Reengenharia com Uso de Padrões de Projeto

Maria Istela Cagnin[♦1]
Rosângela D. Penteado[1]
Fernão S. R. Germano[2]
Paulo C. Masiero[2]

[1]Departamento da Computação – DC
Universidade Federal de São Carlos – UFSCar
Caixa Postal 676 CEP 13.565–905
São Carlos – SP – Brasil

{mcagnin,rosangel}@dc.ufscar.br

[2]Instituto de Ciências Matemáticas e de Computação – ICMC
Universidade de São Paulo – USP – São Carlos
Caixa Postal 668 CEP 13.560– 970
São Carlos – SP – Brasil

{fernão, masiero}@icmc.sc.usp.br

Resumo

O processo de reengenharia de um sistema originalmente desenvolvido orientado a procedimentos em linguagem C foi conduzido com mudança de paradigma para orientação a objetos. Esse processo é precedido pela engenharia reversa orientada a objetos do sistema legado usando o método Fusion/RE. Foi também precedido por uma primeira fase em que é mantida a linguagem C, utilizando um processo de segmentação em que o código é reestruturado para atender a algumas características de orientação a objetos. Neste trabalho usa-se a linguagem Java e o Banco de Dados Relacional Sybase fazendo-se três variantes de implementação de acordo com os três modos alternativos sugeridos no padrão Persistent Layer. O padrão CRUD é usado para o projeto dessa camada de persistência. As operações do padrão CRUD são implementadas: nas classes da aplicação (primeiro modo); em classes específicas para cada classe da aplicação (segundo modo) e em uma classe Broker, que faz o mapeamento de qualquer tipo de objeto para o banco de dados relacional (terceiro modo). Modelos de objetos das diferentes implementações e exemplos de código fonte são incluídos no trabalho para ilustrar a efetividade da evolução alcançada com as três configurações. Essas são usadas para avaliar o processo de segmentação que resulta em uma implementação similar àquela obtida no primeiro modo sugerido para implementar o padrão Persistent Layer, pois a herança envolvida nela é praticamente restrita à assinatura nas classes abstratas. Isto é compatível com a falta de herança da linguagem C.

Palavras-Chaves: Reengenharia, Orientação a Objetos, Padrões de Projeto *Persistent Layer*.

Abstract

The reengineering process of a system originally developed with procedural orientation in C is conducted with change to the object oriented paradigm. This process has been preceded by the object oriented reverse engineering of the legacy system using the method Fusion/RE. It has also been preceded by a first phase in which C is preserved; and uses a segmentation process in which the code is restructured to satisfy some object oriented requirements. Java and SyBase are used in this paper to generate three types of implementation according to the three alternative modes suggested by the Persistent Layer pattern. The operations of the CRUD pattern are implemented in the application classes (first mode), in the specific classes of each application class (second mode) and in a Broker class that does the mapping of all the object types to the relational database (third mode). Object Models of the different implementations and examples of source code are included to ilustrate the evolution obtained with the three configurations. These are used to evaluate the segmentation process which results in an implementation similar to that obtained through the first mode suggested by the Persistent Layer pattern as the inheritance involved in it is practically restricted to the methods signature in the abstract class. This is compatible with the lack of inheritance in the C language.

♦ Bolsita da FAPESPProc. Nº 97/12208-0.

1. Introdução

Uma das maneiras de aplicar a reengenharia em um sistema procedimental, para amenizar seus custos e esforços de manutenção, sem alterar a funcionalidade do sistema, é mudar a linguagem de programação, de procedimental para a orientada a objetos. Processos de engenharia reversa apóiam a fase de reengenharia.

Este trabalho apresenta a reengenharia de um sistema procedimental, desenvolvido em linguagem C, mudando a linguagem de programação para Java e alterando a forma de armazenamento. O sistema original armazena os dados em arquivos-texto; após a reengenharia a persistência é realizada no banco de dados relacional Sybase [1, 2].

Um outro problema surge quando se deseja utilizar uma linguagem orientada a objetos e um banco de dados relacional, pois existem algumas diferenças nesses 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) [3]. Para solucionar esse problema são utilizados padrões de projeto específicos para mapear os objetos, criados na linguagem Java, para o sistema de gerenciamento de banco de dados relacional Sybase.

