UNIVERSIDADE FEDERAL DE JUIZ DE FORA

Departamento de Ciência da Computação

# Aplicação Prática de Técnica de Reengenharia de Software


Rafael Ferreira de Almeida

Universidade Federal de Juiz de Fora
Instituto de Ciências Exatas
Departamento de Ciência da Computação
Bacharelado em Ciência da Computação
Orientadora: Profa. Fernanda C. A. Campos

Juiz de Fora, MG

Julho de 2009

# Aplicação Prática de Técnica de Reengenharia de Software

Rafael Ferreira de Almeida

Monografia submetida ao corpo docente do Departamento de Ciência da Computação do Instituto de Ciências Exatas da Universidade Federal de Juiz de Fora, como parte integrante dos requisitos necessários para a obtenção do grau de Bacharel em Ciência da Computação.

Aprovada pela banca constituída pelos seguintes membros:

____________________________________________________

**Fernanda C. A. Campos** – orientadora
Dra. em Engenharia de Sistemas e Computação, COPPE/UFRJ, 1999

____________________________________________________

**Ely Edison Matos**
MSc. em Modelagem Computacional, UFJF, 2008

____________________________________________________

**Regina Maria Maciel Braga**
Dra. em Engenharia de Sistemas e Computação, COPPE/UFRJ, 2000

Juiz de Fora, MG

Julho de 2009

Agradecimentos

Gostaria de agradecer a todos os que, de alguma maneira, contribuíram não só para a realização deste trabalho, mas que também participaram ao longo de todo o curso.

Aos meus pais e familiares, pela enorme oportunidade que me foi dada, permitindo que eu tivesse a chance de chegar aonde cheguei, e por todo carinho e força.

Aos amigos e colegas que também participaram de algumas batalhas desta grande jornada acadêmica, agradeço pela amizade e pelo apoio.

# Sumário

Sumário ... ii

Lista de Reduções ... iv

Lista de Figuras ... v

Lista de Tabelas ... vii

Resumo ... viii

1. Introdução ... 1

2. Reengenharia ... 3

2.1 - Apresentações, histórico e definições ... 3

2.2 – Reengenharia do Processo do Negócio ... 4

2.3 – Reengenharia de Software ... 5

2.4 – Engenharia Reversa ... 10

2.5 – Reestruturação ... 16

2.6 – Engenharia Avante ... 17

3. Persistência de Dados ... 19

3.1 – Introdução ... 19

3.2 – Visão Geral Sobre Orientação a Objetos ... 20

3.3 – Visão Geral sobre Banco de Dados ... 22

3.3.1 – Modelo Relacional ... 22

3.3.2 – Modelo Entidade Relacionamento ... 23

3.4 – Persistência Objeto Relacional ... 24

3.5 – Camada de Persistência ... 25

3.6 – Mapeamento Objeto Relacional ... 28

3.7 – Framework de Persistência ... 33

4. Estudo de Caso: Reengenharia do módulo SIGA-Biblioteca ... 35

4.1 – Introdução .... 35

4.2 – Reengenharia do módulo SIGA-Biblioteca .... 36

5. Considerações Finais .... 48

Referências Bibliográficas .... 49

Apêndice I .... A1

Visão Geral sobre UML .... A1

A.1 Principais Elementos .... A1

A.1.1 Pacotes .... A1

A.1.2 Classes .... A2

A.1.3 Atores .... A3

A.1.4 Caso de Uso .... A3

A.2 Principais Diagramas .... A3

A.2.1 Diagrama de Classes .... A3

A.2.2 Diagrama de Caso de Uso .... A4

A.2.3 Diagrama de Estados .... A5

# Lista de Reduções

| | |
|---|---|
| BPR | *Business Process Reengineering* |
| CRUD | *Create Retrieve Update Delete* |
| DER | Diagrama Entidade Relacionamento |
| MER | Modelo Entidade Relacionamento |
| MOR | Mapeamento Objeto Relacional |
| OO | Orientação a Objetos |
| RAD | *Rapid Application Development* |
| SGBD | Sistema Gerenciador de Banco de Dados |
| SGBDR | Sistema Gerenciador de Banco de Dados Relacional |
| SIGA | Sistema Integrado de Gestão Acadêmica |
| SQL | *Structured Query Language* |
| UFJF | Universidade Federal de Juiz de Fora |
| UML | *Unified Modeling Language* |
| XML | *Extensible Markup Language* |

# Lista de Figuras

Figura 2.1 – Um modelo de BPR (Pressman, 2006)........ 5

Figura 2.2 – Relacionamentos no Ciclo de Desenvolvimento de Software (Chikofsky e Cross, 1990). .. 6

Figura 2.3 – O Processo de Reengenharia (Sommerville, 2007). ........ 8

Figura 2.4 – Um modelo de processo de reengenharia de software (Pressman, 2006)......... 8

Figura 2.5 – Visualizações de Software no Ciclo de Desenvolvimento (Costa, 1997)......... 11

Figura 2.6 – O Processo de Engenharia Reversa (Sommerville, 2007)......... 12

Figura 2.7 – O Processo de Engenharia Reversa (Pressman, 2006). ........ 13

Figura 3.1 – Camadas que compõem um modelo de gerenciamento de persistência de dados (Bauer & King, 2005)......... 20

Figura 3.2 – Persistência de comandos SQL com regras de negócio (Ambler, 2005). ........ 30

Figura 3.3 – Persistência encapsulando os comandos SQL em classes de dados (Ambler, 2005)......... 31

Figura 3.4 – Implementação de persistência utilizando uma camada robusta (Ambler, 2005). ........ 32

Figura 4.1 – Diagrama de casos de uso Gerente......... 37

Figura 4.2 – Diagrama de classe Catalogação de Exemplar. ........ 38

Figura 4.3 – Diagrama de estados Situação Empréstimo......... 38

Figura 4.4 – DER Circulação......... 39

Figura 4.5 – Mapeamento da classe unidade. ........ 41

Figura 4.6 – Chamada para função de excluir obra........ 43

Figura 4.7 – Função para excluir obra antes da reestruturação. ........ 44

Figura 4.8 – Função reestruturada para exclusão da obra. ........ 44

Figura 4.9 – Função GetCategoria() antes da reestruturação......... 45

Figura 4.10 – Função GetCategoria() reestruturada. ........ 45

Figura A.1 – Exemplo de Pacote em UML. ........ A2

Figura A.2 – Exemplo de Classe em UML. ........ A2

Figura A.3 – Exemplo de Ator em UML. .......... A3

Figura A.4 – Exemplo de Caso de Uso em UML. .......... A3

Figura A.5 – Exemplo de Diagrama de Classes em UML. .......... A4

Figura A.6 – Exemplo de Diagrama de Caso de Uso em UML. .......... A4

Figura A.7 – Exemplo de Diagrama de Estados em UML. .......... A5

# Lista de Tabelas

Tabela 4.1 – Relacionamentos do elemento principal da classe ........................................................ 42

Tabela 4.2 – Relacionamentos dos atributos da classe........................................................................ 42

Tabela 4.3 – Relacionamentos entre as classes ..................................................................................... 42

Tabela 4.4 – Funções dos objetos persistentes...................................................................................... 46

# Resumo

Este trabalho tem por objetivo fornecer um embasamento sobre o processo de manutenção de software, mais especificamente sobre reengenharia. Devido a isso são apresentadas as definições sobre essa modalidade de manutenção e os casos em que se aplicam, bem como a razão de utilizar, a forma de adotar e como realizar esse processo de reengenharia de software.

Além disso, este trabalho mostra uma aplicação de técnica de reengenharia de software no módulo SIGA-Biblioteca, que integra o sistema de gestão acadêmica da Universidade Federal de Juiz de Fora. O módulo passou por um processo de reengenharia a fim de usufruir funcionalidades proporcionadas pela implantação da camada de persistência de objetos.

**Palavras-chave:** Manutenção de Software, Reengenharia de Software, Camada de Persistência, Mapeamento Objeto Relacional, Persistência Objeto Relacional.

# 1. Introdução

A variedade de problemas que envolvem manutenção de software cresce constantemente, sendo que as soluções não acompanham essa evolução. Esses problemas são resultantes de código fonte e documentação mal elaborados ou defasados.

A partir do momento em que um sistema começa a ser utilizado, ele entra em um estado contínuo de mudança. Mesmo que tenha sido construído aplicando as melhores técnicas de projeto e codificação existentes, os sistemas vão se tornando obsoletos em vista das novas tecnologias que são disponibilizadas.

Além das correções de erros, as mudanças mais comuns que os sistemas sofrem são migrações para novas plataformas, ajustes para mudanças de tecnologia de hardware ou sistema operacional e extensões em sua funcionalidade para atender os usuários. Em geral, essas mudanças são realizadas sem que haja preocupação com a arquitetura geral do sistema, produzindo estruturas mal projetadas, documentação desatualizada, lógica e codificação ruins, sendo esses os focos que dificultam a manutenção em um sistema.

Quando o sistema não é fácil de ser mantido, porém, de grande utilidade, ele deve ser reconstruído. Partindo-se do sistema existente, são abstraídas as suas funcionalidades e são construídos o modelo de análise e o projeto do software. Esse processo é denominado reengenharia de software.

O número de aplicações combinando orientação a objetos e banco de dados relacionais tem crescido muito. Possuir habilidade de utilizá-los juntos torna-se primordial para preservar investimentos em sistemas de armazenamento de informações e, ao mesmo tempo, adequar as aplicações da organização aos domínios cada vez mais presentes. Quando os objetos precisam ser armazenados em banco de dados relacionais, o vácuo entre os dois aspectos precisa ser preenchido.

Para corrigir este problema, a abordagem proposta por um framework de persistência é que existirá um mapeamento entre suas classes e as tabelas de seu banco de dados de tal forma que uma simples mudança, como alteração no nome de algum campo, não levará a mudança na classe de negócio.

A principal função de uma camada de abstração de acesso a dados é garantir ao desenvolvedor a total independência entre o modelo de objetos e o esquema de dados do banco, permitindo que a base ou detalhes do esquema de dados sejam substituídos sem impacto nenhum na aplicação. Uma camada de persistência permite o armazenamento e manutenção do estado de objetos em algum meio não-volátil, como um banco de dados, de forma transparente.

Este trabalho apresenta uma abordagem sobre técnicas de reengenharia de software aplicada no módulo SIGA-Biblioteca, que integra o sistema de gestão acadêmica da Universidade Federal de Juiz de Fora. O módulo em questão foi um dos primeiros a ser desenvolvido no sistema, e precisava passar por um processo de reengenharia a fim de usufruir novas funcionalidades proporcionadas por novas versões da linguagem fonte utilizada no desenvolvimento, em particular, a implantação da camada de persistência de objetos.

A monografia se divide em cinco capítulos, além de um apêndice, desta introdução e das referencias bibliográficas.

No segundo capítulo são apresentados conceitos sobre reengenharia e algumas propostas apresentadas por pesquisadores como modelos do processo de reengenharia de software.

O terceiro capítulo aborda os conceitos relacionados à camada de persistência de dados. Apresenta uma visão geral introdutória sobre orientação a objetos e banco de dados, seguido de conceitos sobre persistência do objeto relacional, abordando o mapeamento destes objetos e apontando vantagens de se utilizar um framework persistente.

O quarto capítulo apresenta um estudo de caso, baseado num modelo proposto no segundo capítulo, do processo de reengenharia que o módulo SIGA-Biblioteca do sistema acadêmico da Universidade Federal de Juiz de Fora foi submetido.

O quinto capítulo conclui o trabalho. No apêndice I é feita uma breve revisão sobre conceitos importantes da *Unified Modeling Language*, que é utilizada no capítulo quatro.

# 2. Reengenharia

## *2.1 - Apresentações, histórico e definições*

Atualmente, diversas empresas migram seus sistemas para novas linguagens ou procuram alguma forma de melhorar a qualidade do seu software existente. Isso se deve ao constante crescimento dos problemas que envolvem a manutenção, sendo que as soluções não acompanham essas evoluções.

Reengenharia nada mais é do que reorganizar e modificar o software com o objetivo de torná-lo mais fácil de manter. A partir do momento em que um sistema começa a ser utilizado, ele entra num processo contínuo de mudanças, mesmo tendo sido criado aplicando técnicas de projeto. Ao passo que novas tecnologias são disponibilizadas, os sistemas vão se tornando obsoletos.

Quando um sistema se torna muito difícil de ser mantido, ele precisa passar por uma fase de reconstrução. Alem de correções de erros, as mudanças mais comuns envolvem ajustes de tecnologia, migração para novas plataformas e extensões na sua funcionalidade básica a fim de atender aos usuários.

Pressman (2006) utiliza uma analogia para facilitar essa explicação: considere qualquer produto de tecnologia que tenha servido bem, é utilizado regularmente, mas está ficando velho, quebra com freqüência, leva mais tempo do que gostaria para reparar e deixou de representar a tecnologia mais recente. Se o produto for um hardware a solução seria comprar um modelo mais novo, mas se for um software, feito sob encomenda, essa opção pode não ser possível. Sendo assim, o ideal seria reconstruí-lo, melhorando seu desempenho, confiabilidade e funcionalidade.

Segundo Sommerville (2007), a reengenharia de software se ocupa de reimplementar sistemas legados, para que sua manutenção seja mais fácil. A reengenharia pode envolver redocumentar, organizar e reestruturar o sistema, traduzir o sistema para uma linguagem de programação mais moderna e modificar e atualizar a estrutura e os valores dos dados do sistema. A funcionalidade do software não é modificada e, normalmente, a arquitetura do sistema também permanece a mesma.

## *2.2 – Reengenharia do Processo do Negócio*

Com o atual cenário econômico, as empresas precisam se adequar, objetivando aumentos de produtividade e qualidade para sobreviverem. Por isso é importante que elas moldem seus processos com a finalidade de produzirem seus produtos ou prestarem seus serviços seguindo critérios mais econômicos.

Segundo Cameira (2003), um processo de negócio é um conjunto de tarefas logicamente relacionadas, realizadas para conseguir um resultado definido do negócio. No processo de negócio, fatores como equipamentos, pessoal e recursos materiais são combinados para produzir um resultado específico.

Reengenharia do processo do negócio (*Business Process Reengineering*, BPR) consiste na busca e na implementação no processo do negócio para conseguir resultados inovadores. A BPR é iterativa, ou seja, as metas precisam ser adaptadas de acordo com o ambiente no qual elas estão sendo aplicadas. Sendo assim, não há começo nem fim; ela é um processo evolutivo (Cameira, 2003).

De acordo com Pressman (2006), todo processo de negócio tem um cliente definido, uma pessoa ou grupo que recebe o resultado, seja uma idéia, um relatório, um projeto ou um produto. Alem disso, eles exigem que diferentes grupos organizacionais participem das tarefas logicamente relacionadas que definem o processo (figura 2.1).

