UNIVERSIDADE FEDERAL DE SÃO CARLOS
DEPARTAMENTO DE CIÊNCIAS EXATAS
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIAS DA COMPUTAÇÃO

UM ESTUDO PARA A ESCOLHA DO SGBD PARA SISTEMAS SUBMETIDOS À REENGENHARIA ORIENTADA A OBJETOS

Rinaldo Macedo de Morais

São Carlos-SP
Agosto/2003

Ficha catalográfica elaborada pelo DePT da Biblioteca Comunitária da UFSCar

M827ee

Morais, Rinaldo Macedo de.
Um estudo para escolha do SGBD em processos de reengenharia orientada a objetos / Rinaldo Macedo de Morais. -- São Carlos : UFSCar, 2003.
87 p.

Dissertação (Mestrado) -- Universidade Federal de São Carlos, 2003.

1. Engenharia de software. 2. Reengenharia orientada a objetos. 3. Banco de dados orientado a objetos. 4. Banco de dados objeto-relacional. I. Título.

CDD: 005.1 (20ª)

Dedico especialmente este trabalho ao meu pai.

AGRADECIMENTOS

À professora Rosângela pela dedicação na orientação, pela paciência e pela oportunidade de desenvolver este trabalho.

À minha esposa Regiane pelo incentivo durante este período.

Aos meus filhos e familiares, com os quais espero compensar minha ausência nestes anos.

Ao amigo Frank, pelo companheirismo durante toda a jornada.

Aos docentes do Departamento de Computação, em especial aos professores Mauro, Sandra Fabbri e Marilde pelas sugestões e contribuições a este trabalho.

Ao professor Fernão e colegas Valter, Josiel e Maria Istela pelas sugestões e comentários fundamentais.

Aos colegas de mestrado Calebe, Sergio Borges, Débora, Wanderlei, Edson Recchia e Maria Ângela, Gizelle, Gustavo, Regiane, Val, Dinho, Karina, Fernando Genta, Cláudio, Fernando Balbino, Edson Guimarães, Marcos Vieira, Lucrédio e RAR, pelo companheirismo.

Às colegas Marília e Talita, da Unifran, pela revisão da monografia.

Ao Renato Barossi do Centro Universitário Moura Lacerda.

Às secretárias Cristina e Mírian, do Departamento de Computação.

A todos que, de forma direta ou indireta, contribuíram para a realização deste trabalho.

RESUMO

Um processo para a escolha do sistema gerenciador de banco de dados (SGBD) na etapa de engenharia avante, em um processo de reengenharia orientada a objetos, é apresentado. O processo foi instanciado para dois sistemas gerenciadores de bancos de dados particulares – *Jasmine* e *Caché*, com os quais um estudo de caso foi desenvolvido. O sistema legado tomado como exemplo foi submetido ao processo de reengenharia sendo utilizado o sistema gerenciador de banco de dados relacional *Sybase*. Esse mesmo sistema foi utilizado seguindo o processo descrito neste trabalho tendo os SGBDs *Jasmine* e *Caché* para persistência de dados. Dessa forma, três versões puderam ser obtidas para um mesmo sistema. Uma análise comparativa das três versões também consta deste trabalho.

ABSTRACT

A process for the database management system (DBMS) choice in the forward engineering stage, in an object-oriented reengineering is presented. The process was instanced for two particular DBMSs – Jasmine and Caché, which a forward engineering case study was developed. The legacy system taken as example was submitted to the reengineering process being used the Sybase relational database management system. This same system was used following the process described in this work and having the Jasmine and Caché DBMSs to data persistence. Of this way, three versions could be obtained for a same system. A comparative analysis of three versions also consists of this work.

SUMÁRIO

1 INTRODUÇÃO 1
1.1 CONSIDERAÇÕES INICIAIS 1
1.2 OBJETIVO 2
1.3 RELEVÂNCIA 2
2 REVISÃO BIBLIOGRÁFICA 4
2.1 CONSIDERAÇÕES INICIAIS 4
2.2 ENGENHARIA REVERSA E REENGENHARIA 4
2.3 O MÉTODO *FUSION/RE* 5
2.4 ESTRATÉGIAS INCREMENTAIS DE REENGENHARIA 8
2.5 PADRÕES DE PROJETO 10
2.6 O PADRÃO DE PROJETO *PERSISTENCE LAYER* 14
2.7 A FAMÍLIA DE PADRÕES DE REENGENHARIA FaPRE/OO 16
2.8 CONSIDERAÇÕES FINAIS 17
3 SISTEMAS GERENCIADORES DE BANCOS DE DADOS 18
3.1 CONSIDERAÇÕES INICIAIS 18
3.2. BANCO DE DADOS E PROJETO DE BANCO DE DADOS 18
3.3 SISTEMAS GERENCIADORES DE BANCOS DE DADOS RELACIONAIS 19
3.4 SISTEMAS GERENCIADORES DE BANCOS DE DADOS ORIENTADOS A OBJETOS 21
3.5 SISTEMAS GERENCIADORES DE BANCOS DE DADOS OBJETO-RELACIONAIS 25
3.6 CONSIDERAÇÕES SOBRE A SELEÇÃO DO BANCO DE DADOS 29
3.7 CONSIDERAÇÕES FINAIS 31

4 ENGENHARIA AVANTE COM USO DE SGBDs COM SUPORTE A OBJETOS 33
4.1 CONSIDERAÇÕES INICIAIS 33
4.2 A ETAPA DE ENGENHARIA AVANTE 33
4.3 O PROCESSO DE ENGENHARIA AVANTE COM O USO DO SGBD *JASMINE* 39
4.4 O PROCESSO DE ENGENHARIA AVANTE COM O USO DO SGBD *CACHÉ* 44
4.5 CONSIDERAÇÕES FINAIS 46

5 ESTUDO DE CASO 48
5.1 CONSIDERAÇÕES INICIAIS 48
5.2 DESCRIÇÃO DO SISTEMA LEGADO 48
5.3 ENGENHARIA REVERSA DO SISTEMA LEGADO........ 49
5.4 ENGENHARIA AVANTE COM USO DE SGBD RELACIONAL 50
5.5 ENGENHARIA AVANTE UTILIZANDO O SGBDOO *JASMINE* 53
5.6 ENGENHARIA AVANTE COM USO DO SGBD *CACHÉ* 63
5.7 ANÁLISE COMPARATIVA ENTRE AS VERSÕES DO SISTEMA EXEMPLO.. 74
5.8 CONSIDERAÇÕES FINAIS 82
6 CONCLUSÃO 84
6.1 PRINCIPAIS RESULTADOS 84
6.2 TRABALHOS FUTUROS 86
REFERÊNCIAS BIBLIOGRÁFICAS

LISTA DE FIGURAS

Figura 2.1 Padrão *Factory Method* ... 11
Figura 2.2 Padrão *Abstract Factory* ... 12
Figura 2.3 Padrão *Proxy* ... 12
Figura 2.4 Padrão *Iterator* ... 13
Figura 2.5 Padrão *Memento* ... 13
Figura 2.6 Diagrama de Integração de Padrões ... 15
Figura 2.7 Grupos da família de padrões de reengenharia FaPRE/OO ... 17
Figura 3.1 Conceitos principais do Modelo Relacional ... 20
Figura 3.2 Mapeamento de classes para relações ... 21
Figura 3.3 Padrão *Proxy* para as classes de aplicação em *Jasmine* ... 22
Figura 3.4 *Framework* de suporte a persistência de *Jasmine* ... 24
Figura 3.5 Estrutura de suporte à persistência em *Caché* ... 27
Figura 3.6 *Binding Caché* com Java ... 28
Figura 3.7 Manipulação de objetos *Caché* em Java ... 29
Figura 3.8 Classificação dos gerenciadores pela complexidade ... 31
Figura 3.9 Principais alternativas de persistência ... 32
Figura 4.1 Etapa de engenharia avante em um processo de reengenharia orientada a objetos utilizando bancos de dados de objetos ... 34
Figura 4.2 Mapeamento MAS x Esquemas de Classes para *Jasmine* ... 41
Figura 4.3 Interação entre camadas – modelo de classes de projeto para *Jasmine* ... 42
Figura 4.4 Derivação do Modelo de Projeto a partir do MAS para relacionamentos de especialização ... 43
Figura 4.5 Interação entre camadas – modelo de classes de projeto para *Caché* ... 45
Figura 5.1 Mapeamento de operações entre o Sistema Legado e o MAS ... 50
Figura 5.2 Modelo de Classes de Projeto - 1ª Versão do Sistema ... 51
Figura 5.3 Três Alternativas de Reengenharia para o sistema exemplo ... 52
Figura 5.4 Interface do Cadastro de Materiais – Sistemas Legado (a) e Alvo (b) ... 53
Figura 5.5 Mapeamento lógico e físico de classes em *Jasmine* ... 54
Figura 5.6 Modelo de classes de projeto com uso do SGBD *Jasmine* (visão parcial) ... 55
Figura 5.7 Diagrama de seqüência para caso de uso

CadastrarPrevisaoCompra (2ª versão do sistema exemplo) 56

Figura 5.8 Padrão *Proxy* para a classe Compra ... 57

Figura 5.9 Padrão *Factory Method* no modelo de classes de projeto ... 57

Figura 5.10 Padrão *Iterator* para classe Compra ... 58

Figura 5.11 Ambiente *Jasmine Studio* ... 59

Figura 5.12 Ambiente *Jasmine Browser* ... 59

Figura 5.13 Formulário HTML CadastrarMaterial.html ... 60

Figura 5.14 Java *Servlet* com conexão *Jasmine* ... 61

Figura 5.15 Padrão *Proxy*: exemplo de implementação ... 62

Figura 5.16 Padrão *Iterator* : exemplo de implementação ... 62

Figura 5.17 Padrão *Memento*: exemplo de codificação ... 63

Figura 5.18 Mapeamento lógico e físico de classes para *Caché* ... 64

Figura 5.19 Modelo de classes de projeto com uso do SGBD *Caché* ... 66

Figura 5.20 Diagrama de seqüência para caso-de-uso CadastrarPrevisaoCompra .. 67

Figura 5.21 Padrão *Proxy* nas classes *Caché* ... 68

Figura 5.22 Padrão *Method Factory* e *AbstractFactory* para a estrutura de classes *Caché* ... 68

Figura 5.23 Ambiente *Caché Object Architect* ... 70

Figura 5.24 Tabela Material – visão relacional ... 70

Figura 5.25 Java *Servlet* com conexão *Caché* ... 71

Figura 5.26 Relacionamento em *Caché* utilizando uma classe *container* ... 72

Figura 5.27 Relacionamento em *Caché* utilizando *query* SQL ... 73

Figura 5.28 Recuperação de objetos utilizando *query* SQL em *Caché* ... 73

Figura 5.29 Persistência de coleções em *Caché* ... 74

Figura 5.30 Percorrimento de listas em *Caché* ... 74

Figura 5.31 Visão Arquitetural das três versões do sistema exemplo ... 75

Figura 5.32 Mapeamento conceitual-lógico no projeto de banco de dados para os paradigmas relacional e orientado a objetos ... 76

Figura 5.33 Reutilização nas três versões do sistema exemplo ... 78

Figura 5.34 Atributos e funções membros da classe Compra (1ª. versão) ... 80

Figura 5.35 Implementação utilizando alcançabilidade (versão 2 do sistema exemplo).. 81

Figura 5.36 Implementação na versão 1 do sistema exemplo ... 81

LISTA DE TABELAS

Tabela 2.1 Padrões associados ao padrão *Persistence Layer*..15

Tabela 3.1 Conjunto de classes geradas pela ferramenta *Jasmine Browser*..................23

Tabela 3.2 Operações de persistência e *binding* de *Jasmine* 24

Tabela 3.3 Operações de persistência para o *binding* de *Caché* com Java 28

Tabela 3.4 Critérios para seleção do tipo de SGBD 30

Tabela 4.1 Mapeamento Lógico-Físico em *Caché* 44

Tabela 5.1 Fases do processo de engenharia avante...................... 82

LISTA DE SIGLAS E ABREVIAÇÕES

ANSI – *American National Standard Institute*
ASP – *Active Server Pages*
BLOB – *Binary Large Object*
CAD – *Computer Aided Design*
CAM – *Computer Aided Manufacture*
CASE – *Computer Aided Software Engineering*
CIM – *Computer Integrated Manufacture*
CMM – *Capability Maturity Model*
CRUD – *Create, Read, Update, Delete*
FaPRE/OO – Família de Padrões de Reengenharia Orientada a Objetos
FD – *File Description*
HTML – *Hiper-Text Markup Language*
HTTP – *Hiper-Text Transfer Protocol*
JSP – *Java Server Pages*
KPA – *Key Process Areas*
MAS – Modelo de Análise do Sistema
MASA – Modelo de Análise do Sistema Atual
MVC – *Model-View Controller*
ODMG – *Object Data Management Group*
ODQL – *Object Data Query Language*
OID – *Object Identifier*
OQL – *Object Query Language*
OREF – *Object Reference*
SGBD – Sistema Gerenciador de Banco de Dados
SGBDOO - Sistema Gerenciador de Banco de Dados Orientado a Objetos
SGBDO-R - Sistema Gerenciador de Banco de Dados Objeto-Relacional
SGBDR - Sistema Gerenciador de Banco de Dados Relacional
SQL – *Structured Query Language*
UML – *Unified Model Language*
URL – *Unified Resource Locator*
WIMP – Window Icon Menu Pointer
WWW – *Wide World Web*

Capítulo 1

1. INTRODUÇÃO

1.1. CONSIDERAÇÕES INICIAIS

Novas tecnologias como *World Wide Web*, orientação a objetos e computação distribuída propiciaram o desenvolvimento de novas aplicações nas áreas de comércio eletrônico, sistemas de informação gerenciais e sistemas de apoio à decisão, promovendo um diferencial competitivo às organizações. Por outro lado, apesar da distância do estado-da-arte no que se refere à tecnologia, os sistemas legados desempenham papéis críticos nos contextos em que estão inseridos, constituem repositórios de conhecimento e os riscos envolvidos muitas vezes inviabilizam sua substituição.

A busca de redução de esforços em manutenção, a melhoria da qualidade de produtos de software e a conformidade com a tecnologia emergente têm levado a indústria de software a aplicar procedimentos de reengenharia em seus sistemas legados e é crescente o interesse da comunidade científica nesse processo.

No escopo de reengenharia de software, este trabalho situa-se em uma linha de pesquisa que investiga métodos, processos, padrões e ferramentas para apoiar engenheiros de software na condução de reengenharia orientada a objetos de sistemas legados procedimentais.

Em um processo de reengenharia orientada a objetos a partir de sistemas legados procedimentais, uma das atividades atribuídas ao projetista é tratar a questão da persistência[1] de objetos na aplicação. Os sistemas legados, geralmente, fazem uso de recursos em desuso ou inadequados para armazenar dados e a mudança ou adoção de um sistema gerenciador de bancos de dados (SGBD) se faz necessária para prover serviços de persistência com eficiência e confiabilidade. Dentre as alternativas existentes os sistemas gerenciadores de bancos de

[1] A persistência pode ser definida como a propriedade em que um objeto continua a existir após seu criador cessar sua execução.

dados relacionais (SGBDRs) são predominantemente utilizados em aplicações orientadas a objetos. Seu uso pode ser associado a padrões de projeto para amenizar as diferenças entre os paradigmas da aplicação e do banco de dados. Outra opção é o uso de sistemas gerenciadores orientados a objetos (SGBDOOs) ou objeto-relacionais (SGBDO-Rs), os quais oferecem suporte à orientação a objetos em seus modelos de dados. Muitos desenvolvedores ainda preferem os relacionais aos outros devido às facilidades que eles apresentam em relação aos demais.

1.2. OBJETIVO

O SGBD a ser utilizado em um sistema submetido à reengenharia é um fator que impacta nos esforços despendidos pelo engenheiro de software durante e após esse processo. Como ocorre no desenvolvimento de sistemas a escolha do SGBD influencia todo o seu ciclo de vida, ou seja, desde a concepção do sistema até o momento em que esse deixa de existir; isso não é diferente no processo de reengenharia. Assim, este trabalho tem por objetivo investigar e discutir o aspecto de persistência de dados em sistemas que serão implementados em uma linguagem de programação orientada a objetos, após um processo de engenharia reversa, com utilização de diferentes sistemas gerenciadores de bancos de dados. Serão comentadas as utilizações de SGBDs com suporte a objetos e objetos relacionais. Além disso serão analisadas as diferenças existentes entre esses e em relação ao uso de SGBDs relacionais. Um estudo de caso utilizando exemplo um sistema legado procedimental que passou pelo processo de reengenharia utlizando SGBD relacional será considerado. O uso de SGBDs OO e objeto relacional são implementados neste trabalho e serão comentadas as diferenças e similaridades entre os três SGBDs utilizados.

1.3. RELEVÂNCIA

A crescente utilização de linguagens de programação orientadas a objetos induz a investigação do uso de um banco de dados do mesmo paradigma, pela compatibilidade entre eles, assim como de SGBDs objeto-relacionais, os quais estendem as funcionalidades do modelo relacional acrescentando suporte às abstrações do modelo orientado a objetos. Dessa

forma, o sistema legado procedimental que é submetido ao processo de reengenharia orientada a objetos pode utilizar deferentes SGBDs para a persistência de seus dados. Através das análises realizadas com o estudo de caso, foram elaboradas algumas diretrizes para auxiliar os engenheiros de software quanto à escolha do SGBD mais adequado quando um processo de reengenharia orientada a objetos for utilizado.

Para atingir o objetivo proposto, esta dissertação está organizada da seguinte forma: o Capítulo 2 apresenta os conceitos de engenharia reversa, reengenharia e padrões de projeto, o Capítulo 3 discute os modelos de SGBDs e apresenta as principais características dos gerenciadores *Jasmine* e *Caché*, utilizados neste trabalho. O processo de engenharia avante proposto neste trabalho é descrito no Capítulo 4. No Capítulo 5 é apresentado o estudo de caso desenvolvido. No Capítulo 6 são apresentadas as conclusões desta dissertação.

Capítulo 2

2. REVISÃO BIBLIOGRÁFICA

2.1. CONSIDERAÇÕES INICIAIS

Este capítulo apresenta alguns temas relacionados ao projeto desenvolvido. A Seção 2.2 apresenta os conceitos gerais sobre engenharia reversa e reengenharia; a Seção 2.3 apresenta o método *Fusion/RE*. Na Seção 2.4 são apresentadas estratégias incrementais de reengenharia e as Seções 2.5, 2.6 e 2.7 tratam, respectivamente, sobre padrões de projeto, o padrão de projeto *Persistence Layer* e a família de padrões de reengenharia FaPRE/OO. Comentários finais são apresentados na Seção 2.8.

2.2. ENGENHARIA REVERSA E REENGENHARIA

No modelo de processo de software a fase de manutenção corresponde ao período em que o software é liberado para uso e quatro tipos básicos de atividades de manutenção ocorrem (PRESSMAN, 2001):

- Manutenção corretiva: constitui as atividades relativas ao ajuste do software às suas especificações, diagnóstico e correção de erros;
- Manutenção adaptativa: refere-se às modificações associadas a seu ambiente;
- Manutenção perfectiva: refere-se a possibilidade de incrementar novas capacidades ao software e,
- Manutenção preventiva: corresponde às atividades aplicadas ao software com o objetivo de melhorar sua confiabilidade e manutenibilidade, na qual as técnicas de engenharia reversa e reengenharia são utilizadas.

Chikofsky e Cross (1990) definem reengenharia de software como sendo o exame e alteração de um sistema para reconstituí-lo em uma nova forma e a sua subseqüente

implementação nessa nova forma. Inclui uma fase de engenharia reversa, na qual são identificados os componentes do sistema e seus interelacionamentos, através da criação de modelos que representem a aplicação, seguida de alguma forma de engenharia avante ou reestruturação.

Para Jacobson e Lindström (1991), reengenharia orientada a objetos consiste na somatória de engenharia reversa, Δ e engenharia avante. Δ representa possíveis modificações funcionais e de implementação que podem ocorrer no sistema que passa por esse processo. A reengenharia que não tem mudança de funcionalidade, cujo Δ=0, e a reengenharia com mudança parcial de funcionalidade são os dois tipos apresentados.

Os sistemas legados utilizados nos processos de reengenharia, salvo raras exceções, apresentam documentação pobre, obsoleta ou inexistente, seus mantenedores não participaram de seu desenvolvimento e os especialistas de domínio que estabeleceram suas especificações não estão mais presentes. Dessa forma, a etapa de engenharia reversa é caracterizada por atividades de exame e identificação de seus componentes, normalmente a partir de código-fonte, e especificação de modelos de mais alto nível. Nos processos de reengenharia em que o paradigma orientado a objetos substitui o procedimental, os artefatos de projeto não são relevantes na fase de engenharia reversa devido à mudança de paradigma. Assim, a engenharia reversa é conduzida de forma a se obter um modelo de análise que contemple os requisitos implementados pelo sistema legado. Na próxima seção será apresentado um método para engenharia reversa de sistemas procedimentais para orientados a objetos.

2.3. O MÉTODO *FUSION/RE*

Fusion/RE (PENTEADO, 1996) é um método para apoiar engenharia reversa orientada a objetos de sistemas legados procedimentais, sem mudança de funcionalidade. Inicialmente, os modelos de análise orientada a objetos baseavam-se nos modelos do método *Fusion* (COLEMAN, 1994). Atualmente, os modelos da linguagem UML (OMG, 2003) e a ferramenta *Rational Rose* (RATIONAL, 2003) são utilizados para modelar o sistema tanto para a engenharia reversa quanto para a engenharia avante.

O método *Fusion/RE* tem como início a revitalização da arquitetura, em que a documentação básica do sistema e o código-fonte são analisados com o objetivo de se estabelecer a lógica estrutural procedimental do sistema legado. Os artefatos dessa etapa são diagramas chama-chamado, descrições de módulos e procedimentos e interfaces de cada procedimento. Essa atividade pode ser realizada manualmente ou com o apoio de alguma ferramenta de extração.

O segundo passo corresponde à recuperação do Modelo de Análise do Sistema Atual (MASA). Nesta etapa é desenvolvido um pseudo-modelo orientado a objetos, tipicamente sem relacionamentos de agregação ou especialização, representando as estruturas de dados e procedimentos implementados no sistema legado como pseudo-classes do modelo.

Os procedimentos identificados na primeira etapa são base para determinação dos métodos no modelo definitivo (MAS) e são classificados no MASA como: "c" quando se tratar de um construtor (modifica uma estrutura de dados); e "o" quando for um observador (consulta uma estrutura de dados). Uma vez que se trata de um sistema procedimental, as anomalias são identificadas e cada procedimento pode ser classificado como (oc), (o+), (c+), (oc+), (o+c) ou (o+c+), com o sinal "+" representando o acesso a mais de uma estrutura de dados.

Na terceira etapa, elabora-se o Modelo de Análise do Sistema (MAS), que é uma abstração do MASA, eliminando-se as anomalias de seus procedimentos. As pseudo-classes do MASA que geram hierarquias de classes, bem como um conjunto de classes e relacionamentos, podem ser modeladas como uma única classe no MAS. Os nomes dos métodos e atributos são mais significativos, de acordo com a funcionalidade do sistema, do que os existentes no legado.

Na quarta etapa, o mapeamento do MAS para o MASA é desenvolvido e validado: para cada classe do MAS, seus atributos e métodos são associados aos componentes originais (elementos de dados e procedimentos). Esse mapeamento auxilia a manutenção do sistema atual e sua reimplementação, bem como indica como os elementos do MAS estão implementados, facilitando seu reuso.

A fase de engenharia avante corresponde às atividades de projeto, implementação e

testes a partir do modelo conceitual obtido na etapa de engenharia reversa. Adicionalmente, o processo de reengenharia deve ser acompanhado por atividades de garantia da qualidade.

Diversos trabalhos têm sido desenvolvidos envolvendo engenharia reversa e reengenharia orientada a objetos de sistemas legados procedimentais. Penteado (1996) mostra como o método *Fusion/RE* foi aplicado em um processo de engenharia reversa no ambiente StatSim, ferramenta para edição e simulação de *Statecharts*, com aproximadamente 30.000 linhas de código, escritos em linguagem C.

O método *Fusion/RE* foi empregado para engenharia reversa no processo de reengenharia em um sistema de informação para oficinas de reparos elétricos e mecânicos de automóveis desenvolvido inicialmente em *Clipper* que foi reconstituído em *Delphi* (BRAGA, 1998). Posteriormente, para esse mesmo sistema, foi conduzida a segmentação[2] do código *Clipper* (PENTEADO; BRAGA; MASIERO, 1998). O mesmo sistema foi utilizado com a ferramenta transformacional Draco-Puc para conversão de *Clipper* para Java (PENTEADO *et al*, 1998).

Dois processos de reengenharia foram conduzidos para o ambiente StatSim - o primeiro submetido à segmentação a partir da documentação produzida na engenharia reversa utilizando *Fusion/RE*, considerando o código em linguagem C, com armazenamento em arquivos texto e com características de orientação a objetos (PENTEADO; MASIERO; CAGNIN, 1999). O segundo, a reengenharia foi realizada somente para edição textual de *Statecharts*, utilizando Java como linguagem de programação e implementando o padrão *Persistence Layer* nos três modos propostos por Yoder, Johnson e Wilson (1998), para mapear objetos no banco de dados relacional *Sybase* (SYBASE, 2003). As versões do sistema legado original e as obtidas nesses processos serviram de base a experimentos para avaliação de manutenabilidade e legibilidade (CAGNIN, 1999).

A continuidade do processo de reengenharia iniciado por Cagnin, constou da implementação da interface gráfica para edição de *Statecharts* no ambiente StatSim. Assim,

[2] *Segmentação é uma técnica que pode ser aplicada em processos de reengenharia, que consiste em preservar a linguagem de programação procedimental utilizada no sistema legado, rearranjando o código com a adição de características orientadas a objetos, quando possível.*

Prieto (2001) além do ferramental já citado utilizou o padrão MVC (GAMMA *et al.*, 1995) para avaliar benefícios e restrições desse padrão juntamente com outros já implementados anteriormente.

Camargo (2001) conduziu a reengenharia para WEB de um sistema de informação para controle de estoque, implementado em Cobol Microfocus 85. Inicialmente, foi procedida engenharia reversa utilizando *Fusion/RE* instanciado para sistemas Cobol. Na etapa de engenharia avante, foi utilizada a linguagem de programação Java (DEITEL; DEITEL, 2001), Sybase (SYBASE, 2003) como sistema gerenciador de banco de dados e reutilizada a terceira implementação do padrão *Persistence Layer* (CAGNIN, 1999). A aplicação foi portada para *WEB* utilizando *servlets* (DEITEL; DEITEL, 2001) como tecnologia de *middleware,* para comunicar a interface, escrita em HTML, com a aplicação.

Com base no método *Fusion/RE* e nos padrões de Engenharia Reversa e Reengenharia Orientada a Objetos propostos por Demeyer, Ducasse e Nierstrasz (2000), Recchia (2002) elaborou a família de padrões FaPRE/OO, permitindo ao engenheiro de software a realização completa da reengenharia de sistemas procedimentais para sistemas orientados a objetos. Essa família de padrões será apresentada na Seção 2.7.

Melhorias no processo de reengenharia são sugeridas por Lemos (2002), tornando esse processo evolutivo e cuidando que tanto o processo quanto o produto tenham qualidade. Dessa forma, KPAs (*Key Process Areas*) do nível 2 do CMM-SW (SEI, 1995) foram introduzidas ao processo de reengenharia. Os sistemas alvo são os implementados no ambiente *Delphi*, que foram desenvolvidos de forma estruturada para orientada a objetos.

2.4. ESTRATÉGIAS INCREMENTAIS DE REENGENHARIA

Esta seção discorre sobre abordagens que podem ser aplicadas para migrar a aplicação e os dados a partir de sistemas legados. Em sistemas de grande porte, com tamanhos que podem atingir magnitudes de centenas de milhares de linhas de código, em que o processo de reengenharia levaria anos, pode ser adotada uma estratégia incremental, em que a aplicação é logicamente particionada e a reengenharia é planejada e conduzida de forma iterativa, até que

a última partição seja convertida.

As estratégias incrementais de reengenharia têm sido adotadas por oferecerem menores impactos e riscos, maior flexibilidade, retorno mais imediato de investimentos e escalonamento de custos. Por outro lado, devem ser criados mecanismos de comunicação entre as partições legadas e as que já foram submetidas ao processo de reengenharia. A estratégia *Big-Bang* consiste na substituição completa do sistema legado pela nova versão do sistema. Essa estratégia também é conhecida como reengenharia revolucionária, *cold turkey* ou *lump-sum reengineering* (OLSEM, 1998) e tem as seguintes restrições:

- O não cumprimento de algum requisito no novo sistema pode comprometer seu funcionamento;
- Devem ser previstos planos de contingências na hipótese de falhas;
- A validação pode ser complexa para sistemas legados que não possuem nenhum plano ou histórico de testes;
- Duplo gerenciamento de configuração: as manutenções no sistema legado, enquanto ocorre a reengenharia, devem ser controladas e
- Porte do sistema pode exigir processo de reengenharia de longa duração e em situações reais e dificilmente o sistema pode ser "congelado".

A estratégia *Chicken Little (*BRODIE; STONEBRAKER*,* 1993) propõe que as partições que sofrem reengenharia e as partições legadas coexistam em um ambiente de interoperabilidade, com replicação das bases de dados e utilização de módulos mediadores (*gateways*) que tratam a comunicação entre as aplicações legada e destino e as bases de dados, garantindo a integridade e a consistência enquanto as duas bases de dados coexistem. A redundância imposta pelo método garante a reversibilidade da migração, pois a base de dados legada preserva todas as informações controladas pelo sistema. Além disso, recursos modernos de sistemas gerenciadores de bancos de dados podem ser utilizados pelas novas aplicações no sistema destino. Por outro lado, o uso de *gateways* pode influir na complexidade, sendo que o desempenho do sistema pode ser crítico em aplicações distribuídas e heterogêneas. Outro fator a ser considerado é a pouca disponibilidade de *gateways* comerciais, sendo em sua maioria, produtos para apoiar a migração a partir de bases de dados armazenadas em *mainframes* (OLSEM, 1998).

Na metodologia *Butterfly* (BISBAL *et al.*, 1999) o processo é conduzido mantendo-se isolados os sistemas legado e destino. O sistema legado permanece operacional até ser extinto e seus dados são gradativamente transportados para a base de dados destino. No ambiente destino ocorrem atividades de reengenharia ou redesenvolvimento, testes e validação do novo sistema. A principal contribuição do método é o suporte oferecido a sistemas de missão crítica, pela não interrupção do sistema durante seu processo.

A Reengenharia Iterativa (BIANCHI; CAIVANO; VISAGGIO, 2000) propõe uma abordagem com enfoque em dados, com coexistência dos sistemas legado e destino e um processo evolutivo de migração sem redundâncias nas bases de dados. Os dados do sistema legado são classificados em conceituais, de controle, temporários e calculados e o método baseia-se no princípio de que apenas os dados conceituais devem ser migrados para a nova base de dados, eliminando-se quaisquer tipos de redundância. Estudos de casos realizados indicaram os seguintes resultados após a aplicação do método:

- Melhoria nos sintomas da idade, ou seja, nos efeitos causados pelo envelhecimento do modelo de dados em função de manutenções;
- Melhoria na poluição de dados, com eliminação de dados redundantes e não-essenciais, arquivos temporários e arquivos patológicos (manipulados em vários pontos no sistema legado);
- Pouca alteração no código-fonte; e
- Aplicabilidade de recursos dos SGBDs modernos.

2.5. PADRÕES DE PROJETO

Um padrão de projeto de software descreve uma solução para um problema de software que já foi largamente utilizada, produzindo resultados satisfatórios. A adoção de padrões traz vantagens para qualidade do processo, reduz seus riscos e pode gerar ganhos em produtividade, em função do reuso de suas técnicas e/ou implementação. Os padrões oferecem um vocabulário comum entre projetistas, reduzindo a complexidade pelo uso de abstrações (GAMMA *et al.*, 1995).

O conceito de padrão por vezes confunde-se com os termos *framework* e biblioteca de classes. Enquanto um padrão trata de um problema particular de software, bibliotecas de

classes (*toolkits*) agrupam classes relacionadas para fornecer a funcionalidade por reuso de código. Um *framework* é um conjunto de classes cooperantes que oferecem uma infra-estrutura reutilizável para construção de projetos para um domínio específico (GAMMA *et al.*, 1995).