O ambiente StatSim é o sistema escolhido no experimento deste trabalho. Inicialmente esse ambiente foi desenvolvido em linguagem C, utilizando tipos abstratos de dados (TAD), estações de trabalho *Sun* e linguagem de interface gráfica *Xview*. Devido às dificuldades nas atividades de manutenção e expansão, ele foi submetido à engenharia reversa com o apoio da abordagem Fusion/RE [4, 5], que muda o paradigma de desenvolvimento de um sistema do procedimental para o orientado a objetos.

A Figura 1 mostra o modelo de classes parcial do ambiente StatSim obtido após a aplicação de Fusion/RE, utilizando-se a notação da UML [6]. Em [5] podem ser obtidos os detalhes da aplicação desse processo para o ambiente StatSim. As classes exibidas nessa figura são as utilizadas no processo de reengenharia deste trabalho.

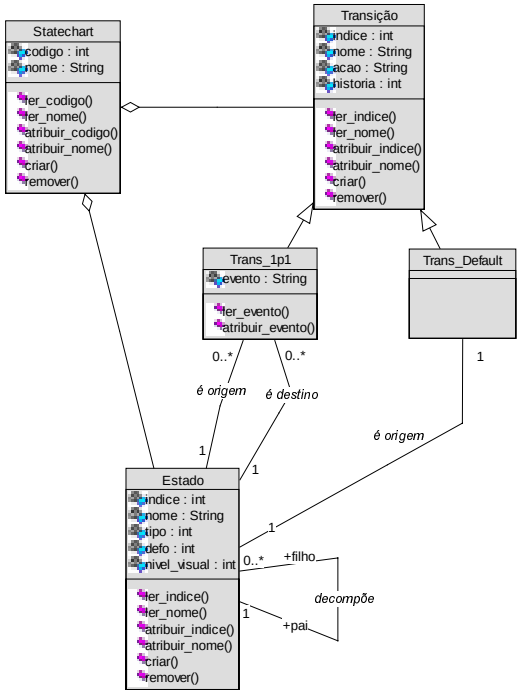

Figura 1 – Modelo de análise do sistema (MAS), parcial, do ambiente StatSim para o tema statechart temporário

Em uma primeira fase, a partir do Modelo de Análise do Sistema (MAS) do ambiente StatSim, obtido no processo de engenharia reversa, foi realizada a reengenharia com mudança de orientação preservando a linguagem C. Parte desse modelo, com os principais componentes de *statecharts* é exibido na Figura 1. Foram adicionadas algumas características orientadas a objetos no código fonte quando possível e esse processo recebeu o nome de segmentação.

O processo de segmentação, descrito em [7, 8], foi realizado sem apoio de uma ferramenta específica. Os elementos do código C são transformados para C com características orientadas a objetos, de acordo com as informações obtidas no processo de engenharia reversa e com a classificação dos TAD's (Tipos Abstratos de Dados). Se o TAD for caracterizado como simples, não é transformado em classe, senão é mapeado como classe na nova versão do sistema. Esse processo preserva totalmente a estrutura da declaração dos TAD's do código legado e somente altera o nome, quando necessário, para mnemônicos mais significativos. Métodos são criados para leitura e atribuição dos atributos do TAD, que foi transformado em classe. Um procedimento com anomalia, isto é, que altera e/ou consulta mais que um TAD é transformado, na segmentação, em dois ou mais métodos. A estrutura seqüencial do código não é alterada e a funcionalidade do sistema é mantida. São adicionadas apenas invocações aos novos métodos criados.

CORET [9] é uma ferramenta específica para transformar procedimentos em métodos, com a seguinte restrição: o procedimento não é dividido em métodos, apenas sofre algumas modificações na sintaxe, devido à mudança de linguagem, de C para C++. A seqüência interna do corpo do procedimento, quando transformado em método, permanece a mesma. Quando um tipo abstrato de dado (TAD) é transformado em classe, os atributos da classe permanecem públicos, não se preocupando com o conceito de encapsulamento de dados do paradigma orientado a objetos.

