve como expressar padrões, sugerindo o uso de um gabarito (“*template*”) para estruturar informações e garantir uniformidade. Organiza os padrões de projeto, categorizando-os em termos de jurisdição e caracterização. Jurisdição é o domínio sobre o qual um padrão se aplica, podendo ser classe, objeto ou composto. Caracterização diz respeito a “o quê” o padrão faz, podendo ser: padrões criativos, que tratam do processo de criação de objetos; padrões estruturais, que lidam com a composição de classes ou objetos e padrões comportamentais, que caracterizam as formas pelas quais classes ou objetos interagem e distribuem responsabilidade. Introduce um catálogo com vinte e três padrões de projeto, explicando detalhadamente três deles: o “*Abstract Factory*”, o “*Strategy*” e o “*Wrapper*”. Relata sua experiência com padrões no projeto e construção de dois

sistemas, uma ferramenta de análise financeira e uma ferramenta de manipulação de restrições. Observa que padrões de projeto motivam os desenvolvedores a irem além dos objetos concretos, isto é, eles objetivam conceitos do domínio do problema que não são tão aparentes como objetos. Comenta a similaridade de seu trabalho com o de Coad [Coa 92], ressaltando que os padrões de Coad são mais próximos da análise que do projeto, pois surgem naturalmente na modelagem do sistema. O único padrão que coincide é o “Broadcast”, que é o mesmo que seu padrão “Observer”.

Figura 2.9 - Exemplo do padrão “Abstract Factory” [Gam 93]

A figura 2.8 mostra o diagrama que representa graficamente o padrão “*Abstract Factory*”. O diagrama é baseado na notação OMT (“*Object Modeling Technique*”), à qual acrescentou-se o pseudo-código dos métodos. A classe “*AbstractFactory*” generaliza as “*ConcreteFactories*”, que por sua vez dão origem aos produtos específicos. Esses são por sua vez generalizados pelas classes “*GenericProducts*”. A figura 2.9 mostra uma instanciação desse padrão a uma aplicação cuja interface contém janelas e barras de rolagem de dois tipos diferentes: “*Motif*” e “*Open Look*”. Nesse caso, deseja-se que o cliente seja independente desses tipos, isto é, que o acesso às classes concretas seja feito apenas pela sua interface genérica (a classe abstrata).

Vlissides, em [Vli 95], diferencia engenharia reversa de “arquitetura reversa”, dizendo que a primeira analisa um sistema de software para recuperar seu projeto, enquanto a segunda analisa diversos sistemas de software para recuperar projetos repetidos e o raciocínio envolvido neles. Na verdade, usa “arquitetura reversa” como um termo mais ousado para o desenvolvimento de padrões de projeto, discutido em [Gam 95]. Mostra, inicialmente, um

exemplo de padrão de projeto tirado desse livro, mais especificamente o “*Strategy*”. Considera que a reflexão é a atividade mais importante na arquitetura reversa. Deve-se reservar um tempo para refletir sobre os sistemas construídos, os problemas que surgiram e as soluções encontradas para resolvê-los (ou não). Tudo deve ser anotado, pois esse será o material básico para extração de padrões. Discute alguns princípios para a escrita de padrões, recomendando, também, o uso de um gabarito para uniformizar a estrutura da informação e facilitar a aprendizagem, comparação e utilização dos padrões de projeto.

Budingsky, em [Bud 96], apresenta uma ferramenta para gerar código automaticamente a partir de padrões de projeto de software, constituída de três componentes: um apresentador, um gerador de código e um módulo para estabelecer correspondência entre os outros dois. O apresentador usa páginas semelhantes às da Web para colocar à disposição do usuário as informações sobre padrões de software disponíveis em [Gam 95]. O gerador de código produz fragmentos de código semelhantes aos exemplos de código que ilustram os padrões em [Gam 95]. Cabe ao usuário escolher em cada caso de geração de código, qual alternativa de implementação deseja adotar entre as mencionadas em [Gam 95], guiando-se pelos “*trade-offs*” lá mencionados. Para tornar esta escolha mais prática, a ferramenta exibe esses “*trade-offs*” na hora da escolha. O módulo que estabelece correspondência entre o apresentador e o gerador de código tem utilidade transparente ao usuário. A ferramenta permite maior flexibilidade para o usuário de padrões, que pode mudar decisões de projeto e ter seu código automaticamente reescrito pela ferramenta, bastando para isso escolher os “*trade-offs*” correspondentes. Ao usuário cabe fornecer informações específicas da aplicação, de modo que o código gerado seja coerente com ela. A ferramenta também é útil para que se possa dominar a aplicação de padrões de software dinamizando a necessária experimentação que com eles deve ser realizada.

Gall, em [Gal 96], comenta a aplicação de padrões na reengenharia, pela identificação e utilização de conceitos reusáveis. Explica que o COREM, um método por ele desenvolvido para reestruturação arquitetural, atualmente requer a intervenção de um especialista humano para fornecer informações perdidas que não podem ser recuperadas de forma automática. Entretanto, diz que está sendo feita uma extensão ao COREM, pela integração de conceitos de padrão ao longo de alguns passos do processo, para reduzir essa intervenção humana. Discute a utilização de padrões na engenharia de software, em diversos níveis de abstração, desde a análise até a codificação. Utiliza os exemplos de padrões apresentados por Coad [Coa 92]. A figura 2.10 mostra as principais atividades no processo do COREM no contexto de camadas de abstração. A

seta contínua representa a engenharia avante, enquanto que a seta pontilhada representa a engenharia reversa. Gall lembra que o uso de padrões orientados a objetos na engenharia avante têm-se mostrado útil e avalia sua adequação à melhoria do processo de engenharia reversa. Para isso, são propostas duas estratégias diferentes: busca por padrões orientados a objetos no código fonte guiada pelos padrões e guiada pelo código fonte. A primeira, parte de um certo padrão e vasculha o código, procurando sua ocorrência. A segunda, parte das estruturas de dados presentes no código fonte e tenta fazer a correspondência com padrões existentes. Conclui que o conhecimento semântico da aplicação é necessário, não sendo possível a automação total. Porém, uma ferramenta pode auxiliar o engenheiro de software, fazendo as tarefas monótonas e oferecendo uma interface sofisticada para o registro de conhecimento semântico da aplicação.

Figura 2.10 - COREM e tipos de padrão básicos [Gal 96]

Krämer, em [Kra 96], propõe a recuperação de projeto pela busca automática por padrões estruturais de projeto em software orientado a objetos. Afirma que a localização de padrões estruturais de projeto, como “*Adapter*”, “*Bridge*”, “*Composite*”, “*Decorator*” e “*Proxy*” (introduzidos por Gamma [Gam 95]), em software produzido sem uso explícito de padrões, pode melhorar sua manutenibilidade, porque pedaços maiores do software podem ser entendidos como um todo. Apresenta uma ferramenta, o sistema “Pat”, que busca por instâncias de padrões de projeto em software existente, implementado em C++. Resumidamente, a abordagem cria representações em PROLOG tanto dos padrões quanto das informações de projeto extraídas do

código-fonte. Então um motor de inferência PROLOG faz a busca, necessitando de pós-processamento manual para remover falsos positivos (instâncias ilegítimas). A figura 2.11 mostra a arquitetura do sistema “Pat”. Krämer utilizou a ferramenta CASE “Paradigm Plus” para fazer a análise estrutural do código, obtendo nomes de classes, atributos, métodos e propriedades, relações de herança, associação e agregação, embora com algumas limitações. Avaliou quatro programas aplicativos, sendo dois sistemas de comunicação e duas bibliotecas de classes largamente usadas. Comparou os resultados obtidos quanto ao número de instâncias verdadeiras e à precisão, tendo obtido precisão entre 14 e 50 por cento de padrões identificados.

Figura 2.11 - Arquitetura do sistema “Pat” [Kra 96].

Campo, em [Cam 97], apresenta uma abordagem para engenharia reversa com base no reconhecimento e visualização de padrões de projeto presentes em um dado *framework*. Fala das vantagens do uso de *frameworks* no aumento de produtividade e qualidade do desenvolvimento de software. Por outro lado, lembra que, para obter o benefício máximo do reuso de *frameworks*, é necessário entender o projeto interno de suas classes, a colaboração entre elas e a forma pela qual instâncias dessas classes colaboram em tempo de execução. Esse entendimento é uma tarefa dispendiosa e demorada. Em *frameworks* complexos, desenvolvem-se estruturas de projeto flexíveis, para possibilitar adaptações dinâmicas. Essa flexibilidade acarreta projetos complexos, difíceis de entender e, conseqüentemente, difíceis de reusar. O objetivo maior dos *frameworks* é o reuso do conhecimento que o projetista possui sobre o domínio de aplicação. A identificação de padrões de projeto embutidos na estrutura do *framework* é importante para oferecer ao usuário visões mais abstratas dessa estrutura, no nível arquitetural. Facilita-se, assim, o reuso, sem prejuízo do entendimento do programa. A abordagem proposta por Campo é baseada no *framework* “Luthier”, para construção de ferramentas para análise e visualização de aplicações.

Prolog é usada na representação das regras para reconhecimento de padrões, construção de visualizações de padrões em potencial e explicações sobre esses padrões, e razões pelas quais sugere-se sua presença no projeto.

Bosch, em [Bos 97], identifica quatro problemas encontrados na implementação de padrões de projeto usando linguagens de programação orientadas a objeto tradicionais. O primeiro problema é a identificação de padrões de projeto no código, pois quase sempre os padrões ficam perdidos durante a implementação, já que as linguagens de programação não oferecem o conceito correspondente a padrão. Assim, os padrões ficam espalhados em partes de um objeto ou múltiplos objetos. O segundo problema é o “auto-problema”. Ele ocorre porque objetos que já não mais existem podem ser referenciados quando uma mensagem é passada adiante e posteriormente delegada a outros objetos. O terceiro problema é a limitação da reusabilidade, pois a implementação do padrão de projeto acaba misturando-o com o domínio de aplicação, fazendo com que apenas o projeto possa ser reusado. O quarto problema é a sobrecarga de implementação, devido ao grande número de métodos simples gerados, geralmente com comportamento trivial. Para tentar resolver tais problemas, propõe um modelo de objetos em camadas, chamado “*LayOM*”, que é uma linguagem orientada a objetos estendida. LayOM permite a representação explícita de padrões de projeto na linguagem de programação. Ilustra como um padrão de projeto pode ser implementado por camadas que encapsulam o objeto, de forma que mensagens enviadas ou recebidas pelo objeto tenham que passar por essas camadas. Essa abordagem é aplicada a oito dos padrões de projeto catalogados por Gamma [Gam 95].

Fowler, em [Fow 97], apresenta setenta e seis padrões de análise que “refletem estruturas conceituais de processos de negócios, ao invés de implementações reais de software”. Define padrão como uma idéia que tem sido útil em um contexto prático e que provavelmente será útil em outros. Diz que um padrão, mesmo que ainda não tenha sido escrito, tem quatro partes essenciais: o estabelecimento do “contexto” no qual ele é útil; o “problema” por ele endereçado; as “forças” que regem a formação da solução; e a “solução” que soluciona aquelas forças. Seus padrões são baseados em sua experiência pessoal de aplicação de modelagem orientada a objetos a grandes sistemas de informação corporativos. Entre os padrões por ele apresentados estão: o *Accountability*, que define responsabilidades entre partes, como por exemplo contratos formais ou informais; o *Observation*, para lidar com informações qualitativas dos objetos; o *Measurement*, para lidar com informações quantitativas dos objetos; o *Inventory* e o *Accounting*, que descrevem como uma rede de contas e regras de lançamento podem formar um sistema de

contabilidade; o *Plan*, para o planejamento e uso de recursos; o *Contract* e o *Portfolio*, para lidar com comércio de recursos, entre outros.

Martin, em [Mar 98], reúne os melhores artigos sobre padrões apresentados nas conferências PLOp'96 e EuroPLOp'96. Nesta seção alguns deles são resumidos. Segundo Martin, os padrões devem ser lidos, entendidos e incorporados ao modelo mental de projeto dos desenvolvedores, para então poderem ser utilizados quando um software específico estiver sendo projetado. Ressalta-se a importância de saber aplicar o padrão correto e de maneira correta, já que os padrões têm finalidade específica e muitas vezes confundida por seus usuários. São discutidos padrões em diversos níveis de abstração, desde os padrões de padrão, padrões de processo, padrões arquiteturais e padrões de análise, até os padrões de projeto e padrões de programação. Os padrões de padrão apresentam heurísticas sobre a forma de apresentar padrões. Os padrões de processo fornecem recomendações sobre como conduzir as várias fases de desenvolvimento de software. Os padrões arquiteturais definem propriedades globais do sistema, ajudando na especificação da estrutura fundamental de um sistema de software. Os padrões de análise e de projeto encontram-se distribuídos em diversos sub-tipos, como os padrões de persistência, padrões de distribuição, padrões de domínio específico, padrões de interface com o usuário e padrões de uso geral. Os padrões de programação descrevem soluções para problemas específicos de programação, tornando o código mais claro de entender, adaptar e modificar. Esse trabalho reforça a idéia de que ainda existem muitos padrões a serem descobertos, por exemplo nos casos de padrões de interface com o usuário e padrões de distribuição.

Johnson, em [Joh 98], descreve o padrão *Type-Object*, mostrando sua aplicação a dois sistemas: uma videolocadora e uma fábrica. Fornece o código fonte em Smalltalk referente à implementação desses exemplos particulares, no qual pode-se entender o relacionamento entre as classes que formam o padrão. Johnson apresenta diversas variações na aplicação do padrão, como por exemplo o uso de *Type-Objects* aninhados ou recursivos, em que pode-se ver a flexibilidade desse padrão. Por outro lado, comenta as consequências de seu uso, como por exemplo o aumento da complexidade do projeto e implementação e a gestão de referências. O padrão *Type-Object* é descrito na seção 5.2.

Boyd, em [Boy 98], descreve padrões para representar a associação entre objetos em sistemas de negócios. São eles: o padrão *Association-Object*, o padrão *Customer Contact* e o padrão *3-Level Order*. O primeiro, descrito na seção 5.2, é comumente encontrado em sistemas de informação, já que representa algo que acontece em um ponto específico do tempo associando

dois outros objetos. O segundo é uma variação do primeiro, sendo uma associação mais específica entre um negócio e um cliente, porém envolvendo uma terceira classe, que é responsável por controlar os diversos contatos feitos com o cliente e estabelecer regras de contato. O terceiro é uma série de associações de objetos, com uma estrutura para separá-los em grupos de responsabilidades.

Figura 2.12 - Diagrama de Interação dos Padrões [Yod 98]

Yoder, em [Yod 98], apresenta dez padrões usados na implementação de objetos de negócios, de forma que possam ser mapeados para bases de dados não orientadas a objetos. Esses padrões tentam amenizar a diferença existente entre objetos, que representam tanto dados quanto comportamento, e bases de dados relacionais, que consistem de tabelas e relacionamentos. O padrão “*Persistence Layer*” fornece uma camada para fazer a correspondência entre os objetos e a base de dados. Essa camada é o centro de interação com os demais padrões, isolando o desenvolvedor dos detalhes de armazenagem e recuperação de objetos. O padrão “*CRUD*” fornece as operações básicas utilizadas na persistência de objetos, a saber: *Create*, para criar o novo objeto na base de dados; *Read*, para ler objetos da base de dados; *Update*, para atualizar os dados do objeto na base de dados; e *Delete*, para eliminar um objeto da base de dados. São sugeridas também duas outras operações que podem ser necessárias: *LoadAllLike*, para carregar

todos os objetos que satisfazem uma dada condição e *LoadAll*, para carregar todos os objetos de uma classe. O padrão “*SQL Code Description*” é usado para construir as chamadas SQL para a base de dados. O padrão “*Attribute Mapping Methods*” faz a correspondência entre os valores na base de dados e os atributos da classe e vice-versa. O padrão “*Type conversion*” faz a transformação de valores da base de dados para o tipo correspondente no objeto e vice-versa, assegurando a integridade dos dados. O padrão “*Change Manager*” acompanha as mudanças efetuadas nos valores dos objetos para que fiquem consistentes com a base de dados. Ele determina a necessidade de escrever ou não os valores para a base de dados. O padrão “*OID Manager*” gera chaves únicas para identificar os objetos durante uma inserção. O padrão “*Transaction Manager*” fornece um mecanismo para manipular transações ao salvar objetos. O padrão “*Connection Manager*” estabelece e mantém a conexão com a base de dados. O padrão “*Table Manager*” gerencia o mapeamento entre um objeto e suas tabelas e colunas na base de dados. O diagrama da figura 2.12 mostra a interação entre esses dez padrões, que juntos estabelecem um mecanismo para mapear objetos persistentes para uma base de dados.

2.5.3 - Conclusões

Os padrões de projeto desempenham diversos papéis no processo de desenvolvimento orientado a objetos. Primeiro, eles fornecem ao projeto um vocabulário comum. Segundo, eles reduzem a complexidade do sistema, nomeando e definindo abstrações. Terceiro, eles constituem uma base de experiência para construção de software reutilizável. Por último, eles atuam como blocos construtivos a partir dos quais projetos mais complexos podem ser construídos [Gam 93].

Os padrões de projeto documentam uma parte repetida de um projeto orientado a objetos de forma que permitem seu entendimento e aplicação em um contexto particular. Com os padrões, pode-se discutir o projeto em um nível de abstração maior do que classes e objetos. Assim, as informações sobre o projeto podem ser divulgadas de forma mais rápida e precisa, facilitando tanto o processo de projeto quanto a documentação [Vli 95].

2.6 - Considerações Finais

Como os tópicos de interesse, conforme mencionado em 2.1, são intimamente inter-relacionados, muitos trabalhos poderiam figurar em mais de uma das seções deste capítulo. A decisão de colocá-los numa certa seção foi subjetiva e determinada, não pelo assunto considerado predominante pelo autor, mas pelo maior interesse apresentado, de acordo com os objetivos deste trabalho mencionados em 1.3.

O entendimento de programas foi investigado exaustivamente, por ser de grande importância para a engenharia reversa de sistemas legados. A engenharia reversa foi bastante explorada, porém considerou-se predominantemente o apoio fornecido pelo trabalho recente de Penteado [Pen 96]. Na reengenharia foi dado enfoque especial a trabalhos que visavam à mudança de paradigma para orientação a objetos. Padrões foram explorados de forma intensa, principalmente visando à descoberta de padrões orientados a objetos a partir da engenharia reversa de sistemas legados.

Apesar de não ter sido dado destaque em separado ao Reuso de Software, ele foi uma preocupação sempre presente no contexto dos tópicos de interesse. O reuso é uma das aplicações tanto de entendimento de programas, quanto de engenharia reversa e padrões de software, conforme destacado nas respectivas seções.

A revisão bibliográfica evidenciou a importância de padrões para melhoria de qualidade e produtividade das atividades de manutenção e reuso de software. Mostrou também sua atualidade, havendo muitos trabalhos recentes enfocando esse assunto.

CAPÍTULO 3 – Engenharia Reversa

3.1 – Considerações Iniciais

Diante dos conhecimentos obtidos com o estudo da bibliografia mencionada no capítulo 2, neste capítulo dá-se uma idéia das pesquisas que desenvolvemos na área de engenharia reversa. Essas pesquisas referem-se sobretudo à extensão da aplicação do Fusion/RE [Pen 96] ao domínio de sistemas de informação. Seus resultados foram objeto do relatório técnico [Bra 98a], de parte da seção 3 de [Pen 98 a] e da seção 3 de [Pen 98b]. Note-se que as aplicações anteriores [Pen 96a, Pen 96b] foram feitas a uma ferramenta chamada STATSIM para edição e simulação de statecharts desenvolvida em linguagem C, que não pertence ao domínio de sistemas de informação. Assim, na seção 3.2 é feito um resumo do Fusion/RE. Na seção 3.3 é apresentado o detalhamento do processo de abstração originalmente descrito em [Bra 98a]. Na seção 3.4 é mostrado o estudo de caso no qual é feita a aplicação da metodologia descrita nas seções 3.2 e 3.3 a um sistema legado do domínio de sistemas de informação. Na seção 3.5 são feitas as considerações finais deste capítulo.

3.2 – Um Resumo do Fusion/RE

O Fusion/RE [Pen 96a] é um processo geral para efetuar engenharia reversa orientada a objetos, tomando-se como ponto de partida sistemas desenvolvidos com orientação procedimental, já citado no capítulo 2. A figura 3.1 mostra o esquema do Fusion/RE e a seguir seus quatro passos são descritos resumidamente, pois em [Pen95, Pen96] encontram-se mais detalhes dos mesmos.

O **primeiro passo** da abordagem consiste na revitalização da arquitetura do sistema com base na documentação existente ou no código fonte, caso não exista. Ao final desse passo, tem-se uma lista de todos os procedimentos, com sua descrição e hierarquia de chamadas (“chama/chamado por”).

O **segundo passo** tem por objetivo recuperar o Modelo de Análise do Sistema Atual (MASA). Nele elabora-se um pseudo Modelo de Objetos do sistema, identificando-se as classes e seus relacionamentos, bem como seus atributos e procedimentos associados.

Figura 3.1 - Esquema do Fusion/RE

Observa-se que a maioria dos procedimentos do sistema legado contém anomalias, ou seja, um mesmo procedimento lida com várias classes. Para classificar os procedimentos, adota-se a convenção: (c) para construtor, quando o procedimento altera a estrutura de dados e (o) para observador, quando o procedimento somente consulta a estrutura de dados. Quando um procedimento consulta e altera uma mesma classe ele é classificado somente como (c). O sinal de + associado às siglas (c) e/ou (o), representa que o procedimento consulta e/ou observa duas ou

mais estruturas de dados. Desse modo, os procedimentos anômalos podem ser classificados como (oc), (o+c), (oc+) e (o+c+). Os procedimentos são classificados como (i) quando referem-se ao tipo de implementação utilizada.

Após a criação do modelo de objetos pode-se, opcionalmente, elaborar os cenários. As entradas e saídas do sistema necessitam de agentes para manipulá-las e, sendo assim, deve-se identificar esses agentes, observando-se o funcionamento do sistema e criando-se os cenários. Um cenário é uma seqüência de eventos fluindo entre agentes e o sistema com algum propósito. Cada cenário envolve agentes, as tarefas que eles querem que o sistema faça e a seqüência de comunicação envolvida para que sejam feitas essas tarefas. Os cenários auxiliam na elaboração do Modelo de Ciclo de Vida e do Modelo de Operações.

A seguir, define-se o Modelo de Ciclo de Vida do sistema, que retrata o seu comportamento global, preocupando-se com a ordem cronológica em que as operações são realizadas. Estas servem como base para a construção do Modelo de Operações do Sistema, no qual cada operação é descrita de forma minuciosa, de acordo com um gabarito preestabelecido. Na verdade, trata-se de um pseudo-modelo orientado a objetos, pois não atende às restrições da orientação a objetos.

O **terceiro passo** visa à criação do Modelo de Análise do Sistema (MAS), dando enfoque ao domínio da aplicação e não à implementação. Os modelos de Objetos, de Ciclo de Vida e de Operações são abstraídos daqueles construídos anteriormente, com alguns cuidados adicionais, como por exemplo a mudança de nomes para outros mais significativos, generalização e especialização de classes. Os procedimentos sem anomalias geram métodos diretamente, enquanto os com anomalias devem ser desmembrados em diversos métodos, de modo que cada método interaja com apenas uma classe.

O **quarto passo** faz o mapeamento do Modelo de Análise do Sistema para o Modelo de Análise do Sistema Atual, descrevendo a relação entre eles. São mapeadas as classes, atributos e métodos do MAS para os elementos correspondentes do MASA, anotando-se possíveis inclusões/exclusões. Esse passo é importante na futura manutenção e reuso do sistema, caso só se faça a engenharia reversa.