A BPR pode ser aplicada em qualquer nível da hierarquia, mas à medida que seu escopo se amplia, os riscos associados crescem bastante. Por esse motivo, a maioria dos esforços focaliza processos individuais ou subprocessos (Pressman, 2006).

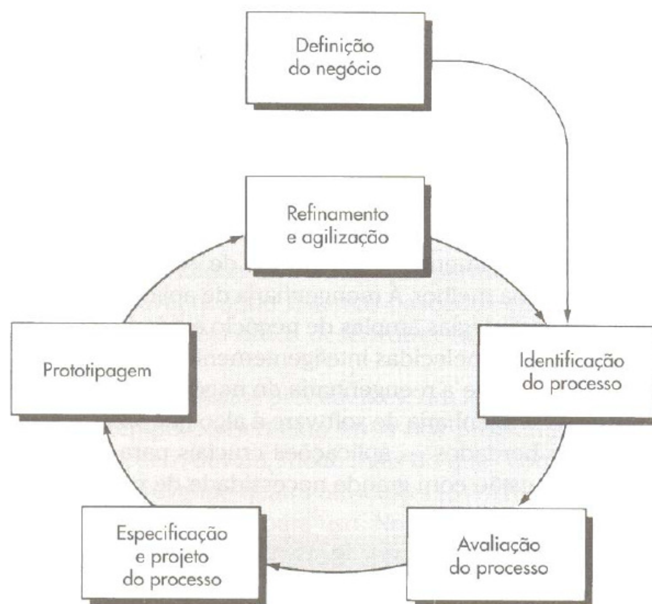


Figura 2.1 – Um modelo de BPR (Pressman, 2006).

## *2.3 – Reengenharia de Software*

Quando um sistema não é fácil de ser mantido, porém, ele é de grande utilidade, deve ser reconstruído. O processo dessa reconstrução, partindo do sistema existente, seja via código fonte, interface ou ambiente, onde são abstraídas suas funcionalidades e construídos o modelo de análise e o projeto do software, dá-se o nome de reengenharia de software.

Segundo Chikofsky e Cross (1990), em qualquer processo de desenvolvimento de software, espera-se que exista interação entre seus estágios. Os estágios iniciais envolvem conceitos mais gerais, independentes da implementação, enquanto os estágios finais enfatizam os detalhes de implementação, como é mostrado na figura 2.2.

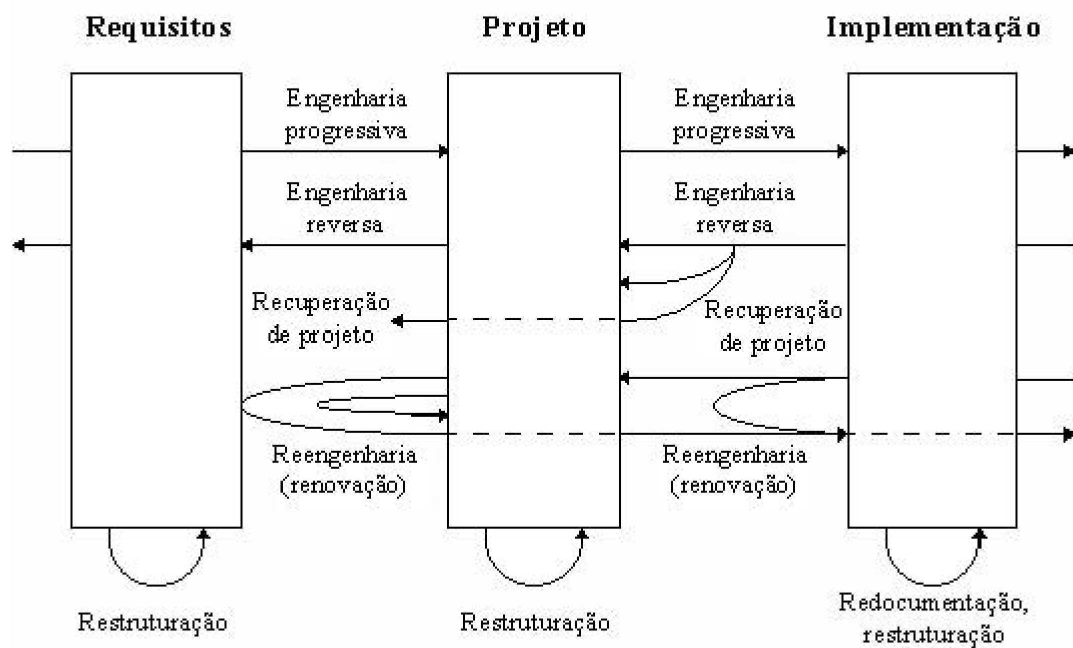


Figura 2.2 – Relacionamentos no Ciclo de Desenvolvimento de Software (Chikofsky e Cross, 1990).

A reengenharia leva tempo, tem um custo significativo, e absorve recursos que poderiam ser usados em outras necessidades emergenciais. O aumento de detalhes durante o processo de desenvolvimento conceitua os níveis de abstração. As informações podem ser representadas em qualquer estágio do desenvolvimento, seja de forma detalhada, com baixo grau de abstração, seja de forma mais sucinta, com alto grau de abstração. Para Chikofsky e Cross (1990), o processo de desenvolvimento do software pode ser definido em três fases, com níveis de abstração bem diferenciados:

- Requisitos: especificação do problema a ser resolvido, incluindo objetivos, restrições e regras de negociação;
- Projeto: especificação da solução;
- Implementação: codificação, teste e adaptação ao sistema operacional.

Conforme Garcia e Prado (2005), podemos considerar engenharia progressiva a técnica tradicional, avançando progressivamente pelas fases do processo de desenvolvimento. Em outras palavras, ela segue a seqüência de desenvolvimento estabelecida no projeto,

visando à obtenção do sistema implementado. Levando em consideração as três fases do processo, entende-se por requisitos a especificação do problema a ser resolvido, incluindo restrições e regras de negócio, enquanto projeto mostra a especificação da solução, e a implementação em si trata da codificação e dos eventuais testes necessários.

Com exceção da engenharia progressiva, as outras transições entre as fases de desenvolvimento são tecnologias utilizadas na manutenção do software. A redocumentação, como uma subárea da engenharia reversa, consiste na revisão de uma representação semanticamente equivalente, dentro do mesmo nível relativo de abstração. Recuperação do projeto também é uma subárea da engenharia reversa na qual o conhecimento do domínio da aplicação é adicionado às observações referentes ao programa. A reestruturação consiste na transformação de uma forma de representação para outra no mesmo nível de abstração relativo, preservando a funcionalidade e a semântica do sistema. A engenharia reversa é o processo de analisar um sistema com a finalidade de criar sua representação de uma forma diferente ou em um nível mais alto de abstração do que o código fonte. Já a reengenharia consiste na reconstrução de algo do mundo real, tendo como propósito a busca por melhorias que permitam produzir algo de qualidade melhor ou comparável ao produto inicial, conforme Garcia e Prado (2005).

Quando se trata de realizar a reengenharia em um projeto de reconstrução de um software, é necessário que se proceda a engenharia reversa do sistema em questão, com o intuito de obter os modelos de análise baseados no software existente.

Sommerville (2007) aponta como sendo as principais características da reengenharia de software os riscos reduzidos, ou seja, a chance de ocorrerem problemas ou erros durante o redesenvolvimento de um software é maior do que o da reengenharia; e os custos reduzidos, que por sua vez diz que o custo da reengenharia é em torno de quatro vezes menor do que o utilizado para desenvolver um software novo. A diferença entre a engenharia convencional e a reengenharia é o ponto de partida de ambas. Enquanto a engenharia começa com uma especificação escrita, a reengenharia se inicia com o próprio sistema antigo, que servirá como especificação para o novo sistema, mostrado na figura 2.3.

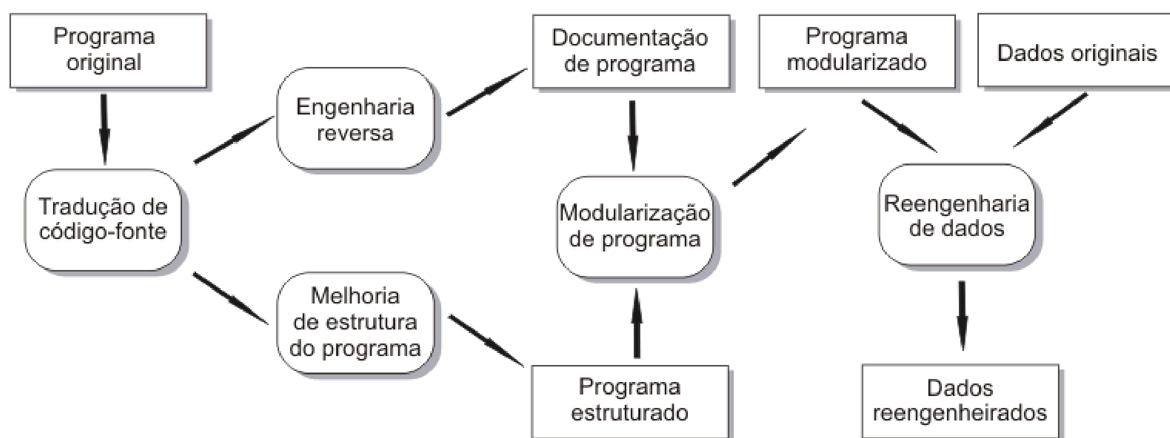


Figura 2.3 – O Processo de Reengenharia (Sommerville, 2007).

Já Pressman (2006) divide as atividades do processo de reengenharia de software em analise do inventário, reestruturação de documentos, engenharia reversa, reestruturação de programas e dados e engenharia avante. Nem sempre essas atividades ocorrem em uma maneira seqüencial, pois esse modelo permite que cada uma dessas atividades seja revisada num ciclo, além de poder terminar a qualquer momento, desde que a atividade em questão seja finalizada. A figura 2.4 mostra o modelo de reengenharia proposto por Pressman (2006).

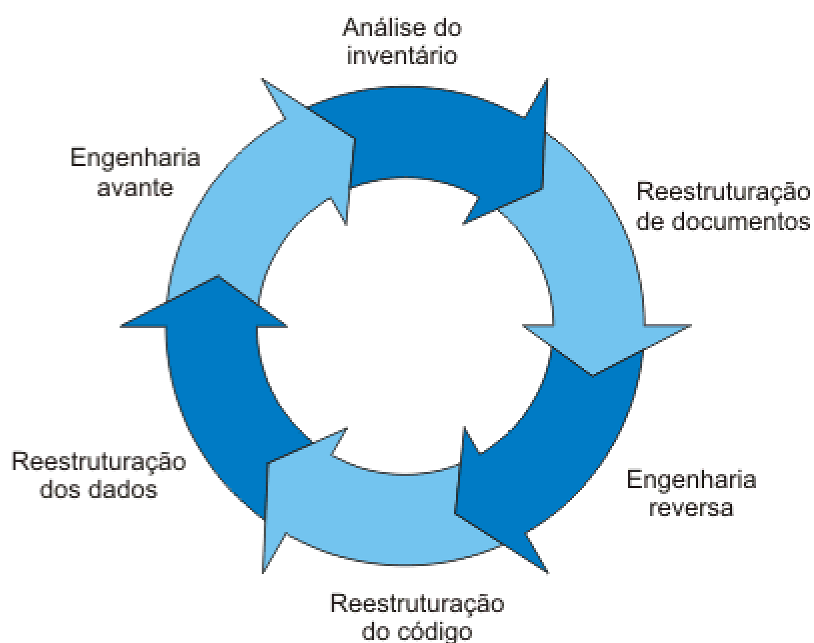


Figura 2.4 – Um modelo de processo de reengenharia de software (Pressman, 2006).

Para Pressman (2006), possuir um inventário de todas as aplicações é muito importante e necessário para qualquer empresa de software. Esse inventário pode ser um modelo de planilha contendo informações detalhadas de cada aplicação ativa. Fazendo a ordenação dessas informações de acordo com a importância dela para o negócio, sua longevidade e sua manutenibilidade corrente, esse inventário deve ser revisado em um ciclo, podendo mudar a ordem de prioridade de reengenharia.

Geralmente os sistemas herdados possuem pouca documentação. Se o sistema funciona, a criação de documentação fica em segundo plano, uma vez que ela consome muito tempo. Pressman (2006) diz que em alguns poucos casos esta abordagem se torna correta, pois se torna inviável recriar documentação para um programa improvável de sofrer alterações ou que esteja no final da sua vida útil. A documentação precisa ser atualizada, talvez não o sistema por completo, mas ao menos a parte que esteja sofrendo modificações.

As empresas se esforçam bastante para conseguir descobrir os segredos dos projetos, geralmente feitos há muitos anos. Esses segredos são difíceis de desvendar a princípio, porque nenhuma especificação jamais foi desenvolvida. Sendo assim, a engenharia reversa do software analisa o projeto, a fim de conseguir representá-lo em uma abstração mais alta que o código fonte, ou seja, é um processo de recuperação de dados (Pressman, 2006).

Segundo Pressman (2006), o tipo mais comum de reengenharia é a reestruturação do código. Alguns sistemas têm uma arquitetura sólida, mas módulos específicos podem ter sido projetados de uma maneira que se tornam difíceis de entender, testar e manter. Em tais casos, o código dos módulos deve ser reestruturado. Diferentemente da reestruturação do código, que ocorre em um nível relativamente baixo de abstração, a reestruturação de dados é uma atividade de reengenharia de escala plena. Na maioria dos casos, a reestruturação de dados começa com uma atividade de engenharia reversa, onde a arquitetura de dados atual é dissecada e os modelos de dados necessários são definidos.

Em uma situação ideal, as aplicações seriam reconstruídas de uma maneira automatizada. A engenharia avante, além de recuperar a informação do projeto de software existente, usa dessa informação para alterar ou reconstituir o sistema existente visando

aperfeiçoar sua qualidade. Não é necessária a utilização de todas as atividades em um único processo de reengenharia, porém pelo menos algumas dessas se tornam necessárias para que se possa desenvolver a reengenharia do software. Isso pode acontecer, por exemplo, se a linguagem do código fonte utilizada seja recente, ou ao menos, tolerável (Pressman, 2006).

## *2.4 – Engenharia Reversa*

Segundo Costa (1997), a engenharia reversa é o processo de análise de software com o objetivo de recuperar o projeto e a especificação, fazendo com que o sistema permaneça inalterado durante esse processo. Em outras palavras, ela é um processo de recuperação de projeto, e, em geral, é aplicada com a finalidade de melhorar um produto ou para analisar algum produto concorrente. Muitas técnicas de engenharia reversa utilizadas em hardware servem para compreender basicamente o sistema e sua estrutura. Entretanto, os objetivos para software são obter uma compreensão suficiente do projeto para auxiliar a manutenção, fortalecer o crescimento e facilitar no suporte.