Os padrões de software podem se referir a diferentes níveis de abstração, como padrões arquiteturais, padrões de projeto e padrões de implementação, entre outros. Os padrões de projeto descrevem soluções para problemas comuns no projeto de software. Um padrão de projeto abstrai e identifica as classes e instâncias participantes, seus papéis, colaborações e responsabilidades e estabelece uma estrutura de projeto comum, para torná-la útil para um projeto orientado a objetos. Gamma *et al.* (1995) enumeram um conjunto de padrões de projeto, dividindo-o em três grupos: padrões de criação - relacionados à instanciação de objetos, estruturais - relacionados à composição e comportamentais - relacionados à interação. Alguns dos padrões de projeto catalogados pelos autores são apresentados a seguir de maneira sintética:

Nome:	*Factory Method*
Tipo:	de criação
Finalidade:	Definir uma interface de métodos para criação de objetos.
Estrutura:	Figura 2.1

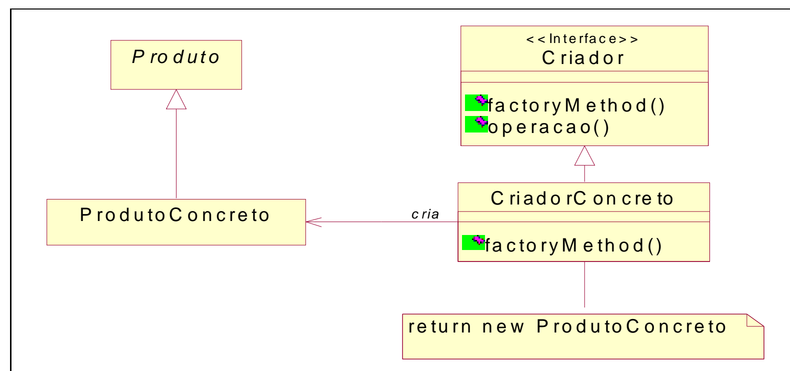


Figura 2.1. Padrão *Factory Method* (GAMMA *et al.*, 1995)

Nome:	*Abstract Factory*
Tipo:	de criação
Finalidade:	Fornecer uma interface para criação de famílias de objetos relacionados ou dependentes, sem especificar suas classes concretas.
Estrutura:	Figura 2.2

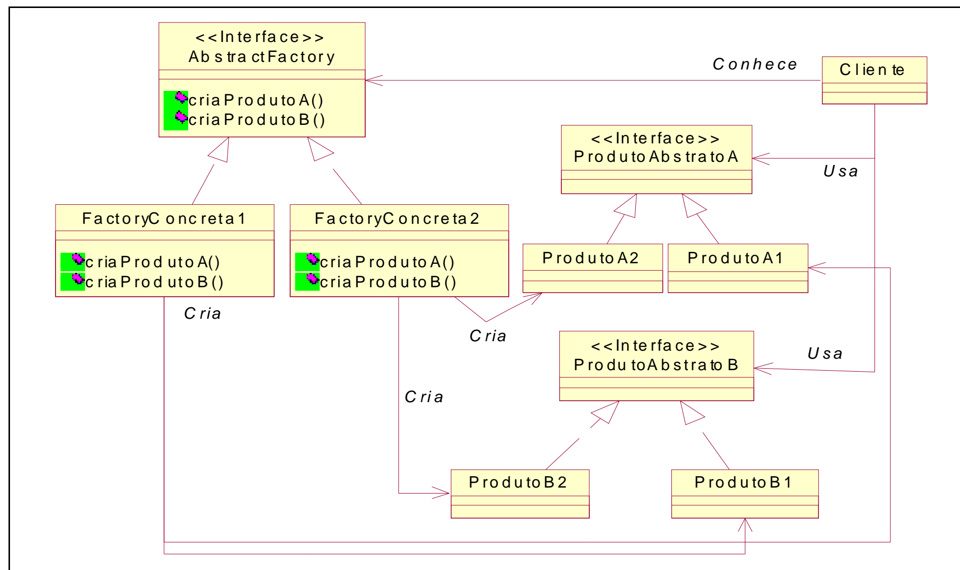


Figura 2.2. Padrão *Abstract Factory* (GAMMA *et al.*, 1995)

Nome:	*Proxy*
Tipo:	estrutural
Finalidade:	Fornecer um representante de outro objeto para controlar o acesso ao mesmo.
Estrutura:	Figura 2.3

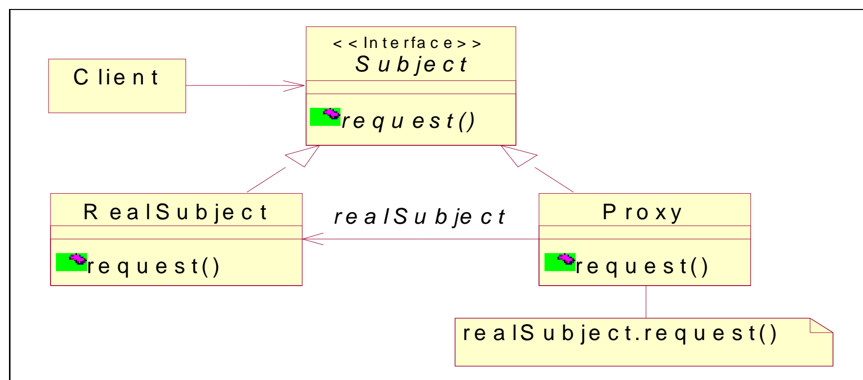


Figura 2.3 - Padrão *Proxy* (GAMMA *et al.*, 1995)

Nome:	*Iterator*
Tipo:	comportamental
Finalidade:	Fornecer acesso seqüencial aos elementos de um objeto composto sem expor sua representação subjacente.
Estrutura:	Figura 2.4

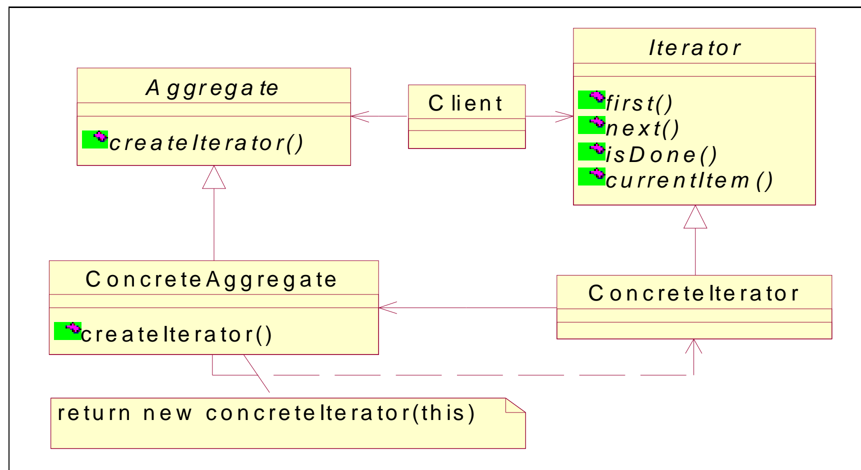


Figura 2.4 - Padrão *Iterator* (GAMMA *et al.*, 1995)

Nome:	*Memento*
Tipo:	comportamental
Finalidade:	Capturar o estado de um objeto de modo que possa ser restaurado para esse estado posteriormente.
Estrutura:	Figura 2.5

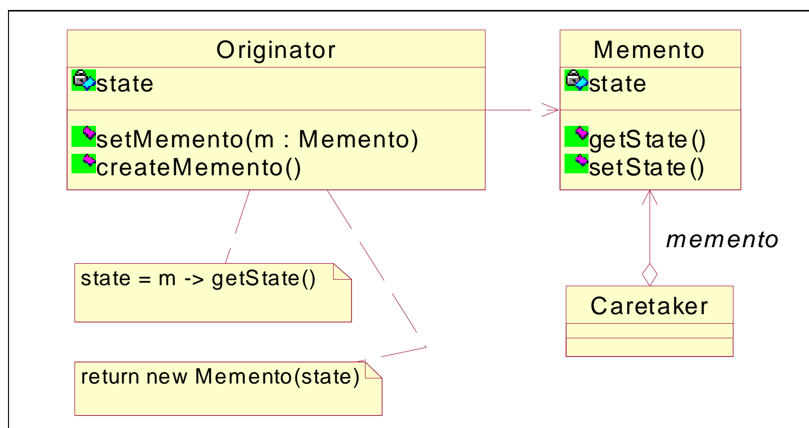


Figura 2.5. Padrão *Memento* (GAMMA *et al.*, 1995)

Tahvildari e Kontogiannis (2002) apresentam uma classificação de relacionamentos entre os padrões de projeto de Gamma *et al.* (1995). Para pares (X,Y) de padrões, os autores catalogaram três relacionamentos primários - os relacionamentos *X uses Y*, *X refines Y* e *X conflicts with Y* – e três relacionamentos secundários, deriváveis a partir de relacionamentos primários – *X is similar to Y*, *X combines with Y* e *X requires Y*. Essa classificação tem por objetivo auxiliar os engenheiros de software na compreensão de relacionamentos complexos entre os padrões de projeto, organizar e descrever novos padrões de projeto a partir de padrões existentes e ser uma referência para construção de ferramentas de suporte à aplicação que utilizam padrões de projeto na reengenharia e reestruturação de sistemas orientados a objetos.

2.6. O PADRÃO DE PROJETO *PERSISTENCE LAYER*

Em aplicações orientadas a objetos, quando se persiste dados utilizando sistemas gerenciadores de bancos de dados (SGBDs) relacionais, ocorre o que é conhecido como impedância entre modelos. Enquanto a relação é a estrutura básica de armazenamento em bancos de dados relacionais, definida como uma estrutura uniforme de dados escalares, objetos podem ser organizados como agregações ou associações com outros objetos, podem possuir heranças, serem compostos de coleções de outros objetos e por tipos não convencionais. Sendo assim, o projetista de software deve tratar o mapeamento entre a aplicação e o banco de dados relacional e implementar esse mapeamento em uma arquitetura orientada a objetos.

O padrão *Persistence Layer* (YODER; JOHNSON; WILSON, 1998) fornece diretrizes para que se utilize uma linguagem de programação orientada a objetos em conjunto com um banco de dados relacional. Para isso define-se uma camada de interface entre a aplicação e o banco de dados. A implementação do *Persistence Layer* faz uso de outros padrões, os quais são descritos na Tabela 2.1, enquanto a interação entre esses vários padrões é apresentada na Figura 2.6.

Yoder, Johnson e Wilson (1998) propõem três modos de implementação do padrão *Persistence Layer*:

1. As classes de aplicação são definidas como subclasses de uma classe abstrata *PersistentObject,* na qual são declarados os protótipos das operações CRUD (*Create*, *Read*, *Update* e *Delete*) definidas para as classes persistentes. As operações do padrão CRUD são implementadas nas classes de aplicação;
2. São criadas classes específicas para as classes persistentes da aplicação, havendo um relacionamento um-para-um entre elas. Essas classes específicas são herdeiras da classe *PersistentObject* e implementam as operações CRUD;
3. Uma classe *Broker* é definida para implementar os métodos do padrão CRUD para qualquer objeto de classes de aplicação. Essa classe deve criar automaticamente as cláusulas no padrão SQL *code description* para manipulação dos objetos no banco de dados. As classes persistentes da aplicação são herdeiras da classe abstrata *PersistentObject* e invocam os métodos da classe *Broker*.

TABELA 2.1. Padrões associados ao padrão *Persistence Layer* (YODER, JOHNSON, WILSON, 1998)

Nome do Padrão	Descrição
CRUD (*Create, Read, Update e Delete*)	Implementa as operações básicas para persistência de objetos em bancos de dados.
SQL *code description*	Linguagem de manipulação de dados para implementar as operações CRUD em um banco de dados relacional.
Attribute Mapping Methods e *Type Conversion*	Mapeiam os atributos dos objetos e do banco de dados, e realizam a translação de valores.
Change Manager	Controla a mudança de valores dos objetos para manter a consistência com o banco de dados.
OID *Manager*	Gera chaves únicas para identificação dos objetos no banco de dados.
Transaction Manager	Provê mecanismos de controle de transações na atualização de objetos.
Connection Manager	Estabelece e encerra conexões com o banco de dados.
Table Manager	Gerencia o mapeamento de um objeto para sua(s) correspondente(s) tabela(s) e coluna(s).

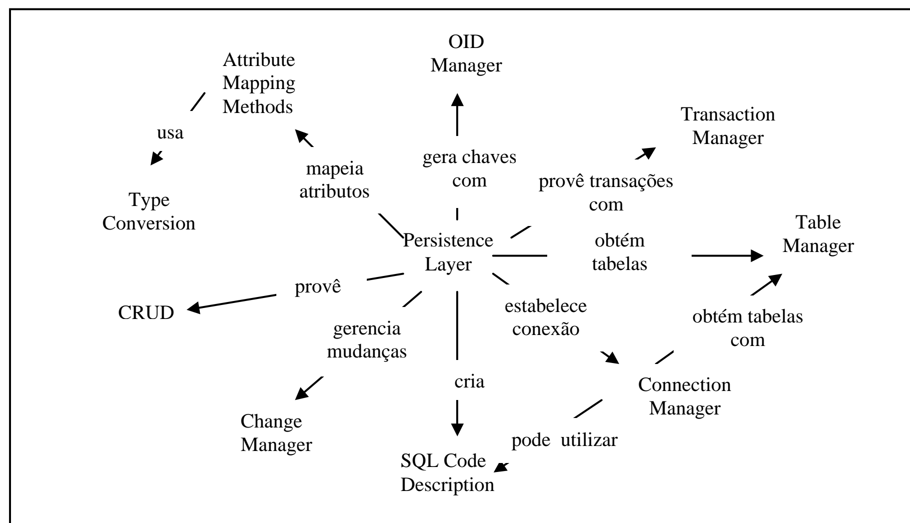


Figura 2.6. Diagrama de Integração de Padrões (YODER, JOHNSON, WILSON, 1998)

O terceiro modo exige maior esforço em desenvolvimento, mas apresenta uma implementação mais elegante: é genérico, seus métodos tratam qualquer objeto das classes de aplicação e é mais fácil de ser reutilizado. Cagnin (1999) realizou experimentos de manutenção em versões de implementação de cada um dos modos, sendo que, nesses experimentos, a implementação do terceiro modo apresentou melhores resultados em manutenibilidade. A classe *Broker* desenvolvida por Cagnin (1999), quando da implementação do terceiro modo proposto por Yoder, Johnson e Wilson (1998), foi reusada em diversos trabalhos: Camargo (2001), Prieto (2001) e Escobal (2000), entre outros.

2.7. A FAMÍLIA DE PADRÕES DE REENGENHARIA FaPRE/OO

Uma família de padrões para condução de reengenharia orientada a objetos a partir de sistemas legados procedimentais foi proposta por Recchia (2002), composta por quatro grupos de padrões, três de engenharia reversa e um de engenharia avante, conforme mostra a Figura 2.7.

O grupo 1 – Modelar os dados do legado – agrupa os padrões que extraem as informações a partir dos dados e do código do sistema legado para gerar o modelo de dados do sistema. .

O grupo 2 – Modelar as funcionalidades do sistema – reúne os padrões para extrair e representar as regras de negócios implementadas pelo sistema legado, para compreensão de suas funcionalidades.

O grupo 3 – Modelar o sistema orientado a objetos – agrupa os padrões para se desenvolver os modelos orientados a objetos a partir do sistema legado. O MAS (Modelo de Análise do Sistema) é o resultado da aplicação desse grupo de padrões e é a base para a engenharia avante.

O grupo 4 – Gerar o sistema orientado a objetos – refere-se aos padrões aplicáveis na engenharia avante do sistema, em que o sistema é reconstruído em uma nova arquitetura.

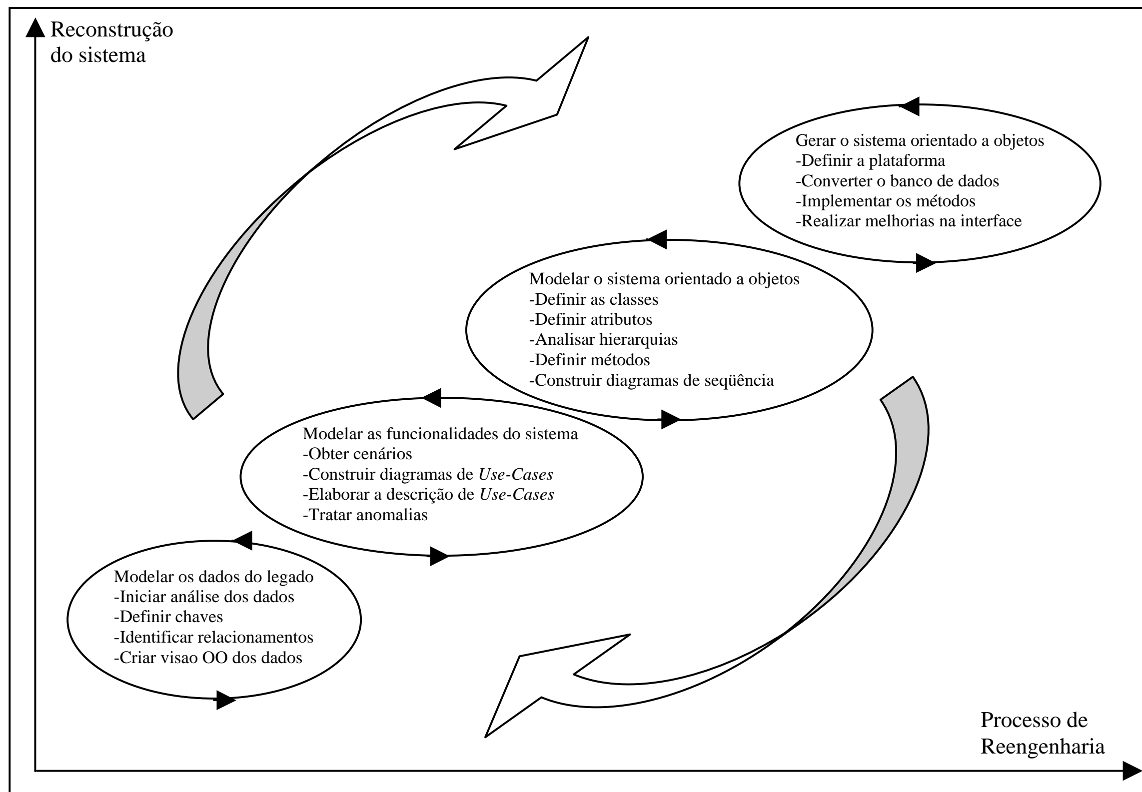


Figura 2.7. Grupos da família de padrões de reengenharia FaPRE/OO (RECCHIA, 2002)

2.8. CONSIDERAÇÕES FINAIS

Neste capítulo foram apresentados temas da literatura e trabalhos associados ao presente estudo. Por se tratar de um trabalho de engenharia reversa e reengenharia orientada a objetos foram apresentados tais conceitos e trabalhos realizados envolvendo-os. No que se refere à persistência, têm sido utilizados, sistematicamente, banco de dados relacionais em processos de reengenharia. Existe pouca literatura sobre persistência em bancos de dados de objetos em processos de reengenharia, sendo esse um dos motivos para a realização deste trabalho. Padrões de projeto foram apresentados por serem identificados durante a reengenharia com adequação para o tipo mais conveniente de SGBD. No próximo capítulo serão apresentados os conceitos e tópicos relacionados a bancos de dados.

Capítulo 3

SISTEMAS GERENCIADORES DE BANCOS DE DADOS

3.1. CONSIDERAÇÕES INICIAIS

Este capítulo apresenta os conceitos e tecnologias de bancos de dados que são utilizados nestre trabalho. São revisados os modelos relacional, orientado a objetos e objeto-relacional e apresentadas as principais características dos SGBDs *Jasmine* (COMPUTER ASSOCIATES, 2003) e *Caché* (INTERSYSTEMS, 2003). O enfoque dado é o de bancos de dados como um componente de um sistema de software e como esse componente integra-se com a aplicação, em nível de projeto e de implementação.

A Seção 3.2 define banco de dados e projeto de banco de dados. A Seção 3.3 refere-se a sistemas gerenciadores de banco de dados relacionais. A Seção 3.4 apresenta sistemas gerenciadores de banco de dados orientados a objetos com uma visão geral do SGBD *Jasmine*. O padrão ODMG (*Object Data Management Group*) também é comentado nessa seção. Os sistemas gerenciadores de bancos de dados objeto-relacionais são discutidos na Seção 3.5, assim como são apresentadas as principais características do sistema gerenciador *Caché*. A Seção 3.6 comenta a respeito de critérios para seleção do SGBD e a Seção 3.7 apresenta as considerações finais do capítulo.

3.2. BANCO DE DADOS E PROJETO DE BANCO DE DADOS

Define-se banco de dados como sendo um conjunto de dados relacionados, armazenados e organizados de forma a ser facilmente manipulado. Sua manipulação é realizada por um software denominado Sistema Gerenciador de Bancos de Dados (SGBD), que possui uma coleção de funções pré-implementadas, as quais gerenciam as operações de inserção, remoção, atualização e consulta dos dados nele armazenados (ELMASRI; NAVATHE, 2000).

Um modelo de dados define a forma em que os dados são organizados e as operações aplicáveis na manipulação desses dados no sistema gerenciador, sendo os modelos mais conhecidos: hierárquico, rede, relacional, objeto-relacional e orientado a objetos. . Dada a importância do banco de dados enquanto componente de um sistema de informação, o projeto do banco de dados é uma atividade essencial no desenvolvimento de software. O projeto de um banco de dados é composto de três fases:

a) Projeto conceitual: tem por objetivo identificar e modelar, em alto nível, as características de dados do domínio do problema e é independente do modelo de dados adotado;

b) Projeto lógico: especifica a estrutura lógica do banco de dados que pode ser processada por um SGBD do modelo de dados adotado;

c) Projeto físico: especifica como o banco de dados pode ser implementado em um particular SGBD, considerando seus recursos e suas restrições.

3.3. SISTEMAS GERENCIADORES DE BANCO DE DADOS RELACIONAIS

Os Sistemas Gerenciadores de Bancos de Dados Relacionais (SGBDRs) implementam o Modelo Relacional, introduzido por Codd[3] (*apud* ELMASRI; NAVATHE, 2000), o qual se baseia no conceito de uma estrutura uniforme de dados denominada relação, sobre a qual se aplica uma fundamentação teórica baseada na álgebra relacional.

A Figura 3.1 mostra como os conceitos do modelo relacional estão relacionados uns com os outros. Os retângulos correspondem aos conceitos chaves do modelo, as linhas rotuladas aos relacionamentos e as setas indicam a direção da leitura. Assim, a figura expressa que uma tabela é a representação concreta de uma relação e compõe-se de um conjunto de tuplas. Toda relação está em uma forma normalizada especificada pela dependência funcional entre seus atributos. Toda tupla é identificada na relação por uma combinação de atributos denominada chave primária e referências a tuplas de outras relações se dá através de chaves estrangeiras. A regra de integridade de entidade especifica que nenhuma chave primária pode ser nula e a de integridade referencial estabelece a consistência entre tuplas de duas relações.

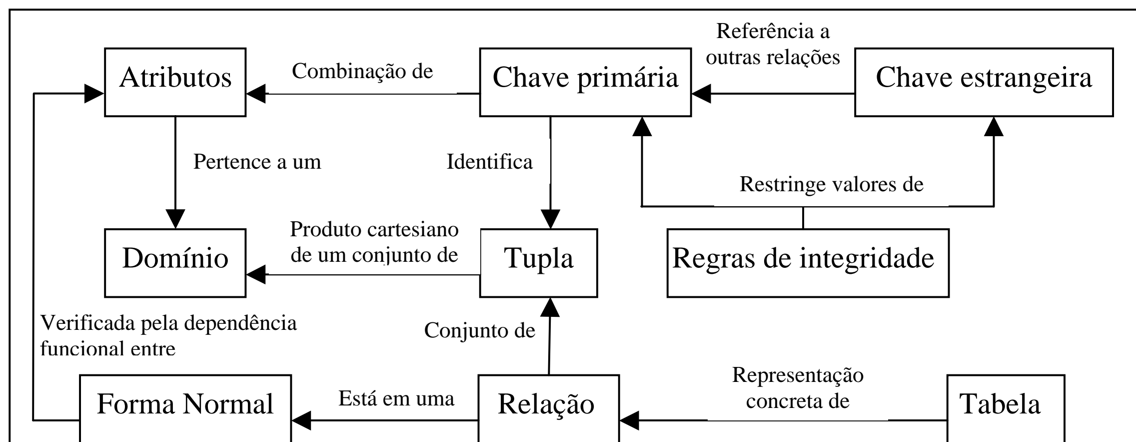


Figura 3.1. Conceitos principais do Modelo Relacional (AGERFALK, 1999)

Quando se utiliza um SGBD relacional para persistir dados em uma aplicação orientada a objetos, na fase de projeto o engenheiro de software deve especificar uma estrutura de tabelas relacionadas equivalente às classes persistentes constantes no modelo conceitual da aplicação, tratando as diferenças semânticas entre os paradigmas.

Elmasri e Navathe (2000) descrevem padrões para o projeto lógico de banco de dados relacionais, através de regras para se obter o esquema relacional a partir de um diagrama entidade-relacionamento estendido. O esquema obtido apresenta os dados organizados de forma normalizada e que pode ser implementado em um SGBD relacional.

Na Figura 3.2 são mostrados exemplos de regras para mapeamento de agregação e herança. O modelo conceitual é representado pela notação da UML por ser a notação utilizada neste trabalho. Um relacionamento de agregação pode ser mapeado em tabelas relacionadas pela definição de uma tabela para representar a classe "todo" e uma tabela para representar a classe "parte", esta última contendo como chave estrangeira uma referência para a primeira tabela. Relacionamentos de especialização podem ser mapeados com a criação de tabelas apenas para as sub-classes nos casos em que a superclasse é abstrata e com uma tabela para cada classe nos casos em que a superclasse é concreta. Nesses casos, as chaves primárias das tabelas são compostas pela mesma combinação de atributos.

[3] CODD. E. F. A relational model for large shared data banks. Communications of the ACM, v. 13, n. 6, p. 377-387, 1970.

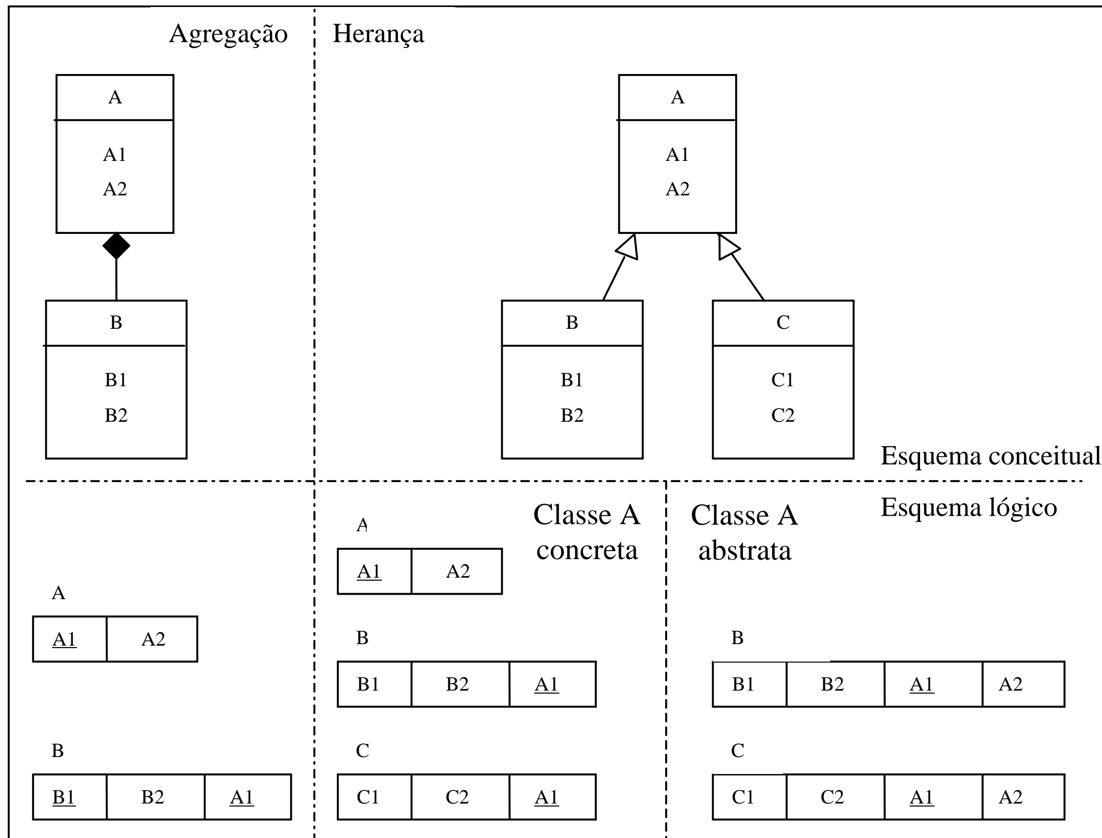


Figura 3.2. Mapeamento de classes para relações

3.4. SISTEMAS GERENCIADORES DE BANCOS DE DADOS ORIENTADOS A OBJETOS

Um Sistema Gerenciador de Banco de Dados Orientado a Objetos (SGBDOO) oferece recursos para definição de classes, métodos para manipulação de objetos, implementam os conceitos de orientação a objetos, como agregação, herança e polimorfismo e as instâncias de classes possuem identificadores únicos (OIDs) gerados pelo sistema gerenciador, que permitem recuperá-las no banco de dados. Enquanto em SGBDs relacionais as estruturas de dados são tabelas compostas de tipos escalares, os SGBDOOs permitem definir estruturas mais complexas de dados, pelo uso de atributos de referência e tipos construtores (*collection types – set, list, array e bag*).

Os gerenciadores de banco de dados orientados a objetos oferecem suporte para representação unificada de objetos, denominado *binding* entre a linguagem de programação e o banco de dados, sendo que suas formas mais comuns são pela extensão da linguagem de

programação, pela extensão do modelo de dados ou através de bibliotecas para suporte ao acesso de dados, sendo que a última é a mais utilizada pelos SGBDs orientados a objetos (LARSON, 1995). Ainda é incipiente a padronização para o *binding* entre a aplicação e o banco de dados e cada particular SGBD disponibiliza seu próprio *framework* de suporte à persistência.

Neste trabalho, o termo *framework* de persistência é utlizado para se referir a um conjunto de classes cooperantes, com um propósito definido, extensível e reutilizável na construção de diferentes projetos utilizando o gerenciador de banco de dados. A seguir, esta seção comenta a respeito do SGBD *Jasmine*, utilizado neste trabalho.

Jasmine (COMPUTER ASSOCIATES, 2003) é um SGBD orientado a objetos, que possui como características adicionais um conjunto de ferramentas para apoio ao desenvolvimento de aplicações para *WEB* e uma hierarquia de classes multimídia, para manipulação de objetos *BLOBs (Binary Large Objects*) e tipos de imagens, animação, áudio e vídeo (KOSHAFIAN; DASANANDA; MINASSIAN, 1999).

A definição de classes de objetos e métodos nesse sistema gerenciador pode ser feita pela ferramenta *Jasmine Studio* ou por *scripts* em ODQL (*Object Data Query Language*), linguagem proprietária do gerenciador para definição e manipulação de classes e objetos baseada na linguagem C/C^{++}.