Portanto, ao final do Fusion/RE dispõe-se de um Modelo de Análise do Sistema orientado a objetos que pode ser utilizado na reengenharia do sistema, seja para mudança de linguagem, seja para mudança do paradigma de implementação ou seja ainda para aperfeiçoamento da funcionalidade.

3.3 – Sobre a abstração do Modelo de Análise do Sistema

No Fusion/RE, após a execução dos passos 1 e 2, é feita a abstração do Modelo de Análise do Sistema, baseada no Modelo de Análise do Sistema Atual. Muitas dificuldades podem surgir ao fazer essa abstração de forma direta. O detalhamento desse passo de abstração foi sugerido para contornar essas dificuldades [Bra 98a] e é ilustrado na figura 3.2. Considera-se que o Modelo de Objetos do Sistema Atual (MOSA) foi construído na fase 2 do Fusion/RE e está disponível durante o processo de detalhamento.

Figura 3.2 - Processo de Transição do MASA para o MAS

Inicialmente é feita a abstração do Modelo de Objetos do Sistema. Essa abstração é feita em seis etapas, produzindo diversos modelos intermediários, denominados “MOS-1” a “MOS-6”.

Na primeira etapa é elaborado o MOS-1. Os pseudo-relacionamentos existentes no MOSA são transformados em relacionamentos, removendo-se os atributos das classes cujo único objetivo era fazer o relacionamento entre tabelas, ou seja, as chaves estrangeiras são removidas.

Na segunda etapa é elaborado o MOS-2. Nele, algumas tabelas do MOSA são transformadas em relacionamentos. Classes do MOS-1 que não contêm atributos, ou que contêm apenas um atributo fraco, podem ser transformadas em relacionamentos. Considera-se como atributo fraco aquele que não identifica unicamente o objeto da classe, podendo ser repetido em diversos objetos. Tais classes são tipicamente tabelas do MOSA com duas chaves estrangeiras e nenhum ou apenas um atributo. Outra característica marcante dessas classes é que cada uma das chaves estrangeiras faz a ligação da classe com outra classe por intermédio de um

relacionamento com cardinalidade “muitos para um”. No caso de haver um atributo na classe sendo eliminada, o relacionamento deverá carregá-lo.

Na terceira etapa é elaborado o MOS-3. Nessa etapa são identificadas as agregações e especializações de classes. Agregações podem ser criadas sempre que uma classe englobar outras. Especializações podem ser criadas para isolar em uma superclasse os atributos e métodos comuns a diversas subclasses.

Na quarta etapa é elaborado o MOS-4. Nele acrescentam-se as entradas mais relevantes e os agentes externos que interagem com o sistema, estabelecendo seus limites.

Na quinta etapa é elaborado o MOS-5, que mostra as saídas do sistema, bem como os agentes externos que as manuseiam.

Na sexta etapa é elaborado o MOS-6, no qual são generalizados os nomes de classes, atributos e relacionamentos referentes ao MOS-4 e MOS-5. O objetivo é dar nomes mais significativos, de acordo com o domínio de aplicação.

Vale observar que essas etapas são mais apropriadas para sistemas desenvolvidos com bases de dados relacionais, podendo ser modificadas ou condensadas dependendo da aplicação.

Após a abstração do Modelo de Objetos do Sistema deve ser feita a abstração dos Modelos de Ciclo de Vida e de Operações. No caso do Modelo de Ciclo de Vida, faz-se a mudança de nomes para outros mais significativos. Também podem ser efetuadas mudanças na sequência em que as operações são invocadas ou no seu agrupamento dentro do ciclo de vida do sistema. No caso do Modelo de Operações, os esquemas devem ser revistos de maneira a referenciar as classes do MOS ao invés do MOSA. Os nomes de atributos e operações também devem ser adaptados de acordo com os novos modelos de objetos e de ciclo de vida.

3.4 – Estudo de Caso

O sistema utilizado como exemplo neste trabalho foi originalmente desenvolvido na linguagem Clipper 5.0, possuindo aproximadamente vinte mil linhas de código. Possui vinte e cinco tabelas do tipo “dbf”, relacionadas por chaves estrangeiras.

Trata-se de um sistema real de Oficina Auto-elétrica e Mecânica que controla os serviços executados em uma oficina de veículos, fazendo também o controle de estoque das peças utilizadas e do seu ressurgimento. O cliente vai até a oficina para solicitar a realização de serviços em seu veículo, sendo que um cliente pode ter diversos veículos. O mesmo veículo pode

voltar à oficina diversas vezes, sendo preparada, em cada uma delas, uma Ordem de Serviço distinta. Essa Ordem de Serviço contém os dados do cliente, do veículo e dos reparos a serem feitos.

Quando o veículo é consertado, a Ordem de Serviço é completada, registrando-se as peças utilizadas e a mão-de-obra executada. Muitas vezes o reparo pode exigir peças não existentes no estoque, que são adquiridas fora da oficina mecânica e também fazem parte da Ordem de Serviço. O registro dessas peças é importante para o gerente da oficina, pois elas são candidatas a serem estocadas no futuro. Os possíveis tipos de veículos devem ser cadastrados pelo sistema, pois são utilizadas tabelas para cobrança de serviços, tanto elétricos como mecânicos, de acordo com o tipo de veículo. A venda de peças em estoque diretamente no balcão também é realizada, sendo que nesse caso não é aberta nenhuma ordem de serviço.

Figura 3.3 - Diagrama da Base de Dados.

3.4.1 - Recuperação da arquitetura do sistema legado

O sistema em questão já possuía uma documentação razoável [Oli 91], constituída da descrição detalhada da base de dados, da relação “chama/chamado por” e de um manual do usuário [Bra 90]. A figura 3.3 mostra o diagrama entidade-relação da base de dados, que foi usado no desenvolvimento do sistema legado.

A figura 3.4 ilustra a documentação complementar do diagrama, detalhando, para cada arquivo, seus campos com as respectivas características, seus índices e os programas que com ele interagem. Na figura estão mostrados os detalhes do arquivo “ORDSERV”, que pode ser visto na parte superior da figura 3.3.

A figura 3.5 mostra um dos módulos que fazem parte da relação “chama/chamado por”, na qual para cada módulo existente no sistema há uma descrição de sua funcionalidade, os arquivos lidos e/ou alterados, as rotinas chamadas e as rotinas que o chamam. No caso específico, refere-se ao módulo “OSTELA1”, que consta na parte inferior da figura 3.4 como um dos programas que alteram o arquivo “ORDSERV”. Note-se que essa interação de “OSTELA1” com “ORDSERV” também está indicada na figura 3.5, em que “ORDSERV” é indicado como um dos arquivos lidos e/ou alterados pelo módulo “OSTELA1”.

A figura 3.6 contém um trecho do código fonte, em Clipper, do módulo “OSTELA1”, com cada linha numerada para futuras referências. Algumas linhas pouco significativas para a lógica do programa foram omitidas, tendo sido colocadas reticências em seu lugar.

A figura 3.7 mostra parte dos menus do sistema legado, nos quais está indicado, à direita, o nome do módulo que é executado quando cada opção do menu é selecionada. Por exemplo, “OSTELA1” é executado quando as opções “1-Lançamento”, “1-Ordem de Serviço” e “1-Lançamentos 1ª parte” são selecionadas em seqüência.

Ao realizar a revitalização da arquitetura do sistema legado, correspondente ao primeiro passo do Fusion/RE, notou-se que essa documentação não estava atualizada. Por exemplo, na figura 3.5 consta como rotina chamada por “OSTELA1” apenas “LOGOTIPO”. Mas, na figura 3.6 nota-se que outras rotinas, tais como “AVISO” (linhas 29, 47, 80 e 99), “BUSCLI” (linha 28) e “IMPOS1” (linha 104), também são chamadas. Outra falha encontrada foi com relação aos arquivos “CONTAPAG” e “CONTAREC”, que constam da figura 3.3, mas o código fonte faz

referência apenas a um arquivo “CONTAS”, que sequer consta do diagrama da base de dados. Os arquivos “VTREP” e “EQUIVPEC” não constam do diagrama da figura 3.3, embora sejam importantes na funcionalidade do sistema e constem da descrição de arquivos. Além disso, os nomes de arquivos, procedimentos, campos e índices muitas vezes são pouco significativos, ora por problemas de limitação do Clipper, ora por descuido do programador.

+=====+				
ESTRUTURA DO ARQUIVO ORDSERV .dbf				
+=====+				
ORDEM DE SERVIÇO				
+-----+				
Nro Campo	Nome do Campo	Tipo do Campo	Nro de Caract.	Descrição
+-----+				
1	OSNRO	NUMERICO	5	NUMERO DA ORDEM DE SERVICO
2	DATAENTRA	DATA	8	DATA DE ENTRADA DA ORD. SERVICO
3	DATACOMPRA	DATA	8	DATA DA COMPRA DAS PECAS P/GAR.
4	CODCLIENTE	NUMERICO	5,0	CODIGO DO CLIENTE
5	NROVEICULO	NUMERICO	4,0	NUMERO DO VEICULO DO CLIENTE
6	SERVAEX1	CARACTER	60	SERVICOS A EXECUTAR - PARTE 1
7	SERVAEX2	CARACTER	60	SERVICOS A EXECUTAR - PARTE 2
8	SERVAEX3	CARACTER	60	SERVICOS A EXECUTAR - PARTE 3
9	ORCAMENTO	CARACTER	1	INDICADOR DO ORCAMENTO (S/N)
10	DATAPROM	DATA	8	DATA PROMETIDA PARA ENTREGA
11	HORAPROM	DATA	8	HORA PROMETIDA PARA ENTREGA
12	CONDPAG	CARACTER	15	CONDICOES DE PAGAMENTO
13	PRECOTOT	NUMERICO	11,2	PRECO TOTAL COBRADO
14	DESCONTO	NUMERICO	11,2	DESCONTO CONCEDIDO
15	DESPEXTRA	NUMERICO	11,2	DESPESAS. EXTRAS INCLUSAS
+-----+				
ARQUIVOS DE INDICE				
CLIORDSE.ntx --> CODCLIENTE				
ORDSERV .ntx --> OSNRO				
Programas que o leem:				
ALTCAREC ELICAREC INDEXACA INSCAREC QUITCARE				
Programas que o alteram:				
OSTELA1 OSTELA2				

Figura 3.4 - Descrição de um dos arquivos da base de dados.

Com base na metodologia resumida nas seções 3.2 e 3.3 e usando esse sistema legado, foi conduzida a engenharia reversa cujos passos são descritos a seguir.

```
-----
Módulo Físico:  OSTELA1
Descrição: PRIMEIRA TELA DA ORDEM DE SERVICO
Arquivos apenas lidos:
Arquivos lidos e/ou alterados:  CLIENTES  CLIVEIC  ORDSERV  TIPOVEIC
Rotinas chamadas:  LOGOTIPO
Rotinas que o chamam:  LANCAMEN
```

Figura 3.5 - Descrição de um dos módulos do sistema, com a relação chama/chamado por.

	* Nome do programa: OSTELA1.PRG	58	@ 12, 5 SAY "Endereco:"
1	DO LOGOTIPO	59	@ 12, 15 GET ENDereco PICT REPL ("!",30)
2	SELECT 2	60	@ 12, 53 SAY "Bairro:"
3	USE ORDSERV INDEX CLIORDSE, ORDSERV	61	@ 12, 62 GET BAIRRO PICTURE REPL ("!",15)
4	SELECT 6	...	
5	USE CLIENTES INDEX CODCLI, NOMECLI	62	@ 14, 30 SAY "Servicos a Executar"
6	SELECT 7	63	@ 15, 30 SAY "*****"
7	USE CLIVEIC INDEX CLICODVE	64	serv1 = SPACE(60)
8	SELECT 8	65	serv2 = SPACE(60)
9	USE TIPOVEIC INDEX CODVEIC, MARCVEI,	66	serv3 = SPACE(60)
10	MARCTIP	67	orcama = " "
11	SELECT 2	68	dtpro = CTOD(" / / ")
12	SET ORDER TO 2	69	horapro = SPACE(5)
13	GO BOTTOM	70	@ 17, 7 GET serv1 PICTURE REPLICATE ("!",60)
14	gravou=.f.	71	@ 18, 7 GET serv2 PICTURE REPLICATE ("!",60)
15	codcli = " "	72	@ 19, 7 GET serv3 PICTURE REPLICATE ("!",60)
16	nrov = 0	73	@ 20, 7 SAY "Orcamento (S/N) :"
17	imp = " "	74	@ 20, 26 GET orcama PICT [!] VALID orcama \$ "SN"
18	nrord = OSNRO + 1	75	@ 21, 7 SAY "Prometido para as horas"
19	dtent = DATE()	76	@ 21, 23 GET dtpro PICT "@E" VALID dtpro>=dtent
20	mv = SPACE(15)	77	@ 21, 37 GET horapro PICTURE [!!!!]
21	tv = SPACE(30)	78	READ
	@ 6, 15 SAY "Lancamento da 1a. Parte da Ordem de	79	grava = " "
22	Servico"	80	DO AVISO WITH ga
23	@ 7, 15 SAY "*****"	81	cl = COLUNA(ga)
24	@ 9, 5 SAY "Data de Entrada:"	82	@ 24, cl GET grava PICT [!] VALID grava \$ "GA"
25	@ 9, 22 GET dtent PICTURE "@E"	83	READ
26	@ 9, 48 SAY "O.S.Nro:"	84	IF grava = "G"
27	@ 9, 56 SAY nrord PICTURE[99999]	...	
28	@ 10, 5 SAY "Codigo do Cliente:"	85	SELECT ordserv
29	@ 10, 24 GET codcli PICTURE [99999] when buscli()	86	APPEND BLANK
	DO AVISO WITH " <99999>- Cliente nao Cadastrado	87	REPLACE OSNRO WITH nrord
30	<0> Abandonar"	88	REPLACE DATAENTRA WITH dtent
31	READ	89	REPLACE CODCLIENTE WITH codcli
32	codcli=val(codcli)	90	REPLACE NROVEICULO WITH nrov
33	IF lastkey()#27 .and. codcli <> 0	91	REPLACE SERVAEX1 WITH serv1
34	nro_os=nrord	92	REPLACE SERVAEX2 WITH serv2
35	SELECT CLIENTES	93	REPLACE SERVAEX3 WITH serv3
36	SET ORDER TO 1	94	REPLACE ORCAMENTO WITH orcama
37	SEEK codcli	95	REPLACE DATAPROM WITH dtpro
38	IF .NOT. FOUND()	96	REPLACE HORAPROM WITH horapro
39	GO BOTTOM	97	gravou=.f.
40	codcli = CODCLIENTE + 1	98	men = "Imprime 1a. Parte da Ordem de Servico?
41	dtmov = DATE()		(S/N) ==> <>"
42	APPEND BLANK	99	DO AVISO WITH men
43	REPLACE CODCLIENTE WITH codcli	100	cl = COLUNA(men)
44	REPLACE DATAMOV WITH dtmov	101	@ 24, cl GET imp PICTURE [!] VALID imp \$ "SN"
45	DO AVISO WITH "Digite os Dados do Novo Cliente"	102	READ
46	novo = "S"	103	IF imp = "S"
47	ELSE	104	DO IMPOS1
	DO AVISO WITH "Verifique os Dados do Cliente,	105	ENDIF
48	Modificando se Necessario"	106	ENDIF
49	novo = "N"	107	ENDIF
50	ENDIF	108	ENDIF
51	ok = "N"	109	CLOSE DATABASES
52	DO WHILE ok = "N"	110	RETURN
53	@ 11, 5 SAY "Cliente:"		
54	IF novo = "S"		
55	@ 11, 14 GET NOME PICTURE REPL ("!",30)		
56	ELSE		
57	@ 11, 14 SAY NOME PICTURE REPL ("!",30)		
	ENDIF		

Figura 3.6 - Trecho de um programa Clipper do sistema legado

1-LANÇAMENTOS	{LANCAMEN}
=====	
1-ORDEM DE SERVIÇO	{MENUOS}
1-LANÇAMENTO 1ª PARTE	{OSTELA1}
2-LANÇAMENTO 2ª PARTE	{OSTELA2}
3-LANÇAMENTO TOTAL	{OSTELA}
4-MODIFICAÇÃO 1ª PARTE	{MODOS1}
5-MODIFICAÇÃO 2ª PARTE	{MODOS2}
6-CANCELAMENTO DA O.S.	{CANCELOS}
 2-COMPRAS	 {MENUCOM}
1-PEDIDO DE PEÇAS	{MENUPP}
1-INserção	{INSEPEPE}
2-MODIFICAÇÃO	{ALTPEPE}
3-ELIMINAÇÃO	{ELIPEPE}
2-COMPRAS DE PEÇAS	{MENUCE}
1-INserção	{INSECE}
2-MODIFICAÇÃO	{ALTPCE}
3-ELIMINAÇÃO	{ELICE}
3-VENDAS DE PEÇAS POR BALCÃO	{MENUVEND}
1-INserção	{INSECVE}
2-MODIFICAÇÃO	{ALTCVE}
3-ELIMINAÇÃO	{ELICVE}
 4-CONTAS A PAGAR	 {CONTASPA}
1-INserção	{INSCAPAG}
2-MODIFICAÇÃO	{ALTCAPAG}
3-ELIMINAÇÃO	{ELICAPAG}
4-QUITACÃO	{QUITCAPG}
 5-CONTAS A RECEBER	 {CONTASPA}
1-INserção	{INSCAREC}
2-MODIFICAÇÃO	{ALTCAREC}
3-ELIMINAÇÃO	{ELICAREC}
4-QUITACÃO	{QUITCARE}

Figura 3.7 - Parte dos menus do sistema legado

3.4.2 - Obtenção do Modelo de Análise do Sistema Atual

Apesar dos defeitos constatados na documentação existente do sistema legado, com base nos resultados do passo de revitalização da arquitetura, foi possível construir o Modelo de Objetos do Sistema Atual (MOSA) mostrado na figura 3.8. Pode-se observar que todas as tabelas presentes no diagrama da base de dados da figura 3.3 foram transformadas em classes no MOSA da figura 3.8. Os relacionamentos entre as classes também foram mantidos. Foram acrescentados

ao MOSA as classes que estavam faltando no diagrama da figura 3.3, “VTREP” e “EQUIVPEC”, conforme explicado na seção 3.4.1.

Figura 3.8 - MOSA – Modelo de Objetos do MASA

Ao invés da notação do método Fusion, preconizada em [Col 94], os Modelos de Objetos são aqui representados utilizando a notação UML (*Unified Modeling Language*) [Eri 98, Fow 98], que é plenamente compatível. Vale observar que, na UML, o modelo equivalente ao Modelo de Objetos é denominado “Diagrama de Classes”. A UML foi adotada por suportar a representação de padrões de software, que é necessária no capítulo 5.

Alguns acréscimos de notação foram feitos. Entre esses pode-se mencionar o uso de linhas tracejadas para denotar os pseudo-relacionamentos, o uso de palavras sublinhadas para denotar atributos que são chaves estrangeiras e o uso de palavras em negrito para denotar atributos que são chave primária da classe. Por exemplo, “Ordserv” tem o pseudo-relacionamento “rel-29” com “Ospecas”, devido à existência da chave estrangeira “osnro” em “Ospecas”. Por motivos de falta de espaço, foram colocados nas classes apenas os atributos mais significativos. Por exemplo, dos quinze campos do arquivo “ORDSERV” que constam da figura 3.4, apenas quatro atributos constam na classe “Ordserv” da figura 3.8. Maiores detalhes dessa obtenção do MASA podem ser encontrados em [Bra 98a]. Note-se que lá é usada a própria notação do Fusion.

Figura 3.9 - Parte do MASA e Procedimentos do Sistema Legado

Terminada a construção do MOSA, é possível analisar alguns procedimentos do sistema legado a fim de detectar a ocorrência de anomalias, conforme descrito na seção 3.2. A figura 3.9 ilustra dois procedimentos do sistema legado, “Ostela1” (P1) e “Altveic” (P2), classificados respectivamente como c+ e c, nos quais pode-se observar claramente a ocorrência de anomalias em P1, já que ele lê e/ou altera quatro classes. No trecho de código mostrado na figura 3.6 é possível constatar algumas dessas anomalias, como por exemplo a alteração da classe “Clientes” (linhas 41 a 43).

```

lifecycle Auto-elétrico e Mecânica:= (Lançamentos | Relatórios | Consultas |
    Manutenção_de_arquivos | Manutenção_de_tabelas | Impressos )*

Lançamentos = (lançamento_prim_parte . [#Ordem_Serviço_impressa]) |
    (lançamento_seg_parte . [#Relat_serviços_executados]) |
    modificação_prim_parte | modificação_seg_parte | cancelamento_os |
    inserção_pedido | modificação_pedido | eliminação_pedido |
    inserção_compra | modificação_compra | eliminação_compra |
    inserção_vendas | modificação_vendas | eliminação_vendas |
    inserção_conta_a_pagar | alteração_conta_a_pagar |
    eliminação_conta_a_pagar | quitação_conta_a_pagar |
    inserção_conta_a_receber | alteração_conta_a_receber |
    eliminação_conta_a_receber | quitação_conta_a_receber)

Relatórios = (rel_ord_serv_pendentes | rel_ord_serv_do_mês | rel_pecas_estoque_critico
    | rel_pecas_em_estoque | rel_precos_por_fabricante | rel_equivalencia_entre_pecas
    | rel_pecas_por_componente | rel_pecas_adquiridas_fora | rel_pedidos_pendentes
    | rel_pedidos_pendentes_por_fornecedor | rel_compras_mês | rel_itens_cancelados
    | rel_vendas_mês | rel_contas_a_pagar | rel_contas_a_receber | rel_balanco_mensal
    | rel_cliente_ord_alfabet | rel_cliente_por_entidade | rel_fornecedor_ord_alfabet
    | rel_fornecedor_por_ramo | rel_fabricante | rel_employado | rel_componentes
    | rel_veiculos | rel_mao_deobra | tabela_sugestao_precos
    | tabela_servicos_eletricos | tabela_servicos_meca Nicos)

Consulta = ( cons_pecas_letra_inicial | cons_pecas_fabricante | cons_pecas_codigo_barras |
    cons_pecas_codigo_original | cons_equivalencia_pecas | cons_estatistica_custos |
    cons_fornecedores | cons_vendas | cons_employados | cons_compras |
    cons_componentes | cons_clientes )

Manutenção_de_arquivos = ( inserção_pecas | alteração_pecas | eliminação_pecas |
    inserção_fornecedores | alteração_fornecedores | eliminação_fornecedores |
    inserção_employados | alteração_employados | eliminação_employados |
    inserção_clientes | alteração_clientes | eliminação_clientes |
    inserção_fabricantes | alteração_fabricantes | eliminação_fabricantes |
    inserção_componentes | alteração_componentes | eliminação_componentes |
    inserção_mão_deobra | alteração_mão_deobra | eliminação_mão_deobra |
    inserção_veiculos | alteração_veiculos | eliminação_veiculos )

Manutenção_de_tabelas = (tabela_sugestão_de_preços | inserção_tabela_valor_trabalho |
    alteração_tabela_valor_trabalho | eliminação_tabela_valor_trabalho |
    inserção_tabela_tempo_reparo | alteração_tabela_tempo_reparo )
    eliminação_tabela_tempo_reparo)

Impressos = (cheques | duplicatas | ord_servico | termo_garantia
    | recibo | cancelamento_protestos | carta_cancelamento | vales
    | etiq_cod_barra_por_fornecedor | etiq_cod_barra_por_pecas
    | etiq_cod_barra_por_compra | etiq_cod_barra_pecas_miudas | etiq_por_estantes
    | etiq_clientes | etiq_pessoa_juridica | etiq_pessoa_fisica | etiq_cli_por_entidade
    | etiq_clientes_desejados | etiq_clientes_duplicatas | etiq_fornecedores )

```

Figura 3.10 - MCVSA - Modelo de Ciclo de Vida do Sistema Atual

Na figura 3.10 é mostrado o Modelo de Ciclo de Vida do Sistema Atual (MCVSA), obtido com base nos menus do sistema legado e no código fonte. Comparando a figura 3.7 com a figura 3.10, pode-se observar que a opção “1-Lançamento” da figura 3.7 deu origem ao símbolo não-terminal “Lançamentos” da figura 3.10. Qualquer uma das opções de lançamento pode ser executada a qualquer momento dentro do ciclo de vida do sistema. Assim, foram criadas operações representando cada uma dessas opções, sendo permitida a execução de qualquer uma delas por meio de um “|” (operador “ou”).