Ainda segundo Costa (1997), o processo de engenharia reversa pode ser aplicado especificamente em um sistema, mesmo fazendo parte de um todo na reengenharia de software. Ao ser utilizada como parte de um processo, ela pode ajudar o engenheiro de software a compreender melhor o sistema e dar continuidade no processo. Baseado nos diferentes níveis e graus de abstração e a partir da engenharia reversa, o software pode ser visualizado de diferentes maneiras:

- Visão em nível implementacional: abstrai características específicas da implementação e da linguagem de programação;
- Visão em nível estrutural: abstrai detalhes da linguagem de programação a fim de revelar suas estruturas a partir de diferentes perspectivas. Tem como resultado uma representação explícita das dependências entre os componentes do sistema;
- Visão em nível funcional: relaciona partes dos programas às suas funções, procurando revelar as relações lógicas entre elas;

- Visão em nível de domínio: abstrai o contexto em que o sistema está operando.

Uma representação similar que foi desenvolvida no processo de engenharia progressiva pode diferir de uma forma de representação extraída do código. A figura 2.5 mostra a correspondência entre as categorias de visualização do software e as diferentes atividades do ciclo de desenvolvimento de software (Costa, 1997).

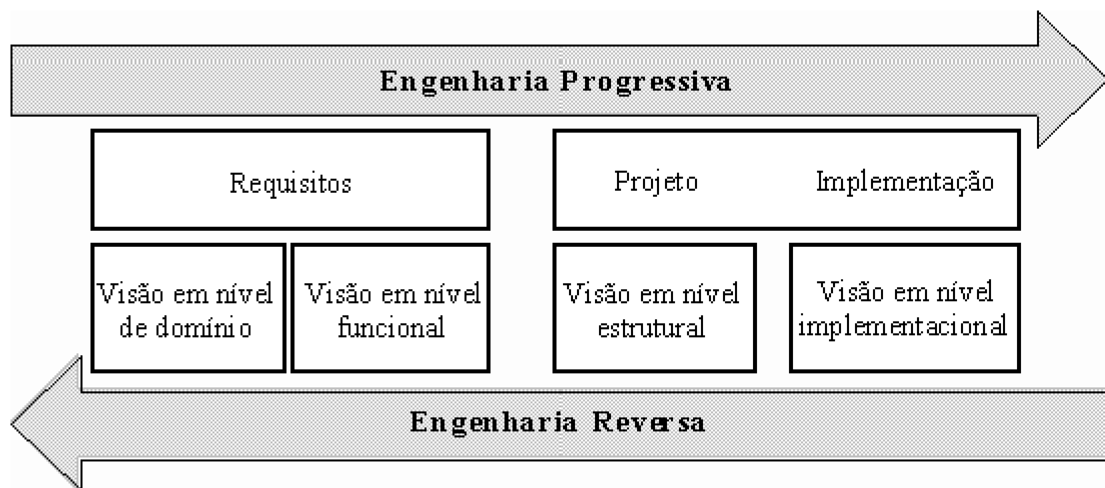


Figura 2.5 – Visualizações de Software no Ciclo de Desenvolvimento (Costa, 1997).

Muitas vezes é necessário incluir informações provenientes de conhecimentos e experiências humanas às informações contidas no código. Conforme as informações usadas, pode-se formular uma categorização dos métodos de engenharia reversa. Se o sistema já possuir um projeto e uma especificação iniciais, elas podem ser utilizadas pela engenharia reversa, ajudando na manutenção do programa.

Segundo Sommerville (2007), com essas informações adicionais, pode não ser necessário aplicar a reengenharia do código fonte no sistema. A análise seria feita por meio de ferramentas automatizadas, com o objetivo de descobrir sua estrutura. Contudo, isso não é suficiente para obter o projeto do sistema, necessitando assim, de informações adicionais descobertas por meio da compreensão do sistema. Essas informações são vinculadas ao código fonte do sistema e mantidas em um grafo direcionado.

As informações obtidas são usadas para comparação entre a estrutura do grafo e o código, a fim de se obter informações extras, que também serão armazenadas no grafo. Este grafo poderá gerar digramas, como o de programas, de estruturas de dados e matrizes de rastreamento. A geração de documentos é repetida, pois as informações do projeto são utilizadas para melhorar as informações contidas no sistema. A figura 2.6 mostra o processo de engenharia reversa segundo Sommerville (2007).

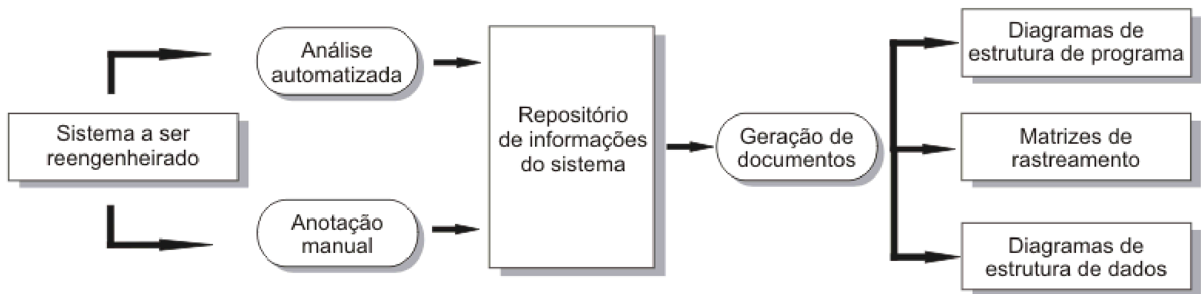


Figura 2.6 – O Processo de Engenharia Reversa (Sommerville, 2007).

Para Pressman (2006), a engenharia reversa de software é um processo de recuperação de projeto, consistindo em analisar um programa, na tentativa de criar uma representação do mesmo, em um nível de abstração mais alto que o código fonte, levando em consideração o processamento, a interface com o usuário que é aplicada e as estruturas de dados ou banco de dados que são usadas.

O propósito da engenharia reversa seria que, fornecendo uma listagem fonte, desestruturada e não documentada, fosse criada uma documentação completa para o programa; porém não existe uma ferramenta capaz de realizar tudo isso. Pressman (2006) diz que para retirar informações de um projeto depende do nível de abstração, da completeza da documentação, da interatividade entre ferramentas e engenheiros e da direcionalidade do processo, que são altamente variáveis.

Conforme Pressman (2006), o nível de abstração se refere à sofisticação da informação do projeto, que pode ser extraída do código fonte. A intenção é que esse nível de abstração seja o mais abrangente possível, ou seja, deve-se criar desde processos de baixo

nível, até modelos entidade-relacionamento, pois quanto mais alto o nível de abstração, mais fácil se tornará para um engenheiro de software entender o programa. A completeza de um processo refere-se ao nível de detalhamento que é fornecido em um nível de abstração, e quanto maior o nível de abstração, menor será seu nível de detalhamento, ou seja, quanto mais vezes forem realizadas análises, maior será a completeza encontrada. A interatividade se refere ao grau em que a pessoa é integrada com ferramentas automáticas para criar um processo efetivo de engenharia reversa. Na maioria dos casos, à medida que o nível de abstração aumenta, a interatividade precisará aumentar, ou poderá diminuir a completeza.

No caso da direcionalidade do processo for a sentido único, toda a informação extraída do código fonte será fornecida ao engenheiro de software, que poderá usá-la em alguma necessidade de manutenção. Caso a direcionalidade seguir dois sentidos, a informação tentará reestruturar ou regenerar o programa antigo. A figura 2.7 mostra o processo engenharia reversa segundo Pressman (2006).

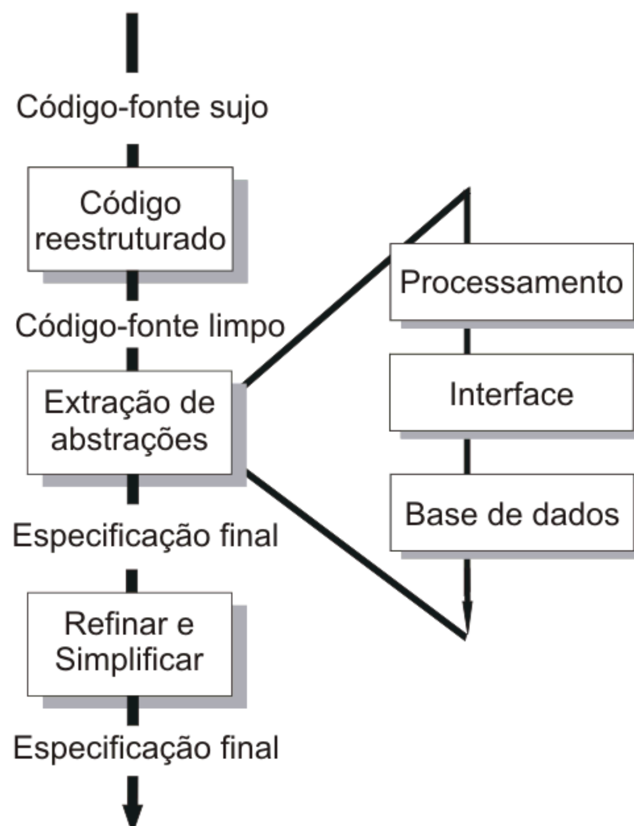


Figura 2.7 – O Processo de Engenharia Reversa (Pressman, 2006).

Para Pressman (2006), a engenharia reversa de dados ocorre em diferentes níveis de abstração. No nível de programa, estruturas de dados internas devem ser submetidas à engenharia reversa como parte da reengenharia global. No nível do sistema, a estrutura de dados globais, como arquivos ou banco de dados, são submetidos à reengenharia para acomodar novos paradigmas de gestão de banco de dados. A engenharia reversa das estruturas de dados globais atuais prepara o sistema para a introdução de um novo banco de dados. Sendo assim, cinco passos podem ser utilizados para definir o modelo de dados existente como precursor de um novo modelo de banco de dados por meio de reengenharia:

- Construir um modelo de objetos inicial;
- Determinar candidatos importantes;
- Refinar as classes provisórias;
- Definir generalizações;
- Descobrir associações.

Conhecendo tais informações, podem-se aplicar transformações para mapear a antiga estrutura de banco de dados para uma nova.

Ainda segundo Pressman (2006), engenharia reversa para entender o processamento começa com uma tentativa de entender, e depois extrair, abstrações procedimentais apresentadas pelo código fonte. Para entender as abstrações procedimentais, o código é analisado em diferentes níveis de abstração: sistema, programa, componente, padrão e declaração.

É necessário entender como funciona todo o sistema do programa, para que se possa fazer um trabalho mais detalhado da engenharia reversa. Cada programa do sistema representa uma abstração funcional em um alto nível de detalhamento. É criado um diagrama de blocos, representando a interação entre as abstrações. Cria-se uma narrativa de processamento para cada um dos componentes. Caso já existam especificações de sistema, ou

programa ou componente, elas são apenas revisadas de acordo com o código que existe (Pressman, 2006).

Para grandes sistemas, a engenharia reversa é geralmente obtida usando uma abordagem semi-automática. Ferramentas automatizadas são usadas para ajudar o engenheiro de software a entender a semântica do código existente. A saída desse processo é então passada para ferramentas de reestruturação e para a engenharia avante, para completar o processo de reengenharia.

Pressman (2006) afirma que um programa ter boa interface é tão importante que seu desenvolvimento se tornou um dos casos mais comuns na reengenharia. Entretanto, para que uma interface seja construída, é importante que tenha sido feito antes a engenharia reversa. Para que se entenda como a interface deve funcionar para o usuário, sua estrutura e comportamentos devem ser explicitados. Foram criadas três perguntas básicas para auxiliar na sua engenharia reversa:

- Quais são as ações básicas que a interface deve processar?
- Qual a descrição compacta da resposta comportamental do sistema a essas ações?
- O que quer dizer substituição ou, mais precisamente, que conceito de equivalência de interfaces é relevante aqui?

As respostas das duas primeiras podem ser obtidas inicialmente observando como a interface reage externamente com o usuário, porém é necessária a observação no código para obter informações mais detalhadas. A interface resultante da engenharia reversa não precisa necessariamente se espelhar na anterior, podendo ser totalmente diferente da antiga e, muitas vezes, são aconselháveis mudanças radicais, principalmente se forem utilizadas novos tipos de tecnologias para o mesmo.

## 2.5 – Reestruturação

Segundo Sommerville (2007), a reestruturação do software modifica o código fonte e os dados com o intuito de deixar suas modificações mais fáceis futuramente. Em geral, a reestruturação não modifica a arquitetura global do programa. Ela se concentra nos detalhes de projeto de módulos individuais e nas estruturas de dados locais definidas nos módulos. Se a reestruturação se estender alem dos limites dos módulos e abranger a arquitetura de software, ela se transformará em engenharia avante. A reestruturação ocorre quando sua arquitetura básica é sólida, porém o interior do software necessita de trabalho. Ela se inicia quando principais partes do programa são reparáveis e uma parte precisa de modificações significativas.

Pressman (2006) sugere que a reestruturação de código seja o tipo mais comum de reengenharia. Não apenas programas que tenham uma arquitetura sólida, porém módulos individuais e difíceis de entender também podem ter de passar pela reestruturação de seu código. Após verificar e registrar as violações da programação atual, o resultado é revisado e testado para garantir que nenhuma anomalia foi produzida, lembrando sempre que a documentação interna também deve ser atualizada.

A reestruturação de código é realizada para executar um projeto que produz a mesma função que o programa original, porém com mais qualidade. O objetivo é pegar um código desarrumado e originar um projeto que respeite a filosofia da programação (Pressman, 2006).

Segundo Sommerville (2007), para muitas aplicações, a arquitetura de dados está mais relacionada à viabilidade do programa em longo prazo do que ao código fonte propriamente dito. Um sistema com uma fraca arquitetura de dados pode se tornar difícil de adaptar ou aperfeiçoar. Na reestruturação a arquitetura de dados atual é repartida e os modelos de dados necessários são definidos. Os objetos de dados e atributos são identificados e as estruturas de dados existentes são revisadas quanto à qualidade. Caso a estrutura de dados seja fraca, os dados passam pela reestruturação. A arquitetura de dados tem uma forte influência na arquitetura do programa e nos algoritmos que o constituem, por isso, suas modificações podem resultar invariavelmente tanto em modificações arquiteturais quanto de código.

Para Pressman (2006), antes que se inicie a reestruturação dos dados deve-se realizar uma atividade chamada análise de código fonte, ou simplesmente análise de dados. As declarações contendo as definições dos dados, descrições de arquivos, entradas e saídas e descrições de interfaces deverão ser avaliadas, a fim de retirar itens de dados e objetos para obter informações sobre o fluxo de dados e entender as estruturas implementadas.

