Alternativas de Reengenharia para um Sistema Implementado em C

Maria Istela Cagnin[1]
Rosângela D. Penteado

Departamento de Computação - DC
Universidade Federal de São Carlos - UFSCar
Caixa Postal 676 CEP 13.565-905
São Carlos - SP - Brasil
{mcagnin,rosangel}@dc.ufscar.br

Resumo

Três diferentes alternativas de reengenharia são apresentadas neste trabalho. Todas preservam a funcionalidade do sistema legado, porém alteram o paradigma de desenvolvimento do procedural para o orientado a objetos. O enfoque especial é dado para a reengenharia que troca a linguagem de implementação de C para Java e utiliza os padrões de projeto para mapear os objetos para o banco de dados relacional. Em um dos processos de reengenharia há troca de linguagem C para C++, sem utilizar padrões de projeto. No outro, a linguagem C é preservada, mas o código é adaptado para simular os conceitos de orientação a objetos. Essas alternativas auxiliam o engenheiro de software na tomada de decisão quanto às possíveis alterações que o código fonte legado possa sofrer. As alternativas de reengenharia são apoiadas pelos modelos orientados a objetos elaborados no processo de engenharia reversa com a abordagem Fusion/RE.

Palavras-Chaves: Engenharia Reversa Orientada a Objetos, Reengenharia, Orientação a Objetos, Padrões de Projeto CRUD.

Abstract

Three different reengineering alternatives are presented on in this work. All three preserve the funcionality of the legacy system, however, they change the development paradigm from conventional to object oriented. Special attention is given to reengineering, that changes the programming language from C to Java and uses the design pattern to map objetcs to a relational data base. In one of the reengineering processes there is a change from C to C++ language, without using design patterns. In another, C language is preserved but the code is adapted to simulate the concepts of object orientation. These alternatives help the software engineer in decision-making regarding the possible changes that way be made to the legacy source code. The reengineering alternatives are supported by object-oriented models constructed in the reverse engineering process with the Fusion/RE approach.

Key-Words: Object-Oriented Reverse Engineering, Reengineering, Object Oriented, CRUD Design Patterns.

[1] Bolsita da FAPESP Nº Proc. 97/12208-0

1. Introdução

A fase de manutenção de software é a mais temida pois é necessário entender e alterar o código fonte, que na maioria das vezes, foi desenvolvido por outras pessoas, com diferentes estilos de programação. A troca desses sistemas, muitas vezes é inviável tanto financeiramente quanto operacionalmente, pode gerar transtornos, exigir treinamento e adaptação dos usuários ao novo sistema, uma vez que estão acostumados com a sua operação. Visando amenizar esses problemas, este trabalho propõe formas alternativas no processo de reengenharia em que a funcionalidade do sistema é sempre mantida. Após a engenharia reversa um sistema pode passar pelo processo de reengenharia, preservando a sua funcionalidade, das seguintes formas: mudança no paradigma de desenvolvimento, porém conservando a linguagem original de implementação (reengenharia com mudança de orientação-segmentação do código), mudança da linguagem de implementação (reengenharia com mudança de linguagem de implementação) e a combinação das duas anteriores (reengenharia com mudança de orientação e da linguagem de implementação).

Neste trabalho são apresentadas as três formas acima descritas. Um sistema real é utilizado nos três casos o que permite apontar os pontos positivos em cada um deles, facilitando a escolha de qual forma pode ser aplicada.

O sistema real foi desenvolvido no paradigma procedural utilizando linguagem C. O paradigma, agora escolhido, é o orientado a objetos e as linguagens, além do C, são: C++ e Java. Os padrões de projeto, também, são utilizados para realizar o mapeamento de objetos para banco de dados relacional na implementação em Java. Os modelos orientados a objetos do sistema, utilizados nos três casos de reengenharia, são os obtidos no processo de engenharia reversa com a abordagem Fusion/RE [Pen95, Pen96, Pen96a].

Este trabalho está organizado da seguinte forma: a seção 2 apresenta uma breve descrição da abordagem Fusion/RE e sua aplicação em um sistema real que, em seguida, sofre reengenharia; a seção 3 introduz os conceitos dos padrões de projeto utilizados na reengenharia com a linguagem Java. A seção 4 comenta como a fase de reengenharia é realizada e os principais resultados obtidos e; na seção 5 são exibidas as conclusões.

2. Abordagem de engenharia reversa utilizada

Diante da dificuldade em expandir e manter sistemas legados a abordagem Fusion/RE foi elaborada, permitindo que abstrações orientadas a objetos sejam feitas a partir de código fonte procedural. Os modelos de análise obtidos seguem os preconizados por Coleman no método Fusion [Col94].