Na figura 3.11 é mostrado um dos esquemas do Modelo de Operações do Sistema Atual (MOpSA), correspondente ao lançamento da primeira parte da ordem de serviço. A elaboração desses esquemas baseou-se no código fonte do sistema legado e também na execução da opção correspondente à operação nos menus do sistema legado. Por exemplo, na figura 3.6 um programador Clipper poderia facilmente reconhecer os trechos referentes à geração do número único de ordem de serviço (linhas 10 a 12 e 17) e criação de um novo cliente (linhas 34 a 43), que são tarefas especificadas na cláusula “Result” do esquema da figura 3.11. Esquemas para outras operações podem ser encontrados em [Bra 98a].

Operation : lançamento_prim_parte
Description : "Faz o lançamento da primeira parte da Ordem de Serviço "
Reads : Supplied : Data_entr : char , Orcam : int , Serv_a_exec : char , Dt_entrega : char , Hora_entrega : char
Changes : new cli : Clientes, new os : Ordserv , new veic : Cliveic new mod : Tipoveic
Sends : Oficina { Ordem_de_Serviço_Impressa }
Result : " Uma new os foi criada Um número único de os foi gerado os.dataentra foi iniciada com Data_entr os.orcamen foi iniciada com Orcam os.servex1 foi iniciada com Serv_a_executar os.dataprom foi iniciada com Dt_entrega os.horaprom foi iniciada com Hora_entrega Se o cliente ainda não estava cadastrado então um new cli foi criado Se era a primeira vez que o veiculo estava sendo consertado então um new veic foi criado Ordem_de_Serviço_Impressa (Ordem de Serviço) foi enviada à oficina Se o modelo de veículo ainda não estava cadastrado então um new mod foi criado"

Figura 3.11 - Esquema para a operação “lançamento_prim_parte” do MOpSA

3.4.3 - Abstração do Modelo de Análise do Sistema

O passo de abstração do MAS foi feito utilizando o processo de detalhamento proposto em [Bra 98a] e descrito na seção 3.3, ao invés do processo recomendado em [Pen 96a]. As diversas etapas da abstração efetuada são mostradas a seguir, englobando as transições do MOSA para o MOS, do MCVSA para o MCVS e do MOpSA para o MOpS.

Na figura 3.12 é mostrado o MOS-1, obtido na primeira etapa, no qual os pseudo-relacionamentos existentes no MOSA da figura 3.8 foram transformados em relacionamentos, removendo-se os atributos das classes cujo único objetivo era fazer o relacionamento entre elas. Por exemplo, da classe “Cliveic” foram removidos os atributos “codcliente” e “codveiculo”, cuja função era relacioná-la às classes “Clientes” e “Tipoveic”, respectivamente. Dessa forma, os pseudo-relacionamentos entre elas (“rel-1” e “rel-4”), que eram denotados por linhas tracejadas, tornaram-se relacionamentos, denotados por linhas contínuas.

Na segunda etapa do processo de detalhamento, diversas classes foram eliminadas, além de modificar os relacionamentos existentes. A figura 3.13 mostra o MOS-2 obtido. As classes “Ramoforn”, “Pecacomp”, “Veipeca” e “Veicomp”, que aparecem na figura 3.12, foram eliminadas de forma direta. O fato delas terem ficado sem atributo algum após a primeira etapa forneceu a indicação de que sua utilidade era apenas de ligação entre duas outras classes. As classes “Temprep” e “Vtrep” foram analisadas cuidadosamente por possuírem apenas um atributo. Como esse atributo não identifica unicamente um elemento da classe, ele pôde ser transferido para o relacionamento resultante entre “Tipoveic” e “Codmobra”. Isso é representado no modelo da figura 3.13 pelos rótulos junto à classe “Codmobra” com os nomes desses atributos (“tempo” e “quantvt”, respectivamente).

A classe “Equivpec” foi eliminada com base no conhecimento sobre o domínio da aplicação. O arquivo do MOSA “Equivpec” possui atributos “codpeca1” e “codpeca2” (ver figura 3.8). Estudando o problema em questão e os procedimentos responsáveis por alimentar esse arquivo, deduz-se que esses atributos representam um auto-relacionamento da classe “Pecas”, indicando que a peça cujo código é “codpeca1” é equivalente à peça cujo código é “codpeca2”. Adotou-se a convenção de dar o próprio nome da classe sendo excluída ao relacionamento resultante após sua exclusão.

Na terceira etapa do processo de detalhamento foram criadas agregações envolvendo as classes “Ordserv”, “Ospecas” e “Osmobra”, as classes “Vendas” e “Itemvend” e as classes

Figura 3.12 - MOS-1 – Modelo de Objetos do MAS – Etapa 1

Figura 3.13 - MOS-2 – Modelo de Objetos do MAS – Etapa 2

Durante a elaboração do MOS-3 percebeu-se que a classe “Compras”, do MOS-2, englobava também os pedidos feitos ao fornecedor. Assim, resolveu-se acrescentar mais uma classe ao MOS-3 que cobrisse esse aspecto, denominada “Pedido”. Foi também necessário acrescentar a classe “Itens do pedido” para tratar dos itens constantes do pedido. Os atributos que antes constavam da classe “Compras” foram distribuídos entre as classes resultantes, “Compra” e “Pedido”, de acordo com seu significado semântico. Isso pode ser observado comparando-se a figura 3.13 com a figura 3.14. Nota-se também que foi acrescentado o relacionamento “refere-se a” entre “Compra” e “Pedido”, já que a compra é feita após o pedido ter sido atendido pelo fornecedor. Outros detalhes importantes tiveram que ser cuidados, por meio de minucioso exame do código fonte. Por exemplo, ao fazer a especialização deve-se examinar os relacionamentos das subclasses com as demais classes, para descobrir se eles devem ser mantidos na subclasse ou se podem ser transferidos para a superclasse. Assim, os relacionamentos “rel-26” e “rel-27” do MOS-2 puderam ser fundidos no relacionamento “rel-26” do MOS-3, já que peças trocadas na ordem de serviço podem ser tanto as adquiridas fora quanto as em estoque. Da mesma forma os relacionamentos “rel-22” e “rel-23” do MOS-2 puderam ser fundidos no relacionamento “rel-23” do MOS-3, já que ambos os tipos de peça possuem fabricante. Já os relacionamentos “rel-24” e “rel-8” foram mantidos ligados à classe “Pecas”, pois peças vendidas no balcão ou compradas do fornecedor referem-se somente às peças em estoque.

Na quarta etapa do processo de detalhamento foi criado o MOS-4, mostrado na figura 3.15. Nele foram acrescentadas algumas entradas mais relevantes do sistema e os agentes externos que com ele interagem, estabelecendo seus limites. Esses agentes são o atendente e o gerente da empresa. O atendente é necessário porque o sistema não é do tipo “self-service”. Existe um atendente intermediário, que faz os lançamentos no computador e recebe alguns relatórios de saída. Outros relatórios mais sofisticados são entregues ao gerente, que os analisa e toma suas decisões. Como existem muitas operações nesse sistema, foram explicitadas apenas as mais relevantes. Por exemplo, é conveniente explicitar o papel do atendente de abrir e fechar uma ordem de serviço. Todavia, ao surgir um novo tipo de veículo haveria necessidade de incluir tempo de reparo e/ou valor de trabalho correspondente, o que seria feito pelo gerente. O mesmo aconteceria quando aparecesse um fornecedor novo, um empregado novo, etc. Operações desse tipo não foram modeladas.

Figura 3.15 - MOS-4 – Modelo de Objetos do MAS – Etapa 4

Na quinta etapa do processo de detalhamento foi elaborado o MOS-5, mostrado na figura 3.16. Esse modelo mostra algumas saídas do sistema, bem como os agentes externos que as manuseiam. Por exemplo, o gerente acompanha o relatório de ordens de serviços pendentes e o relatório de peças com estoque crítico, o atendente utiliza a relação de clientes e o mecânico recebe a ordem de serviço impressa. Como pode ser visto na figura 3.10 existem inúmeros relatórios e impressos nesse sistema que dariam origem a classes no modelo de objetos. Entretanto, por motivos de falta de espaço, apenas alguns deles foram colocados no modelo, principalmente os mais significativos para o sistema.

Figura 3.16 - MOS-5 – Modelo de Objetos do MAS – Etapa 5

Na sexta etapa do processo de detalhamento os nomes de classes, atributos e relacionamentos foram mudados para outros mais significativos dentro do domínio de aplicação. Foram também acrescentados alguns atributos às classes que tinham sido ignorados nos modelos anteriores por questões práticas. O resultado dessas etapas está presente nos modelos apresentados nas figuras 3.17 e 3.18, que juntos formam o Modelo de Objetos do Sistema (MOS).

Figura 3.18 - MOS-6b – Modelo de Objetos do MAS – Etapa 6 – Parte B

Para comprovar essa melhoria de nomes obtida no MOS-6, pode-se consultar a tabela 3.1, apresentada na seção 3.4.4, que faz a correspondência entre as classes do MAS e os arquivos do MASA. A tabela 3.2 mostra alguns nomes de atributos que também foram alterados. Vale ressaltar que algumas classes e relacionamentos foram incluídos nos modelos com o nome definitivo, visto que quando da sua inclusão já se conhecia mais sobre o domínio de aplicação.

Terminada a construção do MOS, o código fonte dos procedimentos com anomalias detectadas no passo dois pode ser analisado de forma a identificar os métodos de cada classe que seriam necessários para resolver essas anomalias. A figura 3.19 mostra parte do MAS e os vinte e quatro métodos identificados na operação “Ostela1”. Esses métodos foram identificados examinando o código fonte da operação do sistema legado e subdividindo em métodos os trechos que possuem anomalias. Cada método é atribuído à classe à qual faz referência, como pode ser observado na figura 3.19. Cabe ressaltar que, nessa etapa, os métodos são apenas identificados, mas não é feita a separação física do código. Essa separação é cuidada durante a reengenharia com mudança de orientação, que é vista na seção 4.2.

Figura 3.19 - Parte do MAS e métodos correspondentes

Para completar a elaboração do MAS, os Modelos de Ciclo de Vida e de Operações foram abstraídos a partir dos respectivos modelos do MASA. A figura 3.20 mostra o Modelo de Ciclo de Vida do MAS (MCVS) e a figura 3.21 mostra um dos esquemas do Modelo de Operações do MAS (MOpS), mais especificamente para a operação “Abre_Ordem_de_Serviço”.

Comparando o MCVSA (figura 3.10) ao MCVS (figura 3.20), nota-se que os nomes das operações foram substituídos por outros mais significativos e as operações foram agrupadas de forma diferente, de modo que possam ser encontradas com mais facilidade de acordo com seu significado semântico. Quanto ao Modelo de Operações do MAS, foi feita a adaptação dos nomes e atributos das classes de acordo com o novo Modelo de Objetos, preservando a funcionalidade e interface do sistema. Exemplos da correspondência entre o MAS e o MASA podem ser vistos na seção 3.4.4.


```

lifecycle Auto-elétrico e Mecânica:=
  (Lançamento | Atualização_arquivos | Impressão | Consulta ) *
Lançamento = ( abre_ordem_de_serviço . [#Ordem_Serviço_impressa] ) | modifica_ord_serv |
  elimina_ord_serv | ( fecha_ordem_de_serviço . [#Relat_serviços_executados] ) |
  insere_atendimento_balcão | altera_atendimento_balcão | elimina_atendimento_balcão |
  insere_compra | altera_compra | elimina_compra | insere_pedido | altera_pedido |
  elimina_pedido | insere_conta_a_pagar | insere_conta_a_receber |
  altera_conta_a_pagar | altera_conta_a_receber | elimina_conta_a_pagar |
  elimina_conta_a_receber | quita_conta_a_pagar | quita_conta_a_receber )
Atualização_arquivos = ( insere_peca | insere_cliente | insere_fabricante | insere_fornecedor |
  insere_empregado | insere_tipo_veiculo | insere_valor_de_trabalho |
  insere_tempo_de_reparo | insere_componente | insere_mao_de_obra |
  altera_peca | altera_cliente | altera_fabricante | altera_fornecedor | altera_empregado |
  altera_tipo_veiculo | altera_valor_de_trabalho | altera_tempo_de_reparo |
  altera_componente | altera_mao_de_obra | elimina_peca | elimina_cliente |
  elimina_fabricante | elimina_fornecedor | elimina_empregado | elimina_tipo_veiculo |
  elimina_valor_de_trabalho | elimina_tempo_de_reparo | elimina_componente |
  elimina_mao_de_obra | atualiza_tabela_sugestao_de_precos )
Impressão = ( rel_ord_serv_pendentes | rel_ord_serv_do_mês | rel_pecas_estoque_critico |
  rel_pecas_em_estoque | rel_precos_por_fabricante | rel_equivalencia_entre_pecas |
  rel_pecas_por_componente | rel_pecas_adquiridas_fora | rel_pedidos_pendentes |
  rel_pedidos_pendentes_por_fornecedor | rel_compras_mês | rel_itens_cancelados |
  rel_vendas_mês | rel_contas_a_pagar | rel_contas_a_receber | rel_balanco_mensal |
  rel_cliente_ord_alfabet | rel_cliente_por_entidade | rel_fornecedor_ord_alfabet |
  rel_fornecedor_por_ramo | rel_fabricante | rel_empregado | rel_componentes |
  rel_veiculos | rel_mao_de_obra | rel_sugestao_precos | rel_servicos_eletricos |
  rel_servicos_mecanicos | cheques | duplicatas | ord_servico | termo_garantia |
  recibo | cancelamento_protestos | carta_cancelamento | vales |
  etiq_cod_barra_por_fornecedor | etiq_cod_barra_por_peca |
  etiq_cod_barra_por_compra | etiq_cod_barra_pecas_miudas | etiq_por_estantes |
  etiq_clientes | etiq_pessoa_juridica | etiq_pessoa_fisica | etiq_cli_por_entidade |
  etiq_clientes_desejados | etiq_clientes_duplicatas | etiq_fornecedores )
Consulta = ( cons_peca_letra_inicial | cons_peca_fabricante | cons_peca_codigo_barras |
  cons_peca_codigo_original | cons_equivalencia_peca | cons_estatistica_custos |
  cons_fornecedores | cons_vendas | cons_empregados | cons_compras |
  cons_componentes | cons_clientes )

```

Figura 3.20 - MCVS - Modelo de Ciclo de Vida do Sistema

3.4.4 - Mapeamento MAS/MASA

Graças ao nível de detalhamento com que foi feita a transição do MASA para o MAS, a realização deste passo foi facilitada. Sua finalidade, conforme explicado na seção 3.2, é estabelecer a correspondência entre o Modelo de Análise do Sistema, que considera principalmente o domínio de aplicação e não a implementação física atual e o Modelo de Análise do Sistema Atual, feito com base na estrutura de dados do sistema legado. Essa correspondência é importante, pois no caso de não ser conduzida a reengenharia do sistema, ela será utilizada durante o processo de manutenção.

<p>Operation : abre_ordem_de_serviço</p> <p>Description : " Dá entrada em uma ordem de serviço "</p> <p>Reads : Supplied : Data_entr : char , Orcam : int , Serv_a_exec : char , Dt_entrega : char , Hora_entrega : char</p> <p>Changes : new cli : Cliente, new os : Ordem de Serviço , new veic : Veículo de Cliente new mod : Tipo de Veículo</p> <p>Sends : Oficina { Ordem_de_Serviço_Impressa }</p> <p>Result : " Uma new os foi criada Um número único de os foi gerado os.Data Entrada foi iniciada com Data_entr os.Orçamento foi iniciada com Orcam os.Serviços a executar foi iniciada com Serv_a_executar os.Data Entrega foi iniciada com Dt_entrega os.Hora Entrega foi iniciada com Hora_entrega Se o cliente ainda não estava cadastrado então um new cli foi criado Se era a primeira vez que o veiculo estava sendo consertado então um new veic foi criado Ordem_de_Serviço_Impressa (Ordem de Serviço) foi enviada à oficina Se o modelo de veículo ainda não estava cadastrado então um new mod foi criado"</p>

Figura 3.21 - Esquema para a operação "abre_ordem_de_serviço" do MOpS

A tabela 3.1 indica a correspondência entre as classes do MOS e os arquivos do sistema legado. Nela pode-se perceber os arquivos eliminados durante a abstração, bem como as classes incluídas durante a mesma. O processo de detalhamento efetuado com base em [Bra 98a] permite que se saiba exatamente em que etapas foram eliminados ou incluídos cada um dos componentes do MOS.

A tabela 3.2 mostra a correspondência entre o MAS e o MASA para as classes Veículo de Cliente e Ordem de Serviço e seus atributos. Pode-se observar nessa tabela os atributos eliminados durante a abstração, mais especificamente durante a etapa 1 do processo de detalhamento.

A tabela 3.3 mostra a correspondência entre algumas operações do MAS e do MASA. Nesse caso, houve apenas uma melhora nos nomes das operações. Não houve inclusão ou exclusão de operações, pois a funcionalidade do sistema foi mantida.

Tabela 3.1 – Correspondência entre classe do MAS e arquivos do MASA

Classe do MAS	Arquivo do MASA
Atendimento por balcão	Vendas
Cliente	Contarec
Componente	Componen
Compra	Compras
Conta	-
Conta a Pagar	Contapag
Conta a Receber	Contarec
Empregado	Empregad
Fabricante	Fabrican
Fornecedor	Fornec
Mão-de-obra da Ordem de Serv.	Osmobra
Ordem de Serviço	Ordserv
Peça	-
Peça adquirida fora	Outpecas
Peça comprada	Itemcomp
Peça da Ordem de Serv.	Ospecas
Peça do estoque	Pecas
Peça pedida	-
Peça vendida	Itemvend
Pedido	-
Ramo	Ramo
Tipo de mão-de-obra	Codmobra
Tipo de veículo	Tipoveic
Veículo de cliente	Cliveic
-	Equivpec
-	Pecacomp
-	Ramoforn
-	Temprep
-	Veipeca
-	Veicomp
-	Vtrep

3.5 – Considerações Finais

A aplicação do Fusion/RE ao domínio de sistemas de informação mostrou que nesse domínio há algumas facilidades, quanto à engenharia reversa dos dados, decorrentes principalmente do uso de tabelas normalizadas no sistema legado. Por outro lado, quanto aos procedimentos, a abstração dos métodos foi facilitada, já que o sistema legado foi desenvolvido em uma linguagem de nível mais alto.

O detalhamento do passo de abstração torna mais ameno o uso do Fusion/RE, que anteriormente exigia um salto maior entre o passo de elaboração do Modelo de Análise do Sistema Atual e o do Modelo de Análise do Sistema.

Tabela 3.2 – Correspondência entre classes/atributos do MAS e arquivos/campos do MASA

MAS	MASA
Modelo de Objetos Classe: Veículo de Cliente	Arquivo Cliveic
Atributos	Campos do Arquivo
- - Placa Cidade Quilometragem Nro de meses Nro sequencial	codcliente codveiculo placa cidade hskm meses nroveiculo
Modelo de Objetos Classe: Ordem de Serviço	Arquivo Ordserv
Atributos	Campos do Arquivo
Nro da Ordem Serv. - - Data de entrada do veic. Defeitos apresentados Preço total Data de saída do veic.	osnro codcliente nroveiculo dataentra servaex1, servaex2, servaex3 precotot datacompra

Tabela 3.3 – Correspondência entre algumas operações do MAS e do MASA

MAS	MASA
Abre_Ordem_de_Serviço	Lançamento_prim_parte
Fecha_Ordem_Serv	Lançamento_seg_parte
Insere_Compra	Inserção_compra
Insere_peça	Inserção_peça
Insere_pedido	Inserção_pedido
Altera_pedido	Alteração_pedido
Elimina_pedido	Eliminação_pedido

Conforme se verá nos próximos capítulos, essa engenharia reversa efetuada é básica tanto para a reengenharia, descrita no capítulo 4, quanto para o reconhecimento de padrões, feito no capítulo 5.

Capítulo 4 – Reengenharia

4.1 – Considerações Iniciais

A engenharia reversa por si só já traz grandes benefícios quanto à manutenibilidade do sistema, já que são criados modelos em níveis mais altos de abstração. Porém, para que esses benefícios possam ser plenamente usufruídos, o código fonte do sistema deve ser compatível com os modelos de análise e projeto. Isso pode ser conseguido de duas formas: por segmentação do sistema legado, mantendo a linguagem de programação original, ou por reengenharia com mudança de linguagem. A primeira alternativa é discutida na seção 4.2, na qual é feita a reengenharia com mudança de orientação e sem mudança de linguagem. A segunda alternativa é apresentada na seção 4.3, na qual a reengenharia com mudança de linguagem, pressupondo que a de mudança de orientação já tenha sido feita, é discutida. Na seção 4.4 apresentam-se as considerações finais deste capítulo.

4.2 – Reengenharia com mudança de orientação e sem mudança de linguagem

4.2.1 – Abordagem proposta

Nesta seção propõe-se uma abordagem para transformar código originalmente orientado a procedimentos para código com as características da orientação a objetos. A linguagem de programação é mantida durante essa transformação. Essa abordagem foi tratada em [Pen 98a] e é ilustrada na figura 4.1. Após a obtenção dos modelos de análise pelo Fusion/RE, dois passos adicionais são executados: o projeto avante do sistema e a segmentação dos programas, descritos a seguir.

No primeiro passo adicional é elaborado o projeto avante do sistema, visando à reengenharia com mudança do paradigma de implementação, mas sem mudar a funcionalidade do sistema. Nele são elaborados: o Grafo de Interação de Objetos, o Grafo de Visibilidade, a Descrição de Classes e os Grafos de Herança.

Figura 4.1 - Abordagem proposta para reengenharia com mudança de orientação

O Grafo de Interação de Objetos é construído com base no Modelo de Operações. Ele mostra a passagem de mensagens entre os objetos, por intermédio dos métodos, além de diferenciar entre a classe controladora da operação e as classes colaboradoras. Um pseudo-código é escrito para o método da classe controladora da operação, mostrando sua lógica e a sequência de chamada dos métodos das classes colaboradoras. Note-se que o método da classe controladora (que implementa uma operação) interage com mais de um objeto, porém o faz por intermédio dos métodos das classes colaboradoras, com o que são evitadas as anomalias.

O Grafo de Visibilidade estabelece a comunicação entre as classes, delimitando quais delas podem trocar mensagens e os tipos de mensagens a serem trocadas. Nos grafos de interação de objetos assume-se que todos os objetos são mutuamente visíveis, podendo trocar mensagens entre si, o que é restringido no Grafo de Visibilidade.

A Descrição de Classes contém documentação complementar ao modelo de objetos do sistema. Assim, para cada classe são reunidas informações que estavam espalhadas pelos vários elementos de documentação até agora produzidos. Do Grafo de Interação de Objetos são retirados os métodos; do Modelo de Objetos são extraídos alguns atributos de dados e do Grafo

de Visibilidade os atributos objeto-valorados. São salientadas, para cada classe, todas as interações com as demais.

Os Grafos de Herança são um mecanismo pelo qual uma classe pode ser definida como uma especialização de outra. Podem ser introduzidas novas classes nesta etapa.