O processo de segmentação e o CORET consideram, no processo de transformação dos procedimentos para métodos, somente os procedimentos que são chamados por outros procedimentos (pelo menos uma vez) no código legado. O conceito de herança não é implementado em nenhum deles. Na segmentação a implementação desse conceito não foi possível pois a linguagem C não suporta essa característica da orientação a objetos. Diferentemente de CORET, a transformação de procedimento para método, na segmentação, é baseada na informação contida no corpo do procedimento, enquanto no CORET são analisados somente parâmetros e tipo de retorno das funções.

Pelo fato de ser realizado "manualmente", a mudança da linguagem original, na segmentação não é uma questão crítica, já para CORET é um grande problema, pois para cada linguagem é necessário criar uma árvore de sintaxe abstrata (AST).

CORET utiliza o processo de engenharia reversa COREM [10, 11] para apoiar o processo de reengenharia de C para C++, enquanto o processo de segmentação é apoiado pelo Fusion/RE [4, 5].

Este trabalho apresenta um experimento em que se usou padrões de projeto para conduzir a reengenharia de um sistema implementado em linguagem procedimental para um implementado em linguagem orientada a objetos, Java, juntamente com o sistema de gerenciamento de banco de dados relacional Sybase. Três variantes de implementação são feitas de acordo com os três modos alternativos sugeridos no padrão *Persistent Layer.*

Na seção 2 apresenta-se, resumidamente, os conceitos dos padrões de projeto utilizados na reengenharia com a linguagem Java. A seção 3 comenta como a fase de reengenharia é realizada, exibindo os três modos de implementação dos padrões de projeto utilizados e os principais resultados obtidos. A seção 4 faz uma avaliação do trabalho

comparando as implementações aqui realizadas com a da segmentação e, a seção 5 apresenta as conclusões.

2. Padrões de Projeto de Software

Estudos mostram que quando especialistas trabalham em um problema particular é raro que inventem uma nova solução completamente diferente das já existentes para atacá-lo. Diferentes soluções de projeto são por eles conhecidas, de acordo com a própria experiência ou a de outros profissionais. Quando se confrontam com novos problemas, freqüentemente lembram-se de outros similares e reusam a solução antiga, pensando em pares "problema/solução", [12].

Gamma e outros [13] propõem padrões de projeto como um novo mecanismo para expressar experiências na elaboração de projetos orientados a objetos. Ensinam os novos projetistas a trabalharem bem e padronizarem a maneira de desenvolvimento. Esses padrões fornecem um vocabulário comum, reduzem a complexidade do sistema usando 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, o projeto é abstraído independentemente da técnica e da linguagem de implementação.

Segundo Yoder e outros [3], desenvolvedores de sistemas orientados a objetos que usam banco de dados relacional geralmente gastam muito tempo na implementação para tornar os objetos persistentes, devido às diferenças entre os dois paradigmas, conforme mencionado anteriormente. Pode-se evitar esse problema com a utilização de sistemas de gerenciamento de banco de dados orientado a objetos, mas isso não ocorre devido aos problemas de alto custo para sua aquisição e treinamento específico para a sua utilização, entre outros. Uma solução para facilitar o mapeamento de objetos para banco de dados relacionais é a implementação do padrão *Persistent Layer*, que é uma camada para proteger a aplicação e/ou banco de dados de mudanças impróprias.

O padrão *Persistent Layer* é implementado utilizando os padrões CRUD, *SQL Code Description*, *Connection Manager* e *Table Manager*. As operações do padrão CRUD (*create, read, update, delete*, em inglês) realizam a inserção, remoção e recuperação de registros no banco de dados e utilizam o padrão *SQL Code Description*, isto é, descrições em comandos SQL. Todo objeto necessita, no mínimo, dessas operações para que possa se tornar persistente. O padrão *Connection Manager* estabelece e encerra uma conexão com o banco de dados e o padrão *Table Manager* gerencia o mapeamento de um objeto para tabelas e colunas do banco de dados. Os padrões *Atribute Mapping Methods*, *Type Conversion*, *Change Manager*, *OID Manager* e *Transaction Manager* podem ser utilizados na implementação do padrão *Persistent Layer*, mas não são utilizados neste trabalho. Yoder e outros [3] mostram que a implementação do padrão *Persistent Layer* utiliza a classe abstrata `PersistentObject`. Essa classe contém assinaturas dos métodos que implementam as operações do padrão CRUD.