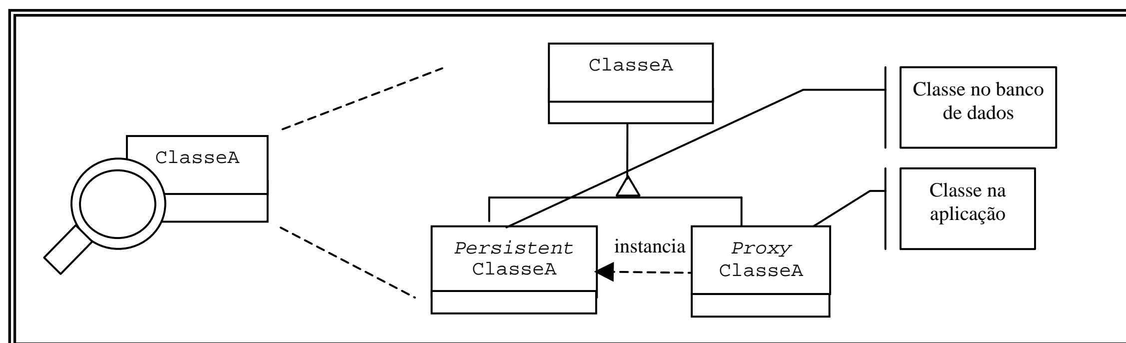


Figura 3.3. Padrão *Proxy* para as classes de aplicação em *Jasmine*

A relação entre um objeto da aplicação e do banco de dados em *Jasmine* é representada na Figura 3.3. O objeto da aplicação é o representante de um objeto correspondente do banco de dados e a persistência é intermediada por esse objeto

representante (*Proxy*). Essa estrutura é similar à definição do padrão *Remote Proxy* definido por Gamma *et al*. (1995).

Jasmine provê um *framework* de suporte à persistência para as classes definidas no banco de dados. Esse *framework* é composto por classes disponíveis em bibliotecas específicas para as linguagens de programação C++, Java e para o padrão COM e por classes que serão geradas a partir de definições do banco de dados. Para cada uma dessas linguagens, a ferramenta *Jasmine Browser* gera o código para acesso a objetos no banco de dados a partir da linguagem hospedeira, que em conjunto com bibliotecas disponibilizadas pelo SGBD oferecem todo o suporte funcional necessário para o acesso e manipulação dos dados da aplicação.

Para a linguagem de programação Java, por exemplo, a ferramenta *Jasmine Browser* gera para cada classe definida no banco de dados seis outras classes, cujas funcionalidades são descritas na Tabela 3.1.

TABELA 3.1. Conjunto de classes geradas pela ferramenta *Jasmine Browser*

Nome da classe gerada	**Funcionalidade**
`ClassName.java`	Definição da classe de aplicação, com construtores, métodos acessórios (*set()s*, *get()s* e *reset()s*) e pontos de chamada a métodos definidos na base de dados.
`ClassNameFactory.java`	Criação e recuperação de objetos da base de dados.
`ClassNameIterator.java`	Definição de um cursor para percorrer objetos agregados.
`ClassNameCollection.java`	Suporte a coleções de objetos da classe.
`ClassNameBeanInfo.java` `ClassNameCustomize.java`	Suporte a *Java Beans.*

A Figura 3.4 mostra o *framework* de classes correspondente ao *binding* de *Jasmine* para a linguagem de programação Java. As classes iniciadas por `ClassName` são as geradas pela ferramenta *Jasmine Browser* e as demais são de suporte à persistência do gerenciador. As classes `Jsession` e `JsessionManager` possuem métodos para serviços de conexão com o banco de dados, validação de usuários e gerenciamento de transações. A classe `JVariant` trata as incompatibilidades de tipos entre o banco de dados e a linguagem de programação e a classe `JException` dá suporte ao tratamento de exceções. A referência completa dessas classes está disponível em *Computer Associates* (2000).

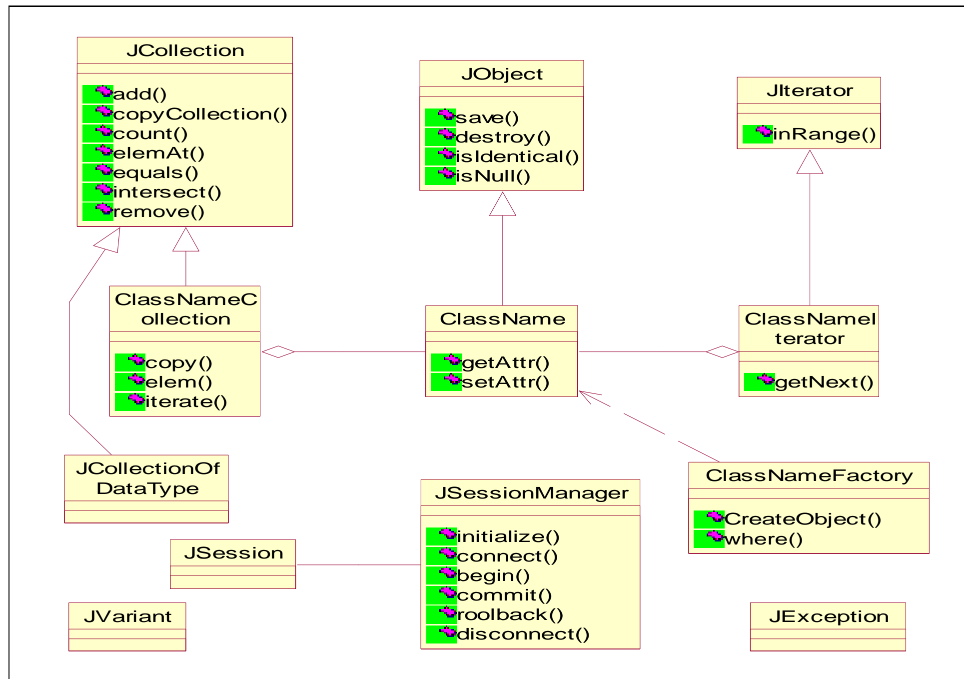


Figura 3.4. ***Framework*** **de suporte à persistência de** ***Jasmine***

A Tabela 3.2 mostra como as operações do padrão CRUD, para persistência de objetos, podem ser implementadas na linguagem de progamação Java utilizando os recursos providos pelo *binding* do banco de dados com a aplicação.

TABELA 3.2. Operações de persistência e *binding* de *Jasmine*

Operação do Padrão CRUD	Recurso disponível no *binding* com a linguagem Java
Create	Método `createObject()` da classe `Factory` e método `save()` da classe abstrata `Jobject`.
Read	Método `where(`*`condição`*`)` da classe `Factory`, que retorna um `iterator` de objetos agregados da classe. A *condição* nula retorna todos os objetos da classe (*FindAll*).
Update	Métodos acessórios `sets()` da classe da aplicação são efetivadas após o método `save()`.
Delete	Método `destroy()` da classe abstrata `Jobject` Método `reset()` para atributos de referência.

O padrão ODMG (*Object Data Management Group*) é composto de um modelo de objetos, uma linguagem de definição de objetos (ODL), uma linguagem de manipulação de objetos (OQL) e *binding* com as linguagens C++, *Smalltalk* e Java (CATTEL *et al.*, 2000). Alguns SGBDs orientados a objetos adotaram o padrão ODMG parcialmente, em geral apenas disponibilizando o *binding* com linguagens de programação orientadas a objetos (ELMASRI; NAVATHE, 2000).

Para a linguagem de programação Java, o *binding* com um SGBDOO definido pelo padrão ODMG é baseado no princípio de que a comunicação entre a aplicação e a base de dados deve ser percebida de modo unificado e não como duas linguagens separadas, com limites estabelecidos entre elas. Assim, a implementação proposta pelo padrão ODMG tem como premissa que "o programador não deve perceber na linguagem expressões disjuntas para operações na base de dados e na aplicação" (CATTEL *et al.*, 2000). Esse princípio define tipos unificados compartilhados pela linguagem de programação e pelo banco de dados, a linguagem não é modificada para acomodar o *binding* e respeita a estrutura semântica de armazenamento de Java.

O *binding* de ODMG com Java estabelece a persistência por alcançabilidade (ou persistência transitiva), em que no encerramento de uma transação (*commit*), todos os objetos alcançáveis a partir de um objeto raiz são atualizados na base de dados.

A implementação do *binding* com a linguagem de programação Java prevê classes pré-existentes que suportam persistência e declarações de classes Java podem ser geradas automaticamente por um pré-processador ODMG. Esses princípios também se aplicam para C++ e *Smalltalk* (CATTEL *et al.*, 2000).

3.5. SISTEMAS GERENCIADORES DE BANCOS DE DADOS OBJETO-RELACIONAIS

Sistemas Gerenciadores de Bancos de Dados Objeto-Relacionais (SGBDORs) mantêm a estrutura de relações como base de armazenamento, mas permitem a representação de objetos sobre as estruturas das relações. Esses gerenciadores incluem suporte a tipos de dados não convencionais, tipos definidos pelo usuário, herança e outras características encontradas em gerenciadores orientados a objetos.

Os SGBDs objeto-relacionais incorporam aspectos que vão desde os convencionais SGBDs relacionais até SGBDs orientados a objetos, formando uma classe bastante heterogênea de produtos. A maioria dos gerenciadores objeto-relacionais adota o padrão ANSI/SQL3 para o modelo de dados e para linguagens de definição e manipulação de dados.

As características que diferenciam os SGBDs orientados a objetos dos objeto-

relacionais podem ser analisadas a partir da origem conceitual desses produtos: enquanto SGBDOOs foram concebidos com a idéia de se unir as características das linguagens de programação orientadas a objetos com as características de bancos de dados (aplicação e banco de dados fortemente acoplados), os objeto-relacionais em geral são sucessores dos tradicionais SGBDs relacionais, concebidos com o princípio da independência de dados (dados e aplicação fracamente acoplados).

Alguns exemplos de gerenciadores objeto-relacionais são Oracle 8*i*/9*i*, *Informix Universal Server*, *HP/Odapter*, *Universal DB/IBM* e *Intersystems/Caché*. Esta seção comenta a seguir a respeito do SGBD *Caché* (Intersystems, 2003), utilizado neste trabalho.

O SGBD *Caché* oferece uma estrutura unificada para representação de classes e tabelas e o sistema gerenciador se encarrega de fazer o mapeamento nas duas direções. A estrutura interna de armazenamento é baseada em *arrays* multidimensionais que permite manipular dados complexos de uma forma eficiente (INTERSYSTEMS, 2003).

Caché apresenta uma arquitetura que suporta acesso ao banco de dados pela aplicação através de bibliotecas de manipulação de objetos e de relações, de modo exclusivo e transparente. O armazenamento interno dos dados é tratado pelo núcleo do gerenciador, que se encarrega de garantir a integridade para ambos os modelos.

A Figura 3.5 mostra a estrutura básica de classes para suporte à persistência em *Caché*. A classe `RegisteredObject` define o método `%New()`, para criação de objetos. Essa classe especializa-se nas classes `Persistent` e `SerialObject`. A classe `Persistent` possui os métodos `%Save()`, para armazenar um objeto, `%OpenId()`, para recuperar um objeto e `%DeleteId()`, para destruir um objeto no banco de dados. Uma classe da aplicação é definida no banco de dados como herdeira de `Persistent` ou de `SerialObject`, quando se tratar de uma classe agregada.

Caché disponibiliza classes de tipos de dados primitivos, como *strings*, numéricos, datas e permite que outros tipos sejam definidos pelo usuário como especializações dessas classes. Para manipular coleções, o SGBD disponibiliza as classes *containers* `RelationshipObject`, `ArrayOfDataTypes`, `ArrayOfObjects`, `ListOfObjects` e `ListOfDataTypes`. Por se tratar de um banco híbrido, em *Caché* os dados também

podem ser manipulados em uma visão relacional, nas relações que são mapeadas pelo SGBD. Para isso, a classe `ResultSet` dá suporte à representação e manipulação de conjuntos de tuplas.

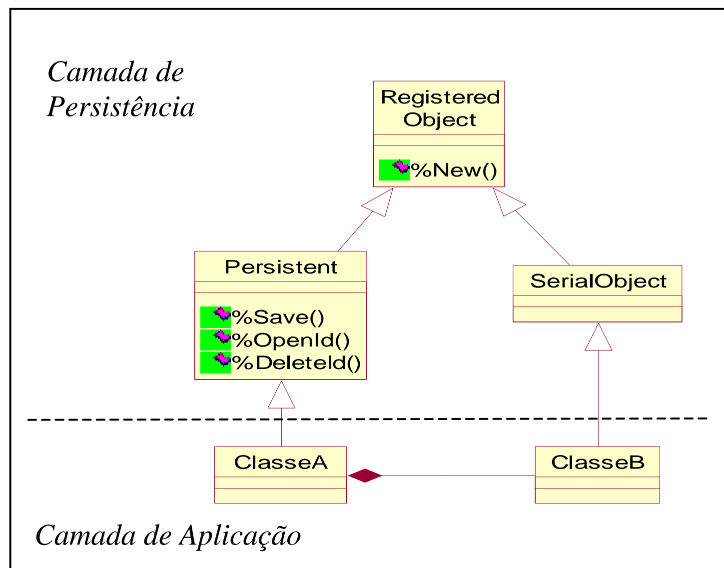


Figura 3.5. Estrutura de suporte à persistência em *Caché*

Relacionamentos entre objetos são classificados em duas categorias: Pai-Filho e Um-para-Muitos. Relacionamentos do tipo Um-para-Muitos possuem integridade referencial e em um relacionamento pai-filhos a exclusão da instância mandatória (cardinalidade pai) automaticamente exclui as instâncias subordinadas (com cardinalidade filho).

Um objeto é identificado na classe pelo seu ID e em todo o banco de dados pelo seu OID, que incorpora o ID, a classe e o nome do pacote referente ao objeto. Em *Caché* o ID pode ser gerado seqüencialmente pelo gerenciador ou pode-se definir que um determinado atributo o seja. O conceito de ID/OID em *Caché* difere do padrão para gerenciadores orientados a objetos, no qual os identificadores de objetos são completamente gerenciados pelo SGBD e não são acessíveis ao programador.

Caché oferece ferramentas tanto para definição de classes de objetos no banco de dados (*Object Architect*) quanto para tabelas SQL (*SQL Manager*). Os métodos definidos no banco de dados são expressos em *ObjectScript*, linguagem proprietária do produto. *Caché* não possui uma linguagem de consulta de objetos (OQL) e faz uso de SQL para expressar cláusulas de consultas "a*d-hoc"* na base de dados, acessando objetos a partir dos IDs obtidos

na consulta.

No *binding* com a linguagem de programação Java, de modo semelhante ao SGBD *Jasmine*, a partir das definições das classes no banco de dados, a ferramenta *ObjectArchitect* gera classes que, em conjunto com a biblioteca `CacheJava.jar`, oferecem ao desenvolvedor os recursos necessários para manipular na aplicação os objetos de classes do banco de dados, conforme indica a Figura 3.6. Para cada classe definida no banco de dados é gerada uma classe *Proxy*. Nessa classe estão definidos os construtores, os métodos acessórios, os métodos da classe e *queries*.

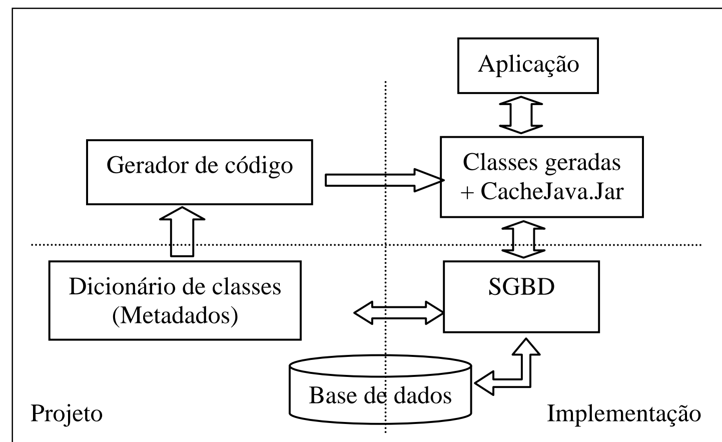


Figura 3.6. *Binding* de *Caché* com Java

TABELA 3.3. Operações de persistência para o *binding* de *Caché* com Java

Operação do Padrão CRUD	Recurso disponível no *binding* com a linguagem Java

Create	`OREF = new object (factory);`
	`OREF.setAtributo(valor);`
	`OREF._save();`

Read	`OREF = new object(factory,ID);`
Update	`OREF.setAtributo(valor);` `OREF._save();`
Delete	`OREF._deleteId();`

Para a linguagem de programação Java, a Tabela 3.3 indica como as operações do padrão CRUD podem ser implementadas, para um objeto OREF genérico.

A Figura 3.7 exemplifica as operações básicas de manipulação de objetos definidos no banco de dados em um programa Java. A linha 1 define a conexão com o banco de dados, estabelecendo a *factory* `f`. Na da linha 2, o objeto instanciado `mat` é uma referência a um

objeto `Material` definido na *factory* `f`. Nas linhas 3 a 5, alguns atributos do objeto material são atualizados, pelo uso de métodos acessórios `sets` e na linha 6 o objeto é persistido no banco de dados. Na linha 7 é lido do banco de dados o material cujo ID tem valor 1, atribuindo-o à OREF `mat`. Nas linhas 8 a 10 são exibidos os conteúdos de alguns atributos do objeto `mat1`, inclusive o atributo `Prateleira`, do objeto embutido `Localizacao`. Na linha 11 ocorre a chamada do método `chegouMinimo()` e na linha 14 o objeto correspondente à OREF `mat` é removido da base de dados.

```
...
1.   ObjectFactory f = new ObjectFactory("cn_iptcp:127.0.0.1[1972]:SUPRIM");
2.   Material mat=new Material(f);
3.   mat.setCodigo(10);
4.   mat.setDescricao("martelo");
5.   mat.setUnidade("pc");
6.   mat._save();
...
7.   Material mat=new Material(f,1);
8.   out.println("descricao:");out.println(mat.getDescricao());
9.   out.println("unidade:");out.println(mat.getUnidade());
10.  out.println("prateleira:");out.println(mat.getLocalizacao().getPrateleira());
11.  if (mat.chegouMinimo()) {
12.  out.println("material atingiu ponto de ressuprimento");
13.  }
...
14.  mat._delete()
...
```

Figura 3.7. Manipulação de objetos *Caché* em Java

3.6. CONSIDERAÇÕES SOBRE A SELEÇÃO DO BANCO DE DADOS

Esta seção referencia alguns autores que apresentam critérios que podem ser adotados quanto à escolha do gerenciador do banco de dados.

Larson (1995) e Elmasri e Navathe (2000) citam que SGBDOOs são mais indicados para aplicações que envolvem objetos complexos com alto grau de associações entre eles, como aplicações para projeto e manufatura em engenharia (CAD/CAM e CIM), computação científica, sistemas de informação geográficos e multimídia.

Case, Henderson-Sellers e Low (1996) descrevem um método geral de análise e projeto orientado a objetos e incluem considerações sobre o sistema gerenciador de banco de dados, definindo a escolha de seu modelo e de seu produto como fases do método. A Tabela 3.4 resume os critérios propostos pelos autores para escolha do modelo do sistema

gerenciador, entre relacional e orientado a objetos, levando em conta os conceitos existentes na época.

TABELA 3.4. Critérios para seleção do tipo de SGBD (CASE; HENDERSON-SELLERS; LOW, 1996)

Critério	**Gerenciadores Orientados a Objetos**	**Gerenciadores Relacionais**
Estrutura de Dados	Objetos complexos Tipos de dados não convencionais Relacionamentos de herança, agregação e associação Grande número de classes de objetos Pequeno número de instâncias	Objetos simples Pequeno número de classes de objetos Grande número de instâncias
Suporte a versões	Diferentes versões dos dados representados	Representação atual dos dados
Transações	Transações de longa duração Transações interativas e cooperativas Transações aninhadas	Transações independentes
Queries	Navegação através de objetos relacionados	Acesso seqüencial por linhas e colunas
Distribuição	Bases de dados distribuídas	Bases de dados centralizadas
Investimentos existentes	Pessoas com experiência e treinamento no uso de SGBDOO e ferramentas integradas com o SGBDOO utilizado	Pessoas com experiência e treinamento no uso de SGBDR e ferramentas integradas com o SGBDR utilizado
Padrões	Nenhum padrão de modelo de dados ou linguagem de consulta a bancos de dados estabelecida	Padrões estabelecidos de modelo de dados e linguagem baseados no modelo relacional
Performance	Performance melhora para objetos complexos	Problemas de performance se ocorrem muitos *JOINS* entre as relações
Regras de negócios	Modelo mais adequado para integrar dados com regras de negócio	Necessidade de definir procedimentos para incorporar regras de negócio com dados
Encapsulamento de dados de outros SGBDs relacionais	Menos trabalho requerido	Mais trabalho requerido

Uma vez definido o modelo do SGBD a ser adotado, na escolha do SGBD dentre os disponíveis comercialmente para o particular modelo, alguns fatores devem ser considerados, tais como conjunto de ferramentas do produto, suporte do fornecedor, suporte à linguagem de programação, integração com sistemas e bases de dados legadas, suporte a trabalho cooperativo, interoperabilidade e desempenho. Na aferição de desempenho, é importante que *benchmarks* sejam feitos de modo realístico, com dados da organização ao invés de se confiar em *benchmarks* públicos. Qualquer *benchmark* deve incluir bases de dados de tamanhos realísticos, estruturas de dados que reflitam os projetos da organização, número reais de usuários concorrentes, plataformas de hardware e infra-estrutura de rede compatíveis (ROTZEL e LOOMIS; LAI e GUZENDA; BUTTERWORTH *apud* CASE; HENDERSON-SELLERS; LOW, 1996).

Elsmari e Navathe (2000) apresentam uma classificação que associa a complexidade dos dados e das consultas de uma aplicação com o modelo de SGBD. Essa classificação é representada na Figura 3.8 em que os eixos indicam a complexidade de armazenamento e

consulta e cada quadrante o modelo sugerido pelos autores.

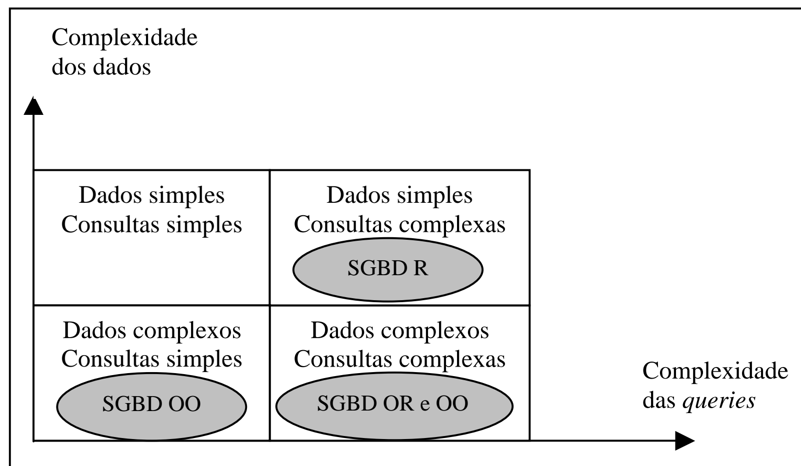


Figura 3.8. Classificação dos gerenciadores pela complexidade (ELMASRI;NAVATHE, 2000)

3.7. CONSIDERAÇÕES FINAIS

Sistemas gerenciadores de bancos de dados de diferentes modelos possuem diferentes arquiteturas e mecanismos de integração com a aplicação. Como conseqüência, em uma aplicação orientada a objetos a seleção do modelo de SGBD a ser utilizado para persistência de dados acarreta impactos no processo de desenvolvimento ou de reengenharia de um sistema. Outros aspectos não funcionais, como manutenibilidade, legibilidade, portabilidade e desempenho também podem ser afetados em função do modelo selecionado.

A Figura 3.9 mostra as três principais alternativas para persistência de dados em aplicações orientadas a objetos. Este capítulo discutiu, para as alternativas (a) e (c), o tratamento da persistência em bancos de dados relacionais e com suporte a objetos, respectivamente. O Capítulo 2 comentou sobre o padrão *Persistence Layer* (YODER; JOHNSON; WILSON, 1998), que insere-se no contexto da alternativa (b) da figura. Na alternativa (a), quando se utiliza um SGBD relacional, o código referente à persistência corresponde às cláusulas SQL que manipulam o banco base de dados e esse código é mesclado com o da linguagem de programação utilizada na aplicação, tornando-o menos legível e com menor manutenibilidade em relação às outras alternativas. Alguns autores enfatizam que o uso de sistemas gerenciadores de bancos de dados relacionais para persistir dados em aplicações orientadas a objetos consomem maior tempo do implementador devido

ao *gap* entre os modelos da aplicação e do banco de dados (YODER; JOHNSON; WILSON (1998); LEAVITT (2000)). Na alternativa (b), também com uso de um gerenciador de banco de dados relacional, é introduzido um padrão de projeto para isolar a persistência. *Frameworks* de persistência em bancos de dados relacionais são boas soluções de projeto por serem reutilizáveis, reduzem o esforço de desenvolvimento, melhoram a manutenibilidade da aplicação por implementar o mapeamento objeto-relacional em uma camada intermediária e por serem acopláveis a vários SGBDs relacionais. Na alternativa (c) utiliza-se um SGBD com suporte a objetos, que iguala o paradigma do banco de dados ao da aplicação e o *binding* do banco de dados com a aplicação dá suporte à persistência de modo transparente.

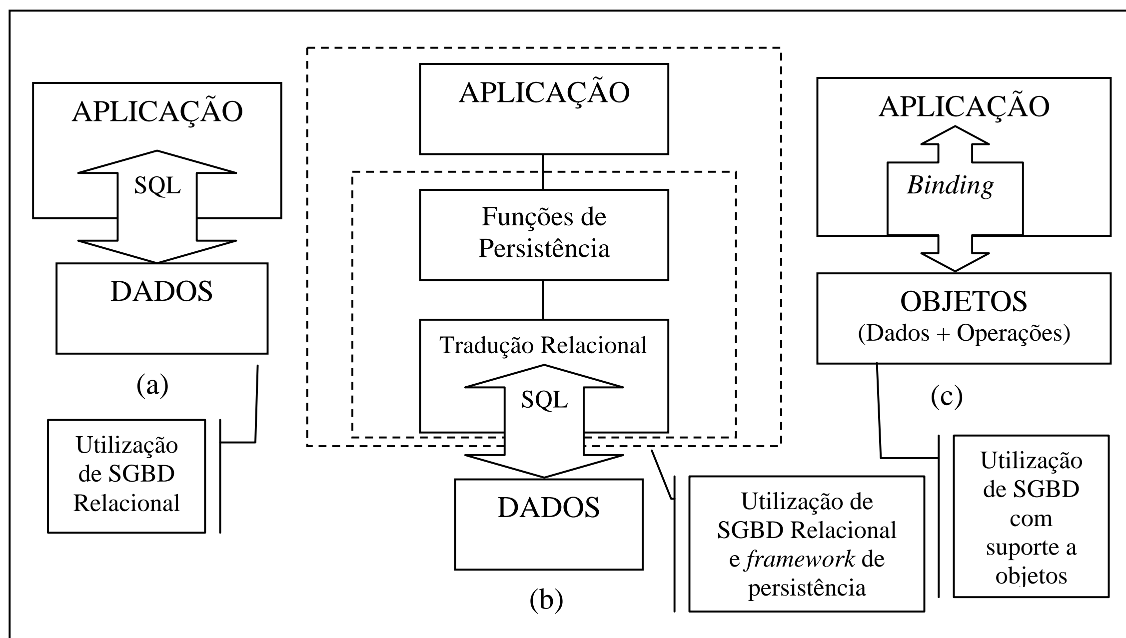


Figura 3.9. Principais alternativas de persistência

Foram apresentados neste capítulo, para os SGBDs *Jasmine* e *Caché*, a infra-estrutura de suporte à persistência desses produtos. O capítulo seguinte apresenta o processo para condução da engenharia avante em um processo de reengenharia orientada a objetos, relativo à utilização de um SGBD com suporte a objetos, especializando-o para os dois particulares SGBDs estudados.

Capítulo 4

4. ENGENHARIA AVANTE COM USO DE SGBDs COM SUPORTE A OBJETOS

4.1. CONSIDERAÇÕES INICIAIS

A etapa de engenharia reversa orientada a objetos permite que um modelo de análise seja recuperado a partir do código fonte. Esses modelos são utilizados na etapa de engenharia avante para completar o processo de reengenharia orientada a objetos e permitem a utilização de diferentes SGBDs – orientados a objetos ou objeto-relacionais – os quais apresentam aspectos inerentes à sua utilização. Um processo para apoiar os engenheiros de software na condução da fase de engenharia avante quanto à utilização de um sistema gerenciador de banco de dados com suporte a objetos pode minimizar os esforços dessa fase.

Este capítulo aprsenta uma proposta de condução de engenharia avante de sistemas orientados a objetos a partir de modelos de análise obtidos na etapa de engenharia reversa orientada a objetos de sistemas legados procedimentais. Assim, a Seção 4.2 apresenta os passos que devem ser realizados durante a etapade engenharia avante. As Seções 4.3 e 4.4 especificam como dois SGBDs – *Jasmine* (COMPUTER ASSOCIATES, 2003) e *Caché* (INTERSYSTEMS, 2003), respectivamente, podem ser utilizados na persistência de dados e na Seção 4.5 são apresentadas as considerações finais.

4.2. A ETAPA DE ENGENHARIA AVANTE

Durante a etapa de engenharia avante os modelos de análise do sistema denominados de MAS pela abordagem *Fusion/RE* (PENTEADO, 1996) e pela FaPRE/OO (RECCHIA, 2002), são utilizados juntamente com o modelo de processo incremental permitindo ao engenheiro de software um aperfeiçoamento gradativo do sistema em desenvolvimento. Dessa forma, as funcionalidades do sistema legado são identificadas, eliminando-se as

anomalias existentes em relação ao tratamento de dados por parte dos procedimentos.

A Figura 4.1 mostra o processo que será utilizado neste trabalho para a realização da engenharia avante de um processo de reengenharia orientada a objetos. A notação utilizada indica as fases do processo através de retângulos com cantos arrendondados, os retângulos internos a esses indicam as atividades em cada fase enquanto que as setas representam a seqüência das atividades. Como o processo é incremental as setas externas ao retângulo representam as iterações que podem existir.

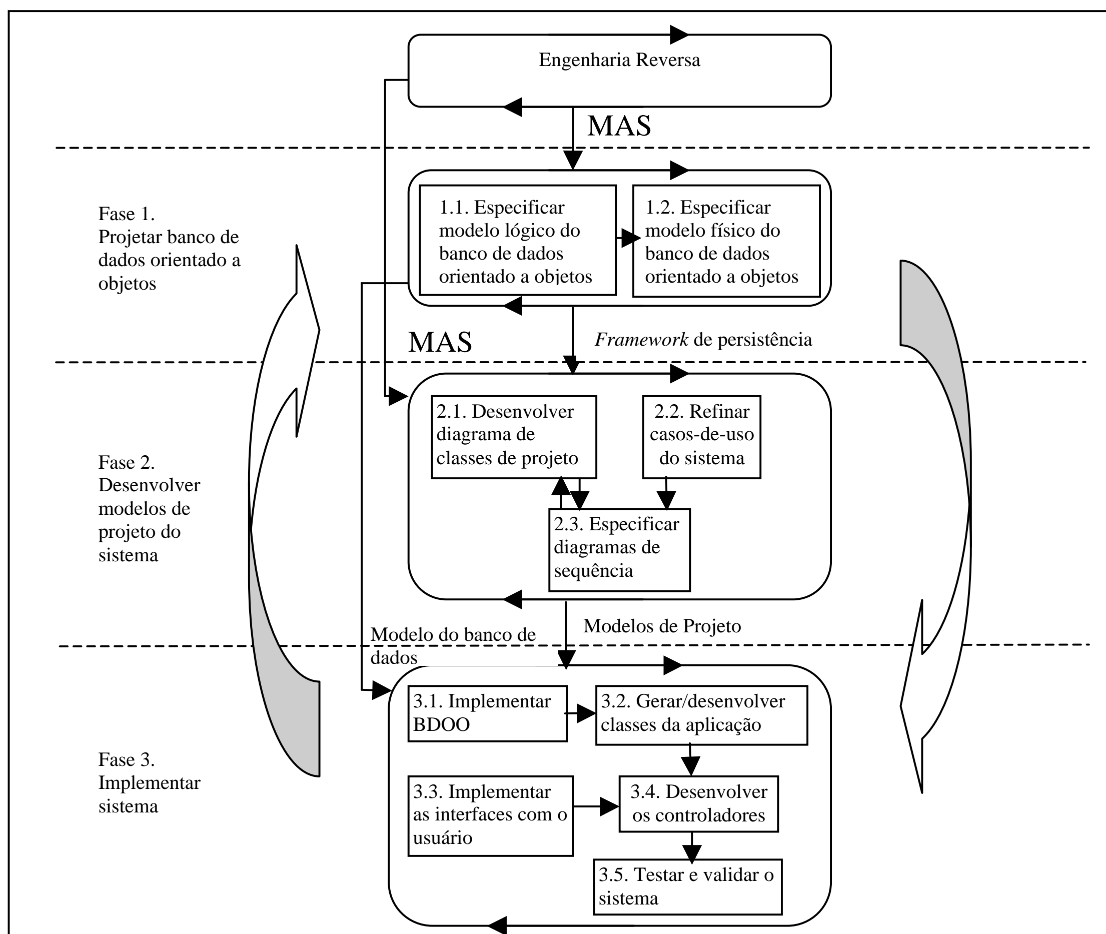


Figura 4.1. Etapa de engenharia avante em um processo de reengenharia orientada a objetos utlizando bancos de dados de objetos

As atividades 1.2, 3.1 e 3.2 são dependentes do SGBD utilizado, enquanto as demais são independentes. Ressalta-se que a atividade 2.1 é realizada considerando o *framework* disponibilizado pelo SGBD. A seguir, serão comentadas as fases e atividades do processo proposto.