Os procedimentos com anomalias podem agora ser examinados. O resultado desse exame vai nortear a realização do passo seguinte.

No segundo passo adicional é feita a segmentação do sistema. Programas originalmente monolíticos e contendo anomalias são aqui desmembrados em métodos, conforme definido nos grafos de interação de objetos. Os programas são percorridos sequencialmente, identificando-se os trechos relacionados a uma das classes identificadas no passo 3 do Fusion/RE. Quanto maiores esses trechos, mais fácil será a segmentação. Esses trechos serão a base para construção de cada método, devendo ser substituídos, no programa que corresponde à operação, por uma chamada de procedimento ou função. Os métodos obtidos devem ficar disponíveis para serem chamados pela operação. O ideal é que sejam métodos da classe correspondente, mas se a linguagem de programação não suportar esse conceito, deverão estar disponíveis por outros meios.

É importante ressaltar que a segmentação do programa é feita considerando o projeto do sistema, desenvolvido no passo de projeto do Fusion, que foi conduzido como se o sistema fosse ser reprogramado usando uma linguagem orientada a objetos. Isso facilita a segmentação, como já foi observado, e elimina a dificuldade de identificação dos métodos. Outra constatação é de que a segmentação não muda a linguagem de programação, não sendo então verdadeiramente orientada a objetos. Trata-se de um código programado com estilo de tipos abstratos de dados e no qual os mecanismos de especialização, generalização e agregação podem ser simulados. Entretanto, após a segmentação, ele pode ser facilmente convertido para uma linguagem orientada a objetos. Essa experiência foi feita por Sneed, transformando o código legado de COBOL para COBOL-OO [Sne 96].

Após a segmentação pode-se proceder o teste do sistema segmentado, com o que se assegura a conservação de sua funcionalidade. O código segmentado pode servir como entrada para uma transformação automática de código, conforme é visto na seção 4.3. Entretanto, mesmo que a mudança de linguagem não ocorra, muitos benefícios da orientação a objetos poderão ser colhidos durante a manutenção do sistema.

4.2.2 – Estudo de Caso

Após ter sido obtido o Modelo de Análise do Sistema, pela aplicação do Fusion/RE, os dois passos adicionais descritos na seção 4.2.1 são aplicados. Para realização da experiência descrita nesta seção foi utilizado o sistema legado cuja engenharia reversa é descrita na seção 3.4. Assim, a documentação de análise orientada a objetos já existe e pode ser aqui utilizada.

Figura 4.2 - Exemplo de um Grafo de Interação de Objetos

No primeiro passo adicional, o Projeto Avante do Sistema é executado, obtendo-se os Grafos de Interação de Objetos, como o exemplificado na figura 4.2, os Grafos de Visibilidade, as Descrições de Classes e os Grafos de Herança. O Grafo de Interação de Objetos mostrado na figura 4.2 é referente à operação “Abre_Ordem_de_Serviço”. Ele mostra os métodos que devem ser invocados de cada classe para que a operação seja executada, destacando a ordem de chamada desses métodos. Os Grafos de Visibilidade, as Descrições de Classes e os Grafos de Herança não têm utilidade imediata para o que se pretende em seguida, que é fazer a segmentação dos programas, mantendo a linguagem de programação original. Como a linguagem “Clipper” não possui recursos para implementar a visibilidade e a herança, esses modelos não foram totalmente desenvolvidos.

Deve-se observar que o Grafo de Interação de Objetos mostrado na figura 4.2 possui influência direta da linguagem de programação em que o sistema legado foi desenvolvido. Por exemplo, métodos como “Abrir_Arquivo” e “Fechar_Arquivo” não seriam necessários em certas linguagens, mas foram considerados nesse grafo por se tratar da linguagem “Clipper”.

No segundo passo adicional, a Segmentação dos programas do sistema legado foi executada. Os procedimentos correspondentes às operações foram tratados um a um. A presença de anomalias foi detectada durante a segunda fase do Fusion/RE e os métodos foram identificados na terceira fase e posteriormente detalhados no Grafo de Interação de Objetos. Isso foi de grande valia, pois pôde-se, de antemão, saber quais métodos seriam esperados. Os programas foram percorridos seqüencialmente, reconhecendo os grupos de comandos correlacionados, que lidavam com cada uma das classes identificadas no MAS. Os métodos foram fisicamente separados do corpo da operação, sendo substituídos por uma chamada a procedimento ou função. Na maioria das vezes houve a necessidade de criar parâmetros de entrada ou saída, para permitir que os resultados da execução do método pudessem ser usados posteriormente por outros métodos.

As limitações da linguagem Clipper obrigaram a adoção de determinadas condutas. Por exemplo, em Clipper não existe a possibilidade de criação de classes que encapsulem dados e métodos. Para contornar esse problema, para cada classe do modelo de objetos foram criados dois arquivos: um contendo os dados (arquivo do tipo “dbf”) e outro contendo todos os métodos da classe (arquivo do tipo “prg”). Isso faz parte da simulação da orientação a objetos, que na verdade é a programação com tipos abstratos de dados. Seria também possível simular a herança,

mesmo que parcialmente, mas isso não foi feito porque os benefícios não compensariam a ilegitimidade causada ao código.

<pre> * Nome do programa: OP_OS1.PRG * Operacao de inclusao da abertura da Ordem de Servico 1 DO LOGOTIPO 2 DO ABRCLIE // Abre_Arquivo_Cli (1) 3 DO ABRCLIVE // Abre_Arquivo_Veic (3) 4 DO ABRORDSE // Abre_Arquivo_OS (2) 5 DO ABRTIPVE // Abre_Arquivo_Tipv (4) 6 nord = 0 7 codcli = 0 8 gravou=.F. 9 nomecli= "" 10 DO OBTPROXO WITH nord // Obtém_Proximo_Num_OS 11 (5) 12 DO TELAOS1 WITH nord, codcli // Obtém_Data_OS (6) 13 IF LASTKEY() # 27 .AND. codcli <> 0 14 enc = .T. 15 DO PROCLI WITH codcli, enc // Escolhe_Cli (7) 16 IF .NOT. enc 17 DO NOVOCLI with codcli // Cria_Cli (8) 18 ELSE 19 DO AVISO WITH "Verifique os Dados do Cliente, 20 Modificando se Necessario" 21 ENDIF 22 ok = "N" 23 nomecli = SPACE(30) 24 DO WHILE ok = "N" 25 DO TELACLI WITH enc,nomecli // Obtem_Dados_Cli (9) 26 ok = CONFDADO(correto) 27 ENDDO 28 DO GRCLIE // Grava_Cli (10) 29 DO TELALIM 30 passou = .T. 31 codvei = 0 32 DO VERCLIVE WITH passou, nomecli // Busca_Veic (11) 33 fim = .F. 34 IF .NOT. passou 35 Nrovei = 0 36 ... 37 @ 7, 25 SAY "Verificacao do(s) Veiculo(s) do Cliente" 38 @ 8, 25 SAY "*****" 39 ... 40 DO WHILE .NOT. fim 41 DO TELACLVE // Exibe_Tela_veic (12) 42 pert = " " 43 men = "Veiculo acima ainda Pertence ao Cliente? (S/N) 44 =====> <>" 45 DO CONFDADO WITH pert, men 46 IF pert = "N" 47 DO APCLIVEI // Elimina_Veic (13) 48 ELSE 49 IF passou = .F. 50 serv = " " 51 men = "Serao Executados Servicos Nesse Veiculo? 52 (S/N) =====> <>" 53 DO CONFDADO WITH serv, men </pre>	<pre> 47 IF serv = "S" 48 * achou o veiculo no qual sera feito o conserto 49 passou = .T. 50 ENDIF 51 ENDIF 52 ... 53 ENDDO 54 ENDIF 55 aband = .F. 56 IF passou = .F. 57 * primeira vez que este veiculo do cliente e consertado 58 ... 59 gr = .T. 60 DO LETIPVEI WITH gr // Obtém_Dados_Tipv (14) 61 IF gr 62 DO GRTIPVEI WITH codvei // Cria_Tipv (15) 63 ELSE 64 aband = .T. 65 ENDIF 66 ENDIF 67 IF .NOT. aband 68 ... 69 ok = "N" 70 DO WHILE ok = "N" 71 DO LECLVEI // Obtém_Dados_Veic (16) 72 DO CONFDADO WITH ok, correto 73 ENDDO 74 DO LEORDSER // Obtém_Dados_Entr_OS (17) 75 grava = " " 76 DO CONFGRAV WITH grava 77 IF grava = "G" 78 DO GRCLIVEI WITH codcli, codvei, nrovei 79 // Cria_Veic (18) 80 DO NOVAORD // Cria_Entr_OS (19) 81 DO GRORDSER WITH nord, codcli, nrovei 82 // Grava_OS (20) 83 gravou = .T. 84 men = "Imprime 1a. Parte da Ordem de Servico? (S/N) 85 =====> <>" 86 imp = "N" 87 DO CONFDADO WITH imp, men 88 IF imp = "S" 89 DO IMPOS1 with nord 90 ENDIF 91 ENDIF 92 ENDIF 93 nro_os=nord 94 // Variável a ser passada como parâmetro para op_os2 95 DO FECLIE // Fecha_Arquivo_Cli (21) 96 DO FECLIVE // Fecha_Arquivo_Veic (23) 97 DO FECORDSE // Fecha_Arquivo_OS (22) 98 DO FECTIPVE // Fecha_Arquivo_Tipv (24) 99 RETURN </pre>
--	---

Figura 4.3 - Programa segmentado correspondente à operação Abre_Ordem_de_Serviço

A figura 4.3 mostra o programa segmentado correspondente à operação “Abre_Ordem_de_Serviço”. Alguns trechos foram omitidos, estando representados por reticências. A figura 4.4 mostra o código de alguns dos métodos invocados no trecho de programa

segmentado da figura 4.3. O método “Obtproxo” da figura 4.4 (a) é invocado na linha 10 do programa segmentado (figura 4.3) e corresponde às linhas 10, 11, 12 e 17 do programa legado (figura 3.6). O método “Novocli” da figura 4.4 (b) é invocado na linha 16 do programa segmentado e corresponde às linhas 34, 35, 38 a 43 do programa legado. O método “Leordser” da figura 4.4 (c) é invocado na linha 71 do programa segmentado e corresponde às linhas 62, 63, 70, 71, 72, 74, 76, 77 e 78 do programa legado. O método “Grordser” da figura 4.4 (d) é invocado na linha 76 do programa segmentado e corresponde às linhas 85 a 96 do programa legado.

<pre>***** FUNCTION OBTPROXO() ***** * metodo de ORDSERV para gerar proximo nro vago de Ordem de Servico. PARAMETERS nord SELECT ORDSERV SET ORDER TO 2 GOTO BOTTOM vOsnro = OSNRO + 1 nord = vOsnro SET ORDER TO 1 RETURN NIL</pre>	<pre>***** FUNCTION LEORDSER() ***** * metodo de ORDSERV para ler dados da Ordem de Servico @ 14, 30 SAY "Servicos a Executar" @ 17, 7 GET vServaex1 PICT REPLICATE ("!",60) @ 18, 7 GET vServaex2 PICT REPLICATE ("!",60) @ 19, 7 GET vServaex3 PICT REPLICATE ("!",60) @ 20, 7 SAY "Orcamento (S/N) : " @ 20, 26 GET vOrcamento PICT "!" VALID vOrcamento \$ "SN" @ 21, 7 SAY "Prometido para as horas" @ 21, 23 GET vDataprom PICT "@E" VALID vDataprom >= vDataentra @ 21, 37 GET vHoraprom PICT "!!!!!" RETURN NIL</pre>
<p style="text-align: center;">(a)</p> <pre>***** FUNCTION NOVOCLI() ***** * metodo de CLIENTE para criar novo registro de Cliente e gerar um numero sequencial único PARAMETERS codcli SELECT CLIENTES SET ORDER TO 1 GOTO BOTTOM vCodcliente = CODCLIENTE + 1 vDatamov = DATE() APPEND BLANK REPLACE CODCLIENTE WITH vCodcliente REPLACE DATAMOV WITH vDatamov DO AVISO WITH "Digite os Dados do Novo Cliente" codcli = vCodcliente RETURN vCodcliente</pre> <p style="text-align: center;">(b)</p>	<p style="text-align: center;">(c)</p> <pre>***** FUNCTION GRORDSER() ***** * metodo de ORDSERV para gravar a nova Ordem de Servico PARAMETERS nord, codcli, nrovei v16Osnro = nord v16Codcliente = codcli v16Nroveiculo = nrovei SELECT ORDSERV REPLACE OSNRO WITH v16Osnro REPLACE DATAENTRA WITH vDataentra REPLACE CODCLIENTE WITH v16Codcliente REPLACE NROVEICULO WITH v16Nroveiculo REPLACE SERVAEX1 WITH vServaex1 REPLACE SERVAEX2 WITH vServaex2 REPLACE SERVAEX3 WITH vServaex3 REPLACE ORCAMENTO WITH vOrcamento REPLACE DATAPROM WITH vDataprom REPLACE HORAPROM WITH vHoraprom RETURN NIL</pre> <p style="text-align: center;">(d)</p>

Figura 4.4 - Alguns métodos do Programa segmentado

Analisando alguns desses métodos e comparando-os com o trecho do programa legado do qual foram extraídos, percebe-se que foi necessário adotar algumas diretrizes para que a

segmentação fosse realizada. Por exemplo, adotou-se a convenção de que os nomes dos atributos das classes seriam armazenados em variáveis globais com a letra “v” precedendo o nome do atributo. Foi necessário tornar globais essas variáveis para que outros métodos, invocados pelo mesmo objeto da classe, tivessem acesso a elas. Porém, após segmentar uma parte do código, começaram a surgir variáveis com mesmo nome, tanto referentes às chaves estrangeiras quanto a nomes que coincidiam. Assim, além do prefixo “v”, foi necessário acrescentar um número que identificasse unicamente cada atributo de classe. Para cada classe que precisasse de um número para diferenciar seus atributos foi atribuído um número único. Por exemplo, à classe “Ordserv” foi atribuído o número 16, enquanto que à classe “Cliente” foi atribuído o número 26. Na figura 4.4 (d), que corresponde às linhas 85 a 96 do programa legado (figura 3.6), três dos atributos receberam o prefixo “v16” (v16Osnro, v16Codcliente e v16NroVeiculo), enquanto que os demais atributos receberam apenas o prefixo “v”, pois seus nomes não coincidem com nenhum atributo de outras classes.

O resultado dessa experiência comprovou que os métodos são muito numerosos e em geral contêm poucas linhas de código. Porém, existe maior possibilidade de reutilizá-los devido à sua funcionalidade bem definida. A facilidade de manutenção também é maior, pois sua localização física é mais próxima em relação ao sistema legado, no qual trechos de programas ficavam espalhados por toda parte e faziam acesso a diversas estruturas de dados.

4.3 – Reengenharia com mudança de linguagem

Conforme mencionado na seção 4.1, a reengenharia com mudança de linguagem de programação, além de atualizar o código para plataformas mais modernas, pode ser realizada para dar continuidade ao processo de engenharia reversa. Isso faz com que o código fique sincronizado com a análise e o projeto, melhorando, portanto, a manutenibilidade futura.

Após ter sido realizada a engenharia reversa, a reengenharia com mudança de linguagem pode ser feita manualmente ou auxiliada por uma ferramenta de transformação. A transformação automática de uma linguagem para outra, se feita de forma direta, na maioria das vezes gera código difícil de compreender e manter. A execução prévia do processo de segmentação permite que essa transformação seja mais amena, o código gerado seja mais compreensível e aproveite melhor as características da orientação a objetos. Tal experimento foi feito usando a máquina

Draco-Puc [Pra 92] e é objeto de um artigo publicado em co-autoria com Penteadó e outros [Pen 98b]. Um resumo desse experimento é relatado nas subseções que se seguem.

4.3.1 – Sistema de Transformação utilizado

A máquina Draco é baseada nas idéias de construção de software por transformação orientada a domínios. Um primeiro protótipo da máquina Draco foi construído por Neighbors [Nei 84]. Posteriormente, ela foi reconstruída na PUC-RJ [Pra 92], usando novas linguagens de plataformas modernas de hardware e de software. Essa versão foi denominada Draco-Puc.

Pela estratégia proposta por Prado [Pra 92], é possível a reconstrução de um software pelo porte direto do código fonte para linguagens de outros domínios. Um domínio, de acordo com a máquina Draco-Puc, é constituído de três partes: um *parser*, um *pretty-printer* e um ou mais transformadores.

Para a reimplementação automática é necessária a construção dos domínios da linguagem origem e da linguagem destino, por intermédio de cinco atividades básicas: construir *parsers*, construir *pretty-printers*, construir um transformador da linguagem origem para a base de conhecimento, construir um transformador da linguagem origem para a linguagem destino e construir bibliotecas, conforme mostra a figura 4.5.

Figura 4.5 - Atividades básicas para a construção dos domínios de origem e destino no Draco

Na atividade “construir *parsers*” obtém-se os *parsers* das gramáticas livres de contexto dos domínios origem e destino, a partir da definição dos seus analisadores léxicos e sintáticos. Essa atividade é auxiliada pelo subsistema gerador de *parsers* do Draco, denominado “Pargen”. As definições dos *parsers* são usadas como entrada nessa atividade. Ao lado das regras gramaticais têm-se as ações semânticas (*makenode*, *makeleaf*, etc.) usadas para construção da DAST (Draco *Abstract Syntax Tree*), que é a linguagem interna usada pelo Draco nas transformações.

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

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

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

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

Após a construção dos domínios origem e destino, os programas escritos na linguagem origem são submetidos ao Draco, que depois de uma série de etapas produz o código fonte na

linguagem destino. O código escrito na linguagem origem é inicialmente analisado pelo Draco usando o *parser* produzido na atividade 1 citada anteriormente. Isso resulta na DAST com a representação interna do código original. Essa DAST é então submetida ao *unparser*, que a formata de acordo com a sintaxe da linguagem origem, fazendo uso do *pretty-printer* da linguagem origem, gerado na atividade 2. O código obtido é então submetido ao transformador construído na atividade 3, que extrai os fatos para a base de conhecimento. Finalmente o código é submetido ao transformador construído na atividade 4 que, usando os fatos da base de conhecimento e as bibliotecas, transforma-o para a linguagem destino.

4.3.2 – Estudo de Caso

O sistema de oficina auto-elétrica e mecânica que passou pela engenharia reversa, conforme mostrado na seção 3.4, e que sofreu segmentação, conforme mostrado na seção 4.2.2, foi transformado de Clipper para Java com o auxílio da máquina Draco. A figura 4.6 mostra, de forma mais detalhada e específica do que a figura 4.5, a construção dos domínios Clipper e Java [Pen 98b]. Cabe ressaltar que não participei diretamente nessa construção, tendo contribuído de outras formas, seja dando indicações de como criar as transformações, seja fornecendo o código segmentado como entrada para a máquina Draco, ou ainda na validação do resultado produzido.

As cinco atividades referidas na seção 4.3.1 foram desenvolvidas. Na atividade “construir parser”, denotada na figura 4.6 como “*parser constructing*”, foram construídos os *parsers* das gramáticas dos domínios Clipper e Java, a partir da definição de seus analisadores léxicos e sintáticos. A linguagem destino, Java, foi escolhida por ser orientada a objetos e bastante atual, além de ser portátil para qualquer plataforma de hardware ou software.

Na atividade “construir pretty-printers”, denotada na figura 4.6 como “*pretty-printers constructing*”, o Draco gera automaticamente os *pretty-printers* Clipper e Java pelo uso de seu sub-sistema “Ppgen”. Isso é feito a partir das definições das regras gramaticais do Clipper e do Java e de primitivas de formatação especiais, colocadas ao lado dessas regras gramaticais.

Na atividade “construir o transformador da linguagem origem para a base de conhecimento”, denotada na figura 4.6 como “*ClipperToKB.tfm constructing*”, foi definido o transformador “ClipperToKb”, capaz de percorrer a DAST Clipper, reconhecendo suas diferentes estruturas e gerando fatos na KB. Conforme dito anteriormente, a construção desse transformador é auxiliada pelo subsistema Tfmgen do Draco.

Figura 4.6 - Construção dos domínios Clipper e Java no Draco [Pen 98b]

<pre> public void OP_OS1() { int nord[] = new int[1]; int codcli[] = new int[1]; boolean gravou; String nomecli[] = new String[1]; boolean encontrou[] = new boolean[1]; String ok[] = new String[1]; boolean passou[] = new boolean[1]; int codvei[] = new int[1]; boolean fim[] = new boolean[1]; int nrovei[] = new int[1]; String pert[] = new String[1]; String men[] = new String[1]; String serv[] = new String[1]; boolean aband; boolean gr[] = new boolean[1]; String grava[] = new String[1]; String imp[] = new String[1]; int nro_os; Utility.LOGOTIPO(); Clientes CLIENTES = new Clientes(); Cliveic CLIVEIC = new Cliveic(); Tipoveic TIPOVEIC = new Tipoveic(); nord[0] = 0; codcli[0] = 0; gravou = false; nomecli[0] = " "; OBTPROXO(nord); TELAOS1(nord, codcli); if (Clipper.lastKey() != 27 && codcli[0] != 0) { encontrou[0] = true; CLIENTES.PROCLI(codcli, encontrou); if (!encontrou[0]) { CLIENTES.NOVOCLI(codcli); } else{ Utility.AVISO("Verifique os Dados do Cliente, Modificando se Necessario"); } ok[0] = "N"; nomecli[0] = Clipper.space(30); while (ok[0].equals("N")) { CLIENTES.TELACLI(encontrou, nomecli); men[0] = Menuprin.correto[0]; Utility.CONFDADO(ok, men); } Utility.TELALIM(); passou[0] = true; codvei[0] = 0; CLIVEIC.VERCLIVE(passou, nomecli); fim[0] = false; if (!passou[0]) { nrovei[0] = 0; CLIVEIC.PROCCLIV(codcli, codvei, nrovei, encontrou); Utility.LIMPA(); if (encontrou[0]) { Clipper.say(7, 25, String.valueOf("Verificacao do(s) Veiculo(s) do Cliente")); TIPOVEIC.PROCTVEI(codvei); TIPOVEIC.TELATVEI(); CLIVEIC.TELAVECA(); Clipper.say(13, 5, String.valueOf("Grupo:")); while (!fim[0]) { TIPOVEIC.PROCTVEI(codvei); CLIVEIC.TELACLVE(); pert[0] = " "; </pre>	<pre> men[0] = "Veiculo ainda Pertence ao Cliente? (S/N) ==> < >"; Utility.CONFDADO(pert, men); if (pert[0].equals("N")) { CLIVEIC.APCLIVEI(); } else { if (passou[0] == false) { serv[0] = " "; men[0] = "Serao Executados Servicos Nesse Veiculo? (S/N) ==> < >"; Utility.CONFDADO(serv, men); if (serv[0].equals("S")) { passou[0] = true; } } } CLIVEIC.AVANCCCLV(codcli, fim); Utility.LIMPA(); } } } aband = false; if (passou[0] == false) { Utility.TELALIM(); TIPOVEIC.TELA(); TIPOVEIC.TELATVEI(); gr[0] = true; TIPOVEIC.LETIPVEI(gr); if (gr[0]) { TIPOVEIC.PR MARTIP(encontrou, codvei); if (!encontrou[0]) { TIPOVEIC.GRTIPVEI(codvei); } } else{ aband = true; } } if (!aband) { TIPOVEIC.TELATVEI(); CLIVEIC.TELAVECA(); ok[0] = "N"; while (ok[0].equals("N")) { CLIVEIC.LECLVEI(); Utility.CONFDADO(ok, Menuprin.correto); } LEORDSER(); grava[0] = " "; Utility.CONFGRAV(grava); if (grava[0].equals("G")) { CLIVEIC.GRCLIVEI(codcli, codvei, nrovei); NOVAORD(); GRORDSER(nord, codcli, nrovei); gravou = true; men[0] = "Imprime 1a. Parte da Ordem de Servico? (S/N) ==> < >"; imp[0] = "N"; Utility.CONFDADO(imp, men); if (imp[0].equals("S")) { OP_IMPO1(nord); } } } } nro_os = nord[0]; CLIENTES.closeDBF(); CLIVEIC.closeDBF(); closeDBF(); TIPOVEIC.closeDBF(); } </pre>
--	--