Neste trabalho são discutidos os três modos de implementação para o padrão *Persistent Layer* sugeridos em [3]:

1. implementar as operações do padrão CRUD nas classes da aplicação que possuem tabelas correspondentes no banco de dados. Essas classes da aplicação, são herdeiras da classe abstrata `PersistentObject`;

2. criar classes específicas para cada classe da aplicação, que possui tabela correspondente no banco de dados, para implementar as operações do padrão CRUD. Essas classes específicas são herdeiras da classe abstrata `PersistentObject`; e,
3. criar uma classe *Broker* [12] para fazer o mapeamento de qualquer tipo de objeto para o banco de dados, isto é, por intermédio das operações do padrão CRUD. Essa classe deve criar automaticamente comandos SQL's para cada tipo de objeto que se deseja recuperar ou atualizar no banco de dados. As classes da aplicação, que possuem tabelas correspondentes no banco de dados, são herdeiras da classe abstrata `PersistentObject` e invocam os métodos da classe *Broker*.

A Figura 2 é um diagrama de classes para uma implementação do padrão *Persistent Layer*. Esse diagrama mostra a utilização das classes da aplicação com os padrões: *Persistent Object* (CRUD e *SQL Code Description* nele incluídos), *Table Manager, Connection Manager*. Para utilizar esses padrões, criam-se três classes de persistência: `PersistentObject`, `ConnectionManager` e `TableManager`. Cada classe da aplicação (nesse caso `ClasseA` e `ClasseB`) é herdeira da classe abstrata `PersistentObject`, tendo que implementar cada operação CRUD; cada classe da aplicação possui um relacionamento de associação com a classe `TableManager` para invocar os métodos que realizam a persistência no banco de dados e a classe `ConnectionManager` possui um relacionamento de associação com a classe `TableManager` para que as operações com o banco de dados sejam realizadas.

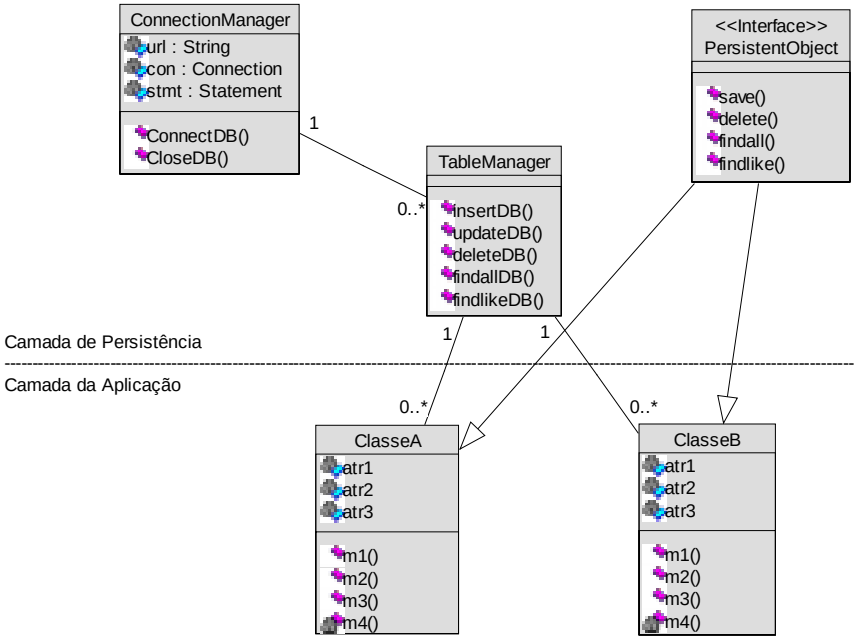

Figura 2 – Classes do padrão *Persistent Layer*

3. Reengenharia com Uso de Padrões de Projeto

Segundo Jacobson [14], existem diversos tipos de reengenharia: com a troca parcial ou completa da linguagem de implementação, com troca do paradigma de desenvolvimento e a combinação das duas anteriores. Porém, em todos os casos preserva-se a funcionalidade

original do sistema. Quando há necessidade da mudança total da funcionalidade deve-se aplicar a engenharia avante.