FASE 1: PROJETAR BANCO DE DADOS ORIENTADO A OBJETOS

O projetista deve desenvolver a modelagem do banco de dados orientado a objetos para as classes persistentes da aplicação definidas no MAS. A modelagem conceitual expressa no MAS, descreve o conteúdo da informação em uma estrutura em alto nível, sendo desenvolvidas as modelagens lógica – dependente do modelo do SGBD e física – dependente do particular SGBD adotado. Este trabalho sugere que seja utilizada a notação do padrão ODMG (CATTEL *et al.*, 2000) para representar os esquemas de classes, devido a sua aceitação pela comunidade de bancos de dados de objetos.

ATIVIDADE 1.1: ESPECIFICAR MODELO LÓGICO DO BANCO DE DADOS ORIENTADO A OBJETOS

O projetista deve desenvolver o esquema lógico de classes para o banco de dados orientado a objetos, definindo para cada classe do MAS os atributos simples, os atributos de referências, os atributos de coleção, os relacionamentos inversos e as interfaces dos métodos.

Cada relacionamento do MAS deve ter o seguinte tratamento:

a) Associação: é definida indicando-se os atributos que a compõe e o relacionamento inverso entre esses atributos. Dependendo da cardinalidade da associação, são utilizados atributos simples ou de coleção.

b) Agregação: É especificado um atributo de referência na classe agregante (“todo”) do tipo do objeto agregado (“parte”), sendo o atributo simples ou de coleção, dependendo da cardinalidade do relacionamento.

c) Especialização: As heranças de uma classe especializada são especificadas.

ATIVIDADE 1.2: ESPECIFICAR MODELO FÍSICO DO BANCO DE DADOS ORIENTADO A OBJETOS

Na modelagem física do banco de dados, o projetista deve especificar as classes considerando as características, as restrições e os recursos que o particular SGBD adotado oferece. O esquema físico descreve a especificação de como as classes podem ser implementadas nesse SGBD, que será a base para o banco de dados a ser criado, testado e tornar-se operacional. As Seções 4.3 e 4.4 apresentam, respectivamente, como o projeto físico

pode ser conduzido especificamente para os gerenciadores *Jasmine* e *Caché*.

FASE 2: DESENVOLVER MODELOS DE PROJETO DO SISTEMA

O projetista deve especificar os modelos de projeto do sistema, relacionados com a solução lógica adotada. O modelo de classes de projeto deve ser desenvolvido representando os aspectos estruturais do sistema. Os aspectos comportamentais, associados às funcionalidades do sistema, são representados com o refinamento dos diagramas de caso de uso e o desenvolvimento dos correspondentes diagramas de seqüência. A ferramenta *Rational Rose* (RATIONAL, 2003) é utilizada para facilitar a diagramação dos modelos UML (OMG, 2003).

ATIVIDADE 2.1. DESENVOLVER DIAGRAMA DE CLASSES DE PROJETO

O modelo de classes de projeto deve ser desenvolvido considerando a arquitetura do sistema. O projetista deve especificar as classes em cada camada arquitetural e as interfaces entre elas: as classes do domínio da aplicação, as classes de suporte à persistência provido pelo gerenciador de banco de dados e as classes que comunicam a interface com a aplicação. Classes de suporte funcional à aplicação, como classes *containers* para manipulação de conjuntos, classes para suporte à conexão com o banco de dados e para controle de transações, também compõem o modelo de projeto.

Os componentes que intermediam a interface com o usuário e a aplicação, tratando os eventos do sistema, também são chamados de controladores de fachada ou controladores de caso-de-uso (LARMAN, 1999). A utilização de controladores é prevista no padrão MVC (*Model-View-Controller*) e torna a aplicação mais flexível e reutilizável (GAMMA *et al.*, 1995). Um controlador, para implementar a funcionalidade específica para o qual é projetado, deve conter as associações com as classes de aplicação, classes de suporte à persistência, classes utilitárias e classes que controlam conexões com banco de dados e transações, que devem estar representadas no diagrama de classes de projeto. Dependendo da característica da aplicação, há controladores específicos, por exemplo, *servlets*, ASP ou JSP para aplicações WEB ou *units* em *Delphi*, para aplicações com interface WIMP (Window, Icon, Menu, Pointer).

ATIVIDADE 2.2: REFINAR CASOS-DE-USO DO SISTEMA

O modelo de casos de uso na fase de análise é apresentado de forma essencial, sem incorporar detalhes de projeto e de interface. Na fase de projeto, os casos de uso devem ser refinados de modo que as particularidades de projeto sejam especificadas.

ATIVIDADE 2.3: ESPECIFICAR DIAGRAMAS DE SEQÜÊNCIA

Ao desenvolver o diagrama de seqüência deve-se observar como a interação entre classes se efetiva através de trocas de mensagens entre objetos, através de métodos, o diagrama de classes pode ser refinado e complementado, incorporando métodos não previstos na atividade 2.1. Os diagramas de seqüência especificam a lógica que deve ser implementada nos controladores de casos de uso para tratar as funcionalidades do sistema.

O diagrama de colaboração da UML também pode, alternativamente, ser utilizado nessa atividade. Optou-se pelo diagrama de seqüência por representar, além da interação entre objetos, a seqüência temporal das ações correspondentes à funcionalidade tratada pelo diagrama.

Outros modelos podem ser acrescidos ao projeto do sistema, como os diagramas de estado e os diagramas de atividades, caso sejam relevantes nesta fase e conforme as características da aplicação.

FASE 3: IMPLEMENTAR SISTEMA

Nesta fase, a implementação deve ser conduzida com base nas especificações desenvolvidas durante a fase de projeto e na linguagem de programação adotada. Os artefatos básicos de implementação que irão compor o sistema incluem as interfaces com o usuário, as classes de domínio da aplicação, o *binding* com o banco de dados e os controladores da aplicação.

ATIVIDADE 3.1. IMPLEMENTAR BANCO DE DADOS ORIENTADO A OBJETOS

A partir do esquema físico de classes, o banco de dados deve ser criado, implementando-se as classes através de *scripts* da linguagem de definição de dados do gerenciador adotado ou pelo uso de ferramenta do ambiente do SGBD que dê suporte à criação e administração de classes de objetos.

ATIVIDADE 3.2: GERAR/DESENVOLVER CLASSES DA APLICAÇÃO

Essa atividade corresponde ao desenvolvimento da camada de aplicação, composta pelas classes de domínio do sistema. Uma vez que se utiliza um banco de dados de objetos e que as classes persistentes já foram definidas no gerenciador de banco de dados, nessa atividade deve ser considerado o acoplamento entre o banco de dados e a linguagem de programação orientada a objetos utilizada, que é particular para cada SGBD.

ATIVIDADE 3.3: IMPLEMENTAR AS INTERFACES COM O USUÁRIO

Em sistemas legados procedimentais, geralmente, as interfaces com o usuário apresentam-se mescladas com o código que implementa as funcionalidades do sistema. Ao se proceder a reengenharia, alguma técnica é aplicada para "separar" a interface da lógica da aplicação na fase de engenharia reversa. Na engenharia avante, as interfaces com o usuário devem ser desenvolvidas ou aprimoradas para o formato especificado (WIMP, WEB, interface texto, etc). Para isso o padrão "Promover melhorias na interface" da FaPRE/OO (RECCHIA, 2002), pode ser utilizado.

ATIVIDADE 3.4: DESENVOLVER OS CONTROLADORES

Nesta atividade a camada de comunicação entre a interface e a camada de domínio é desenvolvida. Essa camada corresponde ao conjunto de controladores de casos de uso do sistema, que implementam as funcionalidades previstas em seus requisitos, tratando os eventos da interface. Esta camada é desenvolvida com base nos diagramas de seqüência, especificados na fase de projeto.

ATIVIDADE 3.5: TESTAR E VALIDAR O SISTEMA

Nesta atividade, os artefatos produzidos na fase de implementação devem ser testados e validados com as especificações do sistema. Como se trata de um sistema orientado a objetos, os testes realizados devem ser adequadas a esse paradigma. Assim, uma abordagem de teste "*bottom-up*" deve ser utilizada, aplicando-se testes a partir de classes de objetos individuais em direção à integração entre os componentes do sistema. Uma estratégia de testes para sistemas orientados a objetos é descrita em Sommerville (2001), em que o autor define quatro níveis de testes:

- Testar as operações individuais associadas com objetos;
- Testar classes de objetos individuais;
- Testar agrupamentos de objetos e
- Testar o sistema orientado a objetos.

Embora a fase de testes em sistemas orientados a objetos seja mais metódica e dispendiosa se comparada a sistemas procedimentais, ressalta-se que uma vez que os componentes individuais são validados, pode-se reutilizá-los em outros projetos.

4.3. O PROCESSO DE ENGENHARIA AVANTE COM O USO DO SGBD *JASMINE*

Nesta seção são comentadas apenas as atividades específicas para o uso do SGBD *Jasmine* (COMPUTER ASSOCIATES, 2003), uma vez que as demais são genéricas para o gerenciador com suporte a objetos adotado. A atividade 2.1 é comentada com o objetivo de detalhar o *framework* de suporte à persistência de *Jasmine*.

ATIVIDADE 1.2: ESPECIFICAR MODELO FÍSICO DO BANCO DE DADOS ORIENTADO A OBJETOS

No projeto físico do banco de dados orientado a objetos com uso de *Jasmine*, para especificar o esquema físico deve-se considerar:

a) Associações: Podem ser projetadas pela definição de um atributo de referência (simples ou de coleção) em cada uma das classes que participam do relacionamento de associação ou em apenas uma delas. Para a primeira opção, deve -se prever o controle da consistência entre os atributos, pois o gerenciador *Jasmine* não dá suporte automático a relacionamentos inversos. Para a segunda opção, a navegação a partir do objeto que não foi definido o atributo de referência é prevista com o uso de um objeto *container* populado através de uma *query*. Associações N:N com outros atributos além das referências, devem ser modeladas com a criação de uma classe associativa com duas associações 1:N .

b) Agregações: As agregações são projetadas pela definição de um atributo de referência para a classe agregada, no caso de agregações simples (cardinalidade 1). Para o caso de agregações de conjuntos, pode-se definir um atributo de coleção na classe Todo ou um atributo de referência na classe Parte.

c) Especializações: *Jasmine* suporta herança simples e herança múltipla e todas as classes em uma hieraquia de herança são definidas como classes concretas.

A Figura 4.2 apresenta o mapeamento para ODMG para relacionamentos entre classes de um modelo de análise, denominado MAS, recuperado a partir da engenharia reversa. Para exemplificar associações, considere o exemplo de um professor que ministra zero ou mais disciplinas: o esquema lógico é construído definindo-se um atributo em cada uma das classes envolvidas na associação e o esquema físico é desenvolvido pela especificação na classe `Disciplina` do atributo `professorAtribuido` do tipo `Professor` e na classe `Professor` é definido o atributo `disciplinas`, do tipo `Collection`. Ressalta-se a semântica mais rica dessa especificação se comparada ao mapeamento relacional, que se dá por meio de definição de chaves estrangeiras. A figura apresenta também exemplos para relacionamentos de agregação e especialização.

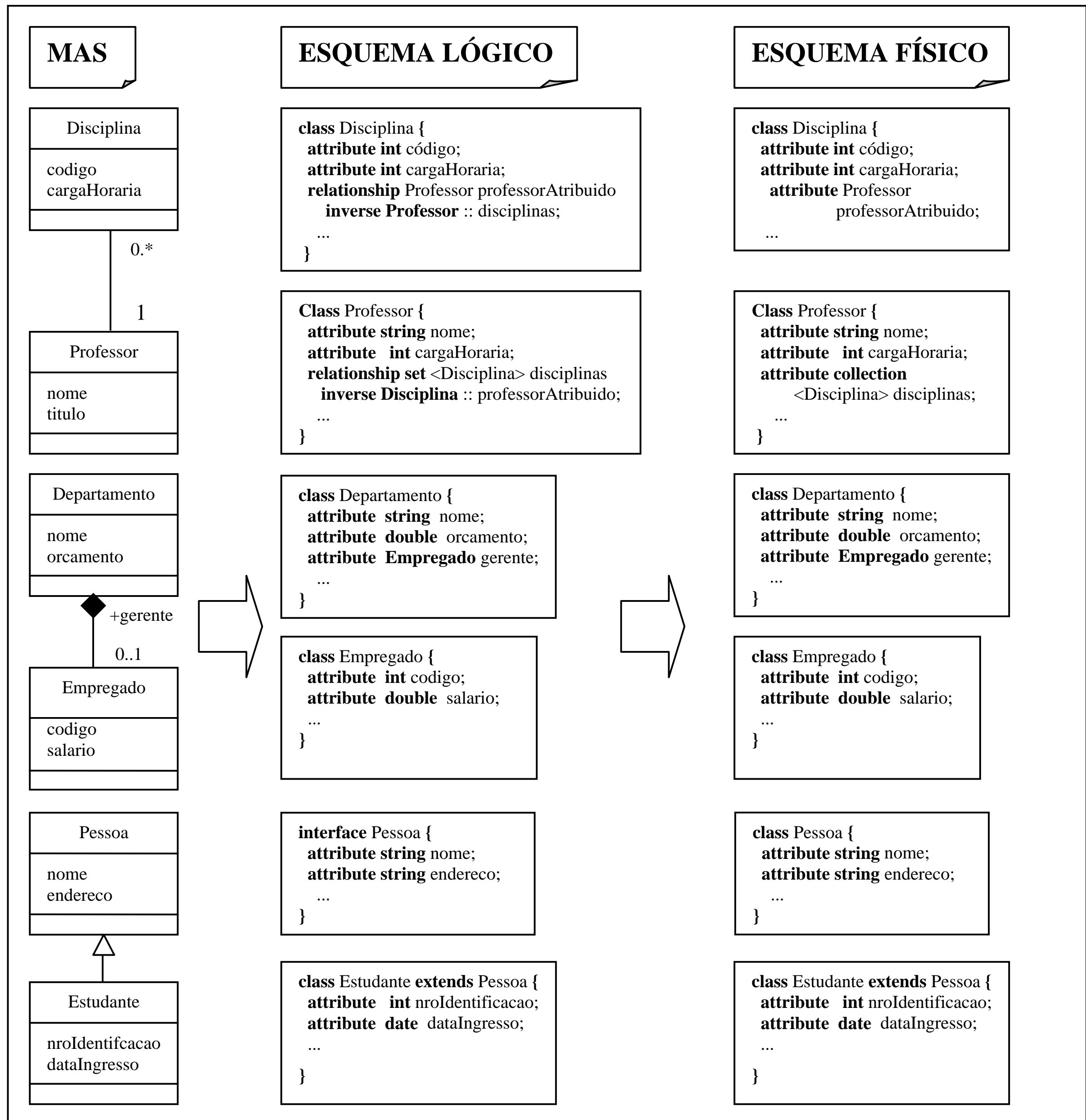


Figura 4.2. Mapeamento MAS x Esquemas de Classes para *Jasmine*

ATIVIDADE 2.1: DESENVOLVER O DIAGRAMA DE CLASSES DE PROJETO

As classes que compõem o modelo de projeto estão distribuídas nas três camadas arquiteturais do sistema: a camada de apresentação – que trata das interfaces com o usuário – a camada de aplicação – que corresponde aos conceitos de domínio – e as classes de persistência – que dão suporte aos serviços de acesso ao banco de dados.

Na composição do diagrama de classes de projeto com uso do SGBD *Jasmine*, deve ser considerado o conjunto de classes que compõem o *binding* do banco de dados com a aplicação.

A Figura 4.3 mostra um exemplo para ilustrar a interação entre as classes de projeto nas camadas de apresentação, aplicação e persistência. Na camada de apresentação `UserInterface` representa um componente de interface com o usuário, que pode ser, por exemplo, uma janela ou uma página em um *web browser*. `ControllerComponent` corresponde a um componente controlador que atende a um evento da interface e interage com as classes de domínio (`ClasseA_` e `ClasseB_`) e com as classes de suporte à persistência providas por *Jasmine*. As classes de domínio são herdeiras de `JObject`, que possui métodos para salvar e destruir objetos. A classe `factory` possui métodos para criar e recuperar objetos do banco de dados. As classes `Iterator e Collection` correspondentes à cada classe de domínio são empregadas como classes utilitárias na aplicação para manipulação de coleções. As classes `JSession` e `JSessionManager` dão apoio à conexão com o banco de dados e com o controle de transações em *Jasmine*.

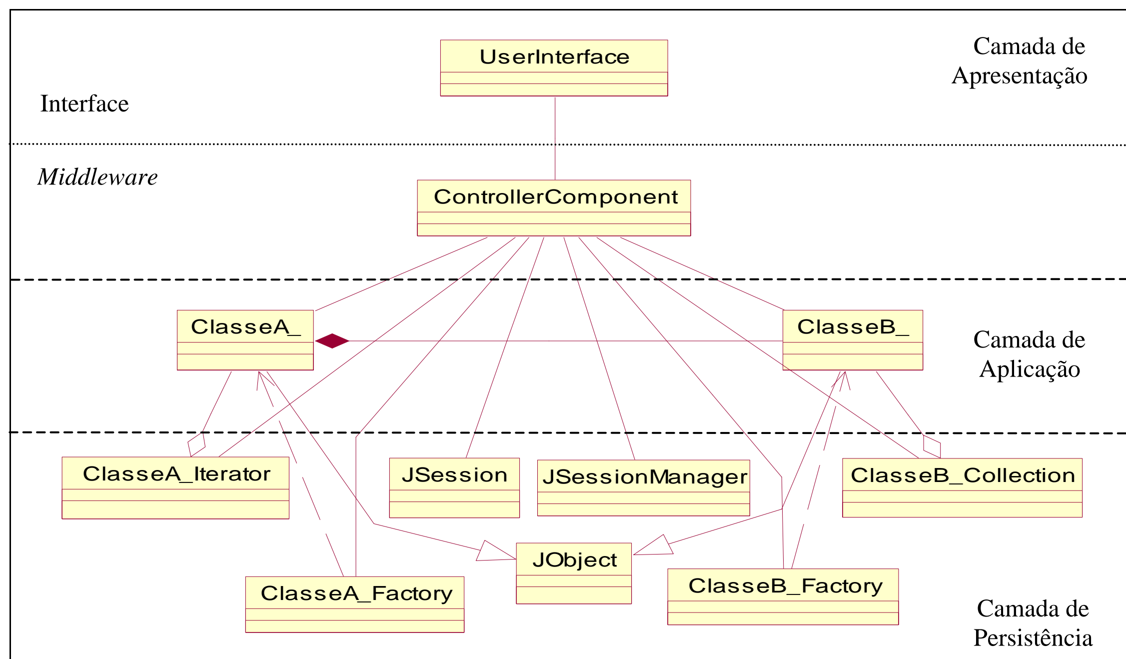


Figura 4.3. Interação entre camadas – modelo de classes de projeto para *Jasmine*

A Figura 4.4 ilustra como deve ser especificado o mapeamento de relacionamentos de herança do diagrama de classes para o modelo de projeto. Nos casos em que a classe pai é concreta, uma classe responsável pela criação de objetos (`factory`) deve ser utilizada.

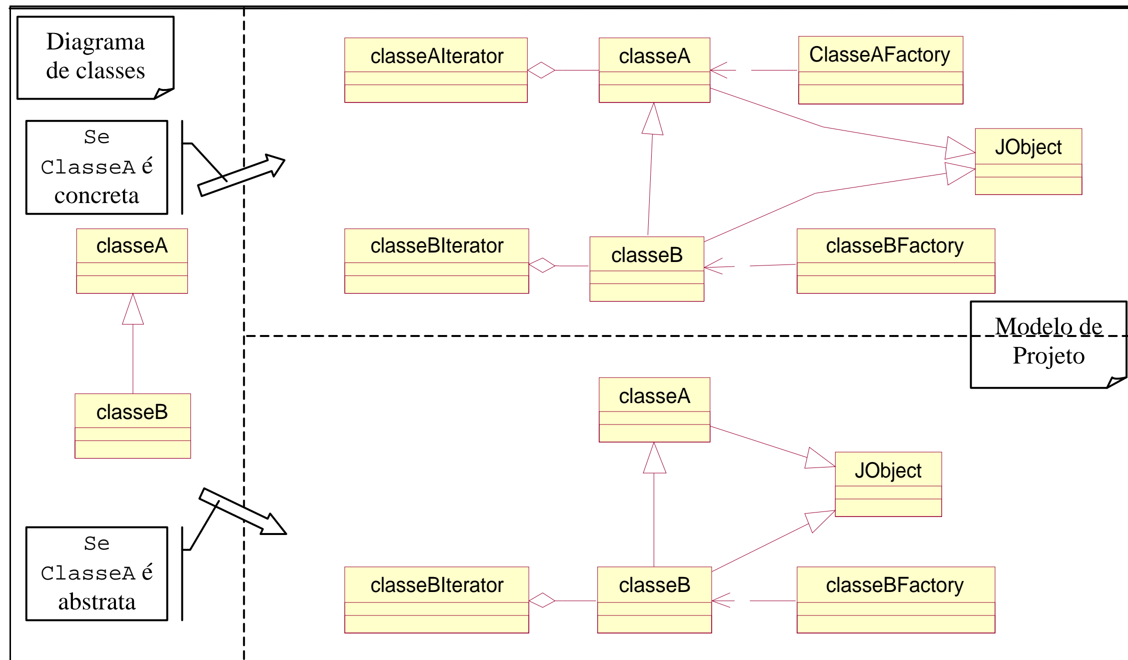


Figura 4.4. Derivação do Modelo de Projeto a partir do MAS para relacionamentos de especialização

ATIVIDADE 3.1: <u>IMPLEMENTAR BANCO DE DADOS ORIENTADO A OBJETOS</u>

O banco de dados é criado com base no esquema físico de classes, implementando-se as classes através de *scripts* da linguagem de definição de dados de *Jasmine* ou pelo uso da ferramenta *Jasmine Studio*. Essa ferramenta provê uma interface para criação e administração de classes e objetos no banco de dados.

ATIVIDADE 3.2: <u>GERAR/DESENVOLVER CLASSES DA APLICAÇÃO</u>

A ferramenta *Jasmine Browser* gera classes descritas na Tabela 3.1 para cada classe definida no banco de dados. Essas classes (Tabela 3.1) em conjunto com outras disponibilizadas pelo gerenciador (*L2 Libraries*) compõem o *binding* da aplicação com o banco de dados.

4.4. O PROCESSO DE ENGENHARIA AVANTE COM O USO DO SGBD *CACHÉ*

Nesta seção será apresentado o processo quando o sistema gerenciador de banco de dados *Caché* (INTERSYSTEMS, 2003) é utilizado na fase de engenharia avante em um processo de reengenharia orientada a objetos.Somente as atividades específicas ao uso deste SGBD no processo são as comentadas nesta seção.

ATIVIDADE 1.2: ESPECIFICAR MODELO FÍSICO DO BANCO DE DADOS ORIENTADO A OBJETOS

Para o projeto físico do banco de dados, a Tabela 4.1 indica, para *Caché*, como as classes no banco de dados podem ser definidas para cada relacionamento constante no esquema lógico do banco de dados.

TABELA 4.1. Mapeamento Lógico-Físico em *Caché*

PROJETO LÓGICO Relacionamento	 Cardinalidade	PROJETO FÍSICO Classe/Tipo
Agregação	1:1	Classe Parte referenciada na classe Todo
	1:N	`Relationship parent-children, List` ou `Array`
Associação	1:1	Atributo de referência em uma das classes
	1:N	`Relationship` *`one-to-many`*
	N:N	Classe associativa com dois relacionamentos 1:N
Especialização		Herança de classes abstratas ou concretas

As associações são representadas no banco de dados pela definição de atributos do tipo Relacionamento (`Relationship`). Para uma classe do banco de dados um relacionamento com outra classe é definido especificando-se uma propriedade, cujo tipo é a classe com a qual o relacionamento é estabelecido e a cardinalidade do relacionamento. As cardinalidades do relacionamento podem ser um-para-muitos ou pai-filhos. As associações um-para-um são previstas pela definição de um atributo de referência. Associações muitos-para-muitos não são suportadas, devendo ser construídas usando dois relacionamentos um-para-muitos. Relacionamentos pai-filhos são empregados em relacionamentos de agregação. Agregações com cardinalidade um-para-muitos com navegação unidirecional também podem ser implementadas utilizando os tipos `list` ou `array`, que representam coleções.

ATIVIDADE 2.1: DESENVOLVER DIAGRAMA DE CLASSES DE PROJETO

A partir do modelo de análise do sistema (MAS) deve-se especificar o modelo de classes de projeto nas camadas de apresentação, de aplicação e de persistência, considerando o conjunto de classes de suporte à persistência de *Caché*. Na Figura 4.5 `ControllerComponent` corresponde a um mediador que recebe os dados da interface com o usuário (`UserInterface`) e interage com as classes de domínio (`ClasseA` e `ClasseB`) e com as classes do *binding* com o banco de dados, na camada de persistência. As classes de domínio são herdeiras de `Persistent` ou `SerialObject` que possuem métodos para criar, salvar, recuperar e destruir objetos (operações do padrão CRUD). A classe `ListOfObject` corresponde a uma das classes que manipulam coleções. A classe `ObjectFactory` serve para conexão com o banco de dados.

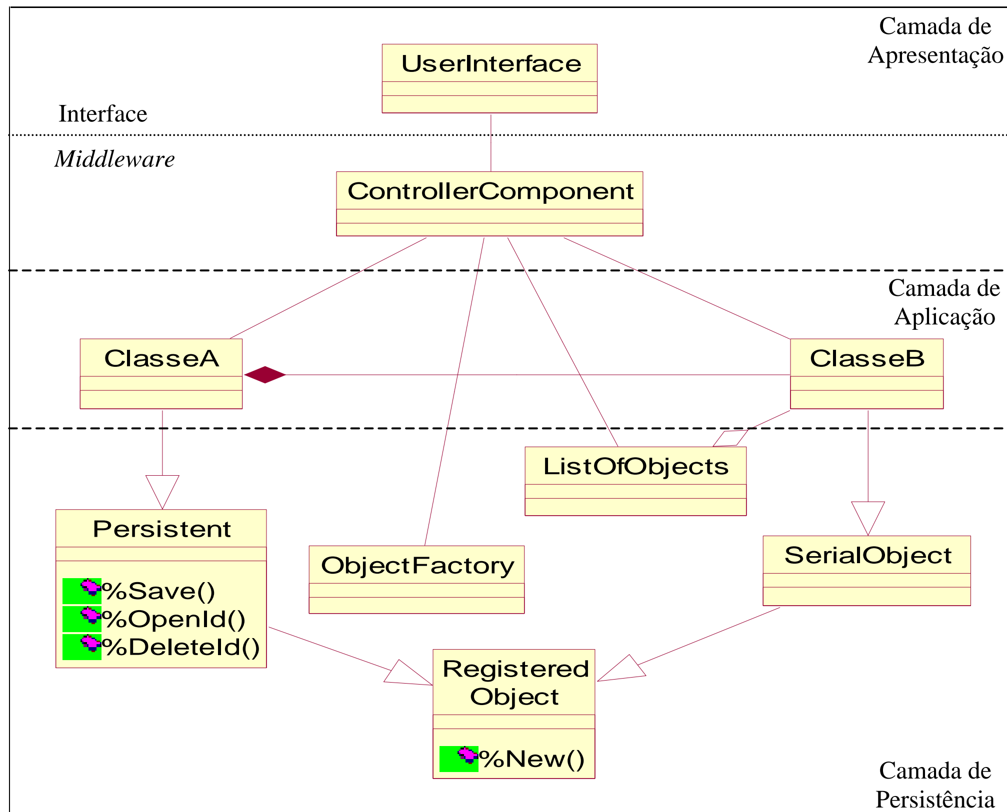


Figura 4.5. Interação entre camadas – modelo de classes de projeto para *Caché*

ATIVIDADE 3.1: IMPLEMENTAR BANCO DE DADOS ORIENTADO A OBJETOS

O banco de dados deve ser definido no gerenciador a partir do esquema físico de classes, através de *scripts* em na linguagem *ObjectScript* ou pelo uso da ferramenta *ObjectArchitect*, do ambiente do SGBD *Caché*.

ATIVIDADE 3.2: GERAR/DESENVOLVER CLASSES DA APLICAÇÃO

Conforme comentado no Capítulo 3, a ferramenta *ObjectArchitect* gera o código para as classes definidas no banco de dados. Para cada classe, é gerada uma classe *Proxy*, que implementa os construtores, métodos acessórios, *queries* e métodos de classe. As classes geradas herdam as operações de persistência definidas nas classes `Persistent`, `SerialObject` e `RegisteredObject`.

4.5. CONSIDERAÇÕES FINAIS

Este capítulo apresentou um processo para a etapa de engenharia avante em um processo de reengenharia orientada a objetos em que utiliza um SGBD com suporte a objetos.

O processo proposto compõe-se de 3 fases para a engenharia avante. A primeira consiste em modelar o banco de dados orientado a objetos, pela especificação das estruturas das classes no banco de dados. Na segunda fase o projetista compõe o modelo de projeto para a arquitetura do sistema, considerando o *binding* do banco de dados com a aplicação. Na terceira fase o sistema é reconstruído.

Dentre as atividades constantes no processo, algumas são genéricas – inviariantes em relação ao SGBD utilizado – e outras específicas, pelo fato de cada gerenciador apresentar sua própria estrutura de *binding* com a aplicação orientada a objetos. Para os SGBDs *Jasmine* e *Caché*, este capítulo apresentou as atividades específicas do processo quando do uso desses gerenciadores.

O uso de banco de dados de objetos iguala os paradigmas do banco de dados e da aplicação, desonerando o projetista de desenvolver o mapeamento de classes para relações e implementar a persistência de objetos em tabelas. Por outro lado, os SGBDs com suporte a objetos possuem diferentes *frameworks* de suporte à persistência e diferentes linguagens de definição e de manipulação de dados, o que dificulta a portabilidade de aplicações e de dados. A portabilidade em bancos de dados relacionais é maior, devido à padronização do modelo relacional e da linguagem SQL.

O próximo capítulo apresenta um estudo de caso no qual foram aplicadas as diretrizes especificadas no processo aqui descrito para um sistema já submetido à engenharia reversa orientada a objetos.

Capítulo 5

5. ESTUDO DE CASO

5.1. CONSIDERAÇÕES INICIAIS

Este capítulo apresenta um estudo de caso para exemplificar a utilização do processo proposto no capítulo anterior. Para isso, um sistema legado implementado originalmente em COBOL e que já havia sido submetido ao processo de reengenharia orientada a objetos usando o método *Fusion/RE* foi utilizado. A reengenharia desse sistema utilizou a linguagem Java sendo disponibilizado para *WEB* por meio de *servlets* com a persistência dos dados realizada com um SGBD relacional (CAMARGO, 2001). Ao final deste processo de reengenharia orientada a objetos tem-se um mesmo sistema implementado em linguagem Java utilizando três diferentes sistemas gerenciadores de banco de dados para a persistência dos dados: relacional, objeto-relacional e orientado a objetos. Assim, é possível realizar uma análise comparativa entre essas versões. Neste estudo de caso o processo apresentado no Capítulo 4 foi especializado para os SGBDs *Jasmine* e *Caché*.