A abordagem Fusion/RE é composta de quatro passos. No primeiro passo, após um estudo detalhado do código fonte do sistema legado, revitaliza-se sua arquitetura, recuperando-se as informações sobre os procedimentos, pela técnica Chama/Chamado. No segundo passo, é elaborado o Modelo de Análise do Sistema Atual (MASA), sendo o modelo de objetos elaborado a partir das estruturas de dados existentes no código fonte do sistema legado. Os modelos de ciclo de vida e de operações são derivados da interface existente. Esse modelo de análise é pseudo-orientado a objetos, uma vez que as classes consideradas são, na maioria, as próprias estruturas de dados existentes no código fonte do sistema legado. No terceiro passo, a partir da abstração do MASA, o engenheiro de software elabora o Modelo de Análise do Sistema (MAS) considerando as classes e os métodos que realmente representam o sistema. No quarto passo é realizado um mapeamento das classes e dos métodos do MAS para o MASA, para que os procedimentos que consultam e/ou modificam diversas

estruturas de dados sejam divididos em métodos de forma que cada um consulte ou modifique somente uma estrutura de dados. Por meio desse mapeamento obtém-se quantos e quais métodos serão necessários para implementar os antigos procedimentos e quais elementos de dados permanecem como atributos das classes. Ressalta-se aqui que há correspondência biunívoca somente para os procedimentos que alteram ou consultam apenas uma estrutura de dados. As informações obtidas do mapeamento do MAS para o MASA, juntamente com o modelo MAS, apóiam o processo de reengenharia do sistema. Mais detalhes sobre essa abordagem podem ser obtidos em [Pen96, Pen96a].

O sistema real utilizado é o ambiente StatSim (*Statecharts Simulator*). Este sistema é um ambiente para edição e simulação de *statecharts* [Har87, Har90], desenvolvido em estações de trabalho *Sun,* utilizando a linguagem C e *Xview*, com aproximadamente 30.000 linhas de código. Esse ambiente passou pelo processo de engenharia reversa utilizando a abordagem Fusion/RE pela necessidade e dificuldade em realizar manutenções e expansões, devido à carência de documentação.

A Figura 1 mostra o modelo de objetos do ambiente StatSim, produzido pelo passo 2 da abordagem Fusion/RE, correspondente ao MAS. O Quadro 1 exibe alguns procedimentos e quais métodos são necessários para implementá-los. Os procedimentos e métodos possuem uma classificação conforme a legenda contida no quadro. As classes da Figura 1 e os métodos do Quadro 1 são os considerados no processo de reengenharia.