Completando a análise de dados, se inicia o reprojeto de dados, uma etapa de padronização de registro de dados, esclarecendo definições de dados para obter consistência entre os itens, os formatos de registro físico ou do formato do arquivo. Outra forma, chamada de racionalização dos nomes dos dados, faz com que todas as convenções de denominação de dados atendam aos padrões locais e que os sinônimos sejam eliminados à medida que os dados fluam através do sistema. Quando a reestruturação se estende além da padronização e racionalização dos dados, é necessário fazer uma modificação física nas estruturas de dados existentes, com o intuito de tornar o processo de dados mais efetivo (Pressman, 2006).

## *2.6 – Engenharia Avante*

De acordo com Pressman (2006), a engenharia avante, também conhecida como engenharia de renovação ou recomposição, recupera informações de projetos do software e as utiliza para alterar ou reconstruir o sistema existente. Geralmente, o software trabalhado por reengenharia implementa funções do sistema existente e adiciona novas funções, melhorando o seu desempenho geral.

Para atender as necessidades de alteração exigidas por um usuário em um programa com muitas linhas de código e pouca documentação, de acordo com Pressman (2006), as seguintes condições devem ser atendidas:

- Sempre que mudanças no projeto forem necessárias, aplicá-las no código fonte;
- Tentar entender o funcionamento interno global do programa com o intuito de tornar as modificações eficazes;

- Fazer um reprojeto, uma recodificação e testar as partes do software que exigem modificação, aplicando uma abordagem de engenharia de software para todos os segmentos revistos;
- Fazer um reprojeto, uma recodificação e testar o programa inteiro utilizando ferramentas CASE para ajudar no entendimento do projeto atual.

Segundo Pressman (2006), a opção correta depende da situação do sistema perante a instituição que a utiliza. Essa abordagem pode ser chamada de manutenção preventiva e é definida como a aplicação de metodologias atuais a sistemas de ontem, para suportar os requisitos de amanhã.

Quando uma empresa tem como seu produto o software, a manutenção preventiva é vista como novas versões do programa. O processo de engenharia avante aplica princípios, conceitos e métodos de engenharia de software para recriar uma aplicação existente. Na maioria dos casos a engenharia avante não cria simplesmente um equivalente moderno de um programa antigo. Requisitos novos do usuário e tecnologia são integrados no esforço de reengenharia. O programa redesenvolvido amplia a capacidade da aplicação antiga.

# 3. Persistência de Dados

## *3.1 – Introdução*

Diante das exigências do mercado consumidor dos dias atuais, fica notório a necessidade de aprimoramento dos processos de informação dos canais produtivos, sob pena de não conquistar um público em quantidade que garanta sua subsistência. Essa realidade torna-se mais complexa pelo fato de que o mercado consumidor acompanha a rápida mudança tecnológica.

Essa conjuntura contemporânea exige mais do que possuir dados armazenados, e isso pode se tornar muito caro. Logo, existe uma necessidade de transformar dados armazenados num banco de dados em informação lapidada, auxiliando o processo de tomada de decisão e possibilitando que as organizações mudem mais rápidas.

Segundo Bauer e King (2005), de uma forma simplificada, Persistência de Dados nada mais é do que armazenar dados em um banco de dados relacional. Um modelo gerencial de persistência atua em camadas, podendo ser feito por intermédio de instruções SQL [1] (*Structured Query Language*). O uso da persistência deixa transparentes as operações básicas de inserção, recuperação, atualização e remoção (CRUD) de dados, contudo, o principal ponto de destaque está no paradigma de Orientação a Objetos para banco de dados, tornando a modelagem e o trabalho de programação muito mais elegante e compreensível.

De acordo com Bauer e King (2005), através do Mapeamento Objeto Relacional (MOR), podemos realizar o mapeamento das entidades em um banco de dados através de uma coleção de metadados que atua conjuntamente com classes especializadas. O MOR pode ser feito tanto manualmente quanto automaticamente. O processo de mapeamento manual é trabalhoso, pois se torna necessário definir, para cada classe do sistema, como esta será

---

[1] SQL, ou Linguagem de Consulta Estruturada, tornou-se uma interface padrão no mercado de sistemas relacionais de bancos de dados, apesar de alguns produtos apresentarem algumas pequenas variações próprias. A linguagem SQL é um grande padrão de banco de dados. Isto decorre da sua simplicidade e facilidade de uso. Ela se diferencia de outras linguagens de consulta a banco de dados no sentido em que uma consulta SQL especifica a forma do resultado e não o caminho para chegar a ele (ANSI, 1992).

armazenada. Já o processo automático torna esse processo mais simples, mas nem sempre o torna transparente. A figura 3.1 mostra as camadas que compõem um modelo de gerenciamento de persistência de dados, Bauer e King (2005).

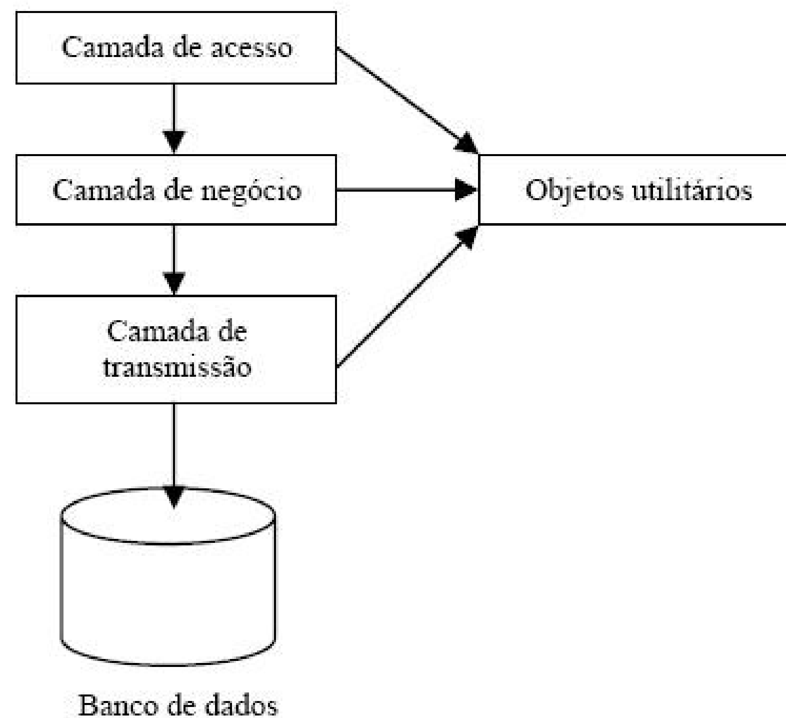


Figura 3.1 – Camadas que compõem um modelo de gerenciamento de persistência de dados (Bauer & King, 2005).

## *3.2 – Visão Geral Sobre Orientação a Objetos*

A orientação a objeto (OO) é um paradigma para o desenvolvimento de aplicações. Segundo Larman (2007), ela aproxima a linguagem de computadores ao mundo real, visando realizar a modelagem e codificação de uma forma mais natural. Com ela, o software fica organizado como uma coleção de objetos que interagem entre si.

Segundo Dall'Oglio (2007), a técnica OO propõe uma maior reusabilidade, através da eliminação de redundância de código, com conseqüente aumento na produtividade,

melhorias na manutenibilidade, pois as modificações no código são bem localizadas e não provoca descontrole em outras partes do software, e também oferece maior confiabilidade, devido ao encapsulamento que impede o acesso direto aos dados dos objetos.

Larman (2007) diz ainda que o objeto é o conceito central da abordagem OO. Ele pode ser entendido como algo do mundo real com limites bem definidos e identidade. Um objeto possui dados, chamados também de atributos, e operações. Quando se define um objeto conceitual a partir de um objeto do mundo real, diz estar fazendo uma abstração. Classe é um conjunto de objetos que possuem características e comportamento em comum.

Cada objeto deve ser criado antes de ser utilizado. O ato de criar um objeto é denominado instanciação. Todo objeto é instancia de uma classe. A instanciação de um objeto é feita através de um método especial denominado construtor. Todos os objetos instanciados terão a mesma estrutura de métodos e atributos de sua classe.

Para Larman (2007), os atributos de um objeto são como variáveis que armazenam dados, e as operações, são o que os objetos podem realizar. Quando uma operação é solicitada ao objeto, diz-se que este objeto recebeu uma mensagem. Esta mensagem pode trazer parâmetros, que são valores passados ao objeto que a recebe. O fato de uma classe reunir tanto características quanto o comportamento dos objetos é chamado de encapsulamento. Assim, outros objetos só terão acesso às operações ou atributos que estiverem declarados como públicos. Dá-se o nome de polimorfismo ao fato de objetos de classes diferentes reagirem de forma particular a uma mesma operação.

Para Dall'Oglio (2007), outro conceito muito importante da OO é a herança. Com ela, uma classe pode herdar todas as operações e atributos de outra classe, acrescentando os seus próprios atributos e operações. A classe que herdou a estrutura é chamada de subclasse, enquanto a que cedeu a estrutura chama-se superclasse. Uma classe pode ter atributo que se refere à outra classe. Essa referência tem o nome de associação.

## *3.3 – Visão Geral sobre Banco de Dados*

Nos estágios iniciais da computação, os dados eram armazenados em sistemas de arquivos, cujos principais problemas associados (como redundância, manutenção, segurança e dependência entre dados e aplicação) fizeram surgir uma nova tecnologia para o gerenciamento dos dados armazenados, denominada Sistema Gerenciador de Banco de Dados (SGBD).

Segundo Piattini e Días (2000), os SGBDs são as ferramentas de software que permitem a definição, criação, manutenção e uso, enfim, o gerenciamento de grandes quantidades de dados inter-relacionados e armazenados em mídia acessível via computador.

Os primeiros SGBDs eram baseados nos modelos Hierárquico e de Rede. Um modelo de banco de dados é um modo de estruturar logicamente as informações. Um usuário de banco de dados hierárquico vê o banco de dados como um conjunto ordenado de árvores de registros de dados. Isto decorre que se dois registros são unidos por um elo, um dos registros é considerado ascendente e o outro descendente. Já a estrutura de dados em Rede pode ser considerada uma forma ampliada da estrutura hierárquica de dados, onde a principal diferença é que na estrutura hierárquica, um registro-filho tem exatamente um pai, enquanto que na estrutura de rede um registro-filho pode ter qualquer quantidade de pais, inclusive zero.

Piattini e Días (2000) descrevem que os modelos Hierárquico e de Rede eram fisicamente implementados via ponteiros, o que garantia o ponto forte desses modelos: a eficiência. Porém, as restrições funcionais necessárias para os conjuntos e ligações de dados eram a causa dos pontos fracos: baixa flexibilidade da estrutura física, dependência entre aplicação e dados, e dificuldades nas linguagens de consulta.

### 3.3.1 – Modelo Relacional

Segundo Lans (2004), o modelo Relacional é uma teoria fundada em conceitos matemáticos simples e sólidos, e amplamente implementado, completa ou parcialmente, por vários SGBDs disponíveis atualmente. Os bancos de dados relacionais são constituídos por

tabelas, cada qual contendo as suas respectivas colunas, que especificam os atributos, e linhas, contendo os dados armazenados nestas tabelas.

O nome deste modelo se refere ao fato de uma tabela de dados corresponder ao conceito matemático de Relação. Cada atributo de uma tabela é comparado a um conjunto matemático de valores, com a mesma quantidade de elementos, os quais formam as linhas da respectiva tabela, a partir de seus pares ordenados.

Para Lans (2004), um SGBD Relacional pode possuir uma grande quantidade de tabelas, cada uma identificada por um nome único. Em geral, cada tabela deve possuir sua chave primária específica, que é um identificador único para cada linha de dados, formado pelo conjunto de uma ou mais colunas da tabela, conforme foi analisada a entidade da vida real que aquela tabela represente no banco de dados. Uma chave primária em uma determinada tabela permite, entre outras características, que outras tabelas se referenciem a esta chave através de chaves estrangeiras. Este outro tipo de chave, também chamada de externa, serve como um apontamento de uma tabela para outra, que o próprio SGBD pode validar e gerenciar durante as atualizações dos dados, realizando as devidas restrições quando necessário, por exemplo, evitando a exclusão de registros de determinada tabela enquanto houverem registros relacionados em outra tabela.

### 3.3.2 – Modelo Entidade Relacionamento

Para Date (2004), o Modelo Entidade Relacionamento (MER) tem como objetivo servir de ferramenta aos projetistas de bancos de dados e analistas de sistemas, acrescentando uma camada intermediária que representa o esquema da empresa no projeto lógico do banco de dados. No projeto lógico é analisada e definida a organização dos dados na melhor forma possível dentro do respectivo SGBD selecionado. Na notação do DER, as entidades são representadas por retângulos, e cada tipo de relacionamento é representado por um losango ligado por linhas as respectivas entidades envolvidas. Nestas linhas também são definidas as possibilidades de quantidade de entidades daquele tipo que podem ser ligadas por aquele respectivo relacionamento. Os tipos possíveis de relacionamento são: um-para-um, um-para-muitos e muitos-para-muitos.

Fowler (2006) referencia o MER como uma clássica abordagem de projeto de banco de dados relacional, e resume que se trata de ter uma tabela para cada entidade da vida real que se queira representar no banco de dados, com um campo para cada elemento de informação desta entidade, mais um campo para cada relacionamento um-para-um ou um-para-muitos com outra entidade. Em resumo, este método de análise e projeto consiste na fase de definição do esquema da empresa usando o Diagrama Entidade Relacionamento (DER), seguido pela fase de tradução do esquema da empresa para o sistema de banco de dados escolhido.

Ainda para Date (2004), uma entidade, para a abordagem MER, pode ser considerada como sendo qualquer coisa da vida real que possa ser identificado, e que segundo o projetista do banco de dados é pertinente ao esquema da empresa. Os relacionamentos entre estas entidades da vida real também são representados no diagrama MER, após serem devidamente classificados conforme o seu tipo.

## *3.4 – Persistência Objeto Relacional*

Para Freire (2006), na orientação a objetos, chama-se de objetos persistentes aquele que permanecem existindo mesmo após o término da execução do programa. Isto somente é possível quando este objeto não possui dados dinâmicos, ou seja, dados que só fazem sentido no contexto do tempo em que estão executando. Os objetos que possuem dados de tempo de execução, se congelados, após sua recuperação os dados que não fazem mais sentido no contexto do novo tempo são ignorados ou perdidos.

Os objetos em uma aplicação OO podem ser transientes ou persistentes. Transientes são os objetos que desaparecem ao termino da execução do programa. Já os persistentes permanecem após o termino da aplicação, podendo ser recuperados ao executar o aplicativo novamente (Freire, 2006).