A versão original do ambiente StatSim foi desenvolvida no paradigma procedimental e armazena os *statecharts* [15] editados em arquivos-texto. Neste trabalho optou-se pela troca do paradigma de desenvolvimento, do procedimental para o orientado a objetos, e pela mudança da forma de armazenamento, preservando a funcionalidade original do sistema. O sistema é migrado da linguagem de implementação procedimental C para a linguagem de implementação orientada a objetos Java. As informações dos *statecharts* editados são armazenadas no Banco de Dados Relacional Sybase.

Para realizar o mapeamento dos objetos para o banco de dados relacional, implementa-se o padrão *Persistent Layer* por meio de operações do padrão CRUD e dos padrões *SQL Code Description*, *Connection Manager* e *Table Manager*. A classe `ConnectionManager`, exibida na Figura 2, implementa o padrão *Connection Manager* pelos atributos `url`, `con` e `stmt`; e pelos métodos `ConnectDB` e `CloseDB`. As classes `Connection` e `Statement` são da biblioteca Java e são utilizadas em todos os modos de implementação. A Figura 3 mostra a implementação da classe `ConnectionManager` em Java. A aplicação tem apenas um banco de dados conectado em um determinado momento, logo há somente um objeto de `ConnectionManager` instanciado durante o tempo de sua execução.

```
public class ConnectionManager{

  static String url;
  static Connect con;
  static Statement  stmt;

 // Conexao – carrega o driver e realiza a
 // conexao com o banco
 public static boolean ConnectDB(String
 DBName, String User, String Password)
 {
  try {
  url = "jdbc:odbc:" + DBName;
  Class.forName("sun.jdbc.odbc.
  JdbcOdbcDriver");
  con = DriverManager.getConnect(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 statment
      con.close();  // encerra a conexão
      System.out.println("BD CLOSED !");
   }
   catch (Exception e) {
      e.printStackTrace();
      System.out.println("Problema no close
      do BD!");
  }
 }
```

Figura 3– Implementação da classe `ConnectionManager`

Nos três modos de implementação de *Persistent Layer* cria-se no banco de dados uma tabela correspondente para cada classe da aplicação: classes são transformadas em tabelas, atributos em colunas, relacionamentos de associação em chaves estrangeiras, relacionamentos de herança em novas tabelas ou em novas colunas da tabela. Quando existe um relacionamento de herança com as propriedades total e exclusivo, pode-se criar tabelas somente para as classes herdeiras; a superclasse é considerada apenas na camada de aplicação. As classes da aplicação que são consideradas na camada de persistência, isto é, que possuem tabelas correspondentes no banco de dados, são chamadas de classes persistentes da aplicação.

O ambiente StatSim é implementado parcialmente, isto é, somente estados e transições (default e 1p1) são considerados nos três modos de implementação do padrão *Persistent*

Layer. A classe da aplicação `Estado` é tomada como exemplo para ilustrar a implementação da camada de persistência nos três modos. O processo utilizado para essa classe repete-se para as demais.

3.1 Primeiro Modo: Implementar as Operações CRUD nas Classes da Aplicação

As operações do padrão CRUD são implementadas em cada classe persistente da aplicação, ou seja, em todas as classes da Figura 4 exceto nas classes da camada de persistência: `ConnectionManager` e `PersistentObject`. As operações do padrão CRUD não são implementadas na classe `Transição`, pois não é criada uma tabela para essa superclasse, somente para as suas classes herdeiras, `Transicao_1p1` e `Trans_Default` pois a propriedade do relacionamento de herança da classe `Transicao` é total e exclusivo. A classe abstrata `PersistentObject` contém somente as assinaturas dos métodos que representam as operações do padrão CRUD (`save`, `delete`, `findall`, `findlike`). Portanto, todas as classes persistentes da aplicação devem ser herdeiras dessa classe e implementar seus métodos. Assim, na Figura 4, a classe `Estado` é herdeira da classe `PersistentObject` e implementa as operações do padrão CRUD, que invocam métodos específicos (`insertDB`, `deleteDB`, `updateDB`, `findallDB`, `findlikeDB`) para manipular informações no banco de dados relacional.