--	--

Figura 4.7 - Código Java resultante da transformação, ref. à operação *Abre_Ordem_de_Servico*

<pre> public void OBTPROXO(int nord[]) { setOrder(2); goLast(); ORDSERV.vOsnro = OSNRO + 1; nord[0] = ORDSERV.vOsnro; setOrder(1); } </pre> <p>(a)</p>	<pre> public void NOVOCLI(int codcli[]) { setOrder(1); goLast(); vCodcliente = CODCLIENTE + 1; vDatamov = Data.today(); appendBlank(); replace("CODCLIENTE", new Integer(vCodcliente)); replace("DATAMOV", vDatamov); Utility.AVISO("Digite os Dados do Novo Cliente"); codcli[0] = vCodcliente; } </pre> <p>(b)</p>	<pre> public void GRORDSER(int nord[], int codcli[], int nrovei[]) { v16Osnro = nord[0]; v16Codcliente = codcli[0]; v16Nroveiculo = nrovei[0]; replace("OSNRO", new Integer(v16Osnro)); replace("DATAENTRA", vDataentra); replace("CODCLIENTE", new Integer(v16Codcliente)); replace("NROVEICULO", new Integer(v16Nroveiculo)); replace("SERVAEX1", vServaex1); replace("SERVAEX2", vServaex2); replace("SERVAEX3", vServaex3); replace("ORCAMENTO", vOrcamento); replace("DATAPROM", vDataprom); replace("HORAPROM", vHoraprom); } </pre> <p>(d)</p>
<pre> public void LEORDSER() { Clipper.say(14, 30, String.valueOf("Servicos a Executar")); Clipper.get(17, 7, String.valueOf(vServaex1), Clipper.replicate("!", 60)); Clipper.get(18, 7, String.valueOf(vServaex2), Clipper.replicate("!", 60)); Clipper.get(19, 7, String.valueOf(vServaex3), Clipper.replicate("!", 60)); Clipper.say(20, 7, String.valueOf("Orcamento (S/N) :")); Clipper.get(20, 26, String.valueOf(vOrcamento), "!", GetField.SIGNAL_IN, String.valueOf("SN")); Clipper.say(21, 7, String.valueOf("Prometido para as horas")); Clipper.get(21, 23, String.valueOf(vDataprom), "@E", GetField.SIGNAL_GREY, String.valueOf(vDataentra)); Clipper.get(21, 37, String.valueOf(vHoraprom), "!!!!"); Menuprin.getReturn = Clipper.read(); vServaex1 = Menuprin.getReturn.isGot(17, 7) ? Menuprin.getReturn.elementAt(17, 7) : vServaex1; vServaex2 = Menuprin.getReturn.isGot(18, 7) ? Menuprin.getReturn.elementAt(18, 7) : vServaex2; vServaex3 = Menuprin.getReturn.isGot(19, 7) ? Menuprin.getReturn.elementAt(19, 7) : vServaex3; vOrcamento = Menuprin.getReturn.isGot(20, 26) ? Menuprin.getReturn.elementAt(20, 26) : vOrcamento; vDataprom = Menuprin.getReturn.isGot(21, 23) ? new Data(Menuprin.getReturn.elementAt(21, 23)) : vDataprom; vHoraprom = Menuprin.getReturn.isGot(21, 37) ? Menuprin.getReturn.elementAt(21, 37) : vHoraprom; } </pre> <p>(c)</p>		

Figura 4.8 - Alguns métodos do programa transformado

Na atividade “construir o transformador da linguagem origem para a linguagem destino”, denotada na figura 4.6 como “*clipperToJava.tfm Constructing*”, foi construído o transformador “ClipperToJava”, que faz o mapeamento semântico da linguagem Clipper para a linguagem Java, utilizando os fatos armazenados na KB. A construção desse transformador também é auxiliada pelo subsistema Tfmgen do Draco.

Na atividade “construir bibliotecas”, denotada na figura 4.6 como “*Libraries constructing*”, foram construídas bibliotecas de mapeamento do Clipper para o Java, com o

objetivo de permitir que se expressem em Java os comandos Clipper referentes a entrada e saída de dados, gerenciamento de tela, impressora e banco de dados.

Isso tudo preparado, foi possível efetuar a transformação, alimentando a máquina Draco com o programa segmentado em Clipper e obtendo a transformação em Java”, seguindo o roteiro apresentado no fim da seção 4.3.1. O resultado é um programa com características de orientação a objetos, obtidas graças ao processo de segmentação.

A figura 4.7 mostra o código Java resultante da transformação referente à operação Abre_Ordem_de_Serviço, cujo código segmentado em Clipper consta da figura 4.3. A figura 4.8 mostra os métodos (em Java) correspondentes aos métodos em Clipper da figura 4.4.

Caso a segmentação não fosse efetuada, apesar do programa em Java ser escrito numa linguagem orientada a objetos, não se teria as vantagens da orientação a objetos, tais como o reuso e a manutenibilidade propiciadas pelo encapsulamento. Esses programas podem ser posteriormente inspecionados e modificados pelo engenheiro de software, se for o caso, para tirar partido das heranças, agregações, polimorfismo, etc.

4.4 – Considerações Finais

A reengenharia de um sistema originalmente procedimental, obtendo-se um sistema orientado a objetos foi efetuada com êxito. Foram explorados dois caminhos em sequência. O primeiro partiu do modelo de objetos obtido pela engenharia reversa e segmentou o código legado, mantendo a linguagem Clipper. O segundo fez a transformação automática de código Clipper para Java, partindo do código segmentado. Em princípio, os códigos produzidos pelos dois caminhos têm a mesma facilidade de manutenção. A transformação para Java mantém toda a estrutura do programa segmentado, sendo, portanto, equivalente em termos de manutenção. Entretanto, o código Java pode ser executado em qualquer plataforma de hardware ou software, o que o torna muito mais portátil do que o código Clipper.

Por outro lado, caso a empresa tenha pessoal especializado em Clipper e não necessite de mudança de plataforma, é mais conveniente utilizar o código segmentado em Clipper. Desta forma não será necessário contratar novo pessoal ou dar treinamento em novas linguagens. A documentação de análise e de projeto é agora compatível com o código, o que facilita a manutenção e eventual futura reengenharia com mudança de funcionalidade. Além disso,

resultados obtidos pela engenharia reversa, engenharia avante e segmentação aumentam a possibilidade de reuso, tanto de código quanto de análise e projeto.

Capítulo 5 - Reconhecimento de Padrões de Software

5.1 – Considerações Iniciais

Conforme mencionado na seção 1.1 existem padrões de software em diversos níveis de abstração, desde a análise, passando pelo projeto e até padrões de código. Neste trabalho, o interesse maior é por padrões que possam ser empregados em sistemas de informação. A seção 5.2 descreve brevemente alguns desses padrões, propostos por autores como Coad [Coa 92], Boyd [Boy 98] e Johnson [Joh 98]. Na seção 5.3 é proposta uma estratégia para reconhecimento de padrões em modelos de objetos de sistemas legados. Na seção 5.4 é feito um estudo de caso usando como material o que foi preparado na seção 3.4. Na seção 5.5 são feitas as considerações finais deste capítulo.

5.2 – Padrões recorrentes para Sistemas de Informação

Existem inúmeros padrões recorrentes no domínio de sistemas de informação, isto é, padrões que ocorrem repetidamente no seu desenvolvimento. Analisando alguns desses padrões e suas principais aplicações, puderam ser escolhidos diversos deles que mais ocorrem em sistemas como os por mim desenvolvidos na prática profissional. Nesta seção são destacados apenas aqueles que se aplicam ao caso em estudo discutido na seção 5.4, que são: o *TypeObject* [Joh 98], o *Association-Object* [Boy 98], o *State across a collection* e o *Behaviour across a collection* [Coa 92]. Outros padrões recorrentes foram mostrados na seção 2.5.2, como o *Roles-played* [Coa 92] e o *Abstract Factory* [Gam 93],

5.2.1 – Padrão *Type-Object*

Um dos padrões encontrados em quase todos os sistemas de informação é o *Type-Object*, descrito por Johnson [Joh 98] e mostrado na figura 5.1. Esse padrão é equivalente ao padrão *Item-Description* apresentado por Coad [Coa 92]. A versão escolhida foi a de Johnson por ser especificada em maior nível de detalhe.

O problema resolvido por esse padrão é o de uma classe com diversos tipos, que podem variar em tempo de execução. A solução é agrupar os atributos comuns à classe em “Class” e criar uma outra classe “TypeClass” para conter os tipos. Os objetos da classe “Class” possuem uma referência a algum dos objetos da classe “TypeClass”. Pode também haver a referência inversa, embora Johnson a considere facultativa.

Figura 5.1 - Padrão Type-Object [Joh 98]

A figura 5.2 mostra o exemplo utilizado por Johnson para ilustrar o padrão *Type-Object*. Em uma videolocadora, um filme possui atributos como título e preço de locação e pode possuir cópias em diversas fitas de vídeo, cada qual alugada para um cliente diferente. O padrão estabelece que “Filme” e “Fita de vídeo” são classes relacionadas, existindo uma referência nos objetos da classe “Fita de vídeo” ligando-os a um objeto da classe “Filme”.

Figura 5.2 - Exemplo do Type-Object [Joh 98]

Para mostrar a semelhança do *Type-Object* com o padrão *Item-description*, esse último é apresentado na figura 5.3. Um exemplo ilustrativo dado pelo autor [Coa 92] é mostrado na figura 5.4, no qual é fácil verificar tal semelhança.

Figura 5.3 - Padrão Item-Description [Coa 92]

Figura 5.4 - Exemplo do Item-Description [Coa 92]

5.2.2 – Padrão Association-Object

Outro padrão bastante comum em sistemas de informação é o *Association-Object*, proposto por Boyd [Boy 98] e mostrado na figura 5.5. É semelhante ao padrão *Time-Association*, descrito por Coad [Coad 92], embora Boyd o tenha apresentado de forma mais detalhada e específica para sistemas de informação.

Figura 5.5 - Padrão Association-Object [Boy 98]

Esse padrão é usado sempre que exista uma associação dinâmica entre dois objetos estáticos a qual deve ser cuidada pelo sistema. Em geral essa associação dura um certo tempo, denotado pelos atributos *Begin_date* e *End_date* da classe *Association* e tem algum custo, tratado pelo atributo *Cost* dessa mesma classe. Além dos atributos em comum, a classe *Association* tem um comportamento comum, presente nos métodos *Allocate_costs* e *Create_current_from_plan*. As classes *STATIC-1* e *STATIC-2* também possuem atributos e comportamento comuns.

A figura 5.6 mostra um exemplo de aplicação do padrão *Association-Object* a um sistema de alocação de salas a departamentos. As classes “Sala” e “Departamento” desempenham o papel de *Static-1* e *Static-2*, respectivamente. A classe “Atribuição Sala-Depto” figura como *Association*. Ao departamento pode ser atribuída uma ou mais salas por períodos predeterminados. Essa atribuição deve definir um percentual de alocação, calculando o custo da alocação. Tanto uma sala quanto um departamento podem querer saber seus custos com alocação. Assim, existem métodos em ambas as classes para calcular os respectivos custos.

Figura 5.6 - Association-Object [Boy 98]

5.2.3 – Padrão State across a collection

O padrão *State across a collection* foi proposto por Coad [Coad 92] para resolver um problema comum em sistemas orientados a objetos: o de agregações, nas quais a superclasse possui atributos que são compartilhados pelas subclasses. A figura 5.7 mostra esse padrão e a figura 5.8 mostra um exemplo de aplicação, no qual a classe “Venda” desempenha o papel *Collection* e a classe “Item da Venda” desempenha o papel de *Member*.

Figura 5.7 - Padrão State across a collection [Coad 92]

5.2.4 – Padrão Behaviour across a collection

O padrão *Behaviour across a collection* também foi proposto por Coad [Coad 92] para resolver o problema de agregações, mas no caso em que a superclasse possua métodos compartilhados pelas subclasses. A figura 5.9 mostra esse padrão e a figura 5.10 mostra um

exemplo de aplicação para um sistema de chamadas telefônicas, no qual a classe “Venda” desempenha o papel *Collection* e a classe “Item da Venda” desempenha o papel de *Member*.

Figura 5.8 - Exemplo do Padrão State across a collection [Coa 92]

Figura 5.9 - Padrão Behaviour across a collection [Coa 92]

Figura 5.10 - Exemplo do Padrão Behaviour across a collection [Coa 92]

5.3 – Uma estratégia para reconhecimento de padrões

Modelos de objetos de diversos sistemas de informação foram inspecionados a fim de neles reconhecer padrões como os discutidos na seção 5.2. Entre esses foram utilizados um sistema de biblioteca, um sistema de consultório médico e o sistema de oficina auto-elétrica e mecânica apresentado na seção 3.4. Essa inspeção permitiu que algumas diretrizes básicas pudessem ser formuladas para o reconhecimento desses padrões em modelos de objetos de sistemas de informação. Essas diretrizes foram generalizadas e deram origem à estratégia apresentada nesta seção.

5.3.1 – Padrão Type-Object

O padrão *TypeObject*, discutido na seção 5.2.1, é freqüentemente encontrado em sistemas de informação. O seguinte algoritmo genérico deve ser seguido para tentar reconhecer esse padrão em um modelo de objetos existente:

- 1) todas as classes do modelo de objetos devem ser analisadas, em busca de relacionamentos com cardinalidade “um para muitos”. Os pares de classes com tais características são selecionados como candidatos a um *Type-Object*, no qual a classe que fica do lado “um” é candidata ao papel “TypeClass” e a classe do lado “muitos” é candidata ao papel “Class”. As demais classes são descartadas.
- 2) em ambas as classes do par candidato deve haver um atributo que identifique unicamente cada objeto da classe e um outro atributo que o descreva. Os pares que não obedecerem a essa regra podem ser descartados.
- 3) finalmente, o valor semântico do relacionamento deverá ser analisado. Lendo-se o relacionamento partindo de “Class” para “TypeClass” deve ser natural pensar em métodos como: a) dado um objeto da classe “Class” informe qual é seu objeto correspondente na classe “TypeClass” e b) atribua a um objeto da classe “Class” um objeto da classe “TypeClass”. Lendo-se o relacionamento no sentido inverso, deve ser possível pensar em um método que, dado um objeto da classe “TypeClass”, retorne todos os objetos da classe “Class”. A existência desses métodos deve ser natural, no sentido de que eles são cabíveis dentro do domínio de aplicação, sendo fácil imaginar sua utilidade para alguém.

Conforme já mencionado, esse algoritmo foi construído com base na experiência prática de reconhecimento de padrões em diversos sistemas de informação. Porém, é possível que outras diretrizes sejam descobertas se forem analisadas outras aplicações. É possível também que algumas dessas diretrizes não se apliquem a todos os casos, apesar dos cuidados que foram tomados ao fazer a generalização.

5.3.2 – Padrão Association-Object

O padrão *Association-Object*, discutido na seção 5.2.2, também é comum em sistemas de informação. Para identificá-lo, pode-se usar o seguinte algoritmo:

- 1) todas as classes do modelo de objetos devem ser analisadas, em busca das que contenham um ou mais atributos do tipo data. Essas classes são candidatas ao papel “Association” do padrão. As demais classes podem ser descartadas.
- 2) deve-se então examinar, para todas as classes candidatas, os relacionamentos do tipo “muitos para um” que elas tenham com outras classes. As classes do lado “um” devem ser estáticas, isto é, devem representar itens tangíveis do domínio, que contenham características inerentes ao longo de seu tempo de vida. Essas classes são candidatas ao papel “static-1” ou “static-2”.
- 3) se não existe relacionamento desse tipo, pode-se descartar a classe candidata.
- 4) se existe apenas um relacionamento desse tipo, mas existe uma classe agregada à classe candidata a “Association” e a partir dessa classe agregada existe um relacionamento como o descrito no item 2, a classe candidata continua sendo examinada. Nesse caso, a classe do lado “um” do primeiro relacionamento identificado assume o papel “Static-1” e a classe relacionada à classe agregada assume o papel “Static-2”. Caso contrário a classe candidata a “Association” pode ser descartada.
- 5) se existem dois relacionamentos desse tipo, associam-se os papéis “Static-1” e “Static-2” às duas classes do lado “um” desses relacionamentos.
- 6) se existem mais de dois relacionamentos desse tipo, devem ser escolhidas duas classes para os papéis “Static-1” e “Static-2”. Executam-se então os demais passos e, caso o padrão não seja identificado, devem ser escolhidas outras duas classes até que todas as combinações possíveis tenham sido exploradas.
- 7) por fim, o significado semântico deve ser analisado. Focalizando na classe candidata ao papel “Association”, deve fazer sentido pensar que ela faz a ligação temporal entre as classes

candidatas a “Static-1” e “Static-2”. Essa ligação deve, preferencialmente, envolver algum custo e deve haver um método na classe “Association” para criar e/ou desfazer a associação entre as duas classes estáticas.

As mesmas observações feitas no padrão *Type-Object* são válidas aqui. Outros sistemas talvez tenham comportamento diferente, que necessite de diretrizes adicionais ou não se encaixem nas diretrizes aqui propostas.

5.3.3 – Padrão State across a Collection

O padrão *State across a Collection*, discutido na seção 5.2.3, é reconhecido usando-se o seguinte algoritmo:

- 1) todas as classes do modelo de objetos devem ser examinadas em busca de classes que possuam subclasses agregadas. Essas classes são candidatas ao papel “Collection” do padrão, enquanto que a subclasse agregada correspondente é candidata ao papel “Member”. As demais classes podem ser descartadas.
- 2) para todas as classes candidatas ao papel “Collection” deve-se examinar seus atributos em busca daqueles que sejam comuns à classe candidata ao papel “Member”. Se tal atributo existe, o padrão ocorre.

Esse padrão aparece com frequência em sistemas orientados a objeto em geral. Seu reconhecimento torna-se fácil devido ao uso de agregação, que já é “meio caminho andado” em busca do padrão. Se o engenheiro de software utilizou agregação para representar certa classe, provavelmente foi devido à ocorrência de atributos ou métodos compartilhados pela classe e subclasse. Assim, a ocorrência desse padrão ou do padrão *Behaviour across a collection*, mostrado a seguir, é garantida.

5.3.4 – Padrão Behaviour across a collection

O padrão *Behaviour across a Collection*, discutido na seção 5.2.4, é reconhecido utilizando-se o seguinte algoritmo:

- 3) todas as classes do modelo de objetos devem ser examinadas em busca de classes que possuam subclasses agregadas. Essas classes são candidatas ao papel “Collection” do padrão, enquanto que a subclasse agregada correspondente é candidata ao papel “Member”. As demais classes podem ser descartadas.
- 4) para todas as classes candidatas ao papel “Collection” deve-se examinar seus métodos em busca de pelo menos um que se aplique a todos os objetos da coleção como um todo. Se tal método existe, o padrão ocorre.

Aqui valem as mesmas observações feitas no padrão *State across a collection*. Na maioria das vezes a ocorrência desse padrão é simultânea com a ocorrência do *State across a collection*, devido ao próprio conceito de agregação.

5.4 – Estudo de Caso

5.4.1 – Reconhecimento dos padrões no modelo de objetos do sistema legado

Conforme mencionado na seção 5.3, a estratégia lá proposta foi possível graças à experiência de reconhecimento de padrões em diversos sistemas de informação. Um desses sistemas é usado para exemplificar o reconhecimento de padrões nesta seção. Trata-se do mesmo sistema utilizado nos capítulos 3 e 4, que é referente a uma oficina auto-elétrica e mecânica de veículos.

O MOS-6a, apresentado na figura 3.17, é tomado como base para o reconhecimento de padrões. Os padrões reconhecidos foram documentados no modelo de objetos, usando a notação recomendada pela UML [Eri 98], mostrada na figura 5.11. O nome do padrão é colocado em uma elipse com linha tracejada. As classes do sistema que correspondem à instância do padrão são colocadas em caixas retangulares. Setas tracejadas são desenhadas partindo da elipse para as caixas retangulares, rotuladas com o nome do papel desempenhado pela classe instanciada.

A figura 5.12 mostra o modelo de objetos do sistema após o reconhecimento de padrões. Sempre que um padrão é reconhecido, os relacionamentos que fazem parte desse padrão são removidos do modelo de objetos examinado.

Figura 5.11 - Notação usada para expressar padrões em UML

Como pode ser observado na figura 5.12, foram reconhecidas duas instâncias do *Type-Object*, quatro instâncias do *Association-Object*, cinco instâncias do *State across a collection* e cinco instâncias do *Behaviour across a collection*. Esse reconhecimento é descrito a seguir.

Inicialmente procurou-se pela ocorrência do padrão *Type-Object*. Todas as classes foram examinadas em busca de relacionamentos do tipo “muitos para um” e foram selecionados os seguintes candidatos ao papel “Class”: Atendimento por balcão, Conta a Receber, Conta a Pagar, Peça vendida, Veículo de cliente, Ordem de Serviço, Peça da Ordem de Serv., Mão-de-obra da Ordem de Serv., Peça, Compra, Peça comprada, Pedido e Peça pedida. Foi então verificada a existência de um atributo identificador e outro descritor nos candidatos a “Class” e nos respectivos candidatos a “TypeClass”. Dessa forma, foram descartados muitos dos candidatos, tendo restado apenas os dois pares de candidatos a “Class”/ “TypeClass”: Veículo de cliente/Tipo de Veículo e Peça/Fabricante. A maioria dos candidatos foi eliminada por não possuir um identificador único ou um atributo que os descrevessem. Foi então analisado o valor semântico do relacionamento e concluiu-se que os dois finalistas realmente formam uma ocorrência do *Type-Object*. É natural pensar em métodos como: dado um veículo de cliente informe qual seu tipo; dado o veículo de placa “XXX-1234” atribua a ele a marca/tipo “Volkswagen-Parati”; e quais veículos da marca/tipo “Fiat-Uno Mille” foram consertados nos últimos seis meses.

Partiu-se então para a busca do padrão *Association-Object*. Na primeira etapa foram classificados, por conterem atributo do tipo data, as classes Atendimento por balcão, Conta, Ordem de Serviço, Compra e Pedido. Foram então examinados seus relacionamentos do tipo “muitos para um” em busca de classes candidatas a “Static-1” e “Static-2”. Para a classe Atendimento por balcão foram encontradas as classes Cliente e Empregado. Para a classe Conta, por intermédio de suas classes herdeiras, foram encontradas as classes Atendimento por balcão, Ordem de Serviço e Compra. Para a classe Ordem de Serviço foi encontrada a classe Veículo de cliente. Para as classes Compra e Pedido foi encontrada a classe Fornecedor.

Figura 5.12 - Reconhecimento de padrões

A classe Atendimento por balcão foi, portanto, classificada sem maiores problemas. É correto pensar que o Atendimento por balcão faz uma associação temporal entre Cliente e Empregado, isto é, existe um atendimento registrando um cliente que foi atendido por um empregado em uma determinada data. A classe Conta possui três relacionamentos candidatos, tendo que ser analisados aos pares. As três demais possuem apenas um relacionamento candidato e devem ser investigadas por possuírem classes agregadas. Ao analisar a classe Conta com os possíveis candidatos a “Static-1” e “Static-2”, o valor semântico foi importante para a decisão de descartar a possibilidade de ocorrência do *Association-Object*. Não faz sentido dizer que Conta é uma associação temporal entre Atendimento por balcão e Ordem de Serviço, ou entre Ordem de Serviço e Compra ou entre Atendimento por balcão e Compra. A classe Ordem de Serviço possui uma classe agregada, Peça da Ordem de Serviço, que por sua vez possui um relacionamento com a classe estática Peça. Assim pode-se eleger a classe Veículo de cliente como candidata a “Static-1” e Peça como candidata a “Static-2”. Pensando no valor semântico desse relacionamento, é correto que a Ordem de Serviço faz uma associação temporal entre um veículo consertado e a peça utilizada, sendo portanto classificado. O mesmo raciocínio foi utilizado para classificar “Compra/Fornecedor/Peça” e “Pedido/Fornecedor/Peça” como ocorrências do *Association-Object*.