A descrição do sistema legado,considerações sobre o processo de engenharia reversa e reengenharia realizados por Camargo (2001), são apresentadas nas Seções 5.2, 5.3 e 5.4, respectivamente. Nas Seções 5.5 e 5.6 a partir dos modelos de análise gerados anteriormente durante a engenharia reversa são utilizados para conduzir a engenharia avante utilizando-se os SGBDs *Jasmine* e *Caché*, respectivamente. A Seção 5.7 apresenta uma análise comparativa entre as versões desenvolvidas por Camargo e as deste trabalho. A Seção 5.9 apresenta as considerações finais.

5.2. DESCRIÇÃO DO SISTEMA LEGADO

Um sistema legado de controle de estoque é tomado como exemplo para o estudo de caso. O sistema foi originalmente implementado em COBOL *Microfocus* 85, com 74 *Kloc*s distribuídos em 498 módulos, sendo 98 programas e o restante *CopyFiles*. Não há documentação disponível sobre o sistema e os responsáveis pela reengenharia nunca tiveram contato com os desenvolvedores.

O sistema gerencia o armazenamento, estoque, solicitações, ressuprimento, compras e recebimento de materiais. O cadastro de um material deve conter seu código, descrição, unidade de medida, quantidade mínima de estoque, preço médio, saldos físico e financeiro e data do último movimento. Para todo material que é cadastrado deve ser criada, previamente, uma conta contábil sintética para ser associada ao material. A conta contábil registra os débitos e créditos do material, quando ocorrem entradas e saídas. Para cada material, o sistema mantém seus saldos de movimentos para o mês corrente, ano corrente e movimento dos últimos três meses. As informações de armazenamento físico do material – almoxarifado e localização - também são registradas pelo sistema. Uma necessidade de material (requisição de ressuprimento) pode ser registrada no sistema manualmente ou quando na solicitação do material é verificado seu saldo físico abaixo do estoque mínimo. Quando uma requisição de ressuprimento está em aberto, o usuário pode informar a previsão de compra para esse item. Quando o material é comprado são registrados o fornecedor, a quantidade pedida, o preço e o número de parcelas, a requisição de ressuprimento passa para a situação "em processo de compra" e a previsão de compra é excluída. Na entrega do material, o sistema registra os dados da nota fiscal – número da nota, fornecedor, data, valores de acréscimo e desconto. Para cada item da nota fiscal, correspondente a um pedido de compra, são atualizados os saldos físico e financeiro do material, seus saldos de movimento do mês atual, a data do último movimento e a requisição de ressuprimento. O recebimento de um material atualiza sua conta contábil, acrescendo o valor de débito, uma ocorrência de fornecimento do item pelo fornecedor da compra é acrescentada e a compra é encerrada. Quando um material é solicitado, a solicitação é cadastrada com a data, número do lote e quantidade. Caso o material esteja com saldo abaixo do mínimo, é disparada uma requisição de resssuprimento. A solicitação de material é atendida pelo setor de expedição, de modo parcial ou total, subtraindo os saldos do material, seu saldo de movimento do mês atual e creditando sua conta contábil.

5.3. ENGENHARIA REVERSA DO SISTEMA LEGADO

Camargo (2001) instanciou o método *Fusion/RE* (PENTEADO, 1996) para sistemas legados implementados em Cobol. No processo de obtenção das classes do MAS, os atributos foram inferidos a partir das estruturas de *Files Descriptions* (FDs) e os métodos a partir dos procedimentos funcionais inseridos nos módulos do sistema legado. A Figura 5.1 mostra o

mapeamento de um módulo do sistema legado para os métodos associados às classes do MAS. As anomalias decorrentes do paradigma procedimental, em que diferentes estruturas de dados são lidas e atualizadas por um mesmo procedimento, foram tratadas definindo-se os métodos necessários para encapsular as operações associadas às estruturas de dados.

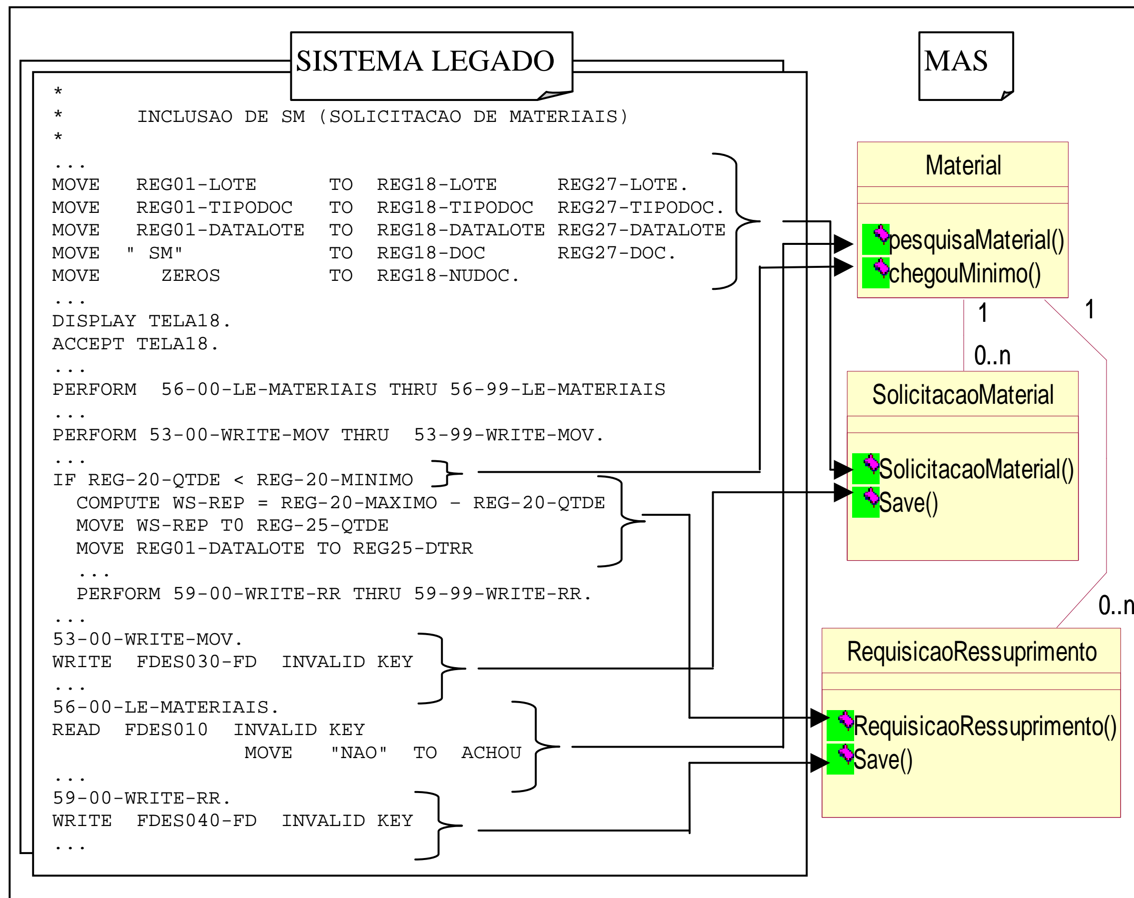


Figura 5.1. Mapeamento de operações entre o Sistema Legado e o MAS
Fonte: Camargo, 2001.

5.4. ENGENHARIA AVANTE COM USO DE SGBD RELACIONAL

A partir dos modelos de análise obtidos na engenharia reversa, Camargo (2001) realizou a engenharia avante para o sistema alvo, em uma arquitetura cliente-servidor com três camadas do tipo *thin-client*, sendo utilizados: a linguagem de programação orientada a objetos Java, a linguagem de marcação HTML para as interfaces, *servlets* da linguagem de programação Java para permitir a comunicação entre essas interfaces, que são executadas no computador cliente, e a aplicação, executada no servidor e o SGBD relacional Sybase (SYBASE, 2003).

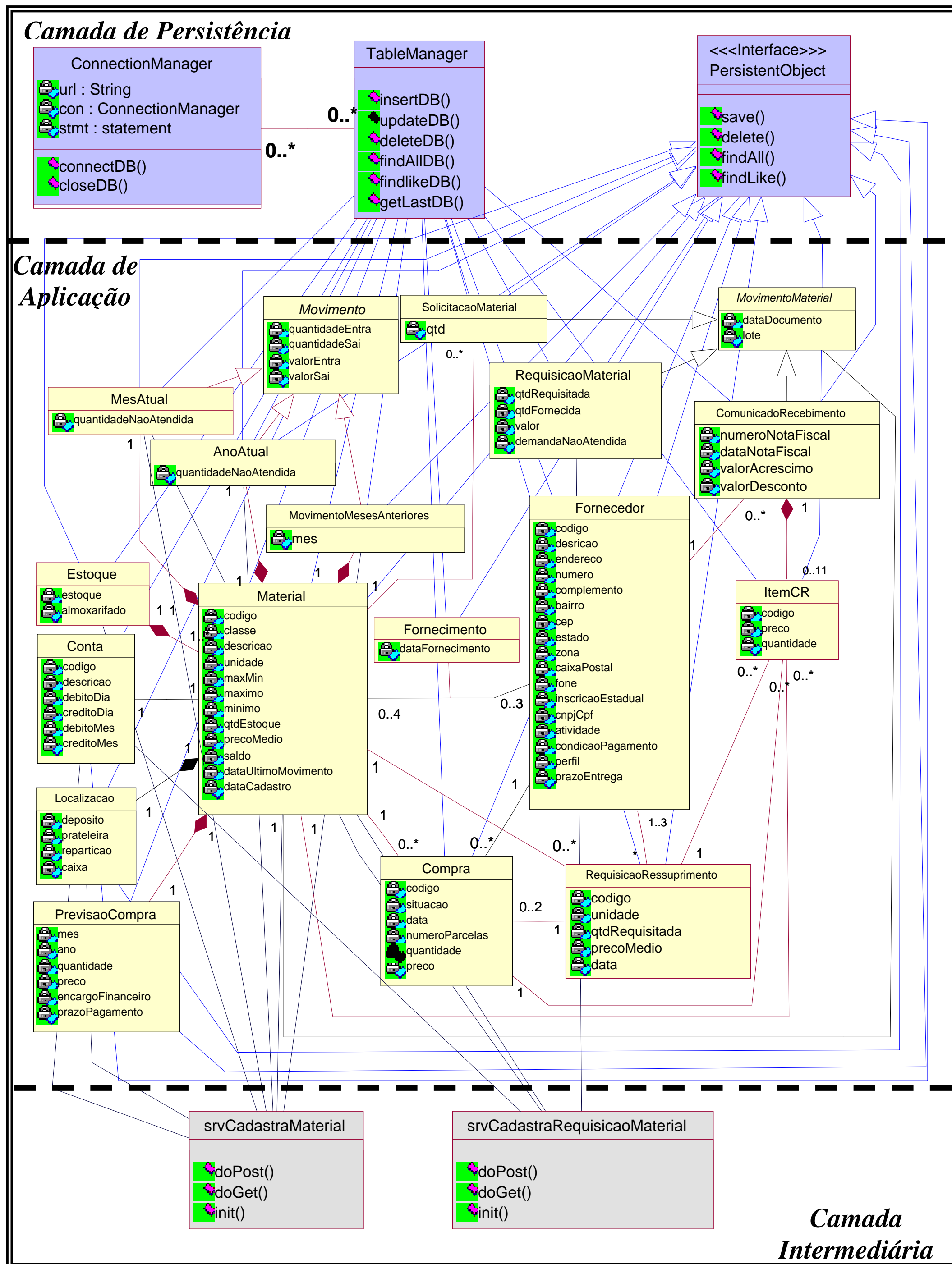


Figura 5.2. Modelo de Classes de Projeto - 1ª Versão do Sistema (CAMARGO, 2001)

O padrão de projeto *Persistence Layer* (YODER; JOHNSON; WILSON, 1998) implementado por Cagnin (1999) foi reutilizado para amenizar as diferenças entre os paradigmas da aplicação, que é orientada a objetos, e do banco de dados, que é relacional. A Figura 5.2 mostra o modelo de classes do sistema em três camadas obtido na fase de

engenharia reversa. A primeira camada é a da persistência e contém as classes que implementam o padrão *Persistence Layer*. A segunda é a da aplicação, contendo as classes específicas de domínio do sistema e a terceira é representada pelos *servlets*, que atuam como mediadores entre os componentes de interface e a aplicação. As classes da camada de aplicação, que devem persistir objetos no banco de dados, herdam da classe `PersistentObject` as operações do padrão CRUD e invocam os métodos da classe `TableManager` para efetivar essas operações no banco de dados relacional.

O projeto do banco de dados foi realizado com base nas classes persistentes da aplicação. Para cada classe foi criada uma tabela correspondente, para cada atributo uma coluna, para relacionamentos de associação ou agregação foram definidas chaves estrangeiras e para relacionamentos de generalização-especificação foram criadas tabelas somente para as sub-classes, adicionando a elas os atributos das superclasses (CAMARGO, 2001).

A partir do modelo de análise orientado a objetos obtido na engenharia reversa realizada por Camargo, o estudo de caso aqui apresentado conta de outras duas versões desse sistema desenvolvidas na mesma arquitetura e com a mesma funcionalidade do anterior. A diferença entre elas é o uso de SGBDs de diferentes paradigmas, uma utilizando o orientado a objetos (*Jasmine*) e a outra usando o objeto-relacional (*Caché*). Essas três versões do sistema exemplo são tratadas para efeito de comparação, neste trabalho, sendo referenciadas como primeira versão a do relacional, a segunda a do orientado a objetos e a terceira a do objeto-relacional, conforme apresentado na Figura 5.3.

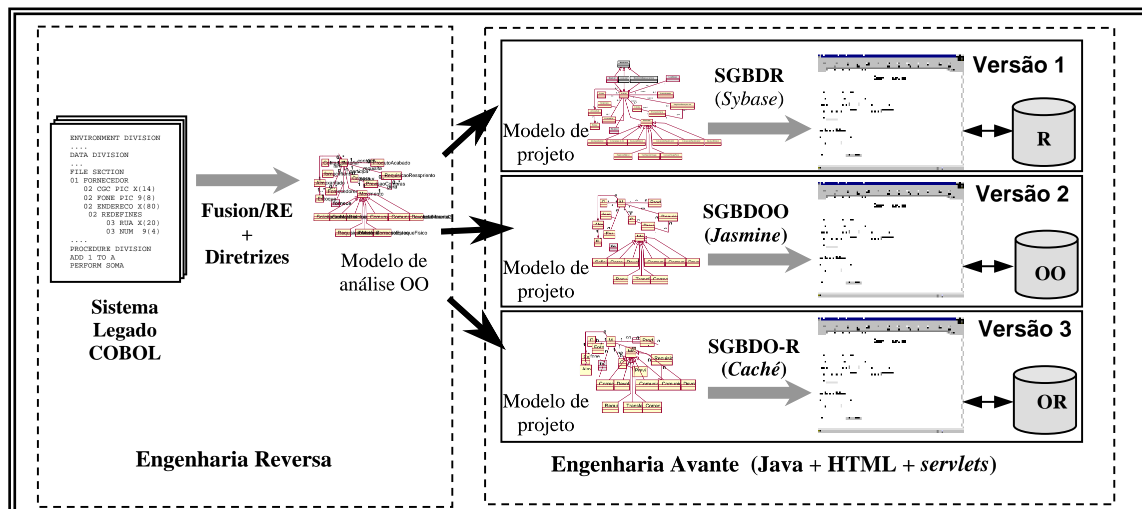


Figura 5.3. Três alternativas de reengenharia para o sistema exemplo

Independente da alternativa de reengenharia, as três versões do sistema exemplo possuem a mesma interface para interação humano-computador. As interfaces para comunicação com o usuário criadas na primeira versão por Camargo (2001) são reusadas, com poucas adaptações, também nas duas versões aqui desenvolvidas. A Figura 5.4.(a) apresenta a interface para cadastro de materiais do sistema legado e a Figura 5.4.(b) essa mesma interface após a reengenharia, quando o sistema é disponibilizado para *WEB*.

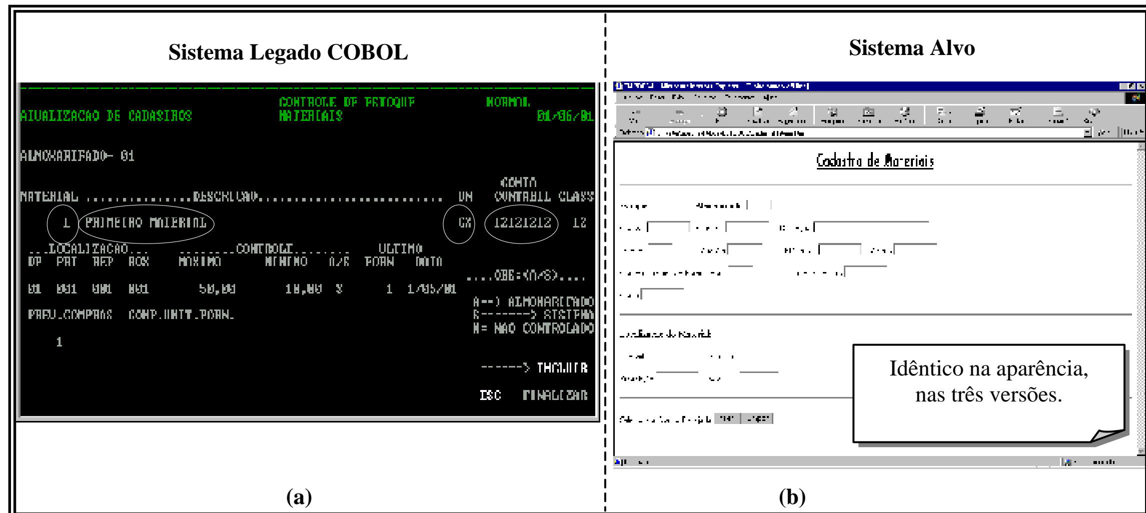


Figura 5.4. Interface do Cadastro de Materiais – Sistemas Legado (a) e Alvo (b)

5.5. ENGENHARIA AVANTE UTILIZANDO O SGBDOO *JASMINE*

Esta seção apresenta a utilização das diretrizes apresentadas no Capítulo 4 para o desenvolvimento da segunda versão do sistema exemplo, a partir do modelo de análise do sistema (MAS), com o sistema gerenciador de banco de dados orientado a objetos *Jasmine* (COMPUTER ASSOCIATES, 2003)

FASE 1: PROJETAR BANCO DE DADOS ORIENTADO A OBJETOS

A partir do MAS, foram desenvolvidos os esquemas de classes, utilizando-se a notação ODMG (CATTEL *et al.*, 2000). Considere classes `RequisicaoMaterial`, `Material` e `PrevisaoCompra`, da Figura 5.2 cujo mapeamento lógico e físico utilizando *Jasmine* é mostrado na Figura 5.5. Na classe `RequisicaoMaterial` foi definido um atributo de referência para um objeto `Material`. Como propriedade inversa, na classe `Material` é definido um conjunto contendo objetos do tipo `RequisicaoMaterial`, de acordo com a

cardinalidade do modelo conceitual para essas classes. Um atributo de referência para a classe `PrevisaoCompra` na classe `Material` foi definido para representar a relação de agregação entre essas classes no MAS. Para essas classes, o mapeamento físico em *Jasmine* foi especificado pela definição de atributos de referência, uma vez que os relacionamentos não são definidos explicitamente.

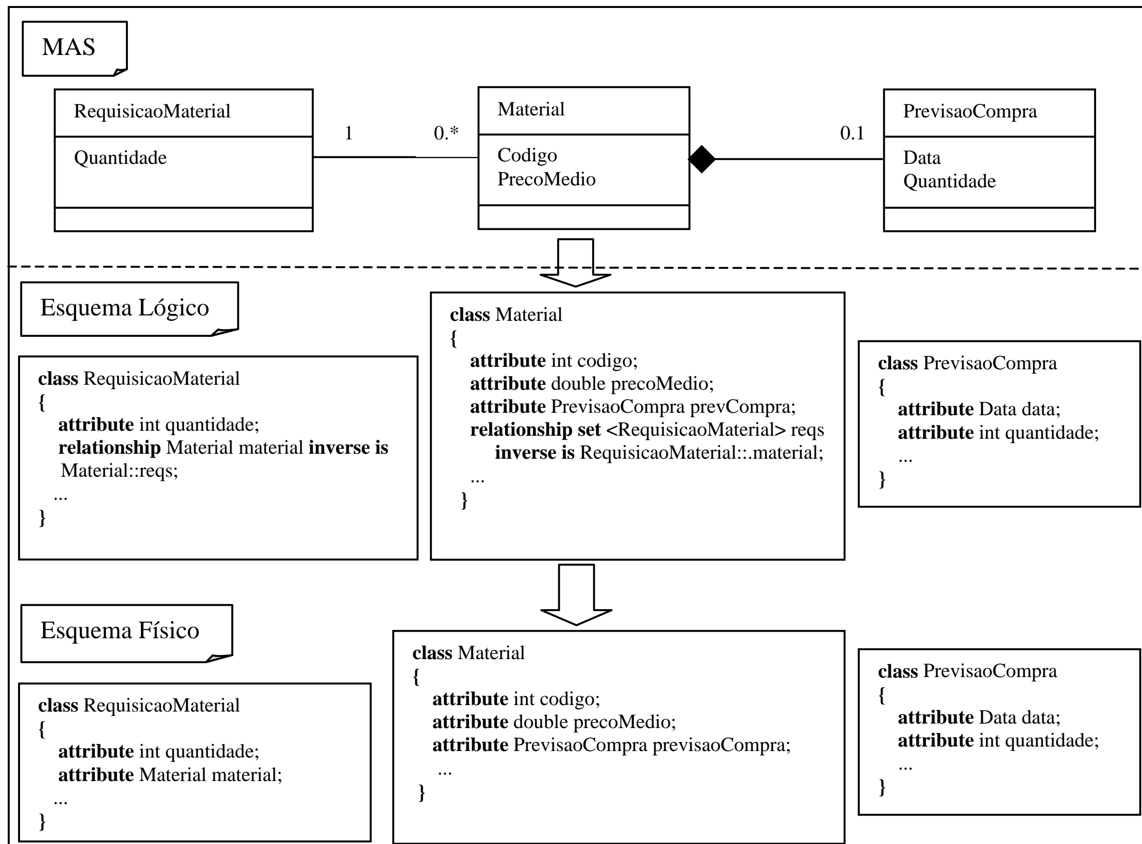


Figura 5.5. Mapeamento lógico e físico de classes em *Jasmine*

FASE 2: DESENVOLVER MODELOS DE PROJETO DO SISTEMA

A especificação de projeto é desenvolvida a partir das classes de domínio da aplicação, considerando a utilização de *servlets* para camada de *middleware* e as classes do *framework* de persistência providas pelo gerenciador de banco de dados. Os *servlets* da camada de *middleware*, utilizados na primeira versão com o padrão *Persistence Layer* e *Sybase*, foram adaptados para uso de *Jasmine*.

A Figura 5.6 mostra parcialmente o modelo de classes de projeto para o sistema exemplo: a camada de aplicação contém as classes inscritas no retângulo (a); as classes previstas para *binding* da aplicação com o banco de dados são as que possuem as palavras *iterator* e *factory* no nome e a camada de *middleware* contém os *servlets*. A Figura 5.7 mostra o diagrama de seqüência para o curso normal do caso de uso `CadastrarPrevisaoCompra`.

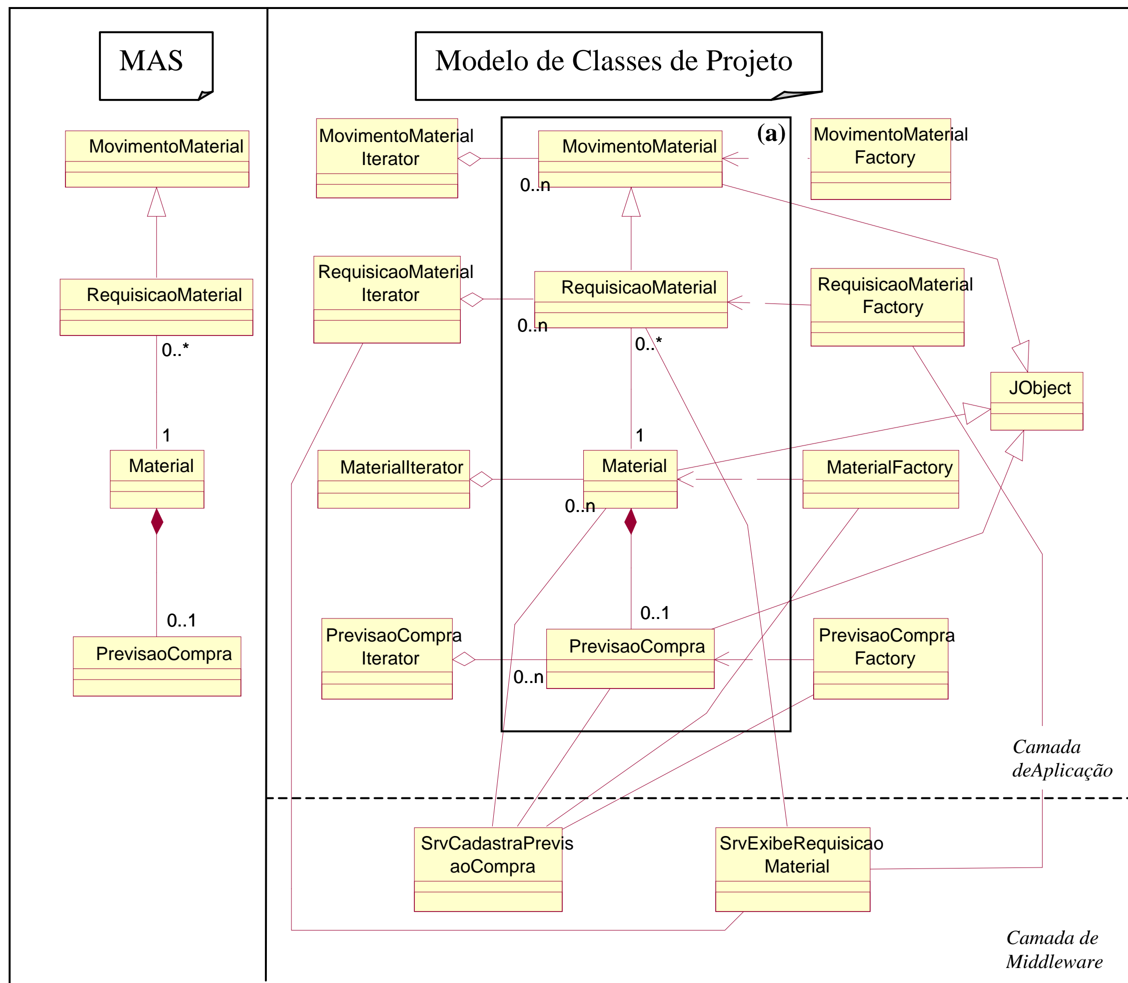


Figura 5.6. Modelo de classes de projeto com uso do SGBD *Jasmine* (visão parcial)

A comunicação entre a aplicação com o banco de dados, nesta segunda versão, ocorre com o uso de um *framework* de suporte à persistência, composto por um conjunto de classes geradas pelo SGBD e a biblioteca de suporte `JavaL2`. Alguns padrões de projeto (GAMMA *et al*, 1995), podem ser identificados em relação ao conjunto de classes que compõe o projeto do sistema e são comentados a seguir.

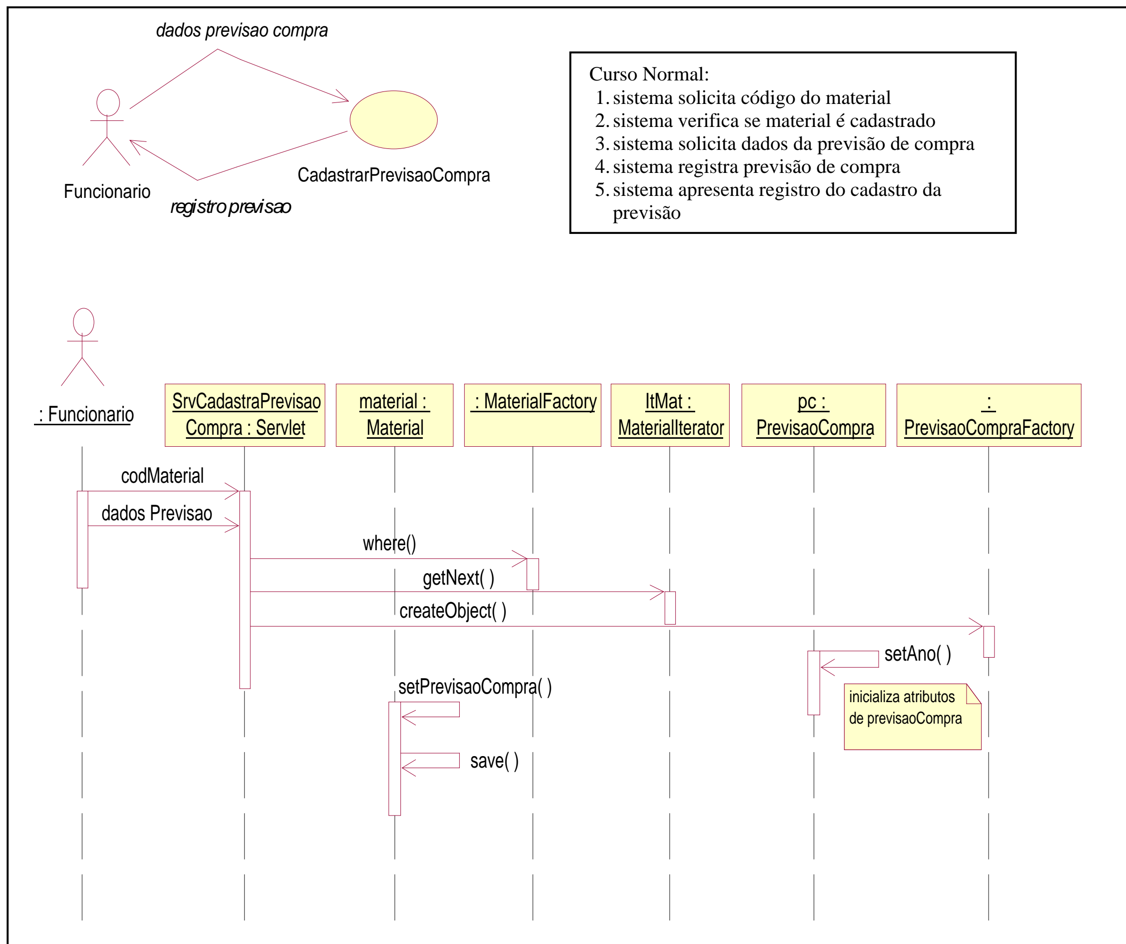


Figura 5.7. Diagrama de seqüência para caso de uso `CadastrarPrevisaoCompra` (2ª. versão do sistema exemplo)

<u>Identificação do padrão *Proxy*</u>: definido pelo uso de um objeto representante de um outro objeto para controlar o acesso ao mesmo. Na utilização do *binding* do SGBD com a linguagem de programação Java, observou-se que todas as operações relacionadas à persistência de objetos (criação, recuperação, atualização e remoção) são definidas e implementadas referenciando um objeto imagem de uma instância no banco de dados. Em *Jasmine*, a persistência faz uso de um objeto *Proxy* (representante), que é uma referência para um objeto da base de dados e toda operação é intermediada por esse objeto representante. A Figura 5.8 exemplifica o uso do padrão *Proxy* para a classe `Compra`. Uma instância de `Compra` é criada pelo método `CreateObject()` da classe `CompraFactory` e a operação `save()` torna o objeto persistente por intermédio de seu *Proxy*.