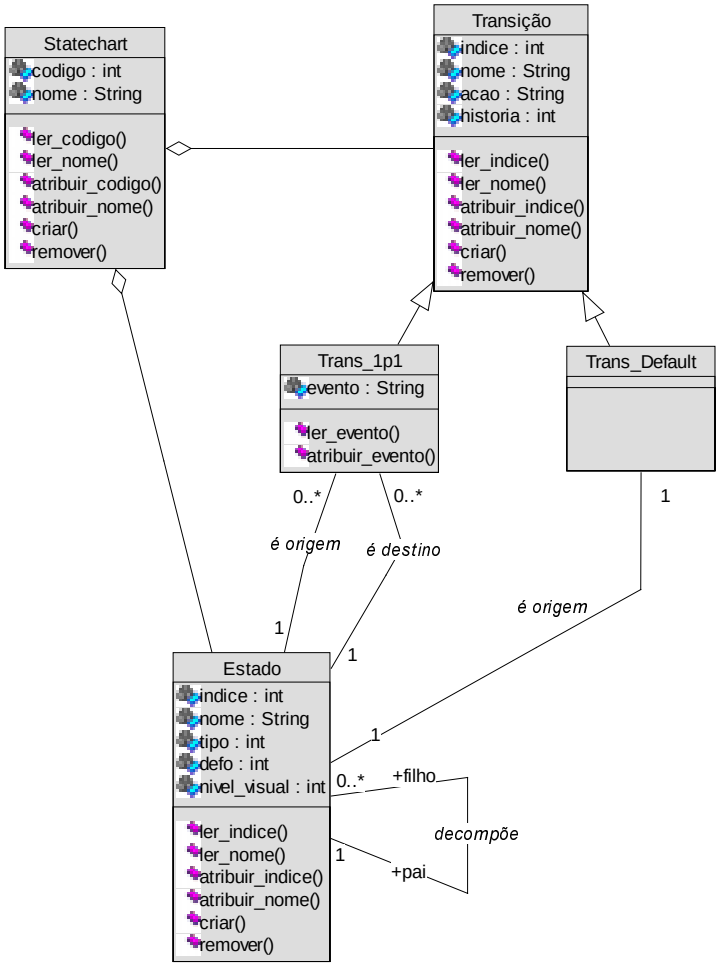

Figura 4 – Classes com atributos parciais e métodos da camada da aplicação e de persistência (primeiro modo)

Os métodos específicos devem ser privados, isto é, somente os métodos da classe podem alterar informações no banco de dados e utilizam o padrão *SQL Code Description* para realizar o mapeamento dos objetos para o banco de dados. A

Figura 5 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 banco de dados é atualizado pela ativação do método `executeUpdate`, da classe `Statement`, por meio do atributo `stmt` da classe `ConnectionManager`. Nesse caso, esse método é utilizado para inserir um objeto da classe `Estado`. Além disso, ele pode também ser usado para modificação ou remoção de dados.

Métodos públicos, que implementam as operações do padrão CRUD, como `save`, `delete`, `findlike` e `findall` são criados para invocar cada um dos métodos específicos. Além dos métodos que implementam as operações do padrão 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 6 exibe a implementação parcial dos métodos públicos aqui referenciados.

Essa maneira de implementar o padrão *Persistent Layer* utiliza poucos benefícios desse padrão, isto é, não utiliza o padrão *Table Manager,* que gera os comandos SQL's automaticamente. Em cada classe persistente da aplicação os métodos que implementam os comandos SQL's são reescritos. Mas é mais fácil de ser desenvolvida, em relação a outras formas de implementação, pois possui métodos de persistência do padrão CRUD na própria classe persistente da aplicação. 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 os comandos SQL's em cada classe.