O modelo de objetos foi então analisado em busca dos padrões *State across a collection* e *Behaviour across a collection*. As agregações existentes foram classificadas: Atendimento por balcão/Peça vendida, Ordem de Serviço/Peça da Ordem de Serviço, Ordem de Serviço/Mão-de-obra da Ordem de Serv., Compra/Peça comprada e Pedido/Peça pedida. Em todas elas existe pelo menos um atributo na classe candidata a “Collection” que é comum à classe correspondente candidata a “Member”. Por exemplo, em Atendimento por balcão, o atributo Data do Atendimento é também a Data em que a peça foi vendida. Também em todas as classes candidatas existem métodos, embora não estejam explicitamente presentes, que se aplicam à coleção como um todo. Por exemplo, na classe Compra, para calcular o Valor total é necessário invocar os métodos individuais de cada peça comprada. Dessa forma, as quatro agregações encontradas são ocorrências simultaneamente dos padrões *State across a collection* e *Behaviour across a collection*.

5.4.2 – Exemplo de implementação de padrão

Após ter sido feito o reconhecimento de padrões, partiu-se para a reengenharia do sistema legado, adotando-se os padrões identificados. A funcionalidade do sistema legado foi mantida durante a reengenharia, embora a interface tenha sido modernizada e adaptada ao estilo Windows.

É ilustrada nesta seção a implementação das duas instâncias do padrão *Type-Object* e de uma das instâncias do padrão *Association-Object* no sistema estudado. Escolheu-se a linguagem Delphi [Can 97, Can 98] para fazer a reengenharia do sistema, por ser uma linguagem comumente utilizada em sistemas de informação comerciais.

Para facilitar a implementação de instâncias do padrão *Type-Object* é proposto um esqueleto de implementação para ele, que agiliza o reuso, bastando para tal particularizar o esqueleto para cada caso particular. O mesmo poderia ser feito com os demais padrões recorrentes vistos na seção 5.2, o que é sugerido como trabalho futuro.

O exemplo de implementação completo do padrão “*Type-Object*” é objeto de um relatório técnico [Bra 98b], no qual constam as listagens de suas duas instanciações, bem como diretrizes para utilizar as facilidades do ambiente Delphi na particularização do padrão para cada instância.

5.4.2.1 – Esqueleto de implementação do padrão *Type-Object*

A figura 5.13 mostra o modelo de objetos do padrão *Type-Object* para o domínio de aplicação aqui abordado. Foram dados nomes genéricos aos atributos e métodos das classes “Class” e “TypeClass”. Os métodos que constam dessa figura foram obtidos após a programação de uma das instâncias do *Type-Object*, que permitiu identificar quais métodos seriam necessários para implementar as operações relativas às classes envolvidas no padrão. Os atributos “Identificador” e “Descritor”, presentes em ambas as classes, são característicos do domínio de sistemas de informação, no qual objetos estáticos possuem uma chave primária para identificá-los e uma chave secundária para descrevê-los de alguma forma. Essa foi uma das exigências para que o padrão fosse reconhecido, conforme a segunda diretriz da seção 5.3.1. O atributo “Tipo” da classe “Class” foi necessário para fazer a ligação com a classe “TypeClass”, já que é utilizada uma base de dados relacional para prover a persistência dos objetos.

Figura 5.13 - Modelo de Objetos detalhado do padrão Type-Object

Com base nesses atributos e métodos com nomes genéricos, foi feito um esqueleto de código para a definição das classes do padrão *Type-Object*. A definição das classes “Class” e “TypeClass” é ilustrada na figura 5.14 e o corpo de alguns métodos consta da figura 5.15.

O padrão “CRUD” [Yod 98], que é um dos padrões sugeridos para conectar objetos a bases de dados relacionais descritos na seção 2.5, foi utilizado na implementação no que se refere ao acesso à base de dados do Delphi. Assim, métodos específicos para acesso às tabelas de dados foram isolados em uma camada. Esses métodos só podem ser chamados pelos métodos da própria classe. Os métodos ilustrados na figura 5.13 são os métodos de interface da classe, isto é, os métodos que podem ser invocados por quaisquer outras classes do sistema.

Para implementar a camada de acesso às tabelas da base de dados foi utilizado o recurso “*DataModules*” (módulos de dados) do Delphi. Um “módulo de dados” contém um componente para fazer referência à tabela em si e alguns componentes para efetuar as operações básicas, em SQL, referentes à inserção, atualização, exclusão e recuperação de dados. Na verdade, um único objeto pode ser armazenado em mais de uma tabela e, nesse caso, o módulo de dados representa o objeto, contendo os componentes necessários para relacioná-lo com as respectivas tabelas. Dessa forma, cada classe faz uso de um único módulo de dados, no qual se concentram as tabelas necessárias para tornar seus objetos permanentes. No caso do padrão *Type-Object*, como cada uma das classes é armazenada em uma única tabela, são necessários dois módulos de dados, cada qual fazendo referência a uma das tabelas.

1	type // Definição da Classe Class do padrao Type_Object
2	AClass = class
3	public // Métodos da interface
4	procedure Salvar;
5	function Excluir:Boolean;
6	function BuscaIdent(x:Integer):Boolean; // Busca objeto por ident
7	function BuscaDescr(x:String):Boolean; // Busca objeto por descr
8	procedure BuscaDescrTodos(x:String; ListaFinal: TListBox); // Busca todos objetos
9	c/descr
10	function ExisteIdent(x:Integer): Boolean; // verifica a existencia por ident
11	function ExisteDescr(x:String): Boolean; // verifica a existencia por descr
12	function GetIdentificador: Integer;
13	function GetDescritor: String;
14	function GetTipo: ATypeClass;
15	procedure SetIdentificador(x: Integer);
16	procedure SetDescritor(x: String);
17	procedure SetTipo(tipo: ATypeClass);
18	function Novo: Integer;
19	private // Atributos Privados
20	Identificador: Integer;
21	Descritor: String;
22	UmTipo: ATypeClass;
23	// Métodos que fazem acesso as tabelas do banco de dados
24	procedure AtribuirObjetoAParametro(x:TQuery);
25	procedure AtribuirModeloAObjeto(x:TQuery);
26	procedure Crie;
27	procedure Atualize;
28	procedure Elimine;
29	function CarregueIdent(x: Integer; atribuir:Boolean): Boolean;
30	function CarregueDescr(x:String; atribuir:Boolean): Boolean;
31	function CarregueDescrTodos(x: String; Lista: TListBox): Boolean;
32	function Ultimo: Integer;
	end;
	(a)
33	type // Definição da Classe TypeClass do padrao Type_Object
34	ATypeClass = class
35	public // Métodos da interface
36	procedure Salvar;
37	function Excluir:Boolean;
38	function BuscaIdent(x:Integer):Boolean; // Busca por ident
39	function BuscaDescr(x:String):Boolean; // Busca por descr
40	procedure BuscaDescrTodos(x:String; ListaFinal: TListBox); // Busca por todos c/descr
41	function ExisteIdent(x:Integer): Boolean; // verifica a existencia por ident
42	function ExisteDescr(x:String): Boolean; // verifica a existencia por descr
43	function GetIdentificador: Integer;
44	function GetDescritor: String;
45	procedure SetIdentificador(x: Integer);
46	procedure SetDescritor(x: String);
47	function Novo: Integer;
48	private // Atributos Privados
49	Identificador: Integer;
50	Descritor: String;
51	// Métodos que fazem acesso as tabelas do banco de dados
52	procedure AtribuirObjetoAParametro(x:TQuery);
53	procedure AtribuirModeloAObjeto(x:TQuery);
54	procedure Crie;
55	procedure Atualize;
56	procedure Elimine;
57	function CarregueIdent(x: Integer; atribuir:Boolean): Boolean;
58	function CarregueDescr(x:String; atribuir:Boolean): Boolean;
59	procedure CarregueTodos(Lista: TListBox);
60	function CarregueDescrTodos(x: String; Lista: TListBox): Boolean;
61	function Ultimo: Integer;
62	end;
	(b)

Figura 5.14 - Esqueleto da declaração da classe “Class” do padrão Type-Object

<pre>// Atribuir valor a Identificador procedure AClass.SetIdentificador(x: Integer); begin Identificador:=x; end;</pre>	(a)
<pre>// Fornecer o Descritor function AClass.GetDescritor: String; begin GetDescritor:=Descritor; end;</pre>	(b)
<pre>// Obter o último identificador registrado na tabela function AClass.Ultimo: Integer; var max: Integer; begin max := 0; with DataModuleClass.TabClass do begin First; while not EOF do begin if Fields[0].AsInteger > max then max := Fields[0].AsInteger; Next; end; end; Ultimo:=max; end;</pre>	(c)
<pre>// Salvar um objeto procedure ATypeClass.Salvar; begin if (not BuscaIdent(Identificador)) and (not BuscaDescr(Descritor)) then // Objeto da classe Class não cadastrado, inseri-lo na tabela Crie else // Objeto da classe Class já cadastrado, atualizar atributos não-chave Atualize; end;</pre>	(d)

Figura 5.15 - Esqueleto de alguns métodos do padrão Type-Object

O módulo de dados referente à classe “Class” é exemplificado na figura 5.16. Ele possui um componente “TTable”, que faz referência à tabela física; um componente “TDataSource” usado para definir a origem dos dados; e cinco componentes “Tquery”: “Inserção”, para inserção de registro na tabela; “Atualização”, para gravar apenas os atributos não-chave da tabela; “Exclusao”, para excluir um registro da tabela, dada sua chave primária; “SelecionaPrim”, para recuperar registros pela chave primária; e “SelecionaSec”, para recuperar registros pela chave secundária. Possui também um procedimento para fazer a gravação permanente dos dados ao fechar a tabela.

Os métodos que fazem o acesso às tabelas da base de dados podem ser vistos nas linhas 23 a 31 e 52 a 61 da figura 5.14, como métodos privados (cláusula *private*, em Delphi). Analisando um desses métodos, como por exemplo o método “Ultimo” da classe “AClass” (figura 5.15 (c)), pode-se notar a referência efetuada ao módulo de dados “DataModuleClass” da classe “Class”. Graças à utilização do padrão “CRUD” [Yod 98], esses são os únicos métodos que devem saber detalhes das tabelas da base de dados, tais como nome das tabelas, localização, nome e posição de campos, etc. Assim, se a base de dados sofrer alguma alteração, poucos métodos precisarão ser modificados.

```

type TdataModuleClass = class(TdataModule)
  TabClass: TTable;
  DataSource1: TDataSource;
  Insercao: TQuery; // insere novo registro na tabela
  Atualizacao: TQuery; // atualiza campos nao chave
  Exclusao: TQuery; // exclui registro da tabela
  SeleccionaPrim: TQuery; // seleciona dado um identificador
  SeleccionaSec: TQuery; // seleciona dado um descritor
  procedure TabClassBeforeClose(DataSet: TDataSet);
end;

var
  DataModuleClass: TDataModuleClass;

...

```

Figura 5.16 - Esqueleto de um dos módulos de dados do padrão Type-Object

O código SQL a ser executado para cada um dos componentes do módulo de dados não fica disponível no código fonte desse módulo de dados, sendo armazenado na sua cláusula “SQL”. As figuras 5.17(a) e (b) ilustram o código SQL para os componentes “Insercao” e “SeleccionaPrim” do módulo de dados “Class”, respectivamente.

```

insert into "CLASS.db"
  (IDENTIFICADOR_MODELO, DESCRITOR_MODELO, TIPO_MODELO)
values
  (:IDENTIFICADOR, :DESCRITOR, :TIPO)

```

(a)

```

select * from "CLASS.db"
where IDENTIFICADOR_MODELO=:IDENTIFICADOR

```

(b)

Figura 5.17 - Código SQL de alguns componentes do módulo de dados “Class”

Esses esqueletos de código podem ser utilizados em qualquer sistema no qual o padrão *Type-Object* seja identificado. Para tal, deve-se fazer a substituição dos nomes genéricos para nomes específicos da instância encontrada, tanto na definição da classe quanto no módulo de dados correspondente. Um procedimento passo a passo de como essa substituição deve ser feita utilizando os recursos do Delphi 3.0 é dado em [Bra 98b].

5.2.4.2 – Instanciação do padrão Type-Object para Peça/Fabricante

A figura 5.18 mostra o modelo de objetos para a instância do padrão *Type-Object*, reconhecida na seção anterior para o par de classes “Peça/Fabricante”. Nota-se que alguns atributos foram acrescentados à classe Peça em relação à classe “Class” do padrão, bem como os métodos para lidar com esses novos atributos.

Figura 5.18 - Modelo de Objetos de uma instância do padrão Type-Object

A definição das classes e dos módulos de dados foi feita com base no esqueleto de código mostrado nas figuras 5.14 a 5.16. As figuras 5.19 e 5.20 mostram as declarações das classes Peça e Fabricante, respectivamente. A figura 5.21 mostra os mesmos métodos da figura 5.15 agora instanciados para as classes Peça e Fabricante. Além de fazer a substituição de nomes genéricos para nomes específicos da instanciação, alguns métodos de acesso às tabelas devem ser revisados

para adaptarem-se ao caso instanciado. Quando outros atributos são adicionados à classe, deve ser feita a inclusão de algumas linhas nos métodos responsáveis por atribuir os valores dos objetos aos parâmetros da SQL e vice-versa. Devem também ser incluídos os métodos para atribuir e recuperar o valor de um atributo (“Set” e “Get”, respectivamente).

```

type                                     // Definição da Classe Peça
Peca = class
public  // Métodos da interface
procedure Salvar;
function Excluir:Boolean;
function BuscaCod(x:Integer):Boolean; // Busca objeto por cod
function BuscaDescr(x:String):Boolean; // Busca objeto por descr
procedure BuscaDescrTodos(x:String; ListaFinal: TListBox); // Busca todos objetos
c/descr
function ExisteCod(x:Integer): Boolean; // verifica a existencia por cod
function ExisteDescr(x:String): Boolean; // verifica a existencia por descr
function GetCodigo: Integer;
function GetDescricao: String;
function GetTipo: Fabricante;
function GetCusto: Currency;
function GetPreco: Currency;
procedure SetCodigo(x:Integer);
procedure SetDescricao(x:String);
procedure SetCusto(c1:Currency);
procedure SetPreco(p1:Currency);
procedure SetTipo(tip:Fabricante);
function Novo: Integer;
private // Atributos Privados
Codigo: Integer;
Descricao: String;
UmTipo: Fabricante;
Custo: Currency;
Preco: Currency;
// Métodos que fazem acesso as tabelas do banco de dados
procedure AtribuirObjetoAParametro(x:TQuery);
procedure AtribuirModeloAObjeto(x:TQuery);
procedure Crie;
procedure Atualize;
procedure Elimine;
function CarregueCod(x:Integer; atribuir:Boolean): Boolean;
function CarregueDescr(x:String; atribuir:Boolean): Boolean;
function CarregueDescrTodos(x:String; Lista:TListBox): Boolean;
function Ultimo: Integer;
end;

```

Figura 5.19 - Código em Delphi referente à declaração da classe Peça

A figura 5.22 mostra o código referente ao módulo de dados criado para fazer acesso à tabela “Peca”. Ele foi criado com base no módulo de dados genérico mostrado na figura 5.16, fazendo a devida adaptação de nomes e inclusão de novos atributos para o caso específico. Deve-se ressaltar que essa mudança de nomes requer alguns cuidados especiais no ambiente Delphi, como por exemplo: desativar a propriedade “Active” do componente “TTable” antes de informar o nome da base de dados, ativando-a novamente logo em seguida; modificar a propriedade “Name” do componente “TDataModule” antes de efetuar a mudança automática dos demais

nomes; e colocar o nome da base de dados em todos os componentes “TQuery” de forma manual, embora possa ser usado o recurso da tecla “Shift” para agilizar essa colocação. Ainda que esse procedimento possa parecer um tanto complicado, a adaptação de um módulo de dados não leva mais do que cinco minutos. Conforme já foi dito anteriormente, o procedimento passo a passo para efetuar essa mudança de forma racional está detalhado em [Bra 98b]. Um exemplo de como o módulo de dados “DataModulePeca” é utilizado pelo método “Ultimo” da classe “Peca” pode ser visto na figura 5.21 (c).

```

type                                // Definição da Classe Fabricante
Fabricante = class
public                               // Métodos da interface
procedure Salvar;
function Excluir: Boolean;
function BuscaCod(x: Integer): Boolean; // Busca por cod
function BuscaNome(x: String): Boolean; // Busca por nome
procedure BuscaNomeTodos(x: String; ListaFinal: TListBox); // Busca por todos c/nome
function ExisteCod(x: Integer): Boolean; // verifica a existencia por cod
function ExisteNome(x: String): Boolean; // verifica a existencia por nome
function GetCodigo: Integer;
function GetNome: String;
procedure SetCodigo(x: Integer);
procedure SetNome(x: String);
function Novo: Integer;
private                             // Atributos Privados
Codigo: Integer;
Nome: String;
// Métodos que fazem acesso as tabelas do banco de dados
procedure AtribuirObjetoAParametro(x: TQuery);
procedure AtribuirModeloAObjeto(x: TQuery);
procedure Crie;
procedure Atualize;
procedure Elimine;
function CarregueCod(x: Integer; atribuir: Boolean): Boolean;
function CarregueNome(x: String; atribuir: Boolean): Boolean;
function CarregueNomeTodos(x: String; Lista: TListBox): Boolean;
procedure CarregueTodos(Lista: TListBox);
function Ultimo: Integer;

end;

```

Figura 5.20 - Código em Delphi referente à declaração da classe Fabricante

As figuras 5.23 (a) e (b) mostram o código SQL executado pelos componentes “Insercao” e “SelecionaPrim”, respectivamente, quando eles são invocados por algum método. Nota-se que, em relação ao esqueleto de código mostrado na figura 5.17, foram acrescentados os outros dois atributos da classe, bem como os parâmetros correspondentes, além de mudar os nomes dos atributos comuns de acordo com as colunas da tabela física. Na implementação das tabelas adotou-se a convenção de colocar o sufixo “_MODELO” nos nomes das colunas das tabelas, para diferenciá-los dos nomes dos atributos na classe correspondente.

<pre>// Atribuir valor a Codigo procedure Peca.SetCodigo(x: Integer); begin Codigo:=x; end;</pre>	(a)
<pre>// Fornecer o Descricao function Peca.GetDescricao: String; begin GetDescricao:=Descricao; end;</pre>	(b)
<pre>// Obter o último identificador registrado na tabela function Peca.Ultimo: Integer; var max: Integer; begin max := 0; with DataModulePeca.TabPeca do begin First; while not EOF do begin if Fields[0].AsInteger > max then max := Fields[0].AsInteger; Next; end; end; Ultimo:=max; end;</pre>	(c)
<pre>// Salvar um objeto procedure Fabricante.Salvar; begin if (not BuscaCod(Codigo)) and (not BuscaNome(Nome)) then // Objeto da classe não cadastrado, inseri-lo na tabela Crie else // Objeto da classe já cadastrado, atualizar atributos não-chave // Atualize; opcao desativada, pois nao existem mais atributos alem dos atributos chave end;</pre>	(d)

Figura 5.21 - Código em Delphi referente a alguns métodos do padrão reconhecido

```
type TDataModulePeca = class(TDataModule)
  TabPeca: TTable;
  DataSource1: TDataSource;
  Insercao: TQuery;
  Atualizacao: TQuery;
  Exclusao: TQuery;
  SeleccionaPrim: TQuery;
  SeleccionaSec: TQuery;
  procedure TabPecaBeforeClose(DataSet: TDataSet);
end;

var
  DataModulePeca: TDataModulePeca;
. . .
```

Figura 5.22 - Módulo de dados da classe Peça

<pre> insert into "PECA.db" (CODIGO_MODELO, DESCRICAO_MODELO, FABRICANTE_MODELO, CUSTO_MODELO, PRECO_MODELO) values (:CODIGO, :DESCRICAO, :TIPO, :CUSTO, :PRECO) </pre>	(a)
<pre> select * from "PECA.db" where CODIGO_MODELO=:CODIGO </pre>	(b)

Figura 5.23 - Código SQL de alguns componentes do módulo de dados “Peca”

A figura 5.24 ilustra a interface para cadastro de peças no sistema exemplo. Essa interface foi elaborada de forma padronizada, de maneira que possa ser facilmente reutilizada. Além disso, o usuário se habitua com seus diversos botões, facilitando o uso e o treinamento.

A figura 5.25 mostra alguns trechos do código relativo à interface da figura 5.24. Nele pode-se observar os seguintes fatos: a utilização das classes do padrão pelo formulário (linha 5); a declaração dos objetos da classe (linhas 18 e 19); a instanciação dos objetos das classes “Peça” e “Fabricante” sempre que o formulário de cadastro de peça é ativado (linhas 24 e 25); as chamadas de métodos (linhas 27, 28 e 35); e a liberação de memória ocupada pelos objetos sempre que o formulário é desativado (linhas 42 e 43).

Figura 5.24 - Interface da operação de cadastro de peças

	// Formulário para Cadastro de Peças
1	unit f_peca;
2	interface
3	uses
4	Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs,
5	StdCtrls, ClassePeca, ClasseFabricante, Mask;
6	type
7	TFFormPeca = class(TForm)
8	CodigoVisao: TEdit;
9	DescricaoVisao: TEdit;
10	ListaFabricantesVisao: TListBox;
11	FabricanteVisao: TMaskEdit;
	...
12	Salvar: TButton;
13	Sair: TButton;
	...
14	procedure SalvarClick(Sender: TObject);
	...
15	end;
16	var
17	FormPeca: TFormPeca;
18	p: Peca;
19	tp:Fabricante;
20	implementation
21	{ \$R *.DFM }
22	procedure TFormPeca.FormActivate(Sender: TObject);
23	begin
24	p:=Peca.Create;
25	tp:=Fabricante.Create;
26	LimparControles(True);
27	tp.BuscaNomeTodos('', ListaFabricantesVisao);
28	p.SetTipo(tp);
29	end;
30	procedure TFormPeca.SalvarClick(Sender: TObject);
31	begin
32	if (CompareStr(Trim(DescricaoVisao.Text), '') <> 0) then
33	begin
34	AtribuirVisaoAObjeto;
35	p.Salvar;
36	end
37	else
38	ShowMessage('Não é possível salvar placa em branco!');
39	end;
	...
40	procedure TFormPeca.FormDeactivate(Sender: TObject);
41	begin
42	p.free;
43	tp.free;
44	end;
45	end.

Figura 5.25 - Trecho de código referente ao formulário de cadastro de peças

5.4.2.3 – Instanciação do padrão Type-Object para Veículo de Cliente/Tipo de Veículo

A figura 5.26 mostra o modelo de objetos para a outra instância do padrão *Type-Object*, também reconhecida na seção anterior, para o par de classes “Veículo de Cliente/Tipo de Veículo”. O código com a declaração dessas classes está na figura 5.27 e os mesmos métodos da figura 5.15 instanciados a esse caso estão na figura 5.28.