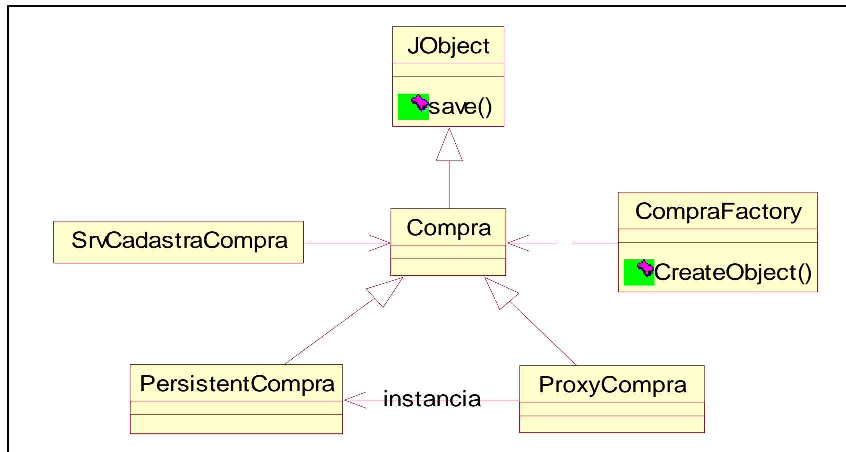


Figura 5.8. Padrão *Proxy* para a classe `Compra`

Identificação do padrão *Factory Method*: o *framework* de persistência em *Jasmine* implementa para cada classe de domínio uma classe *Factory* com um método comum de instanciação `createObject()`, que invoca o construtor da classe correspondente, como é exemplificado na Figura 5.9. As classes *Factories* das classes de aplicação são herdeiras de uma classe genérica `JFactory`, que define o método `createObject()`.

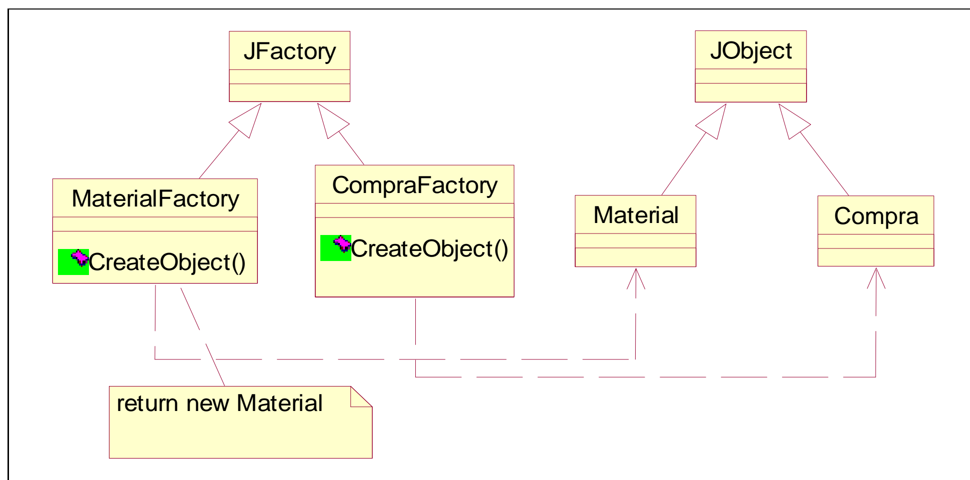


Figura 5.9. Padrão *Factory Method* no modelo de classes de projeto

Identificação do padrão *Iterator*: provê um meio de acesso, seqüencialmente, aos elementos de um objeto agregado, sem expor a sua representação subjacente. Esse padrão fornece uma interface uniforme para percorrer estruturas diferentes (iteração polimórfica).

A biblioteca `JavaL2`, um dos componentes do *binding* com Java, disponibiliza a classe `JIterator`, para percorrer coleções. Para toda classe definida no banco de dados é prevista a geração de uma classe `ClassNameIterator`, herdeira de `JIterator`, que

oferece suporte ao projetista para manipulação de coleções de objetos dessa classe. As interfaces das classes `JIterator` e `ClassNameIterator` foram descritas na Seção 3.4.1. A Figura 5.10 mostra um exemplo de relacionamento entre classes em que foi identificado o padrão *Iterator*.

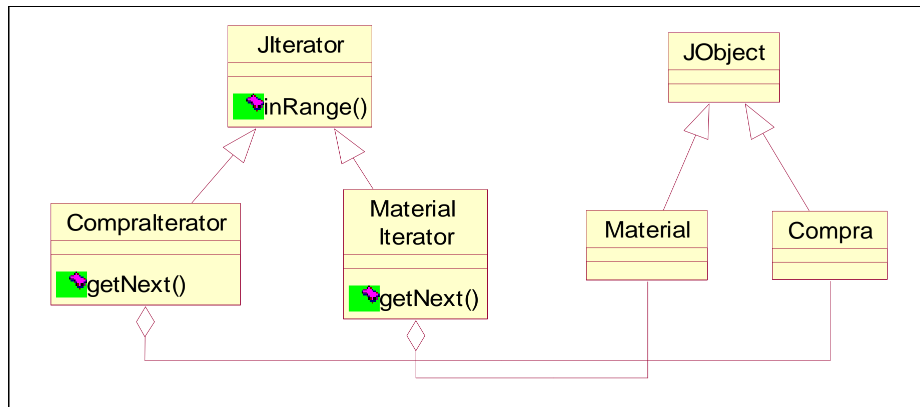


Figura 5.10. Padrão *Iterator* para classe `Compra`

FASE 3: IMPLEMENTAR SISTEMA

Nesta etapa, a partir do esquema de classes do banco de dados orientado a objetos, Figura 5.5, foram criadas as classes no SGBD *Jasmine*, com o uso da ferramenta *Jasmine Studio*. Uma família de classes foi criada no banco de dados e nela foram definidas as classes persistentes da aplicação, registrando-se para cada uma seus atributos e respectivos domínios, seus métodos e os inter-relacionamentos entre as mesmas.

A ferramenta *Jasmine Studio*, utilizada para edição do esquema no banco de dados, é mostrada na Figura 5.11. A janela (a) contém as classes que foram definidas no gerenciador, por exemplo, a relação de herança entre a superclasse `MovimentoMaterial` e as subclasses concretas `RequisicaoMaterial`, `SolicitacaoMaterial` e `ComunicadoRecebimento`. A janela (b) contém os atributos da classe `Material`. As agregações e associações com outras classes podem ser observadas pelos atributos de referência que foram definidos com as classes `Localizacao`, `PrevisaoCompra`, `Fornecedores`, `RequisicaoRessuprimento`, `Conta`, `Estoque`, `MesAtual` e `MovMesesAnt`. A janela (c) exibe a definição do método de instância `chegouMínimo()` da classe `Material`, que retorna um valor lógico resultante da comparação do estoque mínimo com o saldo corrente do material.

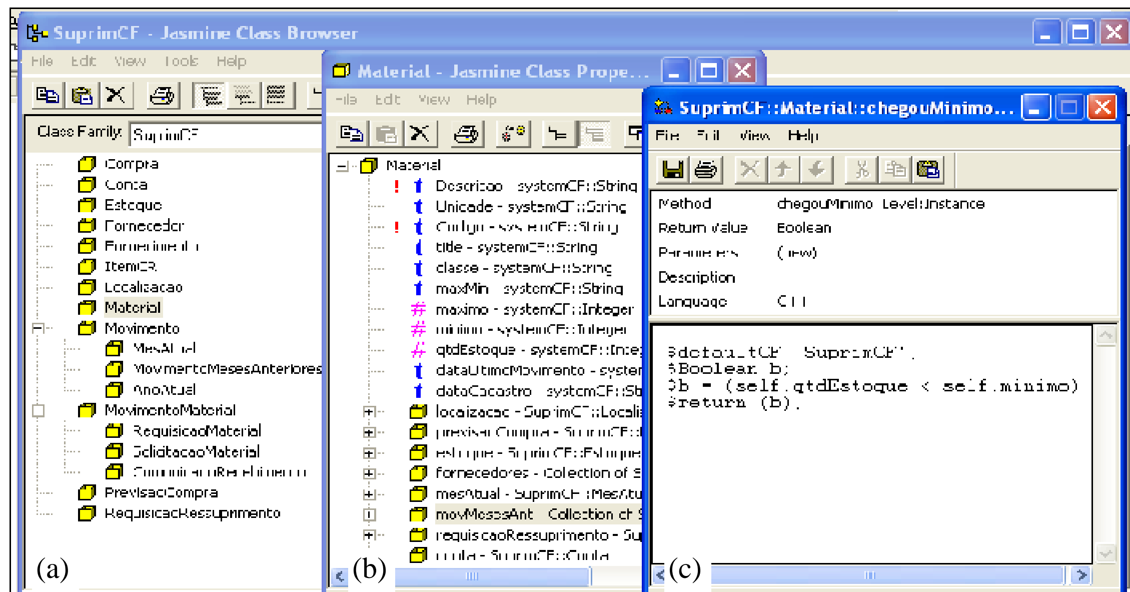


Figura 5.11. Ambiente *Jasmine Studio*

A partir das classes definidas no banco de dados, classes equivalentes foram geradas na linguagem de programação Java, pela ferramenta *Jasmine Browser*, conforme Figura 5.12. Para cada classe definida no banco de dados, seis classes que provêem suporte ao *binding* do banco de dados com a linguagem Java foram criadas. Através dessas classes geradas e da biblioteca de suporte *javaL2* o desenvolvedor pode manipular na linguagem hospedeira os objetos definidos no banco de dados. A funcionalidade de cada classe gerada é a descrita na Tabela 3.1, no Capítulo 3.

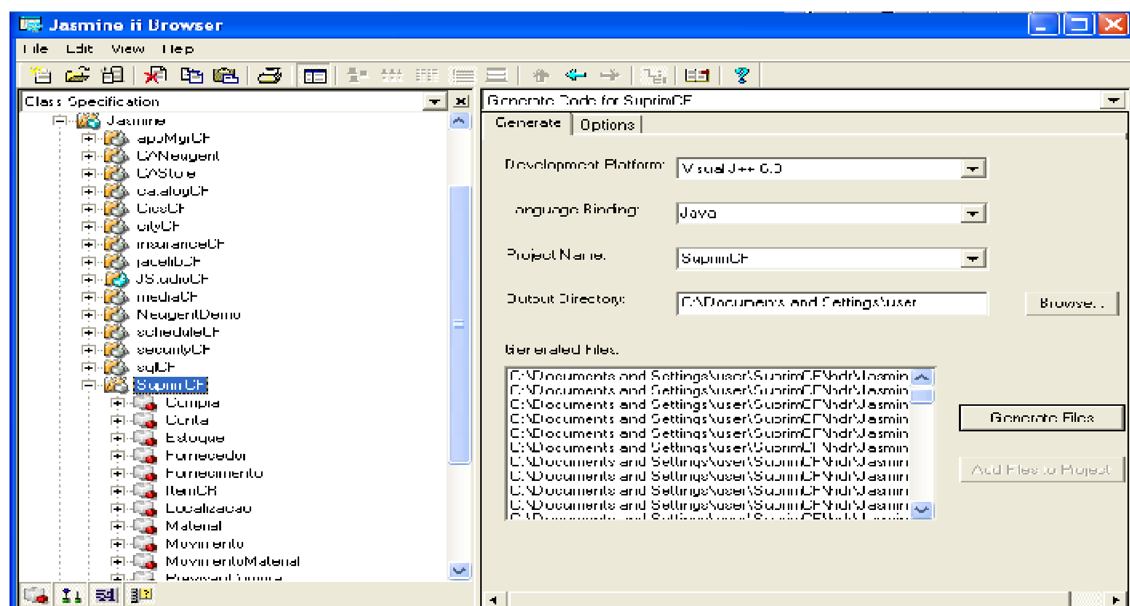


Figura 5.12. Ambiente *Jasmine Browser*

Como o sistema exemplo é para *WEB*, a camada de *middleware*, para comunicar essas interfaces com a aplicação foi desenvolvida utilizando *servlets* da linguagem Java. Os *servlets* correspondem às classes *controllers*, comentadas no Capítulo 4, que tratam os casos de uso do sistema e controlam o fluxo da aplicação.

O servidor *HTTP* utilizado no estudo de caso foi *Jakarta TomCat* (APACHE, 2003). As páginas *HTML* da primeira versão, desenvolvidas por Camargo (2001), foram reutilizadas, alterando-se apenas as *URLs* nas chamadas aos *servlets*. A camada de interface consiste em um conjunto de páginas estáticas, que contêm *links* para outras páginas ou invocam *servlets*. Os *servlets* controlam o fluxo de execução do sistema e criam, dinamicamente, páginas que são exibidas ao usuário. A Figura 5.13 mostra a página *HTML* para a interface `CadastrarMaterial`. Nessa página é definido um formulário que invoca o método `doPost()` do *servlet* `srvCadastrarMaterial`, que atende a essa requisição.

As classes geradas na etapa anterior pela ferramenta *Jasmine Browser*, em conjunto com outras classes que dão suporte ao *binding* da linguagem com o banco de dados, foram utilizadas nos *servlets* para implementar as funcionalidades do sistema, de acordo com o modelo de projeto.

```
<html>
  <head>
   <meta http-equiv="Content-Type" content="text/html; charset=iso-8859-1">
   <meta name="GENERATOR" content="Microsoft FrontPage 4.0">
   <title>EMPRESA</title>
  </head>
  <body>
   <p align="center"><font size="6" face="Architecture"><b><u>Cadastro
   de Materiais</u></b></font> </p>
  <form action=
    "http://localhost:8080/examples/servlet/Suprim.srvCadastraMaterial"
   method="POST">
    <p><b>Estoque</b> <input type="text" size="5" name="Estoque">
    <p><b>Almoxarifado</b>  <input type="text" size="5" name="Almoxarifado"></p>
    ...
    <p>Repartição  <input type="text" size="9" name="Reparticao">  
    <p>Box  ;   <input type="text" size="9" name="Box"> </p>
    <p>Selecione a Opção Desejada 
    <input type="submit" value="Inserir"><input type="reset" value="Limpar"> <br></p>
  </form>
 </body>
</html>
```

Figura 5.13. Formulário HTML `CadastrarMaterial.html` (sistema exemplo)

A Figura 5.14 apresenta o *servlet* `EncontrarMaterial`, que recupera e exibe as informações de um material cadastrado no sistema, a partir de seu código. A parte da Figura rotulada por (a) mostra o método `doGet()` do *servlet*, que é invocado a partir de um

formulário *HTML* da interface. A parte rotulada por (b) atende uma requisição de usuário, inicialmente, estabelecendo a conexão com o banco de dados Na parte (c), o banco de dados é pesquisado para recuperar a informação desejada e, na parte (d) é gerado o texto HTML dinamicamente, para ser exibido no navegador, utilizando os métodos `get_Descricao()` e `get_Unidade()`.

```
…
public class EncontrarMaterial extends HttpServlet
{
// Processando a requisição do usuário.
public void doGet(HttpServletRequest request, HttpServletResponse response)      (a)
  throws IOException, ServletException
    {
    try
      {
       // HTML dinâmico
       response.setContentType("text/html");
       PrintWriter out = new PrintWriter (response.getOutputStream());
      // Inicializa a infraestrutura Jasmine, cria e inicia uma sessão            (b)
       JsessionManager.initialize();
       Jsession session = JsessionManager.connect();
       JsessionManager.begin();
       // Define um objeto do tipo Material
       Material mat1;
         ...
       String query = "Codigo == " + "\"" + codigo + "\"";
       MaterialIterator Itmat = MaterialFactory.where(query,"Root/Jasmine/SuprimCF");   (c)
       while  (Itmat.inRange()) {
          mat1 = Itmat.getNext();
       }
       ...
       // gera HTML dinamicamente
       out.println("<html>");
       out.println("<head><title>Encontrar Material</title></head>");
       out.println("<body>");
       out.println("<H2> Dados do Material </H2>");
       out.print("<BR>") ;
       out.print("<P> Descricao = ");
       out.print(mat1.get_Descricao());                                          (d)
       out.print("<P> Unidade = ");
       out.print(mat1.get_Unidade());
       …
       out.println ("</body>");
       out.println ("</HTML>");
       out.close();
       // Lanca uma excecao em caso de erro
     } catch(Jexception exr) {
...
```

Figura 5.14. Java *Servlet* com conexão *Jasmine*

Testes funcionais foram realizados para verificar a integração entre os *servlets* com as interfaces com o usuário, com as classes da aplicação e com o banco de dados. Os casos de uso foram validados com a implementação desenvolvida. A seguir será comentada a implementação dos padrões de projeto que foram identificados na utilização do *binding* da aplicação com o banco de dados.

A Figura 5.15 mostra o trecho de código correspondente à utilização do padrão *Proxy*. para a criação de um objeto da classe `Compra`. O objeto *`compra`*, declarado na linha 1, é o objeto *Proxy*, criado pelo método da *Factory* correspondente. Nas linhas 2 a 10 pode-se observar a atualizaçãodos atributos do objeto e sua gravação nono banco de dados quando da execução da linha 11, sendo efetivada a transação pelo comando da linha 12. De modo análogo, são utilizados objetos *Proxy* para recuperar e exibir um objeto da base de dados, para modificar o estado de um objeto ou para removê-lo.

```
    ...
1.  Compra compra = CompraFactory.createObject();
2.  compra.set_situacao("P");
3.  compra.set_dataCompra(request.getParameter("Data"));
4.  compra.set_numeroParcelas(numeroParcelas);
5.  compra.set_quantidade(quantidade);
6.  compra.set_valor(valor);
7.  compra.set_valorUltimaCompra(vlUltimaCompra);
8.  compra.set_fornecedorAtual(forn);
9.  compra.set_material(mat);
10. compra.set_nroPedidoCompra(request.getParameter("NumeroPedido"));
11. compra.save();
    …
12. Jasmine.commit();
    ...
```

Figura 5.15. Padrão *Proxy*: exemplo de implementação

A Figura 5.16 apresenta o trecho de código correspondente ao uso do padrão *Iterator*. Para a classe `Compra`, são selecionadas todas as instâncias com situação "`em aberto`". O método `where` da classe `CompraFactory` retorna um objeto `CompraIterator`, através do qual percorre-se a coleção de objetos que atende à condição de pesquisa.

```
...
String query = "Situacao == " + "\"" + em aberto + "\"";
CompraIterator Itcompra = CompraFactory.where(query,"Root/Jasmine/SuprimCF");
while (Itcompra.inRange()) {
  Compra compra = Itconta.getNext();
  System.out.println(compra.get_quantidade());
  System.out.println(compra.get_valor());
  System.out.println(compra.get_material().get_Descricao());
  ...
}
...
```

Figura 5.16. Padrão *Iterator*: exemplo de implementação

O padrão *Memento* é muito utilizado em mecanismos para desfazer operações e permitir ao usuário retroceder ao estado anterior a essas operações. As funções *sets*, definidas para todos os atributos de uma classe, efetivam a valorização das propriedades dos objetos. A análise do código gerado mostra que as funções *sets*, além de atribuir os valores dos aos atributos, registram os valores substituídos em históricos. A execução do método *rollback()*

desfaz a transação, retornando o valor anterior do atributo. A Figura 5.17 mostra, para a classe `Compra`, o código referente à função set para o atributo `fornecedorAtual`. Observa-se, nesse exemplo, que além de alterar o valor do atributo pelo parâmetro passado, também é registrado seu valor anterior.

```
Public void set_fornecedorAtual(Jasmine__SuprimCF.Fornecedor fornecedorAtual) throws
                                  Jexception, PropertyVetoException {
       Jasmine__SuprimCF.Fornecedor old_fornecedorAtual = null;
       Jasmine__SuprimCF.Fornecedor new_fornecedorAtual = (fornecedorAtual);
       try {
            old_fornecedorAtual = (get_fornecedorAtual());
       } catch (JnullPropertyException e) {
       }
       vetos.fireVetoableChange("fornecedorAtual", old_fornecedorAtual,
       new_fornecedorAtual);
       setPropertyValue("fornecedorAtual", new Jvariant(fornecedorAtual));
       changes.firePropertyChange("fornecedorAtual", old_fornecedorAtual,
        new_fornecedorAtual);
    }
```

Figura 5.17. Padrão *Memento*: exemplo de codificação

5.6. ENGENHARIA AVANTE COM USO DO SGBD *CACHÉ*

De modo análogo ao uso do SGBD *Jasmine*, a terceira versão do sistema de Controle de Estoque foi desenvolvida, utilizando o sistema gerenciador de banco de dados *Caché* (INTERSYSTEMS, 2003) e as diretrizes apresentadas no Capítulo 4.

A arquitetura do sistema para a terceira versão é semelhante à da segunda, prevendo a interface com o usuário como páginas HTML, camada de aplicação composta de classes Java, banco de dados objeto-relacional *Caché* e *servlets* na camada intermediária, para comunicar as páginas da interface com as classes de domínio.

FASE 1: PROJETAR BANCO DE DADOS ORIENTADO A OBJETOS

A engenharia avante teve início pela especificação do projeto de banco de dados em ODMG para as classes persistentes da aplicação do MAS. Essa atividade é semelhante à realizada quando da segunda versão do sistema exemplo, uma vez que os dois SGBDs suportam o paradigma orientado a objetos. Assim, o esquema lógico de classes é o mesmo, pois independe do gerenciador utilizado. O projeto físico do banco de dados foi conduzido considerando as características de *Caché* para representação de objetos no banco de dados.

A Figura 5.18 mostra para as classes `MovimentoMaterial`,

`RequisicaoMaterial`, `Material` e `PrevisaoCompra` o mapeamento para os esquemas lógico e físico de classes. A associação entre as classes `Material` e `RequisicaoMaterial` é representada no esquema lógico por um atributo de referência para `Material` na classe `RequisicaoMaterial`, por um atributo de coleção na classe `Material` e por um relacionamento inverso entre esses atributos. No relacionamento de agregação entre `Material` e `PrevisaoCompra` um atributo de referência na classe `Material` é definido.

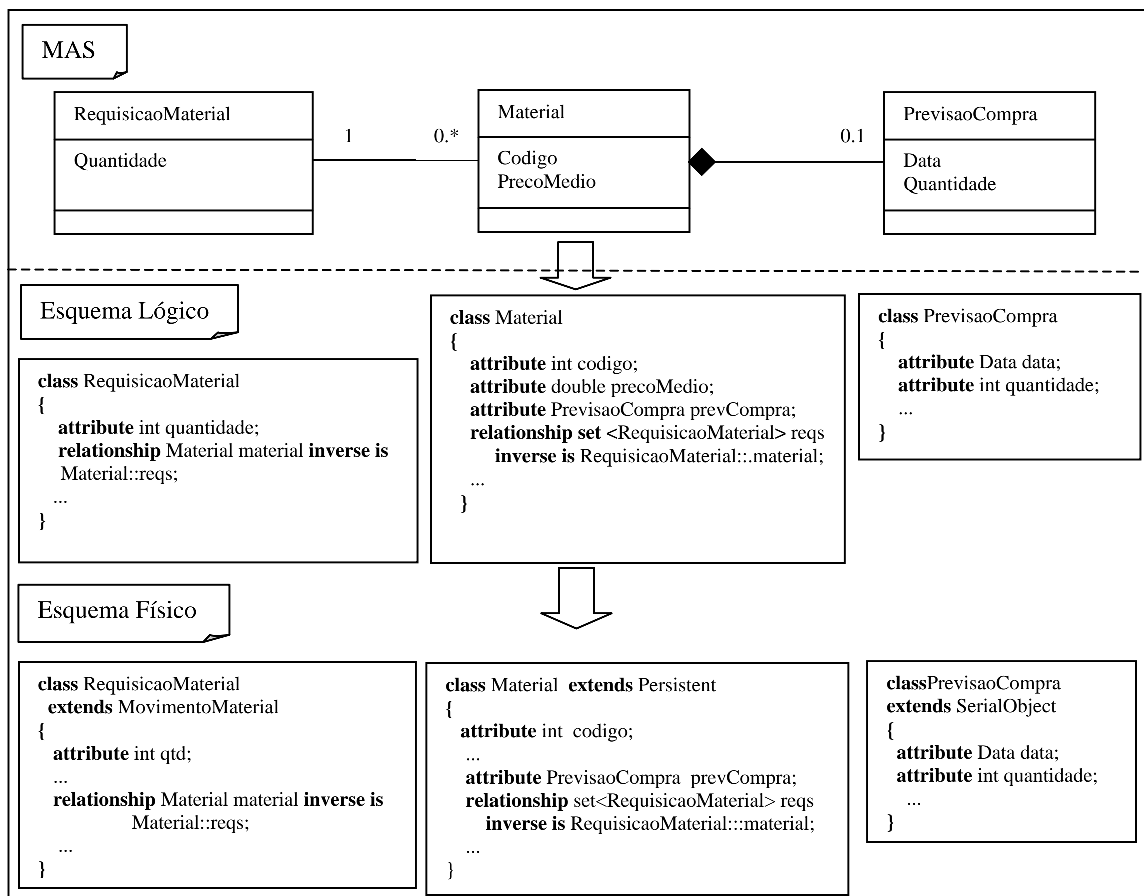


Figura 5.18. Mapeamento lógico e físico de classes para *Caché*

No esquema físico de dados, a associação entre as classes `RequisicaoMaterial` e `Material` é projetada para *Caché* prevendo um relacionamento, definindo-se dois atributos inversos, um em cada classe. Conforme comentado no Capítulo 3, as classes agregadas são definidas em *Caché* como herdeiras de `SerialObject`, o que ocorre com a classe `PrevisaoCompra` para o exemplo da Figura 5.18, em que um objeto dessa classe só será instanciado no banco de dados quando o objeto da classe `Material` for persistido.

FASE 2: DESENVOLVER MODELOS DE PROJETO DO SISTEMA

O modelo de projeto para a terceira versão foi desenvolvido a partir do MAS considerando o *framework* de persistência provido pelo gerenciador. As classes da aplicação foram definidas no modelo de projeto como herdeiras das classes `Persistent` ou `SerialObject`, as quais implementam os métodos para criar, recuperar, salvar ou destruir objetos (métodos do padrão CRUD). As classes `RelationshipObject`, `ListOfObject` e `ArrayOfObject` e `ResultSet` são para suporte às funções do sistema que manipulam coleções e a classe `ObjectFactory` trata os detalhes de conexão da aplicação com o banco de dados. Foi também projetada a camada intermediária, composta de *servlets* para tratar as requisições da interface e que interage com a camada de domínio da aplicação.

A Figura 5.19 apresenta parcialmente o modelo de classes de projeto, as mesmas classes utilizadas na segunda versão. As classes do MAS que correspondem a agregações foram modeladas como herdeiras de `SerialObject`, como por exemplo a classe `PrevisaoCompra`. A classe `Material` herda da classe `Persistent` as operações para salvar e destruir objetos, de modo análogo à segunda versão, em que a classe `JObject` define tais operações. As classes `RelationshipObject` e `ListOfObject` são utilizadas representação de conjuntos e relacionamentos. As classes da camada intermediária, que correspondem aos *servlets*, implementam as funcionalidades da aplicação pela troca de mensagens com a camada de domínio.

A comparação dos modelos de projeto das versões 2 e 3 mostra algumas similaridades: (a) as classes de domínio são herdeiras de classes que implementam operações do padrão CRUD, sendo elas a classes `JObject` para *Jasmine* e `Persistent` para *Caché*; (b) classe *containers* especializadas para manipular agregações de objetos (`Iterator` e `Collection` em *Jasmine* e `ListOfObject`, `ArrayOfObject` e `RelationshipObject` em *Caché*).

Algumas diferenças também podem ser observadas nos modelos de projeto: (a) em *Caché* as agregações são definidas com uso da classe `SerialObject` para objetos da classe Parte; (b) *Jasmine* define uma classe de fábrica específica para cada classe de domínio (classe `Factory`).

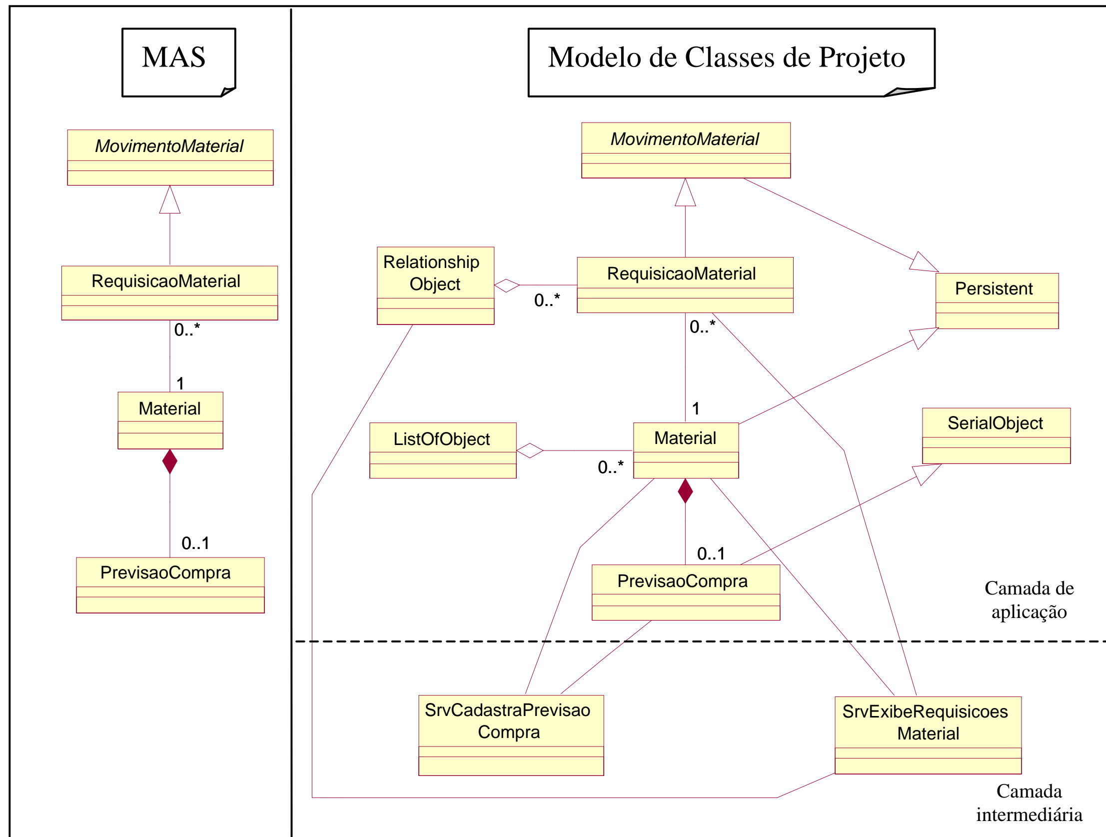


Figura 5.19. Modelo de classes de projeto com uso do SGBD *Caché*

A Figura 5.20 mostra o diagrama de seqüência referente à funcionalidade "Registrar Previsao de Compra do Material", cujo diagrama de caso de uso e a descrição de seu curso normal foram apresentados na Figura 5.7. No exemplo, a operação `save()` aplicada ao objeto `mat` da classe `Material` persiste o objeto agregado da classe `PrevisaoCompra`, por alcançabilidade. Comparando os diagramas de seqüência das duas versões (Figura 5.7 e 5.19) observa-se que as operações de recuperação e armazenamento de objetos envolvem, em *Jasmine*, as classes `Factory` e `Iterator` correspondentes às classes de domínio, enquanto em Caché essas operações são suportadas por herança da superclasse `Persistent`. Deve-se observar, também, que ambos os gerenciadores suportam persistência por alcançabilidade.

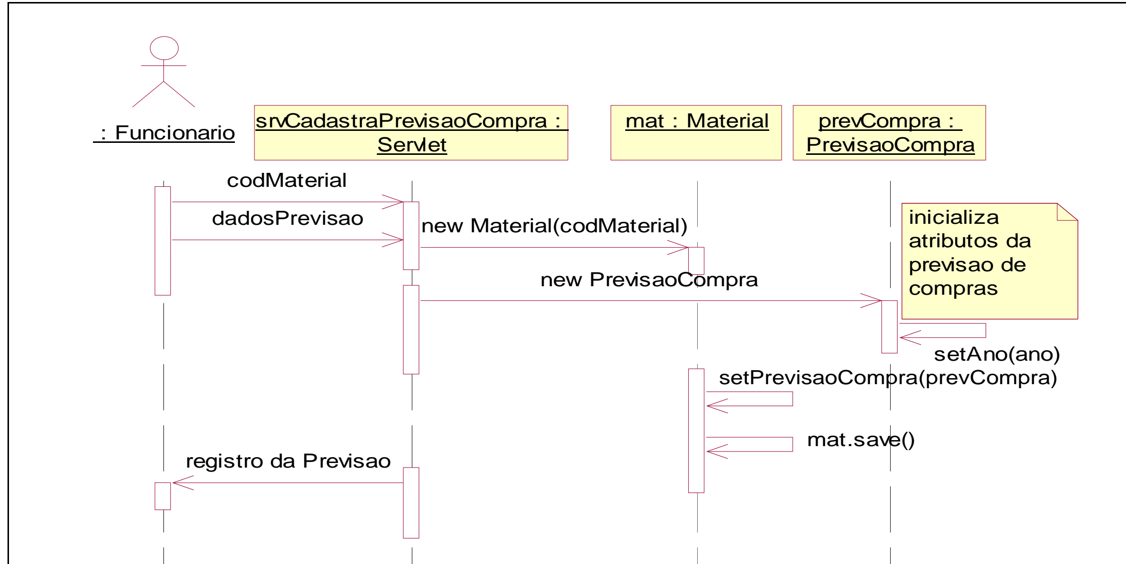


Figura 5.20. Diagrama de Seqüência para caso-de-uso `CadastrarPrevisaoCompra`

Na estrutura dos componentes de projeto para esta versão, também foram identificados alguns padrões de projeto catalogados por Gamma *et al.* (1995), de modo análogo ao da versão 2, que serão descritos a seguir.