3.2. Segundo Modo: Criar Classes Específicas para cada Classe da Aplicação para Implementar as Operações do Padrão CRUD

A Figura 7 exibe o modelo parcial de classes do segundo modo de implementação do padrão *Persistent Layer*. Para cada classe persistente da aplicação, é criada uma classe de persistência. Por exemplo, para a classe persistente da aplicação `Estado`, é criada a classe de persistência `Conj_Estado`, e assim respectivamente para todas as outras, exceto para a classe `Transicao` (pois não é criada uma tabela no banco de dados para essa superclasse, somente para as classes herdeiras, como dito no modo anterior). Cada classe de persistência é herdeira da classe `PersistentObject`, seus métodos são criados para realizar a persistência dos seus objetos no banco de dados relacional. Por exemplo, a classe `Conj_Estado` contém os métodos privados (`insertDB`, `deleteDB`, `updateDB`, `findallDB`, `findlikeDB`) que são invocados pelos métodos que implementam as operações do padrão CRUD (`save`, `delete`, `findalike`, `findall`) e também os métodos `setDBtoObject` e `setObject` que mapeiam os registros, recuperados do banco de dados para objetos. Nessa figura visualiza-se a divisão de responsabilidades entre as classes persistentes da aplicação (por exemplo, `Estado`), que possuem atributos e serviços para manipulação de objetos e as classes de persistência (`ConnectionManager`, `Conj_Estado` e `PersistentObject`) que possuem atributos e serviços para a realização da persistência dos objetos em bancos de dados relacionais.

Pode-se observar uma diferença na assinatura dos métodos da classe `PersistentObject` no primeiro e no segundo modo da implementação do padrão

Persistent Layer. Na Figura 4 os métodos não possuem parâmetro, já na Figura 7 os métodos possuem o parâmetro `obj` do tipo `Object`. Esse parâmetro foi criado porque a camada da aplicação está separada da camada de persistência, como já mencionado. Para que uma classe, que pertence à camada de persistência, mapeie um objeto no banco de dados relacional, é necessário que esse objeto seja passado como parâmetro.

A

Figura 8 mostra os métodos públicos e privados da classe `Conj_Estado`, que implementam a camada de persistência, nesse segundo modo, para a classe `Estado`. Os métodos privados são invocados pelos métodos públicos que implementam as operações do padrão CRUD, da camada de persistência. Como o corpo dos métodos é idêntico aos da Figura 5 e da Figura 6 eles não são esboçados novamente.

O modelo da Figura 7 difere do apresentado na Figura 4 pois apresenta as classes de persistência para cada classe persistente da aplicação, e os relacionamentos: de herança, da classe de persistência `PersistentObject`, e o de associação, da classe `ConnectionManager`, com as classes de persistência criadas.

Essa maneira de implementar o padrão *Persistent Layer* possui a vantagem de que se o engenheiro de software decidir pela mudança da forma de armazenamento somente as classes de persistência criadas para cada classe persistente da aplicação serão alteradas. Dessa forma se poupa esforços de entendimento do código e de manutenção, uma vez que a camada de persistência está desacoplada da camada da aplicação, embora ocorra aumento no número de classes de persistência. Os métodos que implementam as operações do padrão CRUD também são reescritos para cada classe de persistência criada.

```
 public class Estado implements
 PersistentObject{
//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 + "," + ... + ")";
   ConnectionManager.stmt.executeUpdate(
   insertSQL);
   ConnectionManager.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 =
  ConnectionManager.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 5 – Implementação parcial dos métodos privados da camada de persistência (primeiro modo), da classe `Estado`

```
public class Estado implements
PersistentObject
{
//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 6 – Implementação parcial dos métodos públicos da camada de persistência (primeiro modo), da classe `Estado`