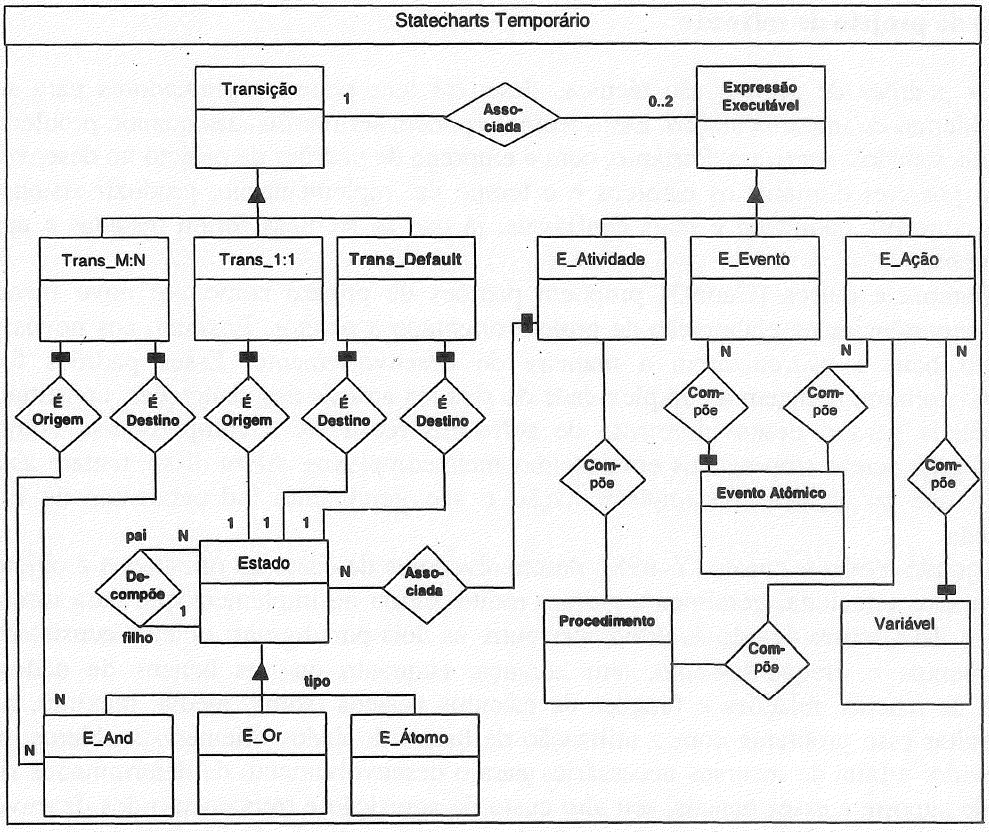

Figura 1 - Modelo de objetos do MAS para o tema statecharts temporário

Quadro 1 - Procedimentos parciais da classe Estado

Classe: Estado			
Arquivos que os Procedimentos do MASA estão inseridos	**Procedimento MASA**	**Método MAS**	**Classe/Tema[2]**
rot_simula.h	Acha_anc_comum (o)	achar_ancestral_comum (o)	Estado/Statecharts Temporário
rot_simula.h	Acha_ancest (o)	achar_ancestral (o)	Estado/Statecharts Temporário
rot_simula.h	Acha_ancest_dest (o+)	achar_ancestral_destino (o) marcar_ancestral_destino (o)	Estado/Statecharts Temporário Trans_MN/Statecharts Temporário
rot_simula.h	Busca_tr_ok_arv (o+)	verificar_est_transicao_ok (o) verificar_trans_ok (o)	Estado/Statecharts Temporário Transicao/Statecharts Temporário
rot_simula.h	Busca_tr_ok (o)	buscar_transicao_ok (o)	Transicao/Statecharts Temporário
bolha1.h	apagar_bolha (c+)	remover_estado (c) remover_bolha (c)	Estado/Statecharts Temporário Bolha/Diagrama

Legenda:
(o) = observa uma estrutura de dados; (o+) = observar duas ou mais estruturas de dados, (c) = altera uma estrutura de dados,
(c+) = altera duas ou mais estruturas de dados, (oc)= observa uma estrutura de dados e altera outra,
(oc+) = observa uma estrutura de dados e altera duas ou mais.

3. Padrões de projeto de software

Os padrões de projeto são técnicas desenvolvidas pelos programadores para solucionarem alguns problemas de implementação. Essas técnicas podem ser reutilizadas quando problemas similares àqueles solucionados, surgirem. Portanto, com o emprego de padrões de projeto no desenvolvimento de software é possível diminuir os esforços e o tempo de implementação, produzir sistemas flexíveis, reusáveis, com alta qualidade e mais confiáveis, já que as técnicas foram testadas e aprovadas por outros sistemas.

Gamma e outros [Gam93] propõem padrões de projeto como um novo mecanismo para expressar experiências na elaboração de projeto orientado a objetos. Ensinam aos novos projetistas a trabalharem bem e padronizarem a maneira de desenvolvimento. Esses padrões fornecem um vocabulário comum, reduzem a complexidade do sistema através das abstrações, constituem uma base de experiência para o desenvolvimento de softwares reusáveis e comportam-se como blocos de construção para serem empregados em sistemas mais complexos. Além disso, tentam abstrair mais o projeto do que as técnicas de implementação e são geralmente independentes da linguagem de programação.

Segundo Yoder e outros [Yod98], desenvolvedores de sistemas orientados a objetos que usam banco de dados relacional geralmente gastam muito tempo na implementação para tornar os objetos persistentes. Isto ocorre devido às diferenças entre os dois paradigmas: objetos consistem de dados e comportamentos e, freqüentemente, têm herança; enquanto que os bancos de dados relacionais consistem de tabelas, relações e funções de cálculos básicos (soma, média, máximo, mínimo, etc). Pode-se evitar esse problema com a utilização de banco de dados orientado a objetos, mas isso não ocorre devido: a falta de recursos necessários para o desenvolvimento de determinados sistemas, pela carência de suporte a esses bancos, seu alto custo de aquisição e treinamento dos desenvolvedores. A implementação de uma Camada de Persistência, isto é, uma camada para proteger a aplicação e/ou

[2] Tema = conceito de assuntos em orientação a objetos. Neste exemplo: statecharts temporário e diagrama (interface)

banco de dados de mudanças impróprias, é uma solução para minimizar o mapeamento de objetos em banco de dados relacionais. Essa camada deve suportar as operações CRUD: inserção, deleção e recuperação de registros que utilizam descrições em SQL. Todo objeto necessita, no mínimo, dessas operações para que possam ser persistidos. Yoder e outros indicam a possibilidade de implementar essa Camada de Persistência de três modos:

1. implementar as operações CRUD nas classes da aplicação;
2. criar classes específicas, para cada classe da aplicação, para implementar as operações CRUD;
3. criar uma classe *Broker* para fazer o mapeamento de qualquer tipo de objeto para o banco de dados, isto é, por intermédio das operações CRUD essa classe deve criar automaticamente SQL's para cada tipo de objeto que se deseja recuperar ou atualizar no banco de dados.