Segundo Cantu (2003), no processo de desenvolvimento de aplicações Orientadas a Objetos, nem sempre é dada a devida atenção à forma como a persistência de objetos será gerenciada e implementada. Na grande maioria dos sistemas, incluir código SQL para acesso ao SGBD em meio ao restante da lógica do sistema é a solução adotada, graças à rapidez de

implementação. Esta é uma escolha perigosa. Sua adoção implica muitas vezes no acoplamento do sistema ao SGBD utilizado, o que dificulta o processo de manutenção de código. Alem disso, quaisquer mudanças na estrutura, ou até mesmo na nomenclatura de colunas, das tabelas existentes no banco de dados trazem o caos à aplicação. Todo o código SQL codificado na aplicação tem que ser reescrito, recompilado e testado novamente.

Para diminuir este acoplamento, surge uma segunda opção, que propõe separar o código SQL das classes da aplicação, de forma que as alterações no modelo de dados requerem modificações apenas nas classes de acesso a dados (Data Access Classes), restringindo o impacto das mudanças no sistema como um todo. Esta estratégia trás um maior controle quanto ao escopo dos possíveis erros gerados por mudanças no esquema de dados do sistema. No entanto, apesar da limpeza de código e melhor divisão de responsabilidades trazidas pela adoção das Data Classes, a solução ainda não é a ideal, por manter objetos e dados relacionais intimamente ligados (Cantu, 2005).

A principal função de uma camada de abstração de acesso a dados é garantir aos desenvolvedores de software a total independência entre o modelo de objetos e o esquema de dados do banco, permitindo que a base ou detalhes do esquema de dados sejam substituídos sem impacto nenhum na aplicação.

## *3.5 – Camada de Persistência*

Segundo Freire (2006), a propriedade de um objeto manter sua integridade ao longo do tempo é chama de persistência. Dentro do contexto de desenvolvimento de sistemas, a persistência de dados é uma forma de manter as informações das aplicações em um meio do qual possam ser recuperados posteriormente.

Conceitualmente uma camada de persistência de objetos é uma biblioteca que permite a realização do processo de persistência, ou seja, o armazenamento e manutenção do estado de objetos em algum meio não volátil, como um banco de dados, de forma transparente. Graças à independência entre a camada de persistência e o repositório utilizado, também é possível gerenciar a persistência de um modelo de objetos em diversos tipos de

repositórios, teoricamente com pouco ou nenhum esforço extra. A utilização deste conceito permite ao desenvolvedor trabalhar como se estivesse em um sistema completamente orientado a objetos, utilizando métodos para incluir, alterar e remover objetos.

Segundo Freire (2006), as vantagens decorrentes do uso de uma camada de persistência no desenvolvimento de aplicações são evidentes. A sua utilização isola os acessos realizados diretamente ao banco de dados na aplicação, bem como centraliza os processos de construção de consultas e operações de manipulação de dados em uma camada de objetos inacessível ao programador. Este encapsulamento de responsabilidades garante maior confiabilidade às aplicações e permite que, em alguns casos, o próprio SGBD ou a estrutura de suas tabelas possam ser modificados sem trazer impacto à aplicação nem forçar a revisão e recompilação de códigos.

Segundo Ambler (2005), uma camada de persistência real deve implementar as seguintes características:

- Dar suporte a diversos tipos de mecanismos de persistência: um mecanismo de persistência pode ser definido como a estrutura que armazenará os dados, seja ele um SGBD relacional, um arquivo XML ou um SGBDOO, por exemplo. Uma camada de persistência deve suportar a substituição deste mecanismo livremente e permitir a gravação de estado de objetos em qualquer um destes meios.

- Encapsulamento completo da camada de dados: onde o usuário do sistema de persistência de dados deve utilizar-se, no máximo, de mensagens de alto nível como *save* ou *delete* para lidar com a persistência dos objetos, deixando o tratamento destas mensagens para a camada de persistência em si.

- Transações: ao utilizar da camada de persistência, o programador deve ser capaz de controlar o fluxo da transação, ou ter garantias sobre o mesmo, caso a própria camada de persistência preste este controle.

- Extensabilidade: a camada de persistência deve permitir a adição de novas classes ao esquema e a modificação fácil do mecanismo de persistência.

- Identificadores de Objetos: a implementação de algoritmos de geração de chaves de identificação garante que a aplicação trabalhara com objetos com identidade única e sincronizada entre o banco de dados e a aplicação.

- Cursores e Proxies: as implementações de serviços de persistência devem ter ciência de que, em muitos casos, os objetos armazenados são muito grandes, e recuperá-los por completo a cada consulta não é uma boa idéia. Algumas técnicas utilizam-se dos proxies para garantir que atributos serão carregados à medida que forem importantes para o cliente e do conceito de cursores para manter registro da posição dos objetos no banco de dados e em suas tabelas especificas.

- Registros: apesar da idéia de trabalhar-se apenas com objetos, as camadas de persistência devem, no geral, dispor de um mecanismo de recuperação de registros, conjuntos de colunas não encapsuladas na forma de objetos, como resultado de suas consultas. Isto permite integrar as camadas de persistências a mecanismos de geração de relatórios que não trabalham com objetos, por exemplo, além de permitir a recuperação de atributos de diversos objetos relacionados com uma só consulta.

- Arquiteturas Múltiplas: o suporte a ambientes de programas *stand-alone*, cenários onde o banco de dados encontra-se em um servidor central e mesmo arquiteturas mais complexas deve ser inerente a camada de persistência, já que a mesma deve visar a reusabilidade e fácil adaptação a arquiteturas distintas.

- Diversas versões de banco de dados e fabricantes: a camada de persistência deve tratar de reconhecer diferenças de recursos, sintaxe e outras minúcias existentes no acesso aos bancos de dados suportados,

isolando isto do usuário do mecanismo e garantindo portabilidade entre plataformas.

- Múltiplas conexões: um gerenciamento de conexões é uma técnica onde vários usuários utilizarão o sistema simultaneamente sem quedas de desempenho.
- Queries SQL: apesar do poder trazido pela abstração em objetos, este mecanismo não é funcional em todos os casos. Para os casos extremos, a camada de persistência deve prover um mecanismo de queries que permita o acesso direto aos dados ou então algum tipo de linguagem de consulta similar a SQL, de forma a permitir consultas com um grau de complexidade maior.
- Controle de concorrência: acesso concorrente a dados pode levar a inconsistência. Para prever e evitar problemas decorrentes do acesso simultâneo, a camada de persistência deve prover algum tipo de mecanismo de controle de acesso. Este controle geralmente é feito utilizando-se dois níveis, as linhas no banco de dados relativas ao objeto acessado por um usuário são travadas e tornam-se inacessíveis a outros usuários até o mesmo liberar o objeto. No mecanismo otimizado, toda a edição é feita em memória, permitindo que outros usuários possam alterar o objeto.

## *3.6 – Mapeamento Objeto Relacional*

Conforme apresentado por Fowler (2006), o processo de Mapeamento Objeto Relacional (MOR) é constituído de padrões de arquitetura, de comportamento e de estrutura. Os padrões de arquitetura definem como as regras do negócio da respectiva aplicação, chamada por ele de lógica de domínio, realizarão o acesso ao banco de dados. Padrões comportamentais estão relacionados principalmente à concorrência dos dados, apresentando como os objetos serão carregados e gravados no banco de dados sem gerar inconsistências ou

dados inválidos. Por fim, os padrões estruturais descrevem diferentes formas de mapear as associações entre objetos, como herança e relacionamentos, para o banco de dados relacional.

Mapeamento objeto relacional corresponde a um modelo de implementação, no qual se acrescenta uma camada que terá a finalidade de mapear as classes e seus atributos para o banco de dados ou fazer o inverso, mapear das tabelas e colunas para as classes de dados e seus atributos, ou seja, é uma técnica de desenvolvimento utilizada para reduzir a impedância entre a programação orientada a objetos e os banco de dados relacionais. As tabelas do banco de dados são representadas através de classes e os registros de cada tabela são representados como instancias das classes correspondentes.

Para Cantu (2005), com essa técnica, o programador não precisa de se preocupar com os comandos em linguagem SQL, pois irá usar uma interface de programação simples que faz todo o trabalho de persistência.

Com o mapeamento não é necessária uma correspondência direta entre as tabelas de dados e as classes do programa. A relação entre as tabelas onde originam os dados e o objeto que os disponibiliza é configurada pelo programador, isolando o código do programa das alterações a organização dos dados nas tabelas do banco de dados. A forma como este mapeamento é configurado depende da ferramenta que é usada.

Segundo Cantu (2005), existem algumas formas de se fazer tal tipo de mapeamento, dentre as quais se destaca o uso de frameworks que utilizam arquivos XML para mapear e persistir os objetos de uma aplicação. Ao utilizar um framework, os comandos SQL para as inserções, alterações, pesquisas e exclusões de dados não serão mais criados pelo desenvolvedor, pois tal responsabilidade passara a ser do framework utilizado.

A utilização de um framework persistente, além de gerar os comandos SQL que realizam as operações de inserção, alteração, seleção e exclusão, é um mecanismo de segurança capaz de eliminar problemas ocasionados com os comandos de SQL.

Ambler (2005) ao demonstrar a modelagem de uma camada robusta de persistência de objetos, inicia destacando três diferentes formas que geralmente são utilizadas nos dias de hoje para implementar a persistência de objetos em aplicações que acessam bancos de dados

relacionais: persistência de comandos SQL com regras de negócio; persistência encapsulando os comandos SQL em classes de dados; e persistência utilizando uma camada robusta.

A figura 3.2 demonstra a maneira mais comum de persistência de objetos, onde os comandos SQL são embutidos diretamente dentro do código fonte das classes de negócio, solução a qual, ainda segundo Ambler (2005), pode trazer vantagens na criação de pequenas aplicações ou protótipos, porém causa um profundo acoplamento entre as regras de negócio e o esquema do banco de dados, pois uma alteração neste esquema, por exemplo, ao renomear uma coluna de uma tabela, necessitará de uma alteração nas classes de negócio.

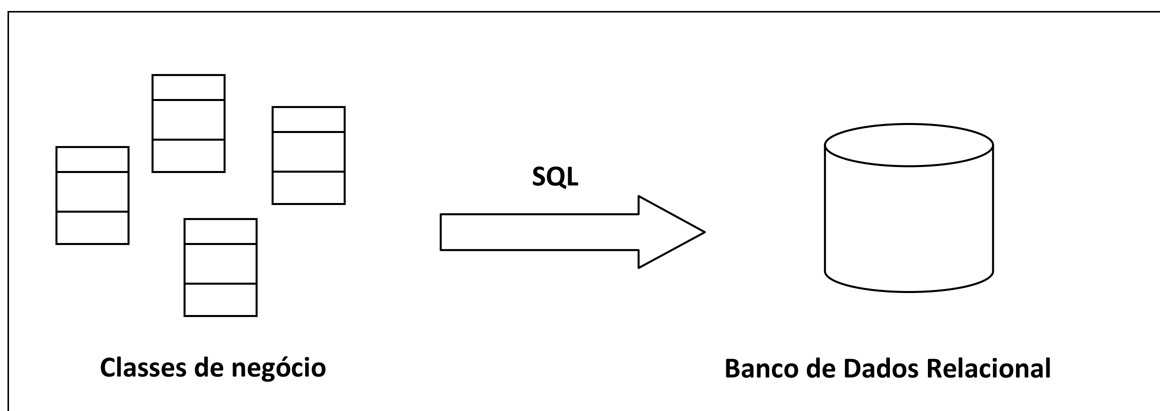


Figura 3.2 – Persistência de comandos SQL com regras de negócio (Ambler, 2005).

Na figura 3.3 a seguir é apresentada outra maneira de implementar a persistência de objetos em bancos de dados relacionais, onde são criadas novas classes que se diferenciam das classes de negócio, por terem como principal objetivo encapsular os comandos SQL, e por isso são chamadas de classes de dados. Esta solução apresenta praticamente as mesmas características que a opção anterior, porém Ambler (2005) destaca que ainda assim, pelo menos o código fonte que contem os comandos SQL que serão executados pelo SGBD ficam centralizados em um lugar diferente de onde se encontram as classes de domínio do problema.

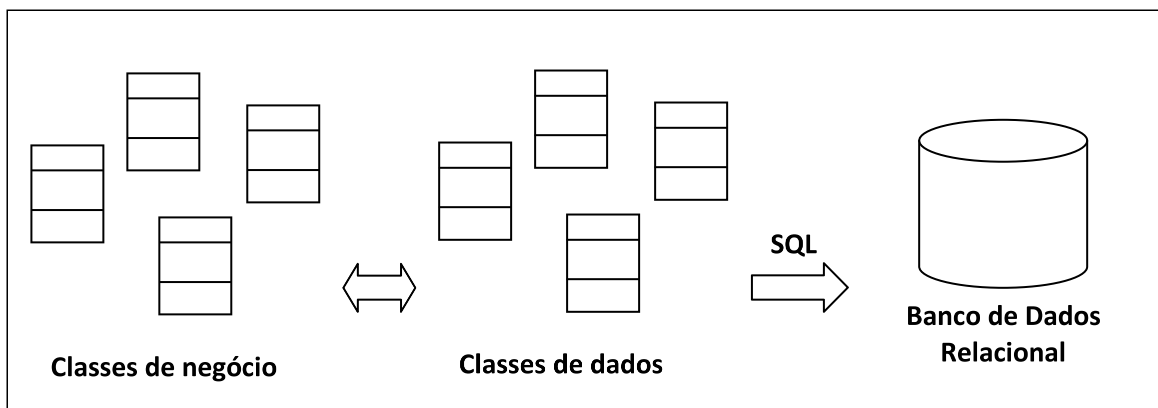


Figura 3.3 – Persistência encapsulando os comandos SQL em classes de dados (Ambler, 2005).

Esta segunda solução de implementação de persistência de objetos serve de base para o entendimento de três padrões apresentados por Fowler (2006) para mapeamento objeto-relacional, chamados de *Gateway* de Registro de Dados, *Gateway* de Tabela de Dados e Mapeador de Dados, pois estes possuem como objetivo principal de retirar do código fonte dos objetos de domínio do negócio quaisquer conhecimentos relacionados ao esquema de modelagem presente no banco de dados ou comandos SQL embutidos.

A diferença entre esses três padrões é que no *Gateway* de Registro de Dados, é modelada uma classe de dados, ou um *gateway*, específico para cada classe de negócio, com os atributos e métodos que representam e manipulam um único registro no banco de dados por vez, enquanto que no *Gateway* de Tabela de Dados, esta classe de dados contém todos os métodos para manipulação de mais de um registro na mesma tabela. Já o mapeador de dados é um padrão sugerido para aplicações mais complexas, que envolvem um modelo de negócio mais elaborado.