Identificação do padrão *Proxy*: Todo objeto persistente é referenciado na aplicação por uma OREF (*Object REFerence*), que corresponde a um *Proxy* do objeto no banco de dados, intermediando todas suas operações. A Tabela 3.4 mostrou as instruções disponibilizadas pelo *binding* de *Caché* com Java para implementar as operações CRUD. A Figura 5.21 exemplifica a uma estrutura no modelo de projeto em que o padrão *Proxy* é utilizado. Uma instância da classe `Material` é um *Proxy* remoto para um objeto correspondente no banco de dados e na execução do método `_save()` o objeto persistido, juntamente com a instância de `Localização`, agregada a ele.

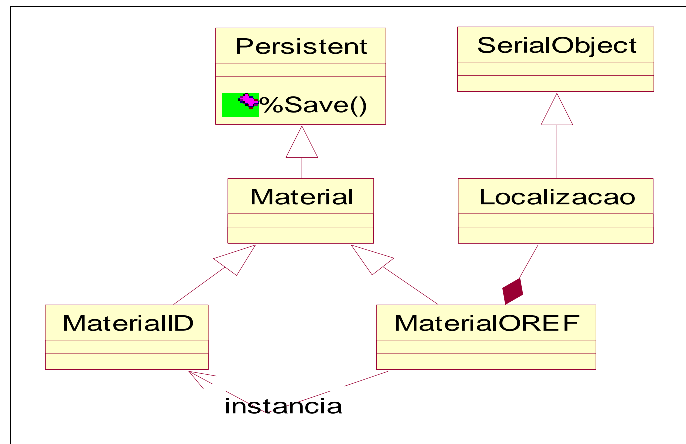


Figura 5.21. Padrão *Proxy* nas classes *Caché*

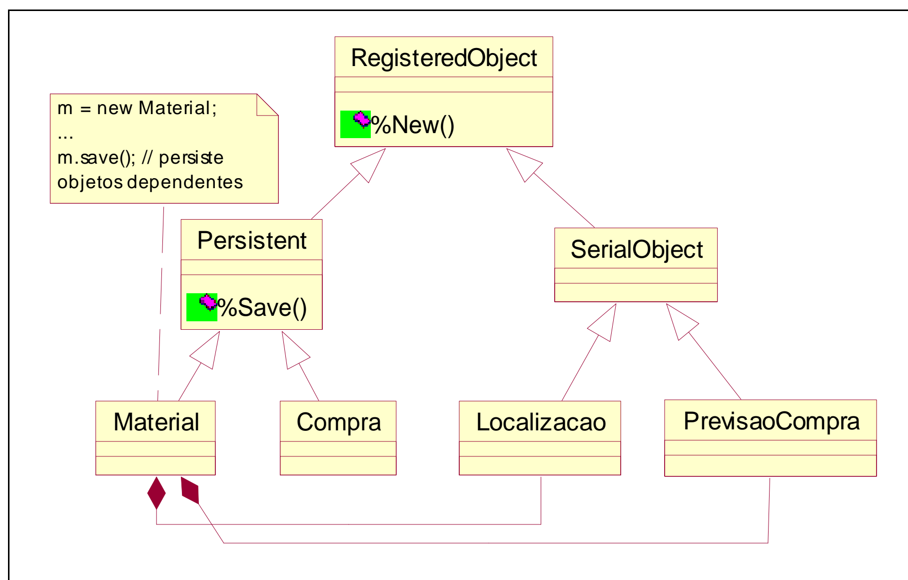


Figura 5.22. Padrão *Method Factory* e *AbstractFactory* para a estrutura de classes em *Caché*

Identificação dos padrões *Factory Method* e *Abstract Factory*: O padrão *Factory Method* define uma interface para criação de objetos, delegando às sub-classes qual objeto instanciar. Na estrutura de classes do SGBD *Caché*, a classe abstrata `RegisteredObject` define a operação para criação de objetos, o método de fábrica. Essa classe é ramificada em outras duas classes abstratas, `Persistent` e `SerialObject`, das quais são herdeiras, respectivamente, as classes persistentes e agregadas da aplicação, Figura 5.22. O padrão *Abstract Factory* está implícito na criação ou recuperação de instâncias que agregam objetos herdeiros da classe `SerialObject` (objetos embutidos), que é realizada por uma única interface.

Identificação do padrão *Iterator*: As classes *containers* RelationshipObject, ArrayOfDataTypes, ArrayOfObjects, ListOfDataTypes e ListOfObjects do pacote cachejava.jar definem métodos para que se tenha acesso e manipuleinstâncias de objetos agregados. Esses métodos permitem que o programador implemente cursores sobre os conjuntos armazenados nessas estruturas, tal como o padrão *Iterator* (GAMMA *et al.*, 1995).

FASE 3: IMPLEMENTAR SISTEMA

Para se definir o esquema de classes do sistema exemplo no gerenciador utiliza-se a ferramenta *Object Architect*. De modo semelhante ao utilizado no ambiente *Jasmine Studio*, essa ferramenta também provê uma interface para criação e administração de classes e objetos no banco de dados. Cada classe criada no banco de dados teve seus métodos escritos na linguagem *ObjectScript*, linguagem de manipulação de dados do gerenciador. A Figura 5.23 mostra o ambiente *Object Architect*, apresentando as classes que foram definidas no banco de dados e detalha a classe Material, com seus atributos. Nessa Figura, pode-se notar que os atributos da classe Material são de tipos primitivos, como String, Integer ou Float ou relacionamentos com outras classes do pacote SuprimCF, em conformidade com o modelo de classes do sistema.

Ao se definir uma classe no banco de dados, o mapeamento para tabelas é realizado automaticamente pelo SGBD. Como exemplo, a Figura 5.24 mostra a tabela Material que acresce todos os atributos de Localizacao (Figura 5.24 linhas 22 a 25 em destaque) e define o atributo conta como chave estrangeira para a tabela Conta (Figura 5.24 linha 5, em destaque).

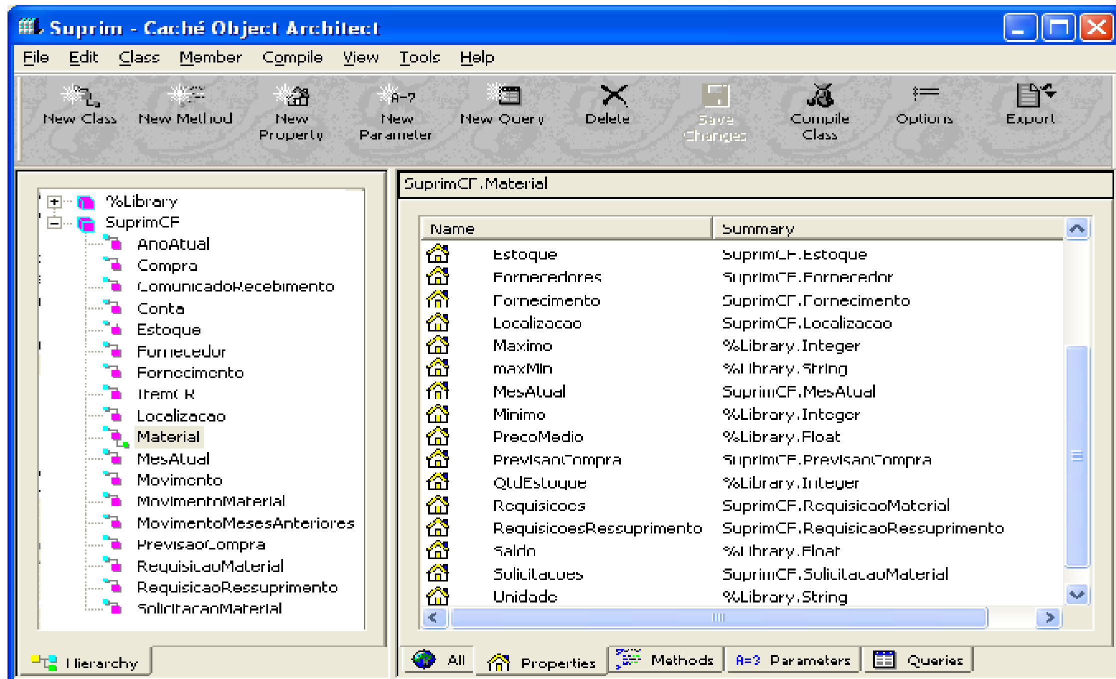


Figura 5.23. Ambiente *Caché Object Architect*

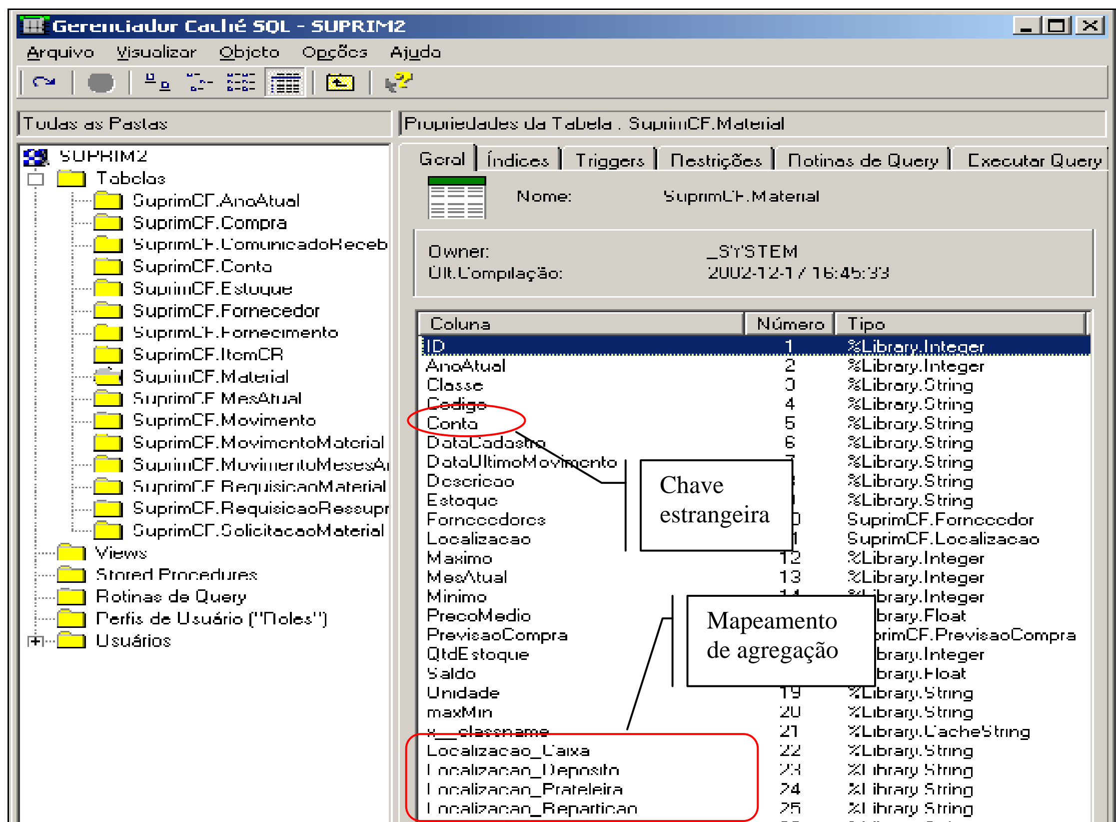


Figura 5.24. Tabela `Material` – visão relacional

A interface da linguagem de programação Java com o banco de dados é provida pela biblioteca de classes `cacheJava.jar` e pelo gerador de código da ferramenta *Object Architect*, o qual cria para cada classe definida no banco de dados uma classe *Proxy* equivalente. Toda a manipulação de um objeto no banco de dados é intermediada através de uma instância dessa classe *Proxy*. Nessa classe estão definidos os construtores, os métodos acessórios *sets* e *gets*, interfaces para métodos e *queries*.

Como o sistema exemplo é para *WEB*, a camada de *middleware*, para comunicar as interfaces com o usuário com a aplicação, foi desenvolvida com uso de bibliotecas Java *servlets,* mas outras tecnologias podem ser utilizadas, como JSP, bibliotecas CGI ou ASP, por exemplo.

Nesta versão do sistema exemplo o servidor HTTP *Jakarta TomCat* (APACHE, 2003), foi utilizado como ocorreu na segunda versão. As páginas HTML, correspondente a interface com o usuário, foram reutilizadas da implementação de Camargo (2001), apenas ajustando os endereços de *URLs* nas chamadas aos *servlets*.

```
...
public class srvCadastraConta extends HttpServlet {

  public void init(ServletConfig config) throws ServletException
  {             // Efetua a conexão com o banco de dadosString        } (b)
    connect="cn_iptcp:127.0.0.1[1972]:USER";
    f=new ObjectFactory(connect);
  ...
  Public void doPost(HttpServletRequest request, HttpServletResponse response) throws  } (a)
  ServletException, IOException              // Processa a requisição do usuário.
  {
    String CodigoS = request.getParameter("Codigo");
    String DescricaoS=request.getParameter("Descricao");
    ...
    try  {
         ...
         Conta conta=new Conta(f);                                    } (c)

         conta.setCodigo(CodigoS);
          conta.setDescricao(DescricaoS);
         conta.setDebitoMes (new Double(DebitoMesR).doubleValue());
         ...
         conta._save();

         out.println("<html>");
         out.println("<head><title> Conta </title></head>");
         out.println("<body>");
         out.println("<P> Código: " + conta.getCodigo () + " </P>");        } (d)
         out.println("<P> Descrição : " + conta.getDescricao () + " </P>");
         out.println("<P> Débito do dia: " + conta.getDebitoDia () + " </P>");
         ...
          conta._close();
       }
       catch (Exception e)
       ...
```

Figura 5.25. Java *Servlet* com conexão *Caché*

A Figura 5.25 apresenta um trecho de código correspondente ao *servlet* `srvCadastraConta`, que trata o caso de uso em que é registrada uma nova conta sintética no sistema. A Figura 5.25(a) mostra o método `doPost` do *servlet*, invocado pela interface. A conexão com o banco de dados é estabelecida na função `init()` do *servlet* - Figura 5.25(b). A criação do objeto, as atribuições dos valores da interface aos atributos e a chamada ao método `save()` para persistir uma instancia no banco de dados ocorre como mostra a Figura 5.25(c) e a parte correspondente ao texto HTML que é gerado dinamicamente, para ser apresentado ao usuário, fazendo uso dos métodos acessórios `getDescricao()`, `getUnidade()` e `getSaldo()` da classe `Conta` é mostrada na Figura 5.25 (d).

Para manipular na aplicação conjuntos de objetos armazenados em coleções, a biblioteca de suporte ao *binding* *`CacheJava.jar`* dispõe ao programador classes *containers* para esse fim. Como exemplo do uso da classe `RelationshipObject`, para manipular relacionamentos, será comentado sobre o caso de uso do sistema em que se deseja consultar as requisições de ressuprimento existentes de um dado material, exibindo sua situação e a quantidade solicitada. A Figura 5.26 mostra trecho do código fonte Java para esse caso de uso. Ao objeto `rro`, da classe `RelationshipObject`, é atribuído o conjunto de todas as referências para requisições de ressuprimento do material `mat` (linha 2). O conjunto é percorrido a partir da chave do primeiro elemento do conjunto (linha 3) pelo uso do método `getObjectNext()` do objeto `rro` (linha 6).

```
        …
1.      Material mat = new Material(f,codMat);
2.      RelationshipObject rro = mat.getRequisicoesRessuprimento();
3.      String key = rro._next("");
4.      StringHolder sh = new StringHolder(key);
5.      for (int i=0;i<rro._count();i++) {
6.        RequisicaoRessuprimento rr = new RequisicaoRessuprimento(f,rro._getObjectNext(sh));
7.        System.out.println(rr.getSituacao()+" "+rr.getQtdRequisitada());
8.      }
        ...
```

Figura 5.26. Relacionamento em *Caché* utilizando uma classe *container*

A funcionalidade apresentada no código fonte da Figura 5.26 também pode ser implementada utilizando-se bibliotecas de manipulação SQL. A Figura 5.27 mostra como a mesma pesquisa é implementada por uma *query* SQL. Neste caso a *query* `pesquisaMaterial`, foi definida no banco de dados na classe `RequisicaoRessuprimento`, para selecionar as tuplas da tabela que satisfazem a condição da pesquisa. No trecho do programa Java, a *query* é executada armazenando no

objeto *ResultSet* `rs` o resultado da pesquisa para um dado parâmetro da interface.

```
        …
1.      ResultSet rs=new ResultSet(f,"SuprimCF.RequisicaoRessuprimento","pesquisaMaterial");
2.      rs.setString(1,codMat);
3.      rs.execute();
4.      while (rs.next()) {
5.      System.out.println(rs.getString(2)+" "+rs.getString(3));
6.      }
        ...
```

Figura 5.27. Relacionamento em *Caché* utilizando *query* SQL

Como o SGBD *Caché* não possui uma linguagem de consulta a objetos, apenas a tabelas, as operações que envolvem a recuperação um ou mais objetos do banco de dados por uma condição, exigem a execução de uma *query* SQL retornando o resultado em um objeto `ResultSet` contendo o *ID* do objeto, para posterior recuperação no banco de dados do objeto através de seu *ID*.

```
    ...
 1. String sit = "em aberto";
 2. ResultSet rs_sit=new
    ResultSet(f,"SuprimCF.RequisicaoRessuprimento","pesquisaSituacao");
 3. rs_sit.setString(1,sit);
 4. rs_sit.execute();
 5. while (rs_sit.next()) {
 6. RequisicaoRessuprimento rr1 = new RequisicaoRessuprimento(f,rs_sit.getString(1));
 7. System.out.println(rr1.getSituacao()+" "+rr1.getQtdRequisitada());
 8. System.out.println(rr1.getMaterial().getDescricao()+"
    "+rr1.getMaterial().getMinimo());
 9. }
    ...
```

Figura 5.28. Recuperação de objetos utilizando *query* SQL em *Caché*

Como exemplo, o caso de uso do sistema que recupera do banco de dados todas as requisições de ressuprimento com situação "`em aberto`" e apresenta a quantidade solicitada, a descrição do material e a quantidade em estoque, pode ser implementado executando-se uma *query* que seleciona para a classe `RequisicaoRessuprimento` todas as instâncias que satisfazem a condição e retorna em um objeto *ResultSet* os *ID*s dos objetos dessa pesquisa. Cada objeto é recuperado a partir de seu *ID* e as informações requeridas no caso de uso apresentadas, conforme o código da Figura 5.28. Nessa Figura, na linha 2 é selecionada a *query* a ser executada, o parâmetro da pesquisa é inicializado na linha 3 e o laço entre as linhas 5 e 9 executa a recuperação na base de dados de todos os objetos que atende à condição. O objeto *ResultSet* foi utilizado como suporte à operação, armazenando os IDs de objetos que foram acessados.

Para exemplificar o uso de coleções no estudo de caso, foi definido atributo `Fornecedores`, do tipo `listOfObject`, para a classe cada `Material`. Quando um material é cadastrado, são registrados na interface com o usuário os códigos de três fornecedores. O *servlet* `srvCadastraMaterial` persiste esses dados na lista, conforme Figura 5.29.

```
...
Fornecedor forn1 = new Fornecedor(f,new Integer(request.getParameter ("Forn1")).intValue());
Fornecedor forn2 = new Fornecedor(f,new Integer(request.getParameter ("Forn2")).intValue());
Fornecedor forn3 = new Fornecedor(f,new Integer(request.getParameter ("Forn3")).intValue());
ListOfObjects listaFornecedores = material.getFornecedores();
listaFornecedores._insert(forn1);
listaFornecedores._insert(forn2);
listaFornecedores._insert(forn3);
material.setFornecedores(listaFornecedores);
...
material._save();
...
```

Figura 5.29. Persistência de coleções em *Caché*

A Figura 5.30 mostra como uma determinada coleção do tipo lista pode ser percorrida, em que o método `getObjectAt()` é utilizado para recuperar um elemento da coleção pela sua posição.

```
...
material=new Material(f,cod);
ListOfObjects listaFornecedores = material.getFornecedores();
int num = listaFornecedores._count();
for (int i=1;i<=num;i++) {
    Fornecedor forn = new Fornecedor(f,listaFornecedores._getObjectAt(i));
    System.out.println(forn.getDescricao()+" "+forn.getCodigo());
}
...
```

Figura 5.30. Percorrimento de listas em *Caché*

5.7. ANÁLISE COMPARATIVA ENTRE AS VERSÕES DO SISTEMA EXEMPLO

Esta seção apresenta uma análise comparativa das duas versões desenvolvidas neste trabalho em relação à primeira, Camargo (2001), quanto aos seguintes tópicos: a) arquitetura, b) projeto de banco de dados, c) projeto do sistema, d) implementação, e) persistência por alcançabilidade e f) persistência de objetos complexos.

A) ARQUITETURA

A diferença arquitetural das versões 2 e 3 em relação à versão 1, refere-se à camada de persistência, em que as classes do padrão *Persistence Layer* foram "substituídas" pelas

bibliotecas do *binding* da aplicação com o banco de dados, conforme mostra a Figura 5.31, sendo que as demais partes permanecem inalteradas.

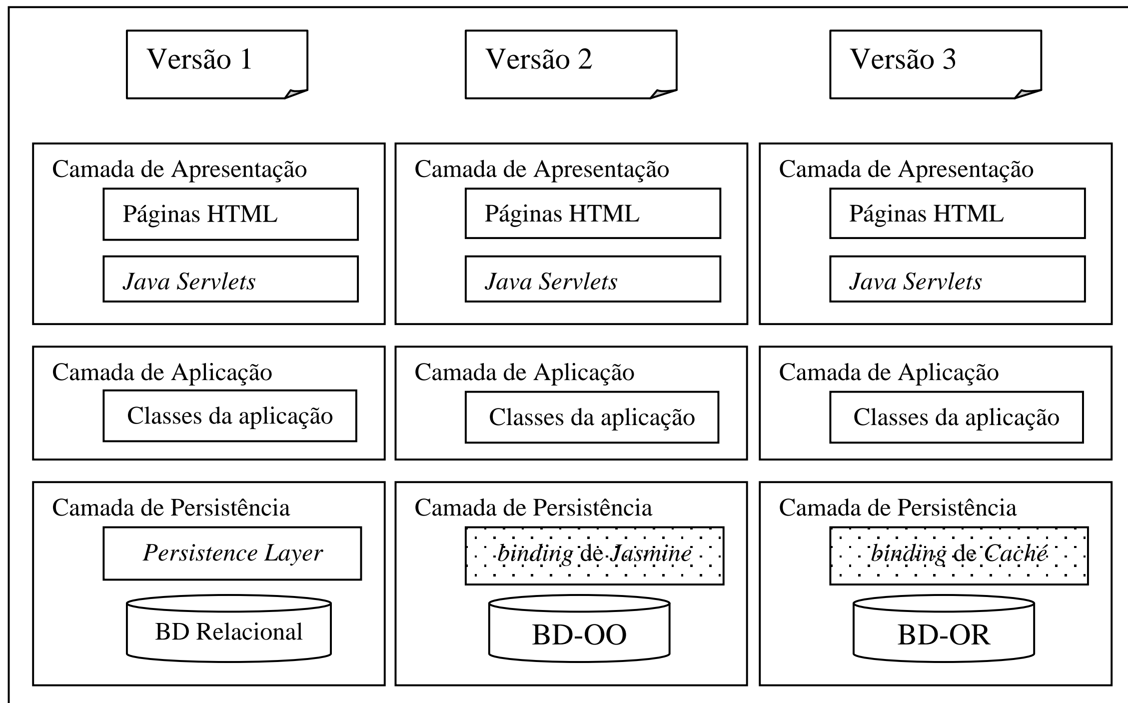


Figura 5.31. Visão Arquitetural das três versões do sistema exemplo

B) PROJETO DE BANCO DE DADOS

Na versão 1 do sistema exemplo, desenvolvido por Camargo (2001), o projeto lógico de banco de dados consistiu no mapeamento das classes de domínio que deveriam ser persistidas para uma estrutura de relações, o esquema relacional. Para cada classe do MAS foi criada uma tabela, para cada atributo uma coluna, para relacionamentos de associação ou agregação foram definidas chaves estrangeiras e para relacionamentos de especialização foram criadas tabelas somente para as sub-classes, adicionando a elas os atributos das superclasses.

Nas versões 2 e 3, o projeto lógico de banco de dados consistiu em transladar as classes persistentes do modelo de análise para um esquema lógico de classes no banco de dados. Para cada classe do MAS foram especificadas as classes no banco de dados, com os atributos simples, os atributos de coleção, os relacionamentos de herança e os de associação, através dos atributos de referência e relacionamentos inversos.

A Figura 5.32 exemplifica a modelagem de dados para os paradigmas relacional e orientado a objetos. Ao se mapear as estruturas de classes para estruturas de tabelas há perda semântica de informações do modelo orientado a objetos, ou seja perde-se a visão de hierarquia de herança por não ser intuitivo o relacionamento entre subtipos e supertipos na estrutura das relações, as associações e agregações são modeladas como repetição de atributos em tabelas distintas e o modelo relacional não dá suporte à representação de conjuntos. Ao se manter o paradigma da aplicação no banco de dados, a semântica é preservada e a base de dados tem melhor legibilidade. Por outro lado, técnicas para obtenção de esquemas relacionais a partir de esquemas conceituais são exaustivamente encontradas na literatura e há maior suporte ferramental, enquanto que para o mapeamento lógico de classes apenas a proposta da ODMG tem razoável consenso na comunidade de banco de dados.

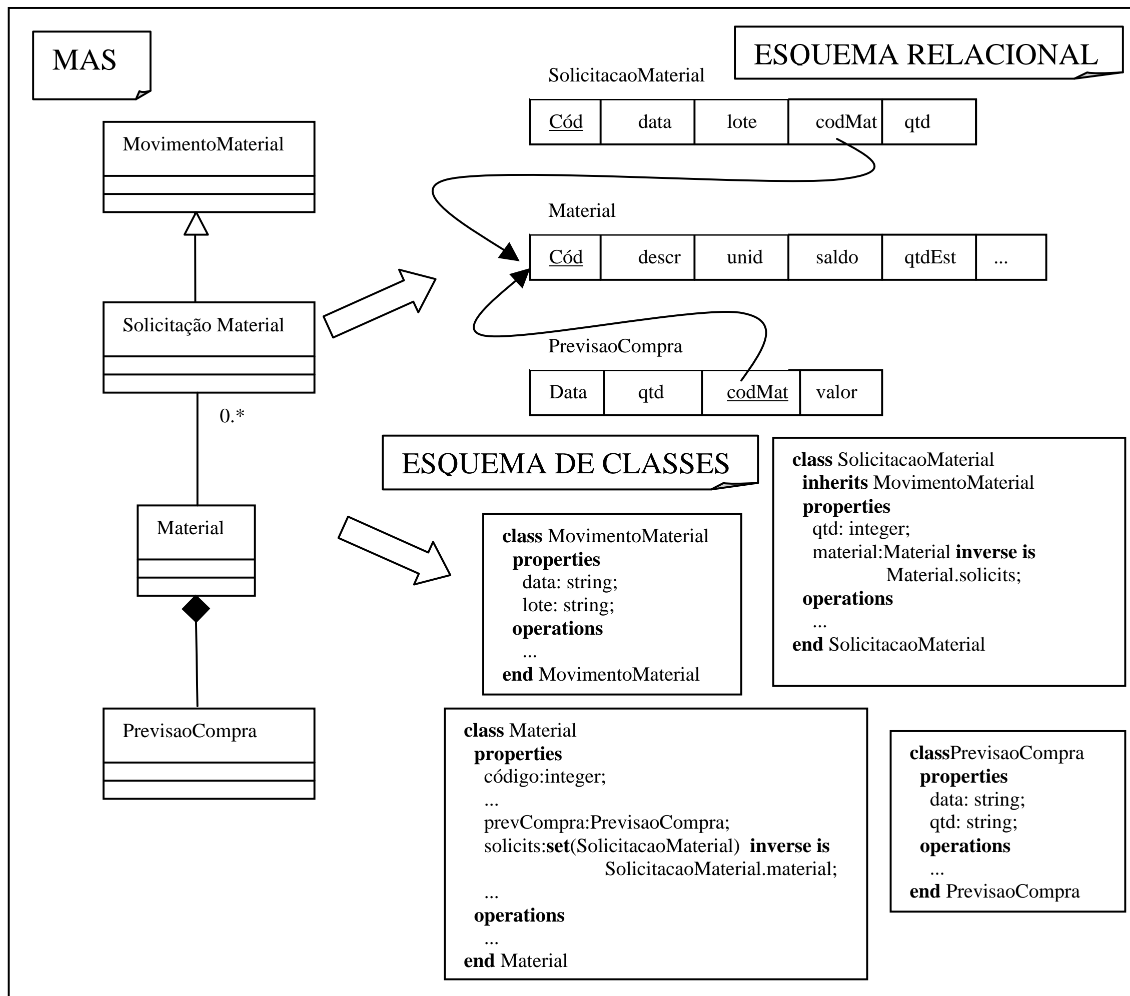


Figura 5.32. Mapeamento conceitual-lógico no projeto de banco de dados para os paradigmas relacional e orientado a objetos.

C) MODELO DE PROJETO DO SISTEMA

Nas versões 2 e 3 a engenharia avante iniciou-se pela modelagem do banco de dados. Foi observado que o projeto físico do banco de dados para cada SGBD adotado define a infra-estrutura para dar suporte à persistência que é utilizada no projeto da aplicação. Na versão 1 a engenharia avante foi iniciada pela especificação das classes de projeto do sistema, incluindo as classes do padrão *Persistence Layer* (YODER, JOHNSON; WILSON, 1998). Esse padrão foi reutilizado de implementaçõ es anteriores (CAGNIN, 1999).

Na primeira versão, observou-se, também, que para o mesmo modelo de projeto poder-se-ia utilizar diversos SGBDs relacionais comercialmente disponíveis, inclusive os de uso livre, por haver uma forte padronização nesses produtos, enquanto que nas versões 2 e 3 foi observado que os componentes de projeto da camada de persistência são completamente associados ao SGBD utilizado. A portabilidade de banco de dados para o paradigma relacional é simplificada quando comparada ao modelo orientado a objetos, mais dependente do SGBD utilizado. Por outro lado, foi observado que os componentes para suporte à persistência e para suporte funcional, que compõem o *binding* do banco de dados com a aplicação, são estruturados baseados em padrões de projeto, que agrega os benefícios associados ao uso de padrões.

A Figura 5.33 indica o reuso dos componentes de projeto para cada uma das versões.

D) IMPLEMENTAÇÃO

No que se refere à implementação do sistema, os seguintes pontos devem ser ressaltados, para cada uma das camadas de arquitetura:

Implementação da interface com o usuário: O código relativo a essa camada é invariante em relação às versões analisadas. Tanto as páginas HTML estáticas quanto às dinâmicas têm conteúdo e volume semelhantes, tendo sido inclusive reutilizadas de uma versão para outra.