4. Alternativas de reengenharia

Segundo Jacobson [Jac91] existem diversos tipos de reengenharia: sem troca da funcionalidade, quer seja com a troca parcial ou completa da linguagem de implementação, com troca do paradigma de desenvolvimento e até mudança total da funcionalidade, que é engenharia avante.

Neste trabalho a reengenharia é realizada de três formas distintas, em todas preserva-se a funcionalidade do sistema legado e troca-se o paradigma de desenvolvimento para o orientado a objetos. Essas formas são: a) segmentação do código fonte do sistema legado; b) troca da linguagem de implementação para C++ e c) troca da linguagem de implementação para Java utilizando padrões de projeto e banco de dados relacional.

O ambiente StatSim é submetido a essas três formas que são comentadas a seguir:

a) Segmentação do código fonte do sistema legado.

O processo de segmentação do código fonte do sistema legado mantém a implementação em linguagem procedural com características orientadas a objetos. Esse processo é realizado em passos que auxiliam os mantenedores de sistemas a organizarem o código fonte de forma a facilitar a sua manutenção e seu entendimento. O completo detalhamento dos passos encontra-se [Cag99, Pen99] para sistemas implementados em C, porém eles podem ser adaptados para outras linguagens procedurais. A partir do código fonte do sistema legado, das classes consideradas, Figura 1, e do Quadro 1, a reengenharia é iniciada. No processo de segmentação foram utilizados o comando utilitário *grep* e os recursos do editor de texto, ferramentas disponíveis no sistema operacional Unix, para auxiliar a transformação dos procedimentos em métodos. No experimento realizado, o ambiente StatSim permanece com a linguagem de implementação original, isto é, C, mas com características de orientação a objetos, como: criação de métodos para realizar o encapsulamento de dados, eliminação de variáveis globais, transformação de procedimentos em métodos, criação de módulos específicos para cada classe, simulação de herança entre as classes, uma vez que a linguagem C não suporta esse conceito de orientação a objetos.

Após esse processo, o código fonte segmentado tem, em torno, de 15% a mais de linhas de código que o código fonte legado, porém é mais legível e fácil de ser mantido. Cada método manipula somente uma classe (estrutura de dados). Há uma parte do código fonte legado, em torno de 15%, que não é transformada em orientação a objetos devido a utilização do *Xview* e de conceitos de orientação a objetos, como polimorfismo e herança, que não são suportados pela linguagem C.

b) Troca da linguagem de implementação para C++.

O processo de reengenharia com troca da linguagem de implementação para linguagem orientada a objetos, sem a utilização de padrões de projetos, é descrito em detalhes em [Biz97, Hen97]. O ambiente StatSim foi transformado parcialmente da linguagem procedural C para C++. Essa transformação utiliza o modelo de análise obtido no processo de engenharia reversa e os modelos de projeto desenvolvidos de acordo com os preconizados por Coleman, no método Fusion. Como *XView* é usado no ambiente StatSim há necessidade de criar métodos "fantasmas" para que as suas funções de biblioteca sejam utilizadas. Ou seja, o código *XView* é utilizado juntamente com o código C++. Jacobson [Jac91] afirma que a reengenharia de um sistema com mudança de linguagem pode ocorrer de forma parcial e gradativa, desde que haja compatibilidade entre elas, isso pôde ser comprovado nesse experimento com as linguagens C e C++. Ou seja, parte do sistema continua implementada em C e é ativada pelos métodos implementados em C++.

A utilização da linguagem C++ permite a criação de métodos efetivos e não apenas a simulação desses, como ocorre com a segmentação do sistema. O sistema todo pode ser implementado em C++ o que reduziria o número total de linhas de código. A conservação da parte gráfica usando *XView* permitiu a integração de linguagens de paradigmas diferentes.

c) Troca da linguagem de implementação para Java utilizando padrões de projetos e banco de dados relacional.

A versão original do ambiente StatSim armazena os *statecharts* editados em arquivos textuais, nesta reengenharia optou-se pela mudança da forma de armazenamento. As informações dos *statecharts* editados são armazenadas no Banco de Dados Relacional SyBase, pois essa forma de armazenamento é mais segura, em relação às falhas, do que a em bancos orientados a objetos.

Neste experimento são utilizados os padrões de projeto que fazem o mapeamento dos objetos para um banco de dados relacional. Para realizar esse mapeamento, implementa-se uma Camada de Persistência, por meio de operações CRUD (inserção, deleção e recuperação). Como foi comentado na seção anterior, existem diversos modos de implementar essa Camada de Persistência [Yod98], aqui ela será feita de duas maneiras: 1) implementar as operações CRUD nas classes da aplicação e 2) criar classes específicas, para cada classe da aplicação, para implementar as operações CRUD.