A terceira e última opção apresentada por Ambler (2005) para implementar a persistência de objetos é apresentada na Figura 3.4, a qual ele utiliza para demonstrar a sua modelagem para uma camada robusta de persistência. Conforme é demonstrado, o objetivo desta arquitetura é fazer com que, por exemplo, ao realizar uma alteração no banco de dados, não seja necessário realizar nenhuma alteração no código orientado a objetos das classes de negócio. Esta solução permite também que os programadores não precisem conhecer a estrutura do esquema modelado no banco de dados, nem sequer saber se seus objetos estão sendo armazenados em um banco de dados relacional ou não. Isto também permite que as

empresas possam construir aplicações cada vez maiores, porém destaca ainda como desvantagem o fato desse processamento impactar na performance da aplicação.

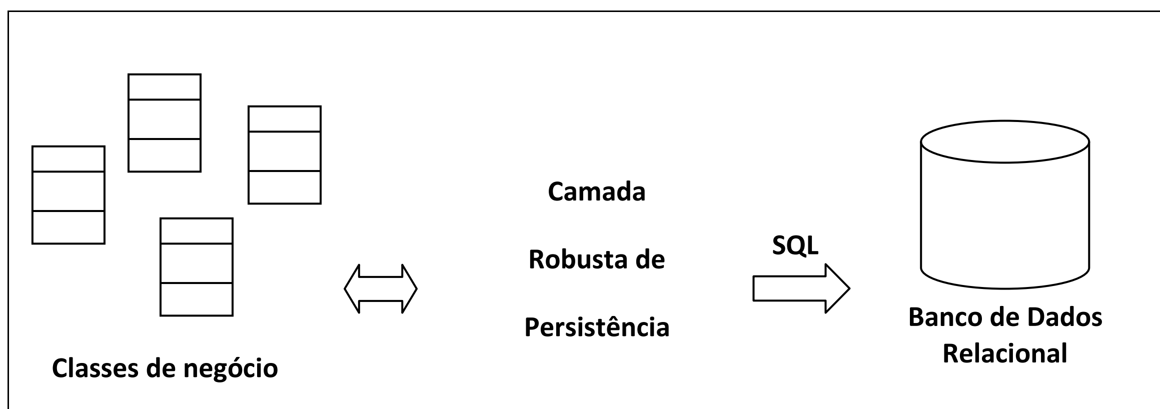


Figura 3.4 – Implementação de persistência utilizando uma camada robusta (Ambler, 2005).

Como pode ser visto, a tarefa de mapeamento objeto-relacional pode envolver muita codificação, apenas para descrever, por exemplo, quais colunas no banco de dados correspondem a quais atributos de objetos na aplicação. Fowler (2006) apresenta também um padrão chamado de Mapeamento de Metadados, o qual pode reduzir drasticamente o processo de codificação ao resolver esse problema utilizando código genérico para realizar as operações necessárias com o banco de dados, em conjunto com a informação de metadados dos objetos a serem persistidos e da respectiva estrutura no banco de dados.

Essas informações de mapeamento de metadados podem ser armazenadas tanto em arquivos fora da aplicação, utilizando XML[2] (Extensible Markup Language), por exemplo, como dentro do próprio banco de dados relacional. Isso permite, por exemplo, que uma alteração no esquema do banco de dados não necessite de uma nova compilação da aplicação, apenas uma alteração no respectivo arquivo de mapeamento de metadados.

Para implementar esse padrão, Fowler (2006) sugere como alternativas a geração automática do código fonte das classes de mapeamento ou programação reflexiva. Em resumo, ele afirma que a geração de código é uma opção menos dinâmica, por requerer

[2] O XML é um formato para criação de documentos com dados organizados de forma hierárquica. Pela sua portabilidade, já que não depende das plataformas de hardware ou de software, um banco de dados pode, através de uma aplicação, escrever em um arquivo XML e um outro banco distinto pode ler estes mesmos dados (XML, 2009).

alguma compilação durante uma alteração neste processo, enquanto que com a programação reflexiva é possível acessar atributos e métodos de um objeto a partir do nome desses atributos e métodos disponibilizados por um arquivo de metadados, para dinamicamente realizar o mapeamento desejado, porém ele ainda aponta como desvantagem o fato desta técnica poder apresentar alguma lentidão.

## *3.7 – Framework de Persistência*

Utilizando um framework de persistência existirá um mapeamento entre suas classes e as tabelas de um SGBDR (Sistema Gerenciador de Banco de Dados Relacional) de tal forma que uma simples mudança não levará a alterações em sua classe de negócio, como por exemplo, mudar o nome de algum campo.

Segundo Freire (2006), algumas vantagens com a utilização de um framework de persistência:

- Resolve os problemas que são causados com o método RAD[3] (Rapid Application Development).
- Faz um mapeamento entre as classes e as tabelas de seu SGBD, ou seja, isolando os acessos realizados diretamente ao banco de dados da aplicação.
- Garante aos desenvolvedores do software a total independência entre o modelo de objetos e o esquema de dados do banco, permitindo alterações no esquema de dados sem impacto para a aplicação.

[3] Metodologia de desenvolvimento de sistemas criada para diminuir o tempo necessário para projetar e implementar sistemas. É um modelo de processo de desenvolvimento de software iterativo e incremental que enfatiza um ciclo de desenvolvimento extremamente curto, onde o acesso ao banco de dados é feito juntamente com as funções da aplicação.

- Centraliza os processos de construção de consultas e operações de manipulação de dados em uma camada de objetos inacessível ao programador.

- O usuário do sistema de persistência de dados utiliza de mensagens de alto nível como save ou delete para lidar com a persistência dos objetos, deixando o tratamento destas mensagens para a camada de persistência em si, ou seja, as instruções SQL são criadas pelo framework.

- Os mecanismos de persistência permitem o armazenamento dos dados em outros tipos de base de dados, garantindo assim portabilidade entre as plataformas.

Ambler (2005) destaca ainda como o código fonte fica robusto ao facilitar sua manutenção devido ao desacoplamento das principais partes da aplicação, e também como essas camadas interagem umas com as outras ao trocarem informações. Enfim, ele também faz uma breve comparação entre comprar um produto pronto de terceiros ou desenvolver uma solução própria para realizar o processo de MOR, enfatizando que desenvolver e manter uma ferramenta destas não é uma tarefa fácil, e sugere que seu desenvolvimento não deve ser iniciado caso seu fim não possa ser alcançado.

# 4. Estudo de Caso: Reengenharia do módulo SIGA-Biblioteca

## *4.1 – Introdução*

Neste capítulo será descrito como o módulo Biblioteca do SIGA[4] (Sistema Integrado de Gestão Acadêmica) passou por um processo de reengenharia, visando adaptar-se aos avanços que as novas versões do MIOLO proporcionaram, em particular, a implantação da camada de persistência de objetos.

Segundo Matos e Gartner (2008), o MIOLO é um framework para criação de sistemas de informações acessíveis via web. Com um projeto modular, baseado em componentes, e uma arquitetura em camadas, o MIOLO atua como "*kernel*" de todos os sistemas criados. Favorecendo a reutilização, os vários sistemas podem ser facilmente integrados, funcionando como módulos de um sistema mais complexo. Além de proporcionar as funcionalidades para o desenvolvimento de sistemas, o MIOLO também define uma metodologia de codificação para que os resultados esperados sejam obtidos de forma simples e rápida, e, atualmente está na versão 2.0 beta 1.

O framework MIOLO tem por objetivo, ainda segundo Matos e Gartner (2008), a construção de sistemas de informação baseados na web, oferecendo a infra-estrutura necessária para que o desenvolvedor se preocupe apenas com o domínio da aplicação e não com os detalhes de implementação. Estes sistemas são construídos através do desenvolvimento de módulos. Um módulo é um componente de uma aplicação. De forma geral, um módulo reflete um subdomínio da aplicação, agregando as classes de negócio que estão fortemente relacionadas e provendo o fluxo de execução e a interface com o usuário para se trabalhar com tais classes. Resumidamente, algumas das principais funções implementadas pela versão mais recente do framework são:

- Controles de interface com o usuário;
- Autenticação de usuários;

[4] SIGA – Sistema Integrado de Gestão Acadêmica é o sistema que atualmente integra todos os processos informatizados da Universidade Federal de Juiz de Fora (UFJF).

- Controle de permissão de acesso;
- Camada de abstração para acesso a banco de dados;
- Camada de persistência transparente de objetos;
- Gerenciamento de sessões e estado;
- Manutenção de logs;
- Mecanismos de *trace* e *debug*;
- Tratamento da pagina como um *webform*;
- Validação de entrada em formulários;
- Customização de layout e temas;
- Geração de relatórios.

Através do módulo SIGA Biblioteca, os usuários podem consultar todo o acervo registrado nas bibliotecas da UFJF, seja de livros, monografias, teses, publicações avulsas, etc. Podem também realizar empréstimos de exemplares, reservas e acompanhar toda sua situação através da web.

## *4.2 – Reengenharia do módulo SIGA-Biblioteca*

Adotando o modelo de processo de reengenharia proposto por Pressman (2006), as atividades do processo de reengenharia de software se dividem em análise do inventário, reestruturação de documentos, engenharia reversa, reestruturação de programas e dados e engenharia avante. Nem sempre essas atividades ocorrem em uma maneira seqüencial, pois esse modelo permite que cada uma dessas atividades seja revisada num ciclo, além de poder terminar a qualquer momento, desde que a atividade em questão seja finalizada.

Seguindo esse ciclo, na análise do inventário não houve mudanças significativas, uma vez que a finalidade do módulo, bem como os níveis de usuários do sistema, não sofreu alteração.

A etapa de reestruturação de documentos foi bem produtiva. Apesar de já existirem alguns diagramas, estes estavam desatualizados e não mostravam a realidade do sistema. É comum criar registros em tabelas, novas classes para novas transações, e devido à necessidade de fazer as mudanças no menor tempo possível, a documentação tinha ficado desatualizada. Sendo assim, diagramas UML[5], como o de casos de usos (figura 4.1), de classe (figura 4.2) e de estados (figura 4.3) foram refeitos.

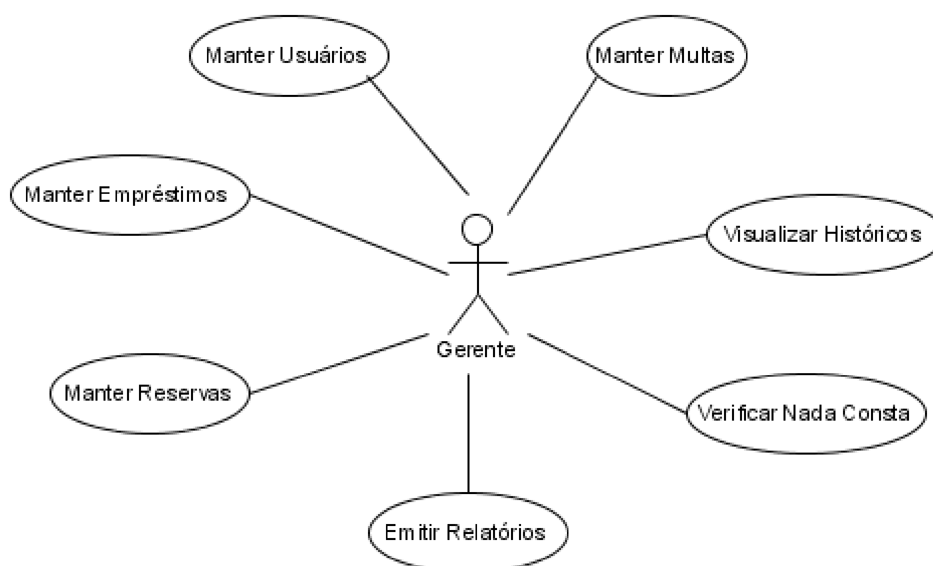


Figura 4.1 – Diagrama de casos de uso Gerente.

[5] UML – A Unified Modelling Language é uma linguagem de modelagem que permite que os desenvolvedores visualizem os produtos de seu trabalho em diagramas padronizados. No apêndice 1 é realizado uma visão geral sobre os principais elementos dessa linguagem.

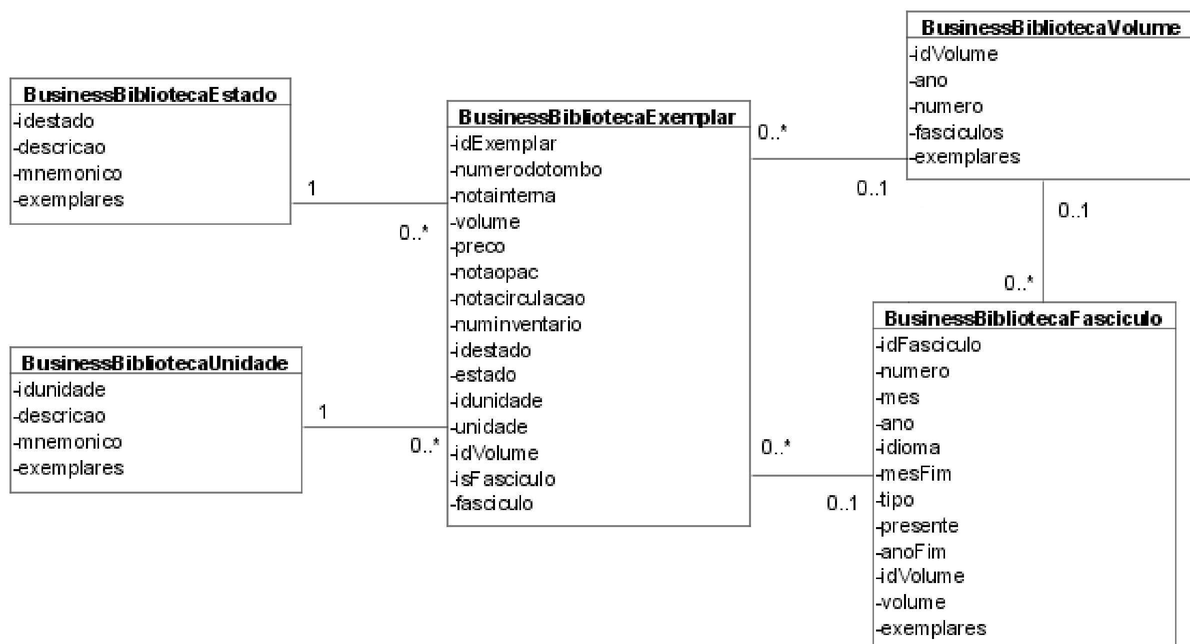


Figura 4.2 – Diagrama de classe Catalogação de Exemplar.