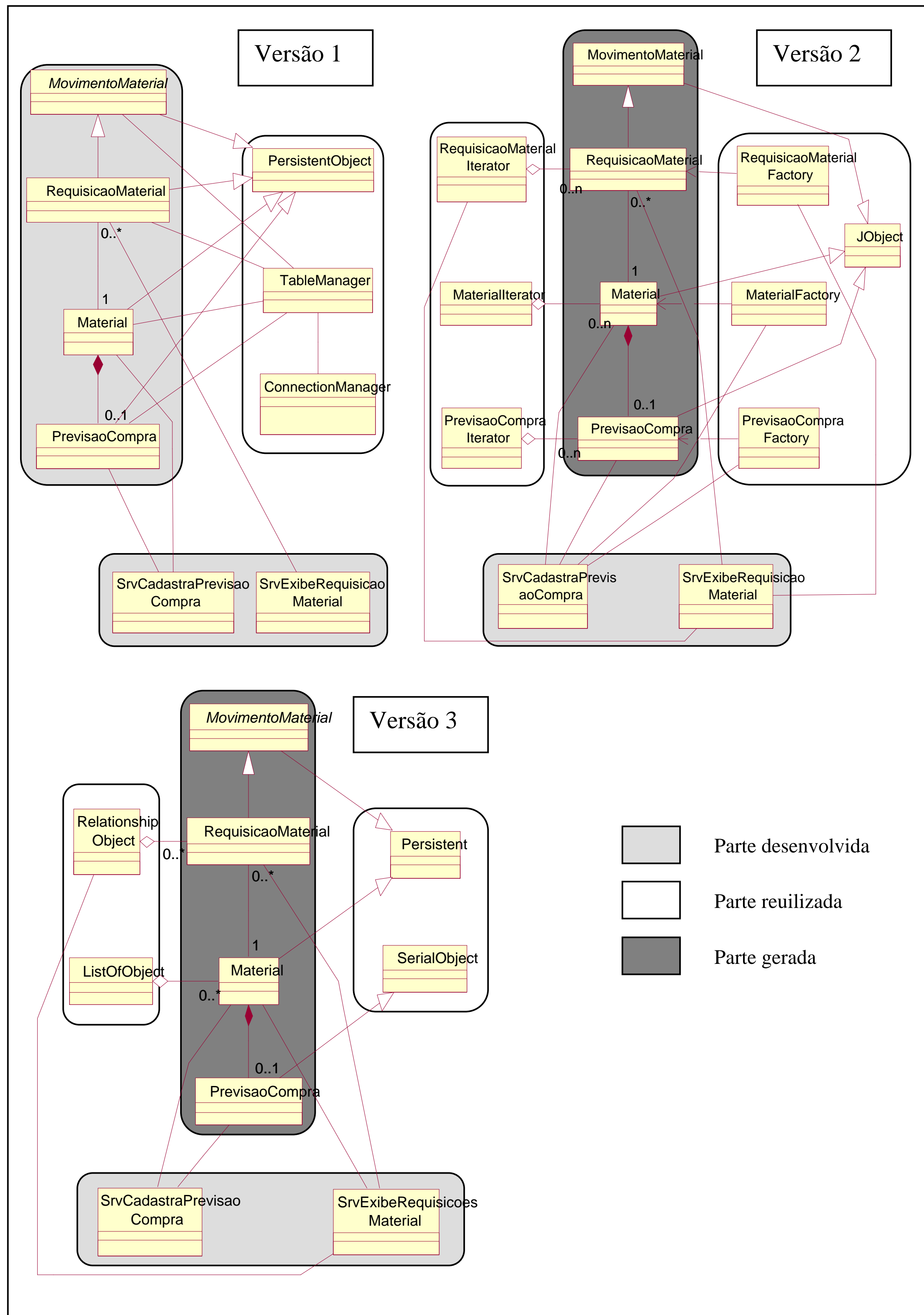


Figura 5.33. Reutilização nas três versões do sistema exemplo

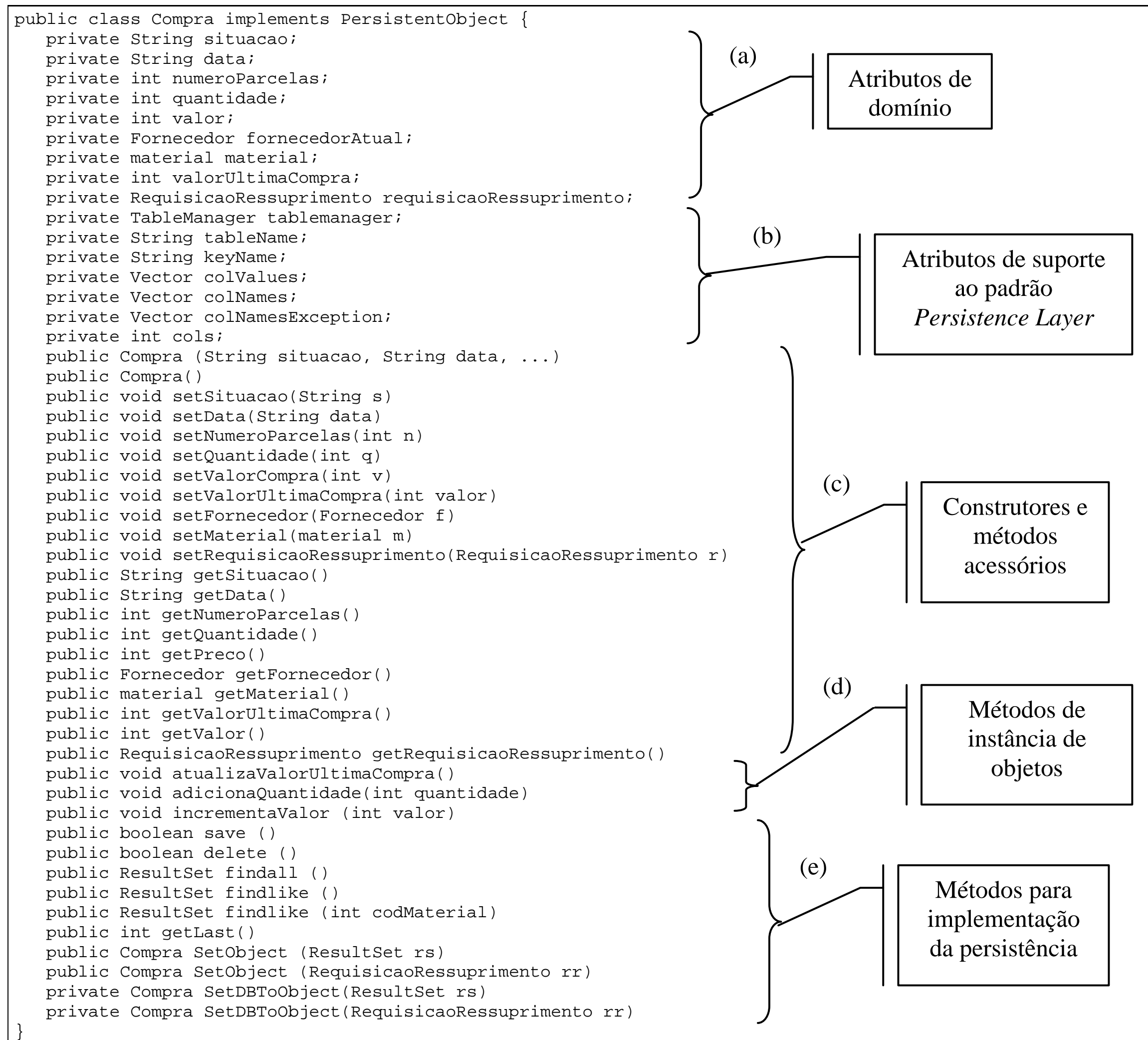
Implementação da camada intermediária: A camada intermediária é composta por *servlets* que comunicam a camada de interface com o usuário com a camada de domínio da aplicação e que coordena o fluxo de execução do sistema. Os *servlets* foram reescritos nas versões 2 e 3 em relação à versão 1, para acomodar a interface com as classes de aplicação e com as classes de suporte à persistência peculiares em cada versão. Não houve variação significativa de volume de código entre as versões.

Implementação das classes da aplicação: O código-fonte referente às classes persistentes de aplicação da primeira versão ou tornou-se desnecessário (código referente à implementação do padrão *Persistence Layer*) ou foi substituído pelo código referente às classes geradas ou foi definido como métodos no banco de dados.

Como exemplo, a Figura 5.34 mostra os atributos e protótipos das funções membros da classe `Compra` da versão 1. Os atributos de domínio indicados por (a) foram definidos na classe `Compra` do banco de dados os construtores e métodos acessórios (c) constam na classe `Compra` gerada nas versões 2 e 3. As partes indicadas por (b) e (e), que correspondem aos atributos de suporte ao padrão *Persistence Layer*, deixam de existir nas versões 2 e 3. Os métodos de instância (d) foram implementados como métodos no banco de dados ou por métodos acessórios.

Embora a geração de código seja característica particular dos SGBDs *Jasmine* e *Caché*, houve uma significativa redução do volume de código desenvolvido nas versões 2 e 3 em relação à versão 1.

É importante ressaltar também que para o código correspondente às classes desenvolvidas na versão 1, observou-se que entre 17 e 57% refere-se ao tratamento do mapeamento objeto-relações, fato que não ocorre quando se utiliza um banco de dados orientado a objetos.



```
public class Compra implements PersistentObject {
   private String situacao;
   private String data;
   private int numeroParcelas;
   private int quantidade;
   private int valor;
   private Fornecedor fornecedorAtual;
   private material material;
   private int valorUltimaCompra;
   private RequisicaoRessuprimento requisicaoRessuprimento;
   private TableManager tablemanager;
   private String tableName;
   private String keyName;
   private Vector colValues;
   private Vector colNames;
   private Vector colNamesException;
   private int cols;
   public Compra (String situacao, String data, ...)
   public Compra()
   public void setSituacao(String s)
   public void setData(String data)
   public void setNumeroParcelas(int n)
   public void setQuantidade(int q)
   public void setValorCompra(int v)
   public void setValorUltimaCompra(int valor)
   public void setFornecedor(Fornecedor f)
   public void setMaterial(material m)
   public void setRequisicaoRessuprimento(RequisicaoRessuprimento r)
   public String getSituacao()
   public String getData()
   public int getNumeroParcelas()
   public int getQuantidade()
   public int getPreco()
   public Fornecedor getFornecedor()
   public material getMaterial()
   public int getValorUltimaCompra()
   public int getValor()
   public RequisicaoRessuprimento getRequisicaoRessuprimento()
   public void atualizaValorUltimaCompra()
   public void adicionaQuantidade(int quantidade)
   public void incrementaValor (int valor)
   public boolean save ()
   public boolean delete ()
   public ResultSet findall ()
   public ResultSet findlike ()
   public ResultSet findlike (int codMaterial)
   public int getLast()
   public Compra SetObject (ResultSet rs)
   public Compra SetObject (RequisicaoRessuprimento rr)
   private Compra SetDBToObject(ResultSet rs)
   private Compra SetDBToObject(RequisicaoRessuprimento rr)
}
```


Figura 5.34. Atributos e funções membros da classe `Compra` (Versão 1)

E) PERSISTÊNCIA POR ALCANÇABILIDADE

Os SGBDs utilizados nas versões 2 e 3 suportam persistência por alcançabilidade, o que reduz o número de linhas de código da aplicação. Isso pode ser observado na Figura 5.35 quando se utiliza esse recurso para a implementação da versão 2 e na Figura 5.36 para a implementação da versão 1.

A persistência por alcançabilidade não é suportada por todos os SGBDs orientados a objetos ou objeto-relacionais, mas está prevista na especificação do *binding* das linguagens de programação Java e *Smalltalk* com bases de objetos no padrão ODMG 3.0 (CATTEL *et al.*, 2000). É um recurso especialmente importante para sistemas em que ocorre um alto grau de interação entre objetos, pois evita que o projetista tenha que prever o controle de atualizações

em objetos complexos. Para sistemas com estruturas simples de objetos, com pouca interação, esse recurso não oferece ganhos significativos.

```
...
Material2.get_localizacao().set_deposito(request.getParameter ("Deposito"));
Material2.get_localizacao().set_prateleira(request.getParameter "Prateleira"));
Material2.get_localizacao().set_reparticao(request.getParameter (Reparticao"));
Material2.get_localizacao().set_caixa(request.getParameter ("Box"));
...
```

Figura 5.35. Implementação utilizando alcançabilidade (versão 2 do sistema exemplo)

```
...
Localizacao localizacao = new Localizacao();
localizacao = new Localizacao(codigoI, request.getParameter ("Deposito"),
                              request.getParameter ("Prateleira"),
                              request.getParameter ("Reparticao"),
                              request.getParameter ("Box"));                (a)
...
localizacao.save ();
Material2.new Material(código,...,localizacao,...);
Material2.save();
...
public boolean save () {
ResultSet rs;
Vector clause = new Vector();  //vetor de cláusulas
Vector parameter = new Vector(); // vetor de parâmetros
Try {
 clause.addElement("codigo = ");  //insere cláusula
 Integer codigo = new Integer(this.codigo);
 parameter.addElement(codigo);  //insere parâmetro
 rs = tablemanager.findlikeDB(this.tableName, clause, parameter);
 colValues.setElementAt(codigo, 0);
 colValues.setElementAt (this.deposito, 1);                                 (b)
 colValues.setElementAt (this.prateleira, 2);
 colValues.setElementAt (this.reparticao, 3);
 colValues.setElementAt (this.box, 4);
 //verifica se o objeto será atualizado ou inserido no BD.
 if (rs.next())
   return tablemanager.updateDB(tableName, colNames, colValues, cols,
                                clause, parameter,colNamesException);
 else
   return tablemanager.insertDB(tableName, colNames, colValues, cols);
 }
 ...
```

Figura 5.36. Implementação na versão 1 do sistema exemplo

F) PERSISTÊNCIA DE OBJETOS COMPLEXOS

Objetos complexos são aqueles que, em sua definição, fazem referências a outros objetos a ele relacionados.

Na versão 1 do sistema exemplo foi observado que na recuperação de objetos complexos ocorre o “cascateamento” de leitura em tabelas do banco de dados, uma vez que os dados do objeto complexo residem em várias tuplas de tabelas distintas e sua recuperação implica no acesso às mesmas. A implementação do padrão *Persistence Layer* utilizada trata a

navegação entre as várias tabelas relacionadas e os métodos `setObject()` e `setDBToObject()`, das classes de aplicação, implementam a montagem da hierarquia de composição do objeto complexo.

Nas versões 2 e 3 do sistema exemplo, por não haver o *gap* semântico entre o SGBD a linguagem de programação orientada a objetos, a recuperação e manipulação de objetos complexos na aplicação são transparentes: quando o objeto é recuperado do banco de dados, as referências e acessos a outros objetos associados a ele já estão presentes. Essa facilidade traz benefícios ao engenheiro de software quando da implementação do mapeamento.

5.8. CONSIDERAÇÕES FINAIS

O processo proposto no Capítulo 4 para a etapa de engenharia avante foi desenvolvido em um sistema que havia sido submetido à reengenharia orientada a objetos (CAMARGO, 2001). A Tabela 5.1 sintetiza as diferenças essenciais entre a versão 1, desenvolvida por Camargo e que utilizou banco de dados relacional, e as versões 2 e 3 desenvolvidas neste trabalho e que utilizaram bancos de dados com suporte a objetos.

TABELA 5.1. Fases do processo de engenharia avante

Fase	Versão 1	Versões 2 e 3
Projeto e implementação do banco de dados	Mapeamento relacional (chaves primárias e estrangeiras, regras de integridade referencial); Definição e criação de tabelas na linguagem de definição de dados de SQL.	Mapeamento de classes (heranças, relacionamentos e atributos de referência); Classes definidas no SGBD e operações de persistência incorporadas no *Binding*.
Projeto da aplicação	Utilização do padrão de projeto *Persistence Layer* para mapeamento objeto-relacional.	*Framework* de suporte funcional e de suporte à persistência baseado em padrões.
Implementação da aplicação	Classes da aplicação implementam operações do padrão CRUD e Reuso dos componentes `ConnectionManager` e `TableManager` (encapsulamento de cláusulas SQL).	Representação unificada de classes, no banco de dados e na aplicação; geração de código.

No estudo de caso observou-se que a utilização de SGBDs com suporte a objetos trouxe vantagens ao engenheiro de software, por eles oferecerem uma semântica mais rica para tratar objetos persistentes; o volume de código desenvolvido foi menor em comparação à versão que utilizou banco de dados relacional e os *frameworks* dos SGBDs usados, baseados em padrões, uniformizam o tratamento da persistência. Essas vantagens sugerem maior produtividade no processo de engenharia avante e maior manutenibilidade do sistema após a

reengenharia.

Relata-se também neste trabalho algumas “armadilhas” que podem surpreender o projetista desatento ou com pouca vivência em sistemas orientados a objetos quando da utilização de bancos de dados orientados a objetos e objeto-relacionais:

- projetar e implementar métodos específicos para tratar a persistência de objetos, ou seja, definir métodos para implementar as operações do padrão CRUD. Esses métodos já existem nos SGBDs orientados a objetos e objeto-relacionais;
- Fazer analogias entre métodos e *stored-procedures* utilizadas para associar código e dados em bancos de dados relacionais;
- Projetar as classes para o sistema alvo com base nas estruturas de tabelas do sistema legado.

No capítulo seguinte serão apresentadas as conclusões deste trabalho.

Capítulo 6

6. CONCLUSÃO

Este capítulo apresenta na Seção 6.1 um resumo dos principais resultados alcançados e na seção 6.2 os trabalhos futuros que podem ser realizados.

6.1. PRINCIPAIS RESULTADOS

Este trabalho apresentou e discutiu o uso de SGBDs orientados a objetos e objeto-relacionais na fase de engenharia avante de um processo de reengenharia orientada a objetos. Um processo para apoiar a engenharia avante foi proposto neste trabalho e instanciado para dois particulares SGBDs: *Jasmine* (COMPUTER ASSOCIATES, 2003) e *Caché* (INTERSYSTEMS, 2003). Além disso, utilizou-se um mesmo sistema exemplo já submetido à reengenharia com a utilização de SGBD relacional (CAMARGO, 2001) para que uma análise comparativa dos três SGBDs fosse realizada.

A versão 1 implementada por Camargo (2001) utilizou o SGBD relacional Sybase (SYBASE, 2003) associado ao padrão de projeto *Persistence Layer* (YODER; JOHNSON; WILSON, 1998) para tratar a diferença entre os paradigmas da aplicação e do banco de dados. O autor apresentou em seus resultados que a definição de uma camada de persistência facilita a manutenção do sistema. Uma vez que o padrão SQL é estável, essa solução é factível para diversos gerenciadores relacionais comerciais, inclusive os de domínio público, como *MySQL*, *PostGres* ou *Interbase*. Apesar desses benefícios, o desenvolvedor tem o ônus de implementar métodos que têm interface com os componentes do padrão *Persistence Layer*.

Nas versões desenvolvidas neste trabalho, com utilização de sistemas gerenciadores de bancos de dados com suporte a objetos, foram observados os seguintes ganhos ao processo de reengenharia:

- os objetos têm uma representação unificada na aplicação e no banco de dados (memória em nível único), não havendo mapeamento entre a camada de aplicação e o

banco de dados;

- maior legibilidade do banco de dados, que expressa a semântica do modelo orientado a objetos,
- os *frameworks* de suporte à persistência são baseados em padrões de projeto, que uniformizam o tratamento da persistência na aplicação;
- melhor suporte para manipulação de objetos complexos;
- a persistência por alcançabilidade simplifica a codificação, principalmente para objetos complexos
- redução do volume de código, em função da não necessidade de se mapear classes de objetos para relações.

Essas vantagens indicam maior produtividade no processo de engenharia avante e maior manutenibilidade do sistema após a reengenharia. Apesar dessas vantagens, a seleção do gerenciador de banco de dados deve levar em consideração também outros fatores, como desempenho e portabilidade. No que se refere a desempenho, a literatura cita que os SGBDs orientados a objetos têm melhor desempenho para estruturas complexas, enquanto os SGBDs relacionais, para estruturas simples (MACIASZEK (2003), AGERFALK (1999), RAO (1998), CASE; HENDERSON-SELLERS; LOW (1996)). Considerando que as restrições de desempenho limitam o uso de SGBDs orientados a objetos a ninhos específicos de aplicações, a adoção de SGBDs objeto-relacionais é uma alternativa viável, pelo suporte ao paradigma orientado a objetos e pela perspectiva evolutiva em relação aos ancestrais relacionais.

Quanto à portabilidade, observa-se que, apesar da busca pela padronização de bases de objetos, como os comitês ANSI/SQL3 e ODMG, os SGBDs com suporte a objetos ainda apresentam aderência parcial a esses padrões, o que dificulta a portabilidade das aplicações e a uniformização do tratamento da persistência. Nesse quesito, os SGBDs relacionais têm mais vantagens que SGBDs orientados a objetos, dentre as quais:

- facilidade de mudança do próprio SGBD, pela padronização da linguagem SQL para definição e manipulação de banco de dados relacionais, enquanto o padrão ODMG, proposto para SGBDs orientados a objetos é parcialmente adotado, o que dificulta a portabilidade (o mesmo ocorre com SGBDs híbridos, uma vez que o padrão SQL3 também é incipiente);
- facilidade de mudança de plataforma de hardware também é maior em SGBDs

relacionais e SGBDs relacionais estendidos;

- compatibilidade com ferramentas CASE no suporte a migração de dados, re-geração de *scripts* de criação de banco de dados e auditoria, etc.
- *frameworks* de persistência aderentes a vários produtos, enquanto para SGBDs com suporte a objetos, são associados ao produto.

Dessa forma, este trabalho fornece ao engenheiro de software uma visão quanto ao uso de SGBDs com suporte a objetos em um processo de reengenharia orientada a objetos, uma visão comparativa em relação a SGBDs relacionais e os principais pontos que devem ser considerados na adoção do SGBD.

6.2. TRABALHOS FUTUROS

Este trabalho pode ser estendido ao estudo de outros SGBDs, particularmente os SGBD's relacionais estendidos, como por exemplo *Oracle 9i* (ORACLE, 2003) e *Informix Universal Server* (INFORMIX, 2003), apresentando quais variações no processo de engenharia avante ocorrem.

O estudo de caso desenvolvido focou somente um sistema de informação comercial, cujas funcionalidades são predominantemente baseadas em transações em bancos de dados. Outros domínios de aplicação podem ser analisados e comparados com o deste trabalho.

Uma análise quantitativa, baseada em métricas de sistemas orientados a objetos, pode ser especificada e aplicada e seus resultados confrontados com os do estudo aqui realizado.

Uma estratégia baseada em prototipagem pode ser avaliada com o uso dos SGBDs com suporte a objetos, pelo nível de reuso dos artefatos de projeto e automatização da implementação. A prototipagem no processo de reengenharia é particularmente interessante sob o ponto de vista de validação do modelo de análise. Considerando que o modelo de análise do sistema orientado a objetos é refinado a partir do MASA, modelo de análise do sistema legado, e que deve ser validado, a prototipagem pode ser utilizada tanto para concepção de um protótipo evolutivo quanto para prova de conceito (protótipo descartável

para validação de requisitos).

Técnicas de *refactoring* em sistemas orientados a objetos para substituição de SGBDs relacionais por objeto-relacionais podem ser avaliadas visando a melhoria do produto e do processo de reengenharia orientada a objetos.

REFERÊNCIAS BIBLIOGRÁFICAS

AGERFALK, P. J. On the combination of objects and relations in systems development. In: INFORMATION SYSTEMS RESEARCH SEMINAR IN SCANDINAVIA, 22., 1999, *Proceedings...* Keuruu, Finlândia, 1999.

APACHE. Disponível em: <http://www.apache.com/JakartaTomCat/>. Acesso em: 27 mar. 2003.

BARRY, D. K. *The object database handbook:* how to select, implement and use object-oriented databases. New York: John Wiley & Sons, 1996.

BIANCHI, A.; CAIVANO, D.; VISAGGIO, G. Method and process for iterative reengineering of data in a legacy system. In: WORKING CONFERENCE ON REVERSE ENGINEERING, 7., 2000, Brisbane, Australia. *Proceedings...* IEEE, 2000.

BISBAL, J.; LAWLESS, D.; WU, B.; GRIMSON, J. Legacy information systems: issues and directions. *IEEE Software*, v. 16, n. 5, p. 103-111, Sept./Oct. 1999.

BRAGA, R. T. V. *Padrões de software a partir de engenharia reversa de sistemas legados*. 1998. Dissertação (Mestrado em Ciências da Computação) - Instituto de Ciências Matemáticas e de Computação, Universidade de São Paulo, São Paulo.

BRODIE, M. L.; STONEBRAKER, M. Darwin: on the incremental migration of legacy systems. *Technical Memorandum of Eletronics Research, Laboratory College of Engineering*. University of California at Berkeley, 1993.

CAGNIN, M. I. *Avaliação das vantagens quanto à facilidade de manutenção e expansão de sistemas legados sujeitos à engenharia reversa e segmentação*. 1999. Dissertação (Mestrado em Ciências da Computação) - Departamento de Computação, Universidade Federal de São Carlos, São Carlos.

CAMARGO, V. V. *Reengenharia orientada a objetos de sistemas COBOL com a utilização de Padrões de Projeto e Servlets*. 2001. Dissertação (Mestrado em Ciências da Computação) - Departamento de Computação, Universidade Federal de São Carlos, São Carlos.

CASE, T.; HENDERSON-SELLERS, B.; LOW, G. C. A generic object-oriented methodology incorporanting database considerations. *Annals of Software Engineering*, v. 2. p. 5-24, 1996.

CATTEL, R. G. G.; BARRY, D.; BERLER, M.; EASTMAN, J.; JORDAN, D.; RUSSEL, C.; SCHADOW, O.; STANIENDA, T.; VELEZ, F. *The object data standard:* ODMG 3.0. San Francisco, California:Morgan Kaufmann Publishers, 2000.

CHIKOFSKY, E. J.; CROSS II, J. H. Reverse engineering and design recovery: a taxonomy. *IEEE Software*, v. 7, p. 13-17, 1990.

COLEMAN, D. *Object oriented development:* the Fusion Method. Englewood Cliffs: Prentice-Hall, 1994.

COMPUTER ASSOCIATES. Disponível em: <http://www.ca.com/products/>. Acesso em: 27 maio 2003.

COMPUTER ASSOCIATES. *Jasmine ii*: language bindings guide. [S.l.] 1 CD-ROM, 2000.

DEITEL, H. M.; DEITEL, P. J. *Java:* como programar. Porto Alegre: Bookman, 2001.

DEMEYER, S.; DUCASSE, S.; NIERSTRASZ, O. A pattern language for reverse engineering. EUROPEAN ON PATTERN LANGUAGES OF PROGRAMMING AND COMPUTING. 5. *Proceedings*... Andreas Rupping Ed. 2000.

ELMASRI, K.; NAVATHE, S. *Fundamentals on Database Systems*. Menlo Park, California:Addison-Wesley Longman Inc., 2000.

ESCOBAL, G. *Estratégia para reconhecimento de padrões de projeto de software em sistemas legados*. Relatório de Pesquisa de Iniciação Científica, PIBIC/UFSCar-CNPq., 2000.

GAMMA, E.; HELM, R.; JOHNSON, R.; VLISSIDES, J. *Design Patterns: Elements of Reusable Object Oriented Software*. Massachussetss: Addison-Wesley, 1995.

INFORMIX. Disponível em: <http://www.informix.com/products>. Acesso em: 26 jul. 2003

INTERSYSTEMS. Disponível em: <http://www.intersystems.com/cache/index.html>. Acesso em: 27 maio 2003.

JACOBSON, I.; LINDSTRÖM, F. Re-engineering of Old Systems to an Object Oriented Architecture. In: OOPSLA'91, 1991, Phoenix, Arizona. *Proceedings...*, ACM, 1991. p. 340-350.

KOSHAFIAN, S.; DASANANDA, S.; MINASSIAN, N. *The Jasmine object database*. Morgan Kaufmann, 1999.

LARMAN, C. *Utilizando UML e Padrões*. Porto Alegre:Bookman, 2000.

LARSON, J. A. *Database Directions:* from relational to distributed, multimedia, and object-oriented database systems. Upper Saddle River:Prentice-Hall, 1995.

LEAVITT, N. Whatever Happened to Object-Oriented Databases? *Computer Magazine*, Aug. 2000. Disponível em <http://www.cee.hw.ac.uk/~trinder/AdvDbSystems/Whatever.pdf>. Acesso em: 15 jul. 2003.

LEMOS, G. S. PRE/OO – *Um processo de reengenharia orientada a objetos com ênfase na garantia da qualidade*. 2002. Dissertação (Mestrado em Ciências da Computação) - Departamento de Computação, Universidade Federal de São Carlos, São Carlos.

MACIASZEK, A. M. *Relational versus object databases: contention or coexistence?* Disponível em: <http://www.comp.mq.edu.au/courses/comp866/oovsrel.html>. Acesso em: 10 jul. 2003.

OLSEM, M. R. An Incremental Approach to Software Systems Reengineering. *Journal of Software Maintenance:* research and practice, v. 10, p. 181-202, 1998.

OMG. Disponível em: <http://www.omg.org/UML>. Acesso em: 27 maio 2003.

ORACLE. Dipsonível em: <http://www.oracle.com/ip/deploy/database/oracle9i/>. Acesso em: 10 jul. 2003.

PENTEADO, R. D.; MASIERO, P. C.; CAGNIN M. I. An experiment of legacy code segmentation to improve maintenability. In: EUROPEAN CONFERENCE ON SOFTWARE MAINTENANCE AND REENGINEERING, 3., 1999, Amsterdam, Holanda. *Proceedings* …

IEEE, 1999. p. 111-119, 1999.

PENTEADO, R. D.; BRAGA, R.T.V.; MASIERO, P. C. Improving the Quality of Legacy Code by Reverse Engineering. In: INTERNATIONAL CONFERENCE ON INFORMATION SYSTEMS, ANALYSIS AND SYNTHESIS, 4., 1998, Orlando, Florida. *Annals*… 1998. p. 364-370, 1998.

PENTEADO, R. D.; MASIERO, P. C.; PRADO, A. F.; BRAGA, R.T.V.. Reengineering of legacy systems based on transformation using the object oriented paradigm. In: WORKING CONFERENCE ON REVERSE ENGINEERING, 5., 1998, Honolulu, Hawaii. *Proceedings ...* IEEE, 1998b. p. 144-153.

PENTEADO, R. D. *Um método para reengenharia orientada a objetos*. 1996. Tese (Doutorado em Física Computacional) - Instituto de Física de São Carlos, Universidade de São Paulo, São Carlos.

PRESSMAN R. S. *Software engineering: a practitioner's approach*. 5th ed. New York:Mc Graw-Hill, 2001.

PRIETO, G. A. *Utilização de padrões de projeto na reengenharia de sistemas*. 2001. Dissertação (Mestrado em Ciências da Computação) - Departamento de Computação, Universidade Federal de São Carlos, São Carlos.

RAO, B. *Persistence in object oriented database systems*. Dept. of Computer Science Wichita State University. Wichita, KS. 1998.

RATIONAL. Dipsonível em: <http://www.rational.com/products/rose/index.jsp/>. Acesso em: 27 maio 2003.

RECCHIA, E. L. FaPRE/OO – *Uma família de padrões para reengenharia orientada a objetos de sistemas legados procedimentais*. 2002. Dissertação (Mestrado em Ciências da Computação) - Departamento de Computação, Universidade Federal de São Carlos, São Carlos.

SEI. SOFTWARE ENGINEERING INSTITUTE - SEI. *The Capability Maturity Model:* guidelines for improving the software process. Massassuchetts: Addison-Wesley, 1995.

SOMMERVILLE, I. *Engenharia de Software*. 6. ed. Addison-Wesley, 2001.

SYBASE. Disponível em: <http://www.sybase.com>. Acesso em: 30 abr. 2003.

TAHVILDARI, L.; KONTOGIANNIS, K. On the role of design patterns in quality-driven re-engineering. In: EUROPEAN CONFERENCE ON SOFTWARE MAINTENANCE AND REENGINEERING, 6., 2002. *Proceedings...*, IEEE, 2002.

YODER, J. W.; JOHNSON, R. E.; WILSON, Q. D. Connecting Business Objects to Relational Databases. In: CONFERENCE ON THE PATTERN LANGUAGES OF PROGRAMS, 5., 1998, Monticello, Illinois. *Proceedings...*, 1998.