Os dois tipos de implementação da Camada de Persistência utilizam a classe `ManagerDB`, que é criada para estabelecer e encerrar a conexão com o banco de dados, pelos atributos `url`, `con` e `stmt`; e pelos métodos `ConnectionDB` e `CloseDB`. Essa classe é implementada em Java, como mostra a Figura 2. A aplicação tem apenas um banco de dados conectado em um determinado momento, logo há somente um objeto de `ManagerDB` durante o tempo de sua execução.

A Figura 3 mostra o relacionamento entre as classes `ManagerDB` e `Estado`. A classe `Estado` cria os métodos específicos para o primeiro modo da implementação da Camada de Persistência.

As operações CRUD são implementadas em cada classe da aplicação sendo que para cada uma cria-se, no banco de dados, uma tabela correspondente. Os métodos específicos (`insertDB`, `deleteDB`, `updateDB`, `findallDB`, `findlikeDB`) que implementam as operações CRUD devem ser privados, isto é, somente os métodos da classe podem alterar informações no banco de dados. Esses métodos utilizam SQL para realizar o mapeamento dos objetos para o banco. A Figura 4 exibe a implementação desses métodos. Como pode ser visto no método `insertDB`, a linha de SQL

chamada `insertSQL` é criada e logo, em seguida, o método `executeUpdate`, do objeto `ManagerDB`, é chamado para realizar uma atualização no banco. Nesse caso, esse método é utilizadc para inserir um objeto da classe `Estado`. Além disso, ele pode também ser usado para modificação ou deleção de dados.

```
public class ManagerDB{

  static String url;
  static Connection  con;
  static Statement   stmt;

 // Conexao - carrega o driver e realiza a
 conexao com o banco
 public static boolean ConnectionDB(String
 DBName, String User, String Password)
 {
  try {
  url = "jdbc:odbc:" + DBName;
  Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
  con = DriverManager.getConnection(url, User,
  Password);
  stmt = con.createStatement();
  con.setAutoCommit(true);
  System.out.println("Database OPENED !");
  Return true;
  }
```

```
catch (Exception e) {
   e.printStackTrace();
   System.out.println("Problema no open do
   Database!");
   Return false;
  }
 }

  // encerra o statement e a conexao
  public static void CloseDB(){
   try {
      stmt.close();       // encerra objeto
statment
      con.close();        // encerra objeto
connection
      System.out.println("Database CLOSED !");
   }
   catch (Exception e) {
      e.printStackTrace();
      System.out.println("Problema no close do
Database!");
  }
 }
```