Figura 5.26 - Modelo de Objetos da outra instância do padrão *Type-Object*

Ao analisar o modelo da figura 5.13 percebe-se que, além da semelhança no nível mais alto de abstração que levou à generalização, há também uma semelhança entre as classes “Class” e “TypeClass”. Isso pode levar à construção de uma superclasse de nível de abstração maior, englobando os atributos e métodos comuns a essas duas classes. Então, a instanciação para os dois casos pode ser feita usando herança. Na verdade houve uma tentativa de fazer essa superclasse em Delphi, mas surgiram problemas na instanciação de listas (“listbox”) e das tabelas da base de dados. Como esses problemas não puderam ser resolvidos a tempo de serem incluídos nesta dissertação, fica como sugestão para trabalhos futuros incluir mais essa generalização.

Comparando a figura 5.19 com a figura 5.27 (a), a figura 5.20 com a 5.27 (b) e a figura 5.21 com a 5.28, pode-se perceber que o reuso é quase total, devendo apenas ser mudado o nome da classe e de seus atributos. Seus métodos podem ter o nome mudado, embora não seja necessário.

```

type                                     // Definição da Classe Veiculo
Veiculo = class
public                                     // Métodos da interface
procedure Salvar;
function Excluir:Boolean;
function BuscaCod(x:Integer):Boolean; // Busca objeto por cod
function BuscaPlaca(x:String):Boolean; // Busca objeto por placa
procedure BuscaPlacaTodos(x:String; ListaFinal: TListBox); // Busca todos objetos c/placa
procedure BuscaClienteTodos(x:Integer; ListaFinal: TListBox); // Busca todos objetos por cliente
function ExisteCod(x:Integer): Boolean; // verifica a existencia por cod
function ExistePlaca(x:String): Boolean; // verifica a existencia por placa
function GetNroSequencial: Integer;
function GetPlaca: String;
function GetCliente: Cliente;
function GetTipo: TipoDeVeiculo;
procedure SetNroSequencial(x: Integer);
procedure SetPlaca(x: String);
procedure SetTipo(tip: TipoDeVeiculo);
procedure SetCliente(cli: Cliente);
function Novo: Integer;
private // Atributos Privados
NroSequencial: Integer;
Placa: String;
UmTipo: TipoDeVeiculo;
UmCliente: Cliente;
// Métodos que fazem acesso as tabelas do banco de dados
procedure AtribuirObjetoAParametro(x:TQuery);
procedure AtribuirModeloAObjeto(x:TQuery);
procedure Crie;
procedure Atualize;
procedure Elimine;
function CarregueCod(x: Integer; atribuir:Boolean): Boolean;
function CarreguePlaca(x:String; atribuir:Boolean): Boolean;
function CarreguePlacaTodos(x: String; Lista: TListBox): Boolean;
function CarregueClienteTodos(x:Integer; Lista: TListBox):Boolean;
function Ultimo: Integer;
end;

```

(a)

```

type                                     // Definição da Classe TipoDeVeiculo
TipoDeVeiculo = class
public                                     // Métodos da interface
procedure Salvar;
function Excluir:Boolean;
function BuscaCod(x:Integer):Boolean; // Busca por cod
function BuscaMarcaTipo(x:String):Boolean; // Busca por marca
procedure BuscaMarcaTipoTodos(x:String; ListaFinal: TListBox); // Busca por todos c/marca
function GetCodigo: Integer;
function GetMarcaTipo: String;
procedure SetCodigo(atr1: Integer);
procedure SetMarcaTipo(atr2: String);
function Novo: Integer;
private // Atributos Privados
Codigo: Integer;
MarcaTipo: String;
// Métodos que fazem acesso as tabelas do banco de dados
procedure AtribuirObjetoAParametro(x:TQuery);
procedure AtribuirModeloAObjeto(x:TQuery);
procedure Crie;
procedure Atualize;
procedure Elimine;
function CarregueCod(x: Integer): Boolean;
function CarregueMarcaTipo(x: String): Boolean;
procedure CarregueTodos(Lista: TListBox);
function CarregueMarcaTipoTodos(x: String; Lista: TListBox): Boolean;
function Ultimo: Integer;
end;

```

(b)

Figura 5.27 - Código em Delphi da declaração das classes Veículo e Tipo de Veículo

```
// Atribuir valor a NroSequencial
```

<pre> procedure Veiculo.SetNroSequencial(x: Integer); begin NroSequencial:=x; end; </pre>	(a)
<pre> // Fornecer o Placa function Veiculo.GetPlaca: String; begin GetPlaca:=Placa; end; </pre>	(b)
<pre> // Obter o último identificador registrado na tabela function Veiculo.Ultimo: Integer; var max: Integer; begin max := 0; with DataModuleVeiculo.TabVeiculo do begin First; while not EOF do begin if Fields[0].AsInteger > max then max := Fields[0].AsInteger; Next; end; end; Ultimo:=max; end; end; end; </pre>	(c)
<pre> // Salvar um objeto procedure TipoDeVeiculo.Salvar; begin if (not BuscaCod(Codigo)) and (not BuscaMarcaTipo(MarcaTipo)) then // Objeto da classe Class não cadastrado, inseri-lo na tabela Crie else // Objeto da classe Class já cadastrado, atualizar atributos não-chave // Atualize; nao existem mais atributos alem dos atributos chave end; end; </pre>	(d)

Figura 5.28 - Código em Delphi referente a alguns métodos da segunda instância do padrão

Neste trabalho optou-se por adaptar os nomes dos métodos de busca, atribuição e recuperação, de acordo com os atributos “identificador” e “descriptor” de cada classe. No entanto, se toda a equipe de desenvolvimento estiver habituada ao uso do padrão, pode ser desejável manter o nome genérico para facilitar a manutenção do sistema. Por exemplo, ao fazer uma busca pela chave primária, o desenvolvedor pode achar mais fácil utilizar sempre o método “BuscaIdent” do que ter que lembrar o nome do atributo para decidir-se por “BuscaCod”, “BuscaNro”, etc. Aliás, quando é utilizada a herança os nomes dos métodos são únicos nas classes e subclasses, podendo sobrepor-se apenas o seu conteúdo.

5.4.2.4 – Instanciação do Padrão Association-Object para Ordem de Serviço

A figura 5.29 mostra o modelo de objetos contendo as classes e relacionamentos necessários para implementar a operação “Abre_Ordem_de_Serviço”. Essa operação foi utilizada como exemplo nas seções 4.2 e 4.3, nas quais houve reengenharia com mudança de orientação e com mudança automática de linguagem para Java, respectivamente. Nesta seção essa mesma operação sofre reengenharia com mudança de linguagem, adotando os padrões reconhecidos.

Figura 5.29 - Modelo de Objetos para a operação “Abre_Ordem_de_Serviço”

Na “Abertura da Ordem de Serviço”, efetuada quando o cliente leva o veículo para ser consertado, estão envolvidas as classes “Cliente”, “Veículo de Cliente”, “Tipo de Veículo” e “Ordem de Serviço”, como pode ser visto na figura 5.29 e também nas figuras 3.9, 3.19 e 4.2. Dessas classes, “Veículo de Cliente” desempenha o papel “Static-2” no padrão *Association-Object*, mas já foi implementada na seção 5.4.2.3, juntamente com a classe “Tipo de Veículo”, por fazerem parte do padrão *Type-Object*. A classe “Ordem de Serviço” desempenha o papel “Association”, conforme visto na seção 5.4.1. A classe “Cliente” precisa ser implementada

devido à funcionalidade da operação “Abre_Ordem_de_Serviço”. O papel “Static-1” é desempenhado pela classe “Peça”, por intermédio da classe “Peça da Ordem de Serv.”, conforme já explicado na seção 5.4.1. A classe “Peça” só é usada no “Fechamento da Ordem de Serviço”, quando o veículo acabou de ser consertado e as peças e mão-de-obra são cadastradas. Como o interesse aqui, para fins de comparação, é pela operação “Abre_Ordem_de_Serviço”, o padrão *Association-Object* é implementado então de forma parcial, deixando de tratar da parte que lida com a classe “Static-1”.

A classe “Cliente” foi implementada com base no esqueleto da classe “Class”, mostrada na figura 5.14 (a), já que possui características semelhantes às classes do padrão *Type-Object*: um atributo identificador e um atributo descritor (“Codigo” e “Nome”, respectivamente). Os métodos também puderam ser aproveitados, fazendo apenas a adaptação de nomes e inclusão dos demais atributos da classe “Cliente”.

Para implementação da classe “OrdemDeServico”, apesar de não haver tanta semelhança quanto nos outros casos, foi também utilizado o esqueleto da classe “Class”. Diversos métodos tiveram que ser eliminados, já que “OrdemDeServico” não possui um atributo descritor. Porém, o processo de reuso foi válido, tendo havido um rendimento bem maior do que se a classe tivesse sido construída a partir do “zero”. A figura 5.30 mostra a definição dessa classe e as figuras 5.31 (a) e (b) mostram o corpo de alguns de seus métodos. Os métodos mostrados na figura 5.31 (a) correspondem ao método “OBTPROXO”, cujo código segmentado consta na figura 4.4 (a) e cujo código transformado para Java consta na figura 4.8 (a). Os métodos mostrados na figura 5.31 (b) correspondem ao método “GRORDSER”, cujo código segmentado consta na figura 4.4 (d) e cujo código transformado para Java consta na figura 4.8 (d). O método “GRORDSER” foi desmembrado em quatro métodos para isolar o acesso às tabelas da base de dados pelo uso do padrão “CRUD”, conforme explicado na seção 5.4.2.1.

Alguns métodos que foram necessários devido às características da linguagem Clipper não possuem o correspondente na implementação em Delphi, como, por exemplo, os métodos para abertura e fechamento de arquivos e os métodos para leitura de dados de interface. Em Delphi, não existe um método que implemente a lógica da operação “Abre_Ordem_de_Serviço”. Em vez disso, o formulário de entrada da Ordem de Serviço é que faz a chamada dos métodos e sua sincronização, de acordo com a lógica desejada. Esse formulário de entrada também faz a leitura de dados por intermédio de componentes visuais. A abertura e o fechamento dos arquivos

da base de dados são de responsabilidade do módulo de dados de cada classe, sendo programados pelo desenvolvedor em tempo de projeto.

```

type                                     // Definição da Classe Ordem de Serviço
OrdemDeServico = class
public  // Métodos da interface
procedure Salvar;
function Excluir:Boolean;
function BuscaNroOS(x:Integer):Boolean; // Busca objeto por NroOS
function ExisteNroOS(x:Integer): Boolean; // verifica a existencia por NroOS
function GetNroOrdemServ: Integer;
function GetDataEntrada: TDateTime;
function GetDefeitos: String;
function GetPrecoTotal: Currency;
function GetDataSaida: TDateTime;
function GetVeiculo: Veiculo;
procedure SetNroOrdemServ(x: Integer);
procedure SetDataEntrada(x: TDateTime);
procedure SetDefeitos(x: String);
procedure SetPrecoTotal(x: Currency);
procedure SetDataSaida(x: TDateTime);
procedure SetVeiculo(x: Veiculo);
function Novo: Integer;
private // Atributos Privados
NroOrdemServ: Integer;
DataEntrada: TDateTime;
Defeitos: String[80];
PrecoTotal: Currency;
DataSaida: TDateTime;
UmVeiculo: Veiculo;
// Métodos que fazem acesso as tabelas do banco de dados
procedure AtribuirObjetoAParametro(x:TQuery);
procedure AtribuirModeloAObjeto(x:TQuery);
procedure Crie;
procedure Atualize;
procedure Elimine;
function CarregueNroOS(x: Integer; atribuir:Boolean): Boolean;
function Ultimo: Integer;
end;

```

Figura 5.30 - Definição da classe “OrdemDeServiço”

A figura 5.32 mostra a tela de abertura do sistema, no momento em que a opção de Abertura da Ordem de Serviço é requisitada pelo usuário, por intermédio de um menu. A figura 5.33 mostra a interface para Abertura da Ordem de Serviço. Deve-se observar que a disposição dos botões na parte inferior da janela é semelhante à das demais operações do sistema, como a mostrada na figura 5.24. Três botões foram adicionados: “Cliente...”, para permitir que o usuário faça a busca de clientes já cadastrados com maior rapidez, inclua novos clientes ou modifique dados de um determinado cliente; “Novo Veículo...”, para inclusão de um novo veículo, se for o caso; e “Eliminar Veículo”, para eliminar os veículos da lista que não pertençam mais ao cliente.

```

// Obter um novo nro de objeto
function OrdemDeServico.Novo: Integer;
begin
  Novo:=Ultimo+1;
end;

// Obter o último nro de ordem de serviço registrado
function OrdemDeServico.Ultimo: Integer;
var max: Integer;
begin
  max := 0;
  with DataModuleOrdemDeServico.TabOrdemDeServico do
  begin
    First;
    while not EOF do
    begin
      if Fields[0].AsInteger > max then
        max := Fields[0].AsInteger;
      Next;
    end;
  end;
  Ultimo:=max;
end;

```

(a)

```

// Salvar um objeto
procedure OrdemDeServico.Salvar;
begin
  if (not ExisteNroOS(NroOrdemServ)) then
    // Objeto não cadastrado, inseri-lo na tabela
    Crie
  else
    // Objeto já cadastrado, atualizar atributos não-chave
    Atualize;
end;

// Criar novo registro
procedure OrdemDeServico.Crie;
begin
  AtribuirObjetoAParametro(DataModuleOrdemDeServico.Insercao);
  DataModuleOrdemDeServico.Insercao.ExecSQL;
end;

// Atualizar registro
procedure OrdemDeServico.Atualize;
begin
  AtribuirObjetoAParametro(DataModuleOrdemDeServico.Atualizacao);
  DataModuleOrdemDeServico.Atualizacao.ExecSQL;
end;

// Atribuir objeto aos parametros de busca
procedure OrdemDeServico.AtribuirObjetoAParametro(x:TQuery);
begin
  with x do
  begin
    ParamByName('NroOrdemServ').AsInteger:=NroOrdemServ;
    ParamByName('DataEntrada').AsDateTime:=DataEntrada;
    ParamByName('Defeitos').AsString:=Defeitos;
    ParamByName('DataSaida').AsDateTime:=DataSaida;
    ParamByName('PrecoTotal').AsCurrency:=PrecoTotal;
    ParamByName('CodigoVeiculo').AsInteger:=UmVeiculo.GetNroSequencial;
  end;
end;

```

(b)

Figura 5.31 - Alguns métodos da classe “OrdemDeServiço”

Figura 5.32 - Interface do sistema mostrando um dos menus

Abertura da Ordem de Serviço

Nro da Ordem de Serviço: Data de Entrada:

Cliente:

Veículos do Cliente:

Placa	Marca
CFU-3400	Fiat Tempra 16V
BEM-1234	Volkswagen Parati GLS

Veículo a ser consertado: Placa: Marca:

Defeitos Apresentados:

Figura 5.33 - Interface da operação de Abertura da Ordem De Serviço

--	--

```

1  unit f_ordemdeservico;           // Formulário para Cadastro de Veiculos de
2  Cliente
3  interface
4  uses
5      Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs,
6      ClasseVeiculo, ClasseTipoDeVeiculo, ClasseCliente, ClasseOrdemDeServico,
7      StdCtrls, Mask;
8  type
9      TformAberturaOrdemDeServico = class(TForm)
10         NroOrdemServVisao: TEdit;
11         PlacaVisao: TEdit;
12         Buscar: TButton;
13         NovaOS: TButton;
14         Salvar: TButton;
15         Sair: TButton;
16         Excluir: TButton;
17         ClienteVisao: TEdit;
18         BotaoCliente: TButton;
19         BotaoVeiculo: TButton;
20         DataEntradaVisao: TMaskEdit;
21         MarcaVisao: TEdit;
22         DefeitosVisao: TMemo;
23         ListaVeiculosVisao: TListBox;
24         Eliminarveiculo: TButton;
25         ...
26         procedure SalvarClick(Sender: TObject);
27         procedure LimparControles;
28         procedure AtribuirVisaoAObjeto;
29         procedure AtribuirObjetoAVisao;
30         procedure FormActivate(Sender: TObject);
31         procedure BuscarClick(Sender: TObject);
32         procedure NovaOSClick(Sender: TObject);
33         procedure SairClick(Sender: TObject);
34         procedure FormDeactivate(Sender: TObject);
35         procedure ExcluirClick(Sender: TObject);
36         procedure BotaoClienteClick(Sender: TObject);
37         procedure BotaoVeiculoClick(Sender: TObject);
38         procedure ClienteVisaoExit(Sender: TObject);
39         procedure ListaVeiculosVisaoClick(Sender: TObject);
40         procedure EliminarveiculoClick(Sender: TObject);
41     end;
42     ...
43     procedure TformAberturaOrdemDeServico.NovaOSClick(Sender: TObject);
44     begin
45         LimparControles;
46         NroOrdemServVisao.Text:=IntToStr(os.Novo);
47         os.SetNroOrdemServ(StrToInt(NroOrdemServVisao.Text));
48         DataEntradaVisao.text:=DateTimeToStr(date);
49         DataEntradaVisao.SetFocus;
50     end;
51     procedure TformAberturaOrdemDeServico.BotaoClienteClick(Sender: TObject);
52     begin
53         FormCliente.NomeVisao.Text:=ClienteVisao.Text;
54         FormCliente.FormActivate(self);
55         FormCliente.Buscar.Click;
56         FormCliente.ShowModal;
57         ClienteVisao.Text:=FormCliente.NomeVisao.Text;
58     end;
59     procedure TformAberturaOrdemDeServico.ClienteVisaoExit(Sender: TObject);
60     begin
61         if ((ClienteVisao.Text <> '') and (c.BuscaNome(ClienteVisao.Text+'%'))) then
62         begin
63             ClienteVisao.Text:=c.GetNome;
64             v.BuscaClienteTodos(c.GetCodigo,ListaVeiculosVisao);
65         end;
66     end;
67     ...

```

Figura 5.34 - Trechos do código referente ao formulário de Abertura da Ordem De Serviço

A figura 5.34 mostra trechos do código referentes ao formulário de abertura da Ordem de Serviço. Conforme citado anteriormente, esse formulário corresponde à operação

“Abre_Ordem_de_Serviço”. A lógica envolvida na operação é ditada pelos eventos possíveis em cada um dos componentes do formulário. Por exemplo, nas linhas 40 a 47 da figura 5.34 está o método executado quando o usuário ativa o botão “Nova OS” e nas linhas 48 a 55 está o método referente à ativação do botão “Cliente”. No primeiro caso, o método “Novo” da classe “OrdemDeServico” é invocado (linha 43) e os componentes do formulário são atualizados para refletir a nova ordem de serviço. No segundo caso, outro formulário de entrada de dados é ativado (linhas 50 a 53) e, ao retornar desse, faz-se a atualização do componente que mostra o nome do cliente (linha 54).

Conforme pode ser observado nas linhas 24 a 38, existem métodos associados à ativação de todos os botões do formulário e também métodos associados à navegação por alguns dos componentes de entrada de dados. Por exemplo, quando o usuário digita o nome do cliente e avança para outro componente, é gerado o evento “Exit”, que dispara o método “ClienteVisaoExit” (linha 36). Esse método dá o tratamento adequado a esse evento, mostrado nas linhas 56 a 63, que basicamente trata de verificar a existência de tal cliente e buscar seus veículos cadastrados, colocando-os na lista de veículos do cliente.

5.5 – Considerações Finais

A ocorrência de diversas instâncias do mesmo padrão no sistema aumenta significativamente o reuso, acelerando o processo de reengenharia. A estratégia proposta é centrada na importância de saber identificar a ocorrência de padrões, para poder tirar proveito deles. O código resultante do uso de padrões é claro e de fácil entendimento.

As diversas experiências de implementação efetuadas mostraram que o tempo gasto para instanciar o padrão *Type-Object* é significativamente menor do que se ele fosse programado a partir do zero. Em cerca de quarenta minutos pode-se ter uma instância do padrão programada e em condições de ser executada.

Algumas classes participam de mais de um padrão, como a “Veiculo de cliente”, que desempenha o papel “Class” no padrão *Type-Object* e o papel “Static-2” no padrão *Association-Object*. Nesse caso, após implementar um dos padrões já se tem a classe programada, o que facilita a implementação do outro padrão no qual ela ocorre.

CAPÍTULO 6 – Conclusões Finais e Sugestões para novas pesquisas

6.1 – Considerações Iniciais

Foi proposta nesta dissertação uma abordagem para reengenharia de sistemas, com uso de padrões de software para maximizar o reuso e melhorar a manutenibilidade. Algumas atividades intermediárias foram desenvolvidas. A engenharia reversa do sistema legado foi efetuada, tendo sido proposto um detalhamento do passo de abstração do Modelo de Análise do Sistema. A reengenharia com mudança de orientação foi realizada, transformando o sistema originalmente procedimental em um sistema com características da orientação a objetos. A reengenharia com mudança de linguagem de programação foi feita, transformando o código fonte de Clipper para Java. Foi, então, estabelecida uma estratégia para reconhecimento de alguns padrões recorrentes de sistemas de informação, a partir do modelo de objetos produzido na engenharia reversa. Finalmente, foi executada a reengenharia com uso de padrões de software, utilizando a linguagem Delphi.

Na seção 6.2 são feitas as conclusões gerais de todo o trabalho desenvolvido. Na seção 6.3 são apresentadas as contribuições deste trabalho e na seção 6.4 estão as sugestões para novas pesquisas.

6.2 – Conclusões Gerais

A engenharia reversa forneceu a documentação de análise orientada a objetos necessária para a reengenharia pretendida e para o reconhecimento de padrões.

A realização da reengenharia com mudança de orientação do paradigma de desenvolvimento do sistema obteve êxito. Um sistema originalmente desenvolvido com orientação procedimental foi transformado segundo o paradigma da orientação a objetos, sem mudança de funcionalidade ou linguagem de programação. Com isso, obteve-se maior compatibilidade entre o modelo de análise obtido pelo Fusion/RE e o código fonte. Se somente a engenharia reversa tivesse sido efetuada, haveria maior dificuldade em associar o código aos componentes do modelo de análise.

A realização da reengenharia com mudança automática de linguagem tornou possível a integração do processo geral Fusion/RE com a máquina Draco-Puc, pelo acréscimo de dois passos novos ao Fusion/RE e pela construção dos elementos necessários na máquina Draco-Puc para que ela pudesse trabalhar com outros domínios. A reconstrução de software para que ele possa ser executado em novas plataformas de hardware e software permite sua melhora sem perder a funcionalidade. A abordagem utilizada acrescenta a isso melhor documentação do sistema em níveis mais altos de abstração, essencial para uma boa manutenibilidade. O código Clipper poderia ter sido transformado para Java sem a segmentação prévia, mas dessa forma o código resultante não tiraria proveito dos recursos de orientação a objetos, resolvendo apenas o problema de portabilidade. A fase de segmentação é que garante as características do sistema resultante.

A identificação de padrões no modelo de análise obtido pela engenharia reversa tornou possível a implementação em Delphi com maior racionalidade. A estratégia para reconhecimento de padrões proposta mostrou-se eficiente nos casos em que foi aplicada.

A comparação do sistema legado em Clipper com os sistemas obtidos após a reengenharia (Sistema Segmentado Clipper, Sistema transformado para Java e Sistema em Delphi com uso de padrões), é mostrada na tabela 6.1.

Todos os sistemas obtidos pela reengenharia são mais fáceis de manter em relação ao código legado, mas existe uma vantagem marcante do código que usa padrões de software. Quanto à legibilidade, o código transformado automaticamente para Java deixa um pouco a desejar. Da forma como foi definida a transformação de comandos de Clipper para Java, a maioria

das funções Clipper foi implementada como um método em Java. O código resultante parece ser uma mistura do Clipper e do Java, não sendo compreendido facilmente nem por um programador Clipper, nem por um programador Java. As vantagens da reengenharia para Java foram quanto à portabilidade, já que essa linguagem é independente de plataforma, ao passo que as outras não, e quanto ao tempo gasto na transformação, que é insignificante após a segmentação e construção dos domínios apropriados. Deve-se observar que parte dos problemas notados podem ser resolvidos com a definição de outras estratégias de transformação, com base na experiência e aprendizado obtidos com este primeiro experimento.