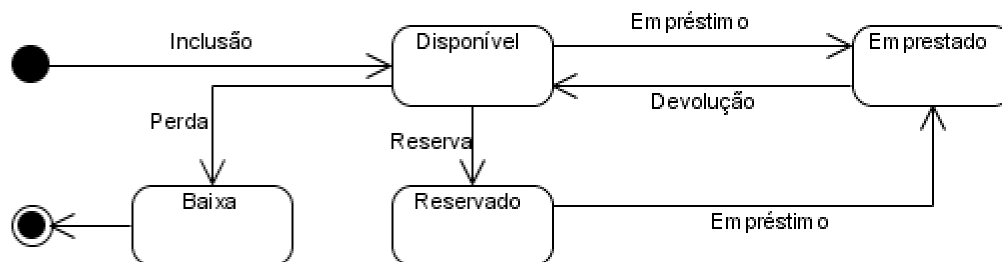


Figura 4.3 – Diagrama de estados Situação Empréstimo.

De acordo com Pressman (2006), a engenharia reversa de software é um processo de recuperação de projeto, consistindo em analisar um programa, levando em consideração o processamento, a interface com o usuário que é aplicado e as estruturas de dados ou banco de dados que são usadas. O propósito da engenharia reversa seria que, fornecendo uma listagem fonte, desestruturada e não documentada, fosse criada uma documentação completa para o programa.

Sendo assim, essa etapa consistiu em:

- Abstrair características específicas da implementação e da linguagem de programação;
- Abstrair detalhes da linguagem de programação a fim de revelar suas estruturas a partir de diferentes perspectivas;
- Relacionar partes dos programas às suas funções, procurando revelar as relações lógicas entre elas;
- Abstrair o contexto em que o sistema está operando.

O nível de abstração se refere à sofisticação da informação do projeto, que pode ser extraída do código fonte. A intenção é que esse nível de abstração seja o mais abrangente possível, ou seja, deve-se criar desde processos de baixo nível, até modelos entidade-relacionamento, pois quanto mais alto o nível de abstração, mais fácil se tornará para um engenheiro de software entender o programa (Pressman 2006). A figura 4.4 a seguir, representa um DER criado. O DER Circulação mostra como é feito o empréstimo de um exemplar, onde um usuário consulta sua disponibilidade, podendo reservá-lo, ou até mesmo gerando uma multa por atraso na devolução.

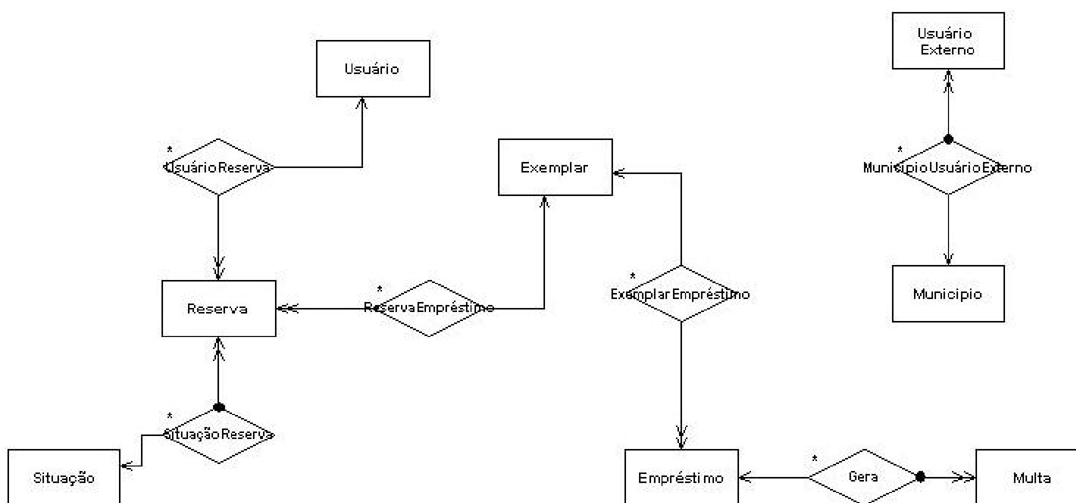


Figura 4.4 – DER Circulação.

Pressman (2006) sugere que a reestruturação de código seja o tipo mais comum de reengenharia. Não apenas programas que tenham uma arquitetura sólida, porém módulos individuais e difíceis de entender também podem ter de passar pela reestruturação de seu código. Realmente a etapa de reestruturação de código foi a mais trabalhosa, e após verificar e registrar as devidas alterações da programação atual, o resultado foi revisado e testado para garantir que nenhuma anomalia foi produzida.

A medida adotada para reestruturar o código do módulo SIGA-Biblioteca foi verificar o handler[6] acionado na operação e em seguida analisar o respectivo formulário, que são os responsáveis por criar a interface com o usuário. Ao longo dos formulários são usados objetos das classes de negócio, bem como funções relacionadas a estes.

Para Matos e Gartner (2008), no contexto da camada de persistência, o mapeamento diz respeito a como são representados os objetos, seus atributos e suas associações, em termos do modelo relacional. Nesta implementação estão sendo usados arquivos XML para representar o mapeamento das classes persistentes e das classes associativas. Foram criados mapas para as classes, para os atributos, para as associações, para as colunas e para as tabelas, permitindo uma grande flexibilidade no processo de mapeamento. A figura 4.5 mostra o mapeamento de uma das classes do sistema.

```
<?xml version="1.0" encoding="ISO-8859-1" standalone="yes"?>
<map>
  <moduleName>biblioteca</moduleName>
  <className>unidade</className>
  <tableName>bt_unidade</tableName>
  <databaseName>sigabib</databaseName>

  <attribute>
    <attributeName>idunidade</attributeName>
    <columnName>idunidade</columnName>
    <idgenerator>seq_bt_unidade</idgenerator>
    <key>primary</key>
  </attribute>
```

[6] Handlers são classes que representam a parte funcional da aplicação, criadas pelo desenvolvedor para fazer o tratamento dos dados enviados pelo cliente. Eles definem o fluxo e implementam os casos de uso. Em uma visão de interface, eles representam os menus.

```xml
  <attribute>
    <attributeName>descricao</attributeName>
    <columnName>descricao</columnName>
  </attribute>

  <attribute>
    <attributeName>mnemonico</attributeName>
    <columnName>mnemonico</columnName>
  </attribute>

  <attribute>
    <attributeName>exemplares</attributeName>
  </attribute>

  <association>
    <toClassModule>biblioteca</toClassModule>
    <toClassName>exemplar</toClassName>
    <cardinality>oneToMany</cardinality>
    <target>exemplares</target>
    <retrieveAutomatic>false</retrieveAutomatic>
    <saveAutomatic>false</saveAutomatic>
    <entry>
      <fromAttribute>idunidade</fromAttribute>
      <toAttribute>idunidade</toAttribute>
    </entry>
  </association>

</map>
```

Figura 4.5 – Mapeamento da classe unidade.

Seguindo Matos e Gartner (2008), são construídas tabelas que representam as associações utilizadas no mapeamento da classe unidade. A tabela 4.1 representa o elemento principal do mapa XML, que é denominado <map>. Já a tabela 4.2 que representa os atributos, que são definidos dentro da tag <atribute>. As associações representam os relacionamentos existentes entre as classes. Sao definidas a partir do ponto de vista da classe sendo mapeada. Suas caracteristicas estão dentro da tag <association>, conforme mostra a tabela 4.3.

Tabela 4.1 – Relacionamentos do elemento principal da classe

| | |
|---|---|
| <moduleName> | nome do módulo onde a classe de negócios está definida |
| <className> | nome da classe de negócios, dentro do módulo <moduleName> |
| <tableName> | nome da tabela que mapeia a classe. Uma classe pode ser mapeada para apenas uma tabela. |
| <databaseName> | nome da configuração do banco de dados |

Tabela 4.2 – Relacionamentos dos atributos da classe

| | |
|---|---|
| <attributeName> | nome do atributo conforme definido na classe |
| <columnName> | nome da coluna que representa o atributo na tabela <tableName>. Atributos que são usados apenas para representar a associação entre duas classes geralmente não têm colunas associadas. |
| <key> | indica, quando este atributo é uma chave da tabela, qual o tipo de chave. Pode assumir dois valores: primary e foreign. |
| <idgenerator> | quando o atributo é chave, esta tag pode indicar qual o nome do IdGenerator . |

Tabela 4.3 – Relacionamentos entre as classes

| | |
|---|---|
| <toClassModule> | nome do módulo da classe associada. |
| <toClassName> | nome da classe associada, dentro do módulo <toClassModule>. |
| <cardinality> | cardinalidade da associação. Refere-se a quantos objetos da classe associada estão relacionados com a classe sendo mapeada. Os valores possíveis são: oneToOne, oneToMany e manyToMany. |
| <target> | indica qual o atributo vai receber o objeto (ou o array de objetos) que for recuperado através da associação. É também o nome usado para identificar a associação. |
| <retrieveAutomatic><br><saveAutomatic><br><deleteAutomatic> | indica se as operações de recuperação, armazenamento e remoção (respectivamente) serão realizadas automaticamente na classe associada. Estas tags são opcionais e o valor default é false. Cuidado especial deve ser tomado nesta definição pois a recuperação (ou armazenamento ou remoção) de um simples objeto pode ocasionar a recuperação automática de dezenas de outros, dependendo do valor |

| | |
|---|---|
| | destas tags. De certa forma é usada para mapear (e sobrepor) as definições de integridade referencial declaradas no esquema relacional. |
| <entry> | Nas associações oneToOne e oneToMany descreve a associação entre os atributos da classe mapeada e os atributos da classe associada. Dentro da tag <entry>, <fromAttribute> indica o atributo da classe mapeada e <toAttribute> indica o atributo da classe associada. Podem existem várias tags <entry> (por exemplo, no caso de chaves compostas). |

Ainda segundo Matos e Gartner (2008), as consultas ao banco de dados são realizadas através de um mecanismo chamado "*query by criteria*". Através deste mecanismo, definimos um "*criteria*", ou seja, um critério para a consulta. O mecanismo de persistência está encapsulado de uma maneira que tornam os objetos de negócio virtualmente persistentes. Métodos como *save*, *delete* e *retrieve*, que tratam automaticamente o acesso ao banco de dados, estão disponíveis. A figura 4.6 mostra uma função de um formulário que trata de exclusão de obras.

```
function btnExcluir_click()
{
    global $MIOLO,$module;
    $objObra = $this->manager->getBusiness('biblioteca','obra',$this->idObra);

    try
    {
        $objObra->DeleteObra();

        $MIOLO->Information("Obra excluída com sucesso.");
    }
    catch (Exception $e)
    {
        $MIOLO->Error($e->getMessage());
    }
}
```

Figura 4.6 – Chamada para função de excluir obra.

A função btnExcluir_click() possui um objeto da classe obra (objObra), responsável por acionar a função DeleteObra().

```
function DeleteObra()
{
    global $MIOLO,$module;

    $commands = array();
    $sql = new sql('','bt_camposfixos','idobra = ?');
    $commands[] = $sql->delete($this->idObra);
    $sql = new sql('','bt_indicadores','idobra = ?');
    $commands[] = $sql->delete($this->idObra);
    $sql = new sql('','bt_material','idobra = ?');
    $commands[] = $sql->delete($this->idObra);
    $sql = new sql('','bt_exemplar','idobra = ?');
    $commands[] = $sql->delete($this->idObra);
    $sql = new sql('','bt_obra','idobra = ?');
    $commands[] = $sql->delete($this->idObra);

    $this->execute($commands);
}
```

Figura 4.7 – Função para excluir obra antes da reestruturação.

```
function DeleteObra ()
{
    global $MIOLO, $module;
    try
    {
        $this->retrieveAssociation('camposfixos');
        $this->retrieveAssociation('indicadores');
        $this->retrieveAssociation('materiais');
        $this->retrieveAssociation('exemplares');
        $this->beginTransaction();
        $this->delete();
        $this->endTransaction();
    }
    catch (Exception $e)
    {
        throw new Exception('Não é possível excluir');
    }
}
```

Figura 4.8 – Função reestruturada para exclusão da obra.

A figura 4.7 mostra como era a função DeleteObra() antes do processo de reestruturação de código enquanto a figura 4.8 mostra o resultado de como ficou após a reestruturação do código. Já outra função, a GetCategoria(), responsável por fazer uma consulta ao banco (a figura 4.9 mostra como era a função antes do processo de reestruturação de código), depois da etapa de reestruturação de código ficou como mostra a figura 4.10.

```
function GetCategoria()
{
    $sql = new sql('f.categoria', 'bt_ficha f,
                    bt_obra o, bt_genero g','(idobra = ?)
                    and (o.idgenero = g.idgenero)
                    and (g.idficha = f.idficha)');
    $query = $this->Query($sql,$this->idObra);
    return $query->result[0][0];
}
```

Figura 4.9 – Função GetCategoria() antes da reestruturação.

A função GetCategoria() seleciona a categoria da ficha buscando nas tabelas bt_ficha, bt_obra e bt_genero no banco de dados, atendendo aos critérios do campo idgenero ser o mesmo nas tabelas bt_obra e bt_genero, o campo idficha ser o mesmo nas tabelas bt_genero e bt_ficha e o campo idobra é informado pelo usuário, de uma maneira transparente, conforme selecionado na aplicação.

```
function GetCategoria ()
{
    $criteria = $this->getCriteria();
    $criteria->addColumnAttribute('genero.ficha.categoria');
    $criteria->addCriteria('idObra', '=', "$this->idObra");
    $query = $criteria->retrieveAsQuery();
    return $query->result[0][0];
}
```

Figura 4.10 – Função GetCategoria() reestruturada.

A tabela 4.4 mostra os métodos persistentes utilizados na função deleteObra(), mostrada na figura 4.8 e na função GetCategoria, mostrada na figura 4.10.

Tabela 4.4 – Funções dos objetos persistentes

| | |
|---|---|
| retrieveAssociation | Recupera os dados referentes a uma associacao e preenche o atributo correspondente com o objeto |
| delete | Remove um objeto no banco de dados. A remoção pode envolver uma ou mais exclusões ou atualizações, em uma ou mais tabelas no banco de dados. As associações podem ser automaticamente excluídas conforme a configuração feita no mapeamento. |
| getCriteria | Retorna um objeto usado para executar consultas customizadas no banco de dados |
| addColumnAttribute | Acrescenta um campo a ser recuperado na consulta. Se não for definido nenhum campo deta forma, todos os atributos da classe serão retomados. |
| addCriteria | Acrescenta uma operação a ser executada no critério. |
| retrieveAsQuery | Retorna uma query como resultado da consulta |

Terminada a etapa de reestruturação de código, segundo Pressman (2006), antes que se inicie a reestruturação dos dados deve-se realizar uma atividade chamada análise de código fonte, ou simplesmente análise de dados. As declarações contendo as definições dos dados, descrições de arquivos, entradas e saídas e descrições de interfaces foram avaliadas, a fim de retirar itens de dados e objetos para obter informações sobre o fluxo de dados e entender as estruturas implementadas.

Após a análise de dados, se inicia uma etapa de padronização de registro de dados, esclarecendo definições de dados para obter consistência entre os itens. A padronização do código não consistiu uma etapa propriamente dita. Procurou-se atingir essa padronização naturalmente ao longo da reestruturação de código, mas não houve depois uma análise detalhada.