Figura 2 – Implementação da classe `ManagerDB`

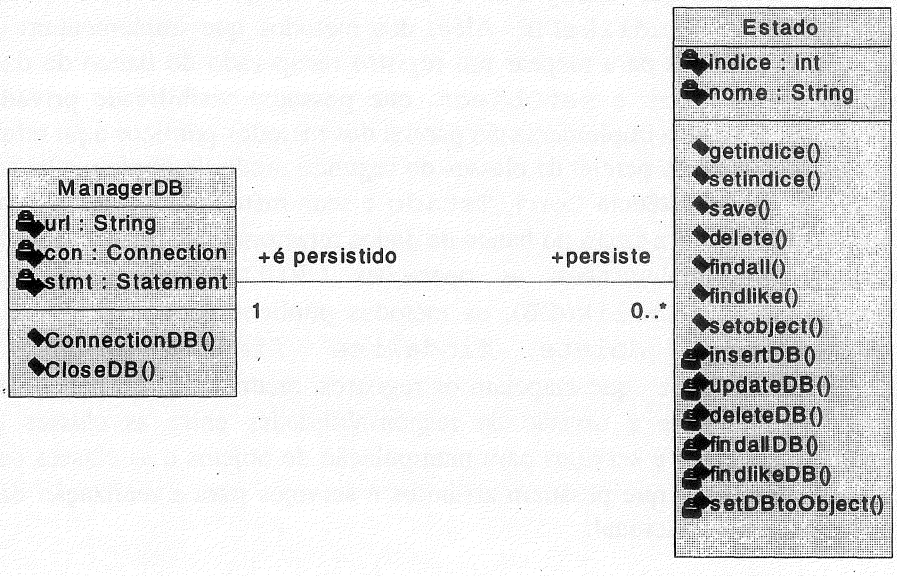

Figura 3 – Classe `Estado` com atributos e métodos do nível de aplicação e de persistência (primeiro modo)

```
public class Estado{
// metodos privados que cuidam da manipulação
 no banco de dados
private boolean insertDB(){
 try {
   String insertSQL = "INSERT INTO
   Estado (statch, indice, ... ) VALUES "
   + "(" + statch.getcodigo() + "," + indice
   + "," + ... + ")";
   ManagerDB.stmt.executeUpdate(insertSQL);
   ManagerDB.con.commit();
   return true;
 }
 catch(SQLException e){
   ...
   return false;
 }
 return false;
}

...

private boolean updateDB(){
   ...;
}

private boolean deleteDB(){
   ...;
}
```

```
private ResultSet findallDB(){
 try{
   String findallSQL = "SELECT * FROM
   Estado WHERE statch = " +
   statch.getcodigo();
   ResultSet rs =
   ManagerDB.stmt.executeQuery(findallSQL);
   return rs;
 }
 catch(SQLException e){
  ...
  return null;
 }
}

private Estado setDBtoObject(ResultSet rs){
Estado estado;
 try{
   int indice = rs.getInt("indice");
   String nome = rs.getString("nome");
   ...
return estado = new Estado(indice, nome,
...);
 }
 catch(SQLException e){
    ...;
   return null;
 }
}
```

Figura 4 - Trecho da implementação dos métodos privados da camada de persistência, da classe `Estado`

Métodos públicos como `save`, `delete`, `findlike`, `findall` são criados para invocar cada um dos métodos que manipulam o banco de dados (`insertDB`, `deleteDB`, `updateDB`, `findallDB`, `findlikeDB`). Além dos métodos que implementam as operações CRUD há também outros métodos para mapear um registro recuperado do banco de dados para um objeto, chamado `setDBtoObject` e `setObject`, que possuem visibilidade privada e pública, respectivamente. A Figura 5 exibe a implementação parcial dos métodos públicos aqui referenciados.

A Figura 6 exibe o modelo parcial de classes do segundo modo de implementação da Camada de Persistência. A classe de persistência `Conj_Estado` e seus métodos são criados para realizar a persistência dos objetos da classe `Estado` no banco de dados relacional. A classe `Conj_Estado` tem os métodos privados que implementam as operações CRUD (`insertDB`, `deleteDB`, `updateDB`, `findallDB`, `findlikeDB`), os métodos públicos para invocar cada um desses métodos de persistência (`save`, `delete`, `findalike`, `findall`) e também os métodos `setDBtoObject` e `setObject` que mapeiam os registros, recuperados do banco de dados, para objetos. Na Figura 6 visualiza-se a divisão de responsabilidades entre as classes da aplicação (`Estado`), que possuem atributos e serviços para manipulação de objetos e as classes de persistência (`ManagerDB` e `Conj_Estado`) que possuem atributos e serviços para a realização da persistência dos objetos em banco de dados relacional.

```
public class Estado
{
// metodos públicos que são intermediários
   aos métodos que cuidam da manipulação no
   banco de dados
public boolean save(){
  ResultSet rs;
  Try{
    Rs = findlikeDB();
    // verifica se o objeto será atualizado
       ou inserido no DB
    if (rs.next())
      return updateDB();
    else
      return insertDB();
    }
  catch(SQLException e){
     ...
return false;
  }
} ...
```

```
public boolean delete(){
  ...;
}

public ResultSet findall(){
  return findallDB();
}

public ResultSet findlike(){
  return findlikeDB();
}

public Estado setObject(ResultSet rs){
  return setDBtoObject(rs);
}
...
}
```

Figura 5 – Trecho da implementação dos métodos públicos da camada de persistência, da classe `Estado`