O reuso é superior no sistema que usa padrões, já que são reusados desde os conceitos de análise e projeto até os detalhes de implementação. Todas as formas de reengenharia implicam

em custos. Porém, a segmentação parece ser mais viável economicamente, pois pode ser feita aos poucos, utilizando o próprio pessoal habituado à linguagem Clipper.

Tabela 6.1 – Comparação dos resultados

Característica	Sistema Legado em Clipper	Sistema Segmentado em Clipper	Sistema transformado para Java usando Draco-Puc	Sistema em Delphi, com uso de padrões
Legibilidade	Boa, pois o programa está estruturado	Boa, embora os parâmetros incluídos na segmentação aumentem o grau de acoplamento	Boa, pois a transformação mantém a estrutura do código segmentado, apesar do código não ser típico de um programa em Java.	Excelente, devido aos nomes mnemônicos e à estruturação permitida pelo uso de classes.
Manutenção	Dificultada devido às anomalias	Facilitada, pois os métodos ficam agrupados no arquivo de cada classe	Facilitada, apesar da legibilidade do código não ser boa	Facilitada, pois a estrutura do sistema é uniformizada
Reuso	Limitado a sistemas com funcionalidade semelhante	Bom, devido ao grande número de módulos com funcionalidade bem definida	Bom, devido ao grande número de módulos com funcionalidade bem definida, apesar do problema de legibilidade	Muito bom, pois abrange desde a análise, projeto até a implementação das classes, módulos de dados e formulários
Esforço para reengenharia	-	Pode ser feita de forma gradual, focalizando primeiramente os módulos de manutenção mais crítica. Não há necessidade de mudança de pessoal.	O maior custo está na fase de segmentação e na construção dos domínios origem e destino. Uma ferramenta de auxílio poderia diminuir esse custo. Há necessidade de pessoal especializado para uso dessa técnica.	Deve haver um custo no treinamento do pessoal no uso de padrões e, se for o caso, no uso de uma linguagem orientada a objetos.
Portabilidade	Específico para plataforma DOS	Específico para plataforma DOS	Independente de plataforma de hardware ou software	Específico para plataforma Windows
Interface com o usuário	Pobre	Pobre	Pobre	Rica
Usabilidade	Regular	Regular	Regular	Boa
Desempenho	Ótimo	Ótimo	Bom	Bom

Quanto à interface com o usuário, o sistema legado apresenta uma interface pobre, baseada em janelas do ambiente DOS, sem muitos recursos para facilitar sua operação. O sistema segmentado mantém essa interface, assim como o sistema transformado para Java. Desta forma, a reengenharia para Delphi é a única que fez a modernização da interface, oferecendo ao usuário todos os recursos dos ambientes atuais. Deve-se ressaltar que a mudança de interface

pode também ser feita no sistema em Java, para aproveitar melhor seus recursos, mas essa mudança deve ser feita manualmente, após o processo de transformação. A interface influi também na usabilidade, que pode ser melhorada empregando os recursos das linguagens modernas, tais como botões, listas, ajuda sensível ao contexto, etc.

O desempenho dos sistemas, em termos de tempo de execução, foi testado em máquinas Pentium 233 Mhz com Windows 95. Obteve-se melhor resultado nos sistemas Clipper do que nos sistemas Java e Delphi. Isso é principalmente devido ao fato de que aplicações DOS são bem mais enxutas em relação a aplicações WINDOWS.

A experiência de reuso efetuada durante este trabalho é extremamente compensadora. Quando comparado ao reuso que tenho feito dentro de minha prática profissional, conforme mencionado na seção 1.2, sua eficiência fica ainda mais evidente. Unidos às vantagens oferecidas pela orientação a objetos, os padrões aceleram o processo de desenvolvimento, por embutirem soluções para problemas na maioria das vezes comuns, mas cuja frequência de ocorrência acarreta tempo substancial de desenvolvimento.

6.3 – Contribuições deste trabalho

Além de alcançar o objetivo de reconhecimento de padrões de software úteis no reuso, ampliação de funcionalidade e manutenção de sistemas, vários subprodutos originaram-se deste trabalho: o detalhamento da abstração do modelo de análise do sistema, a abordagem para mudança de orientação do sistema como preparação para a mudança automática de linguagem e os esqueletos de classes para a implementação do padrão *Type-Object*.

A contribuição mais significativa é o fornecimento de uma estratégia para reconhecimento de padrões específicos de sistemas de informação, tanto na engenharia reversa de sistemas legados quanto na engenharia avante de sistemas. O uso de 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. A estratégia geral proposta pode ser seguida por outros engenheiros de software durante a engenharia reversa ou avante de sistemas. Por meio dessa estratégia o reuso poderá ser expandido a níveis mais altos de abstração, ao invés de se manter puramente no nível do código. Na engenharia reversa esses padrões podem ser reconhecidos após a recuperação do Modelo de Objetos do Sistema. Na engenharia avante esses padrões podem ser reconhecidos após o projeto

do sistema e utilizados na implementação. A realização da reengenharia com mudança de linguagem adotando os padrões reconhecidos permitiu a visualização dos padrões depois de programados em uma linguagem comercial, o Delphi no caso. Com isso, pôde-se avaliar a melhoria da manutenibilidade do sistema que possui padrões refletidos no código fonte.

O detalhamento da abstração do modelo de análise do sistema reforçou o método Fusion/RE, facilitando sua aplicação a sistemas de informação. A transição do Modelo de Objetos do Sistema Atual, construído com influência da implementação atual, para o Modelo de Objetos do Sistema, que deve refletir o domínio de aplicação num nível de abstração mais alto, é complexa. O detalhamento sugerido ameniza essa transição, diminuindo o salto que era antes dado entre um modelo e outro.

O estabelecimento de uma abordagem para mudança de orientação do paradigma de um sistema, de procedimental para orientado a objetos, por intermédio da segmentação, torna possível a reengenharia de mudança de orientação sem mudança de linguagem de programação. Sistemas importantes para as empresas, mas com alto custo de manutenção, podem passar por esse processo e ter facilitada sua manutenção. A segmentação elimina as anomalias no código fonte, tornando-o um código programado pelo conceito de tipos abstratos de dados. Isso é feito em continuidade ao trabalho de Penteado [Pen 96], no qual as anomalias são detectadas durante a engenharia reversa pelo Fusion/RE e os métodos necessários para resolver essas anomalias são indicados de forma textual, não chegando a ser implementados no código. O código segmentado Clipper é também utilizado como entrada para a transformação automática pela máquina Draco-Puc, que nesse experimento o transforma para a linguagem Java. Se a transformação fosse feita sem a segmentação prévia, o código resultante não traria as vantagens da orientação a objetos.

O fornecimento de esqueletos das classes para servirem como base para implementação, em Delphi, do padrão *Type-Object*, facilita o uso desse padrão. Desenvolvedores podem ter um ponto de partida que os estimule ao uso de padrões de software e ao desenvolvimento de uma biblioteca com os padrões usados com mais frequência. A linguagem Delphi, comumente utilizada em sistemas de informação comerciais, provou ser adequada para a implementação de padrões, embora a literatura consultada não mencione esse fato.

6.4 – Sugestões para futuras pesquisas

O processo de segmentação foi executado de forma manual, auxiliado somente por um editor de textos. Inicialmente esse processo consumiu bastante tempo mas, depois de adquirir alguma experiência, houve um aumento na eficiência. Isso é explicado pelo caráter repetitivo das tarefas envolvidas na segmentação. Assim, uma possível pesquisa poderia ser a construção de uma ferramenta de auxílio computadorizado à segmentação. Tarefas tais como a identificação de anomalias, isolamento de trechos em métodos (*slices*), etc. poderiam ser facilitadas por tal ferramenta, eliminando muito do trabalho tedioso e repetitivo. Acredita-se que a intervenção do engenheiro de software seja fundamental durante o processo de segmentação, não sendo possível, portanto, a segmentação totalmente automática.

O reconhecimento de padrões a partir dos modelos de análise do sistema também foi feito de forma manual, embora existam diversas indicações de que alguns padrões pudessem ser descobertos com o auxílio de uma ferramenta especificamente projetada para tal. O conhecimento do domínio de aplicação é essencial no reconhecimento de padrões de análise, mas a construção sintática também pode ajudar, descartando classes sem chance de se encaixar no padrão. A ferramenta poderia auxiliar na análise da construção sintática.

Após o reconhecimento dos padrões, sua utilização em futuros desenvolvimentos poderia ser mais encorajada se eles ficassem disponíveis em um repositório. Esse repositório de padrões poderia englobar classes prontas, pré-programadas em diversas linguagens, que poderiam ser usadas facilmente instanciando-se o padrão para a aplicação específica, de forma similar aos esqueletos apresentados na seção 5.4.2.1.

A implementação dos esqueletos de classes para o padrão *Type-Object* poderia evoluir para um *framework* que englobasse diversos padrões recorrentes, facilitando ainda mais seu reuso em novas aplicações.

Bibliografia

- [Arm 98] Armstrong, M. N.; Trudeau, C. *Evaluating Architectural Extractors*. In: IEEE Working Conference on Reverse Engineering, 5., Honolulu, Hawaii, EUA, outubro de 1998. *Anais*. LosAlamitos-CA, EUA, IEEE Computer Society, p. 30-39.
- [Ben 92] Benedusi, P.; Cimitile, A; Carlini, U. *Reverse Engineering Processes, Design Document Production and Structure Charts*. J. Systems Software, V.19, p. 225-245, 1992.
- [Ben 96] Benedusi, P. *Improving reverse engineering models with test-case related knowledge*. Information and Software Technology, V.38, p.711-718, 1996.
- [Big 89] Biggerstaff, Ted. J. *Design Recovery for Maintenance and Reuse*. Computer-IEEE, p. 36-49, julho 1989.
- [Big 94] Biggerstaff, Ted J.; Mitbender, Bharat G.; Webster, Dallas E. *Program Understanding and the Concept Assignment Problem*. Communications of the ACM, V.37, nº5, maio 1994.
- [Bos 97] Bosch, Jan. *Design Patterns as Language Constructs*. Disponível a partir da Wide World Web <URL:<http://st-www.cs.uiuc.edu/users/patterns/papers/>>, 1997.
- [Boy 98] Boyd, L. Business Patterns of Association Objects. In: “Martin, R.C.; Riehle, D.; Buschmann, F. *Pattern Languages of Program Design 3*. Reading, MA, Addison-Wesley, 1998”, p. 395-408.
- [Bra 86a] Braga, Rosana T.V. *Construção pelo método de Jackson de um protótipo em Dbase II do sistema de processamento de Concurso de Jornal especificado por Jackson*. São Carlos, USP, 1986. (Relatório apresentado à FAPESP, dentro do plano de bolsa de iniciação científica referente ao processo 85/0405-9)
- [Bra 86b] Braga, Rosana T.V. *Aplicação de um instrumento de apoio por computador ao método Gane*. São Carlos, USP, 1986. (Relatório apresentado à FAPESP, dentro do plano de bolsa de iniciação científica referente ao processo 85/0405-9)
- [Bra 90] Braga, Rosana T.V. *Manual de Uso do Sistema de Oficina Auto-elétrica e Mecânica*. São Carlos, 1990. (Documento de Trabalho)
- [Bra 98a] Braga, Rosana T.V.; Masiero, Paulo C. *Detalhamento do passo de abstração do modelo de análise do método Fusion/RE*. São Carlos, USP, março de 1998. (Relatório Técnico do ICMSC-USP, 70)
- [Bra 98b] Braga, Rosana T.V.; Germano, Fernão S.R.; Masiero, Paulo C. *Experimentos para implementação do padrão Type-Object em linguagem Delphi*. São Carlos, USP, 1998. (Relatório Técnico do ICMSC-USP, em elaboração)

- [Bud 96] Budinsky, F.J, et al. *Automatic code generation from design patterns*. IBM Systems Journal, V. 35, nº 2, p. 151-171, 1996.
- [Bus 97] Buschmann, Frank, et al. *Pattern-Oriented Software Architecture*. European Conference on Object-Oriented Programming, 11., Finland, Junho 1997. *Anais*. Finland.
- [Cam 97] Campo, Marcelo; Price, R.T. *Automated Recognition of Design Patterns for Framework Understanding*. (Comunicação particular), 1997.
- [Can 97] Cantu, Marco. *Dominando o Delphi 2*. São Paulo, Makron Books, 1997.
- [Can 98] Cantu, Marco; Gooch, Tim; Lam, John F. *Delphi Developer's Handbook*. Alameda-CA, Sibex, 1998.
- [Chi 90] Chikofsky, Elliot J.; Cross, James H. *Reverse Engineering and Design Recovery: a Taxonomy*. IEEE Software, p. 13-17, janeiro 1990.
- [Coa 92] Coad, Peter. *Object-Oriented Patterns*. Communications of the ACM, V. 35, nº9, p. 152-159, setembro 1992.
- [Col 94] Coleman D. et al. *Object Oriented Development - The Fusion Method*. Prentice Hall, 1994.
- [Cop 92] Coplien, J.O. *Advanced C++ Programming Styles and Idioms*. Reading-MA, Addison-Wesley, 1992.
- [Eri 98] Eriksson, H.; Penker, M. *UML Toolkit*. New York, Wiley Computer Publishing, 1998.
- [Fer 95] Ferreira, Aurélio B.H. *Dicionário Aurélio Básico da Língua Portuguesa*. São Paulo, Editora Nova Fronteira, 1995.
- [Fin 97] Finnigan, P. J. et al. *The Software bookshelf*. IBM Systems Journal, V. 36, nº 4, p. 564-593, 1997.
- [Fow 97] Fowler, M. *Analysis Patterns*. Menlo-Park-CA, Addison-Wesley, 1997.
- [Fow 98] Fowler, M.; Scott, K. *UML Distilled – Applying the standard object modeling language*. Reading, Massachusetts, Addison-Wesley, 1998.
- [Gal 95] Gall, Harald C., Klösh, René R.; Mittermeir, Roland T. *Architectural Transformation of Legacy Systems*. International Conference on Software Engineering, 11., Abril 1995. (Technical Report nº CS95-418)
- [Gal 96] Gall, Harald C., Klösh, René R.; Mittermeir, Roland T. *Application Patterns in Re-Engineering: Identifying and Using Reusable Concepts*. International Conference on Information Processing and Management of Uncertainty in Knowledge-Based Systems, 6., Julho 1996. *Anais*. p. 1099-1106.

- [Gam 93] Gamma, E.; Helm, R.; Johnson, R.; Vlissides, J. *Design Patterns - Abstraction and Reuse of Object-Oriented Design*. LNCS, nº 707, p. 406-431, julho de 1993.
- [Gam 95] Gamma, E.; Helm, R.; Johnson, R.; Vlissides, J. *Design Patterns - Elements of Reusable Object-Oriented Software*. Reading-MA, Addison-Wesley, 1995.
- [Gan 83] Gane, Chris, Sarson, Trish. *Análise Estruturada de Sistemas*, Rio de Janeiro, Livros Técnicos e Científicos Editora S. A., 1983.
- [Hai 96] Hainaut, J-L et al. *Structure Elicitation in Database Reverse Engineering*. In: Working Conference on Reverse Engineering (WCRE), 3., Monterey-California, 1996. *Anais*. IEEE, 1996, p. 131-140.
- [Hai 96a] Hainaut, J-L. *Ajuda on-line do DB-MAIN - Demo Version 2.01*. 1996. (Manual técnico)
- [Jac 83] Jackson, M. *System Development*. Englewood Cliffs-NJ, Prentice-Hall International Inc., 1983.
- [Jac 91] Jacobson, Ivar e Lindstrom, Fredrik. *Re-engineering of old systems to an object-oriented architecture*. In: Conference on Object Oriented Programming Systems, Languages and Applications (OOPSLA). *Anais*. 1991, p. 340-350.
- [Joh 98] Johnson, R.; Woolf, B. *Type Object*. In: “Martin, R.C.; Riehle, D.; Buschmann, F. *Pattern Languages of Program Design 3*. Reading-MA, Addison-Wesley, 1998”, p. 47-65.
- [Klo 96] Klösh, René R. *Reverse Engineering: Why and How to Reverse Engineer Software*. In: California Software Symposium (CSS), California, EUA, abril de 1996. *Anais*. University of Southern California, 1996, p. 92-99.
- [Kra 96] Krämer, Christian; Prechelt, Lutz. *Design Recovery by Automated Search for Structural Design Patterns in Object-Oriented Software*. In: Working Conference on Reverse Engineering (WCRE), 3., Monterey-CA, EUA, 1996. *Anais*. IEEE, 1996, p. 208-215.
- [Mar 94] Markosian, Lawrence, et al. *Using an Enabling Technology to Reengineer Legacy Systems*. Communications of the ACM, V.37, nº5, p. 58-70, maio 1994.
- [Mar 98] Martin, R.C.; Riehle, D.; Buschmann, F. (eds.) *Pattern Languages of Program Design 3*, Reading-MA, Addison-Wesley, 1998.
- [May 95] Mayrhauser, Anneliese von; Vans, A. M. *Program Comprehension During Software Maintenance and Evolution*. Computer-IEEE, p.44-55, agosto 1995.
- [May 96] Mayrhauser, Anneliese von; Vans, A. M. *Identification of Dynamic Comprehension Processes During Large Scale Maintenance*. IEEE Transactions on Software Engineering, V.22, nº 6, junho 1996.

- [Mas 88] Masiero, Paulo C.; Germano, Fernao S.R. *JSD as an Object Oriented Design Method*. Software Engineering Notes, V.13, nº 3, julho 1988.
- [Nei 84] Neighbors, J.M. *The Draco Approach to Constructing Software from Reusable Components*. IEEE Transactions on Software Engineering, V.SE-10, nº 5, p. 564-77, 1984.
- [Nin 94] Ning, Jim Q.; Engberts, A.; Kozaczynsku, W. *Automated Support for Legacy Code Understanding*. Communications of the ACM, V.37, nº 5, p. 50-57, maio 1994.
- [Oli 91] Oliveira, Jacqueline A.; Braga, Rosana T.V. *Documentação Técnica do Sistema de Oficina Auto-elétrica e Mecânica*. São Carlos, 1991 (Documento de Trabalho)
- [Pen 95] Penteado, R.D., Germano, F.; Masiero, P.C. *Engenharia Reversa Orientada a Objetos do Ambiente StatSim: Método Utilizado e Resultados Obtidos*, In: Simpósio Brasileiro de Engenharia de Software, 9., Recife-PE, 1995. *Anais*. Recife, UFPE, 1995. p. 345-351.
- [Pen 96] Penteado, R.D. *Um método para Engenharia Reversa Orientada a Objetos*. São Carlos, 1996. 251p. Tese (Doutorado) - Instituto de Física de São Carlos, Universidade de São Paulo.
- [Pen 96a] Penteado, R.D., Germano, F.; Masiero, P.C. *An Overall Process Based on Fusion to Reverse Engineer Legacy Code*. In: Working Conference on Reverse Engineering, 3., Monterey-CA, 1996. *Anais*. IEEE, 1996, p. 179-188.
- [Pen 96b] Penteado, R.D., Germano, F.; Masiero, P.C. *Melhoria na Manutenibilidade de Software Legado com Base em Engenharia Reversa*. In: Conferência Internacional de Tecnologia de Software - Qualidade de Software, 7., Curitiba-PR, 1996. *Anais*. Curitiba, PUC-PR, p. 155-169.
- [Pen 98a] Penteado, Rosangela D.; Braga, Rosana T.V.; Masiero, Paulo C. *Improving the Quality of Legacy Code by Reverse Engineering*. In: International Conference of Information Systems Analysis and Synthesis, 4., Orlando-FL, EUA, julho de 1998. *Anais*. Orlando-FL, EUA, International Institute of Informatics, p. 364-370.
- [Pen 98b] Penteado, Rosangela D.; Masiero, Paulo C.; Prado, Antonio F.; Braga, Rosana T.V. *Reengineering of Legacy Systems Base on Transformation Using the Object Oriented Paradigm*. In: IEEE Working Conference on Reverse Engineering, 5., Honolulu-HI, EUA, outubro de 1998. *Anais*. LosAlamitos-CA, EUA, IEEE Computer Society, p. 144-153.
- [Pra 92] Prado, A.F. *Estratégia de Reengenharia de Software Orientada a Domínios*. Rio de Janeiro, 1992. 333p. Tese (Doutorado) – Pontifícia Universidade Católica do Rio de Janeiro.
- [Pre 95] Pressman, Roger S. *Engenharia de Software*. São Paulo, Makron Books, 1995.
- [Rek 85] Rekoff, M. G. *On Reverse Engineering*. IEEE Transactions on Systems, Man and Cybernetics, V.SMC-15, nº 2, p. 244-252, Março-Abril 1985.

- [Ric 90] Rich, Charles; Wills, Linda M. *Recognizing a Program's Design: A Graph-Parsing Approach*. IEEE Software, p. 82-89, Janeiro 1990.
- [Rob 91] Robson, D.J.; Bennet, K.H.; Cornelius, B.J.; Munro, M. *Approaches to Program Comprehension*. J.Systems Software, V.14, p. 79-84, fevereiro 1991.
- [Sag 95] Sage, Andrew P. *Systems Engineering and Systems Management for Reengineering*, Journal Systems Software, V.30, p. 3-25, 1995.
- [Sem 95] Semmel, R. D. e Winkler, R. P. *Integrating Reengineered Databases to Support Data Fusion*. Journal Systems Software, V.30, p. 127-135, 1995.
- [Shn79] Shneiderman, Ben; Mayer, Richard *Syntatic/Semantic Interactions in Programmer Behavior: A Model and Experimental Results*. International Journal of Computer and Information Sciences, V.8, nº 3, p. 219-238, 1979.
- [Sne 95] Sneed, Harry M.; Nváry, Erika. *Extracting Object-Oriented Specification from Procedurally Oriented Programs*, In: Working Conference on Reverse Engineering (WCRE), 2., Toronto, Canada, 1995. *Anais*. IEEE, 1995, p. 217-226.
- [Sne 95a] Sneed, Harry M. *Planning the Reengineering of Legacy Systems*. IEEE Software, p. 24-34, janeiro 1990.
- [Sne 96] Sneed, Harry M. *Object-Oriented COBOL Recycling*. In: Working Conference on Reverse Engineering (WCRE), 3., Monterey-CA, EUA, 1996. *Anais*. IEEE, 1996, p. 169-178.
- [Sol 84] Soloway, Elliot; Ehrlich, Kate. *Empirical Studies of Programming Knowledge*. IEEE Transactions on Software Engineering, V.SE-10, nº 5, p. 595-609, setembro 1984.
- [Vli 95] Vlissides, John. *Reverse Architecture*, Position Paper for Software Architectures Seminar, Schloss Dagstuhl, Germany, disponível a partir da Wide World Web <URL:<http://st-www.cs.uiuc.edu/users/patterns/papers/>>, agosto 1995.
- [Wel 95] Welch, Lonnie R. et al. *Reengineering Computer-Based Systems for Enhanced Concurrency and Layering*. Journal Systems Software, V.30, p. 45-70, 1995.
- [Wil 95] 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.
- [Won 95] Wong, Kenny; et al. *Structural Redocumentation: A case Study*. IEEE Software, p. 46-54, janeiro 1995.
- [Yod 98] Yoder, J.W.; Johnson, R.E.; Wilson, Q.D. *Connecting Business Objects to Relational Databases*. Conference on the Pattern Languages of Programs, 5., Monticello-IL, EUA, 1998. *Anais*. St.Louis, Missouri, EUA, agosto de 1998. (Relatório Técnico nº WUCS-98-25 da Universidade de Washington).