A etapa engenharia avante, segundo Pressman (2006), recupera informações de projetos do software e as utiliza para alterar ou reconstruir o sistema existente. Geralmente, o

software trabalhado por reengenharia implementa funções do sistema existente e adiciona novas funções, melhorando o seu desempenho geral. Na maioria dos casos a engenharia avante não cria simplesmente um equivalente moderno de um programa antigo. Requisitos novos do usuário e tecnologia são integrados no esforço de reengenharia. O programa redesenvolvido amplia a capacidade da aplicação antiga.

No caso do módulo Biblioteca apenas foram implementadas funções já existentes no sistema, ou seja, a etapa de engenharia avante não foi desenvolvida, uma vez que a atividade em questão já havia sido finalizada com sucesso.

# 5. Considerações Finais

A forma de realizar a reengenharia em um software depende de muitos fatores, relacionados, por exemplo, com a forma de trabalho da empresa, a origem do software em questão e com o desenvolvimento de tecnologias para apoiar essa atividade. A importância da reengenharia está em possibilitar que todo conhecimento agregado ao software não seja perdido, o que aconteceria se a opção fosse pelo desenvolvimento de um novo software.

O processo de extração do conhecimento de um software é realizado pela engenharia reversa, que fornece visões mais abstratas do que o código do sistema ou alguma outra documentação possivelmente existente. Com essas visões é possível compreender o software e o sistema em que está inserido, possibilitando a produção de modelos de análise que servirão à reengenharia.

Da reengenharia de um software resulta uma nova versão, que agrega toda a informação do software legado, as inovações tecnológicas e novas funcionalidades desejadas. Dessa forma, a reengenharia de software não trata apenas de modernizá-lo, mas também adaptá-lo de acordo com as novas necessidades do processo em que está inserido.

Esse trabalho mostrou o acompanhamento do processo de reengenharia de software adotado no módulo SIGA-Biblioteca. O estudo já foi concluído, uma vez que o sistema passou por etapas que foram consideradas satisfatórias, onde houve uma reestruturação de documentação e, principalmente, de código fonte, a fim de aproveitar as vantagens proporcionadas pela camada de persistência de dados, que passou a ser efetivamente utilizada.

Para trabalhos futuros vislumbra-se a possibilidade de continuidade no ciclo da reengenharia. Novas mudanças acontecem à medida que o sistema cresce, seja por algum campo a mais em uma tabela no banco de dados, ou até mesmo uma mudança estrutural.

# Referências Bibliográficas

AMBLER, Scott W. The Design of a Robust Persistence Layer For Relational Databases. 2005. Disponível em: < http://www.ambysoft.com/essays/persistenceLayer.html >. Acesso em: junho de 2009.

ANSI, SQL-92 Standard. Disponível em <http://www.sqlteam.com/article/ansi-sql-92-standard>. Último acesso em junho de 2009.

BAUER, Christian; KING, Gavin. Hibernate em Ação: Arquitetura em Camadas. Ed. Rio de Janeiro: Ciência Moderna, 2005. 560p.

CANTU, Marco. Mastering Delphi 7. Alameda, California: Sybex, 2003. 1011 p.

CAMEIRA, Renato Flórido. Hiper-Integração: Engenharia de Processos, Arquitetura Integrada de Sistemas Componentizados com Agentes e Modelos de Negócios Tecnologicamente Habilitados. Rio de Janeiro, 2003. 432p. Dissertação (Doutorado em Engenharia de Produção) – COPPE, Universidade Federal do Rio de Janeiro.

CHIKOFSKY, E. J.; CROSS, J. H.. Reverse Engeneering and Design Recovery: A Taxonomy. IEEE Software, v. 7, n. 1, 1990, p. 13-17.

COSTA, R. M.. Aplicação do Método de Engenharia Reversa FUSION-RE/I na Recuperação da Funcionalidade da Ferramenta de Teste PROTEUM. São Carlos: ICMSC-USP, 1997. Dissertação (mestrado).

DALL' OGLIO, Pablo. PHP Programando com Orientação a Objetos. 1ed. São Paulo, SP: Novatec, 2007. 576p.

DATE, C. J.. Introdução a Sistemas de Banco de Dados. 8ed. Rio de Janeiro, RJ: Campus, 2004. 900p.

FOWLER, Martin. Padrões de Arquiteturas de Aplicações Corporativas. 1 ed. Porto Alegre: Bookman, 2006. 492p.

FREIRE, Herval. Mapeando Objetos para Bancos de Dados Relacionais: técnicas e implementações. 2006

GARCIA, Vinicius Cardoso; PRADO, Antonio Francisco do. Phoenix: Uma Abordagem para Reengenharia de Software Orientada a Aspectos. São Carlos [2005?] – Universidade Federal de São Carlos.

LANS, Rick F. Van Der. Introduction to SQL: Mastering the Relational Database Language. 4th. Addison Wesley Professional, 2004. 1056p.

LARMAN, Craig. Applying UML and patterns: an introduction to object oriented analysis and design and iterative development. 3rd ed. Upper Saddle River, N.J.: Prentice Hall PTR, 2007. 703 p.

MATOS, Ely Edison; GARTNER, Vilson. MIOLO 2.0 UserGuide. 2009. 51p.

OMG (Object Management Group). Unified Modeling Language Specification – Versão 2.2. Fevereiro de 2009

PIATTINI, Mario G.; DIAS, Oscar. Advanced database technology and design. Norwood, Massachusetts: Artech House, 2000. 558p.

PRESSMAN, Roger S.. Engenharia de Software. 6.ed. São Paulo: Makron Books, 2006. 1056p.

SOMMERVILLE, Ian. Engenharia de Software. 8.ed. São Paulo: Addison Wesley, 2007. 568p.

Extensible Markup Language. Abril de 2009. Disponível em <http://www.w3.org/XML/> Último acesso em junho de 2009.

# Apêndice I

## *Visão Geral sobre UML*

Segundo OMG (2009), a UML é uma linguagem para especificação, visualização, construção e documentação de artefatos de sistemas de software, assim como para modelagem de outros tipos de sistemas que não softwares. Ela representa uma coleção das melhores praticas de engenharia que se comprovaram eficientes para o projeto de sistemas amplos e complexos. Adotada pela comunidade de desenvolvimento OO como padrão para a criação de modelos e diagramas conceituais, a UML proporciona também a criação de variações através de um mecanismo de extensão próprio.

Os artefatos principais da linguagem são representações gráficas de componentes e formas de relacionamento de um sistema. As etapas de concepção, especificação, construção, testes e entrega de uma solução podem ser descritas através deste conjunto de diagramas, que suportam conceitos de alto nível (estrutura, padrões e componentes). É importante ressaltar também que a UML não substitui nem está atrelada a qualquer linguagem de programação especifica. Para a modelagem de sistemas e soluções OO, a UML possui grande destaque por apresentar-se como uma alternativa completa. A UML é uma linguagem cheia de recursos, que permite não apenas capturar as informações, mas também expressá-las de forma clara e objetiva (OMG, 2009).

## *A.1 Principais Elementos*

### A.1.1 Pacotes

Pacotes são elementos responsáveis por agrupar outros elementos. Um pacote UML pode conter qualquer outro elemento, inclusive outros pacotes. Todos os diagramas da linguagem podem ser organizados em pacotes. Na figura A.1 tem-se um exemplo de pacote.

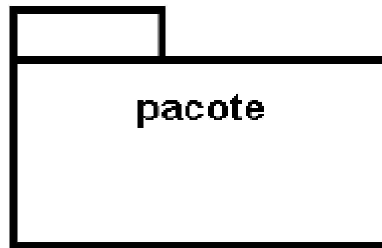


Figura A.1 – Exemplo de Pacote em UML.

### A.1.2 Classes

Classes definem elementos com estrutura, relacionamentos e comportamento similares. A UML oferece notação para a declaração de classes e suas propriedades assim como para a sua utilização em diversas situações. A figura A.2 representa diferentes classes na linguagem UML. A notação de classe possui algumas restrições como: classes e operações abstratas devem ter o nome em itálico, todos os atributos e operações devem ter o nome iniciado em letras minúsculas etc..

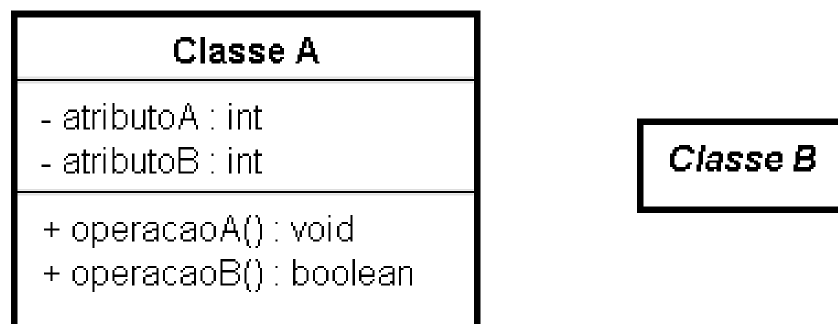


Figura A.2 – Exemplo de Classe em UML.

### A.1.3 Atores

Representam grupos de entidades, não necessariamente pessoas, que são aptas para realizar uma tarefa determinada por outra entidade. São representados por bonecos como o da figura A.3.

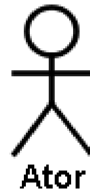


Figura A.3 – Exemplo de Ator em UML.

### A.1.4 Caso de Uso

Uma funcionalidade de uma classe, de um sistema ou de outro elemento pode ser representada como um caso de uso. Casos de uso devem possuir um nome capaz de identificá-lo e são apresentados como elipses com este nome em seu centro. Um exemplo desta representação gráfica é apresentado na figura A.4.

Figura A.4 – Exemplo de Caso de Uso em UML.

## *A.2 Principais Diagramas*

### A.2.1 Diagrama de Classes

Representam a estrutura estática de um sistema, incluindo seus componentes, a estrutura interna destes e as relações entre eles. A figura A.5 apresenta um diagrama de classes simplificado.

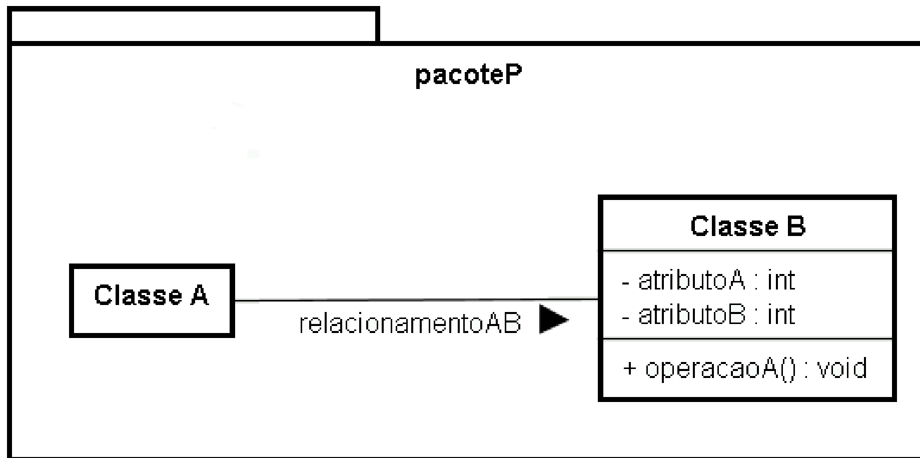


Figura A.5 – Exemplo de Diagrama de Classes em UML.

### A.2.2 Diagrama de Caso de Uso

Mostra os relacionamentos entre os atores de um sistema e outras entidades. Estas entidades podem representar funcionalidades de uma classe ou um pacote, por exemplo. A notação deste diagrama é simples e apresenta basicamente os atores relacionados diretamente com as entidades esperadas de acordo com o sistema. Pode-se usar um retângulo para determinar a ligação entre um conjunto de atores e entidades a algum requisito. Na figura A.6 um diagrama de caso de uso é ilustrado.

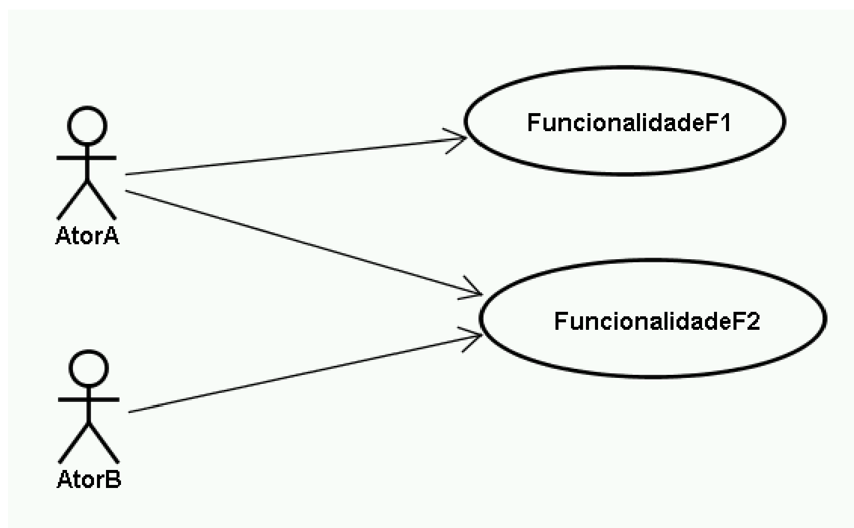


Figura A.6 – Exemplo de Diagrama de Caso de Uso em UML.

### A.2.3 Diagrama de Estados

Em um diagrama de estados, um objeto possui um comportamento e um estado. O estado de um objeto depende da atividade na qual ele está processando. Um diagrama de estado mostra os possíveis estados de um objeto e as transações responsáveis pelas suas mudanças de estado. Na figura A.7 um diagrama de estados é ilustrado.

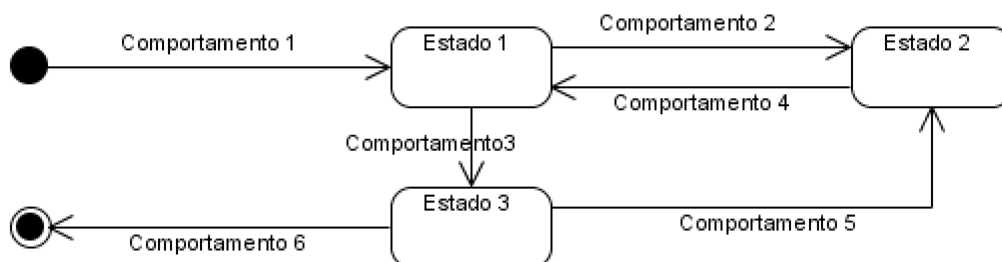


Figura A.7 – Exemplo de Diagrama de Estados em UML.