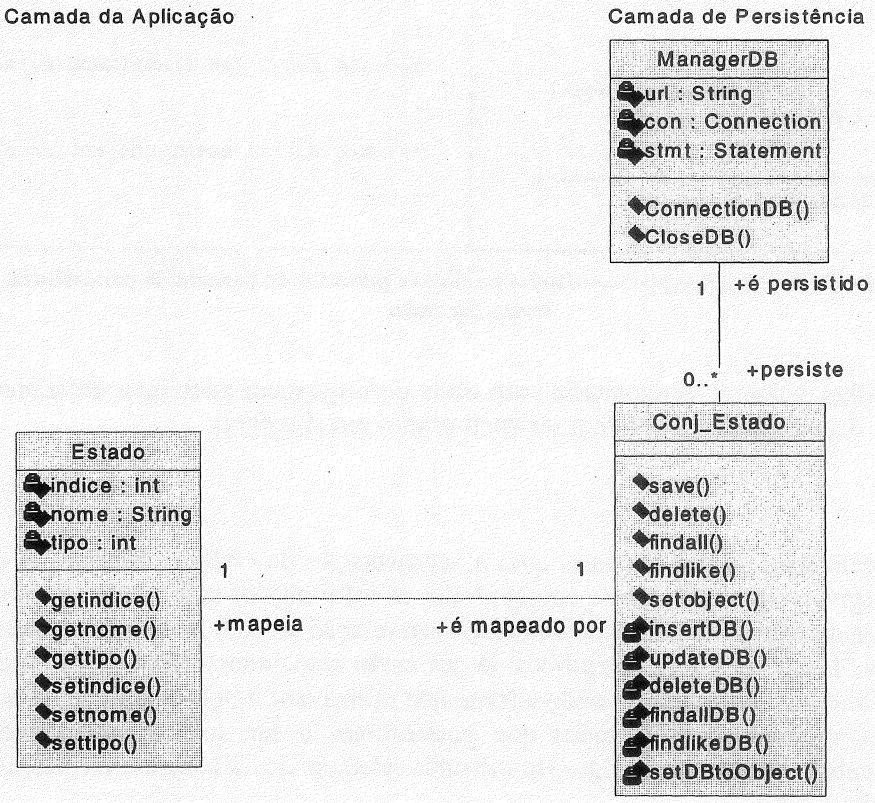

Figura 6 – Classe `Estado` com atributos e métodos específicos do nível de aplicação e classe `Conj_Estado` com atributos e métodos específicos do nível de persistência (segundo modo)

A Figura 7 mostra os métodos privados, implementação das operações CRUD, e os públicos da Camada de Persistência. Como o corpo dos métodos são idênticos aos da Figura 5 e Figura 4 eles não são esboçados novamente. Não é necessário muito esforço para passar do primeiro para o segundo modo de implementação.

As alterações realizadas no código fonte do segundo modo de implementação da Camada de Persistência quando comparado com o primeiro são: a criação da classe de persistência (`Conj_Estado`) e a adição de um parâmetro nos métodos `save`, `findall`, `findlike`, `insertDB`, `updateDB`, `findallDB`, `findlikeDB`. Esse parâmetro é o objeto que será inserido, alterado, removido ou recuperado do banco de dados.

```
public class Conj_Estado
{
  private Conj_Estado conj_estado;
  public Conj_Estado(){
    conj_estado = null;
  }
// metodos públicos que são intermediários aos
   métodos que cuidam da manipulação no banco de
   dados
  public boolean save(Estado estado){
  ...
  }
...
  public ResultSet findall(Estado estado){
    return findallDB(Estado estado);
  }
  public ResultSet findlike(Estado estado){
    return findlikeDB(Estado estado);
  }
  public Estado setObject(ResultSet rs){
    return setDBtoObject(rs);
  }
// metodos privados que cuidam da manipulação
   no banco de dados
  private boolean insertDB(Estado estado)
    ...
  }
...
  private ResultSet findallDB(Estado estado){
    ...
  }
  private ResultSet findlikeDB(Estado estado){
    ...
  }
  private Estado setDBtoObject(ResultSet rs){
    ...
  }
```

Figura 7 - Trecho da implementação dos métodos públicos e privados da camada de persistência, da classe `Conj_Estado`

Mais detalhes sobre a reengenharia com troca de linguagem para Java utilizando padrões de projeto e banco de dados relacional, podem ser encontrados em [Cag99a].

5. Conclusão

Após a realização da reengenharia com a segmentação do código observou-se que o código fonte ficou mais legível e que cada classe contém todas as informações sobre seus atributos e métodos. O número de linhas de código segmentado aumentou em relação ao código original, porém os métodos são menores e mais fáceis de serem entendidos, já que estão associados a uma única classe e possuem nomes mais significativos. As linhas de código fonte que utilizavam funções da linguagem *Xview*, antes espalhadas, foram agrupadas em métodos que possibilitam o seu melhor entendimento e reuso. Também é significativo o reuso de código dos métodos relacionados à leitura e atribuição de atributos (encapsulamento de dados).

O sistema que foi submetido à reengenharia com troca da linguagem de implementação para C++, não utilizando padrões de projeto, tem os métodos bem delineados permitindo fácil entendimento de seu código para futuras manutenções. O reuso também é uma das vantagens desse sistema, uma vez

que foi desenvolvido utilizando o paradigma orientado a objetos. Apesar de apenas uma parte do sistema ter sido implementada em C++, observa-se que os métodos associados a cada classe têm a funcionalidade bem determinada, quando comparada com a parte correspondente do código fonte do sistema legado.

No primeiro modo de implementação da Camada de Persistência, após reengenharia, utilizando Java e padrões de projeto para mapear objetos em banco de dados relacional, observou-se que além do sistema fornecer as vantagens de um sistema desenvolvido em linguagem orientada a objetos também possui métodos que persistem objetos em banco de dados relacional. Esta implementação utiliza poucos benefícios do padrão de Camada de Persistência, mas é mais fácil de ser desenvolvida, em relação às outras maneiras que existem para implementá-lo. Embora necessite de um pequeno código em cada classe, específico para a persistência no banco de dados, esse é reusado e pequenas alterações são feitas para adaptar as SQL's em cada classe.

A vantagem do segundo modo de implementação da Camada de Persistência, em Java, é que se o engenheiro de software decidir pela mudança da forma de armazenamento, somente a classe `Conj_Estado` será alterada, poupando assim esforços de entendimento do código e de manutenção, uma vez que a Camada de Persistência está desacoplada da Camada de Aplicação. Os métodos que implementam as operações CRUD também são reescritos para cada classe de persistência criada.

Uma terceira versão do sistema em Java que está sendo estudada pelas autoras deste trabalho é a de implementar a Camada de Persistência, por meio de uma classe *Broker*. Essa classe deve ser básica para realizar o mapeamento de qualquer tipo de objeto no banco de dados relacional, por meio das operações CRUD. A classe *Broker* deve reconhecer o formato do objeto que deve ser mapeado para o banco de dados, gerando automaticamente SQL's específicas para inserir, deletar ou recuperar registros. Esta solução possui a vantagem de ser utilizada com mais facilidade, embora necessite de muita infraestrutura, uma vez que os métodos dessa classe devem ser genéricos para mapear qualquer tipo de objeto no banco de dados.

Referências Bibliográficas

[Biz97] **Bizarri, G. L., Penteado, R. D.** Reengenharia do Ambiente StatSim – Implementação Orientada a Objetos - Módulo Estado. Relatório Técnico. Bolsa de Iniciação Científica PIBIC-CNPq/UFSCar, São Carlos, 1997.

[Cag99] **Cagnin, M. I.; Penteado, R. A. D.** Passos para condução de reengenharia de sistemas procedurais para sistemas orientados a objetos, preservando a linguagem de implementação original. São Carlos: Departamento de Computação/UFSCar, 1999. 18 p. Documento de Trabalho.

[Cag99a] **Cagnin, M. I.; Penteado, R. A. D.** Passos para condução de reengenharia de sistemas procedurais para sistemas orientados a objetos, utilizando padrões de projeto. São Carlos: Departamento de Computação/UFSCar, 1999. 20 p. Documento de Trabalho.

[Col94] **Coleman, D. et al.** Object-Oriented Development - The Fusion Method. Prentice Hall, 1994.

[Gam93] **Gamma, E., Helm, R., Johnson, R., Vlissides, J.** Design Patterns - Abstraction and Reuse of Object-Oriented Design, LNCS n. 707, p. 406-431, July 1993.

[Har87] **Harel, D.** STATECHARTS: A Visual Formalism to Complex Systems. Science of Computer Programming, v. 8, p. 231-274, 1987.

[Har90] **Harel, D.** STATEMATE: A Working Environment for the Development of Complex Reactive Systems. IEEE Transactions on Software Engineering, v. 16, n. 4, p. 403-414, 1990.

[Hen97] **Henrique, M. G., Penteado, R. D.** Reengenharia do Ambiente StatSim – Implementação Orientada a Objetos – Módulo Transição. Relatório Técnico. Bolsa de Iniciação Científica PIBIC-CNPq/UFSCar, São Carlos, 1997.

[Jac91] **Jacobson, I.; Lindström, F.** Re-engineering of Old Systems to an Object-Oriented Architecture, In: OOPSLA'91, 1991. Proceedings. ACM, p. 340-350.

[Pen95] **Penteado, R., Masiero, P. C., Germano, F.** Engenharia Reversa Orientada a Objetos do Ambiente StatSim: Método Utilizado e Resultados Obtidos. In: Simpósio Brasileiro de Engenharia de Software, 9, 1995, Recife-PE, Anais, p. 345-360.

[Pen96] **Penteado, R. A. D.** Um Método para Engenharia Reversa Orientada a Objetos. São Carlos, 1996. 237 p. Tese (Doutorado em Física Computacional) - Instituto de Física de São Carlos, Universidade de São Paulo.

[Pen96a] **Penteado, R., Germano, F., Masiero, P. C.** An Overall Process Based on Fusion to Reverse Engineering Legacy Code. In: Working Conference Reverse Engineering, 3, 1996, Monterey-California. Proceedings. IEEE, p. 179-188.

[Pen99] **Penteado, R.; Masiero, P. C; Cagnin, M.I.** An Experiment of Legacy Code Segmentation to Improve Maintainability. In: European Conference on Software Maintenance and Reengineering (CSMR'99), 3, 1999, Amsterdan-The Netherlands. Proceedings. IEEE, p. 111-119.

[Yod98] **Yoder, J.W.; Johnson, R.E.; Wilson, Q.D.** Connecting Business Objects to Relational Databases. In: Conference on the Pattern Languages of Programs, 5, 1998, Monticello-IL, EUA. Proceedings.