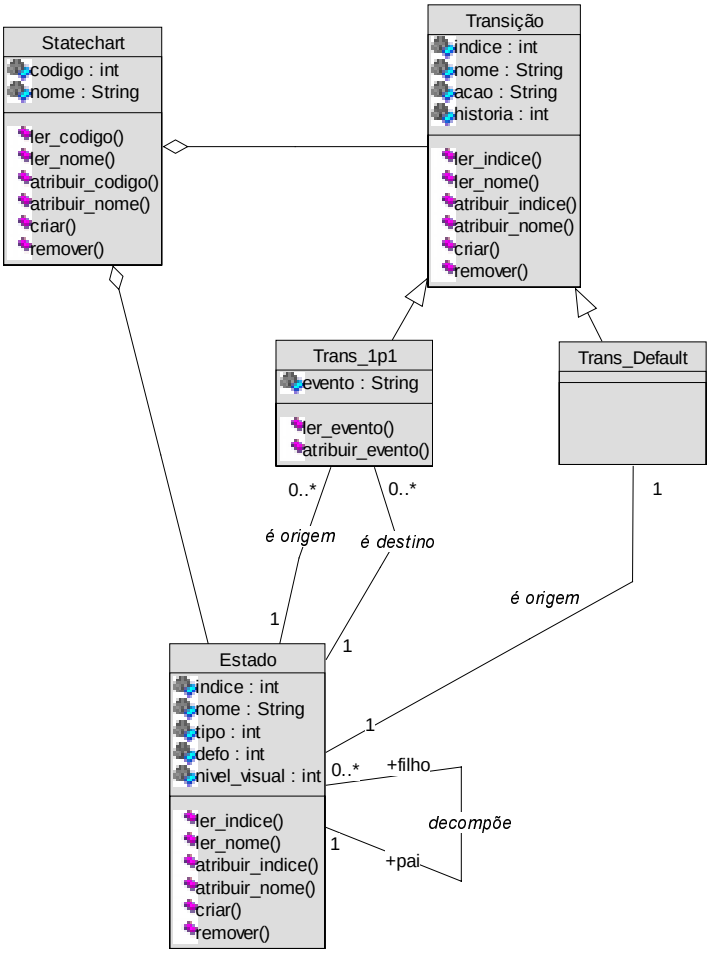

Figura 7 – Classes com atributos parciais e métodos específicos da camada da aplicação e de persistência (segundo modo)

```
public class Conj_Estado implements
PersistentObject
{
  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 8 - Implementação parcial dos métodos públicos e privados da camada de persistência (segundo modo), da classe `Conj_Estado`

3.3. Terceiro Modo: Criar uma Classe *Broker* para Fazer o Mapeamento de qualquer Objeto para o Banco de Dados

Uma classe *Broker* chamada `TableManager` é criada neste modo para a implementação do padrão *Persistent Layer.* Essa classe é básica para realizar o mapeamento de qualquer tipo de objeto no banco de dados relacional. A classe `TableManager` reconhece o formato do objeto que deve ser mapeado para o banco de dados e gera automaticamente o comando SQL específico para inserir, remover ou recuperar registros.

Os métodos públicos como `save`, `delete`, `findlike` e `findall` são criados nas classes persistentes da aplicação (por exemplo, `Estado`) e também são utilizados, como nos modos anteriores, para invocar cada um dos métodos que são responsáveis pela manipulação dos objetos no banco de dados (`insertDB`, `deleteDB`, `updateDB`, `findallDB`, `findlikeDB`), criados na classe `TableManager`. Aqui também, além dos métodos que implementam as operações do padrão CRUD, há 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 e são implementados nas classes persistentes da aplicação. A Figura 9 exibe o relacionamento entre as classes da camada de persistência (`ConnectionManager`, `TableManager` e `PersistentObject`) e as classes persistentes da aplicação (por exemplo, `Estado`). Nesse modelo nota-se redução no número de classes de persistência quando comparado com o modelo do segundo modo, apresentado na Figura 7. As classes da camada de persistência que correspondiam às classes `PersistentObject` e `ConnectionManager` e a cada uma das classes persistentes da aplicação, agora são substituídas pela classe `TableManager`.

Os seguintes atributos são adicionados nas classes persistentes da aplicação: `tableManager`, do tipo `TableManager`, realiza as operações no banco de dados; `tableName` armazena o nome da tabela em que o objeto é inserido; `keyName` armazena o

atributo que é chave primária da tabela; `colNames` contém os nomes de todas as colunas da tabela; `colValues` contém os valores de todas as colunas da tabela; `colNamesException` contém os nomes dos atributos que são chave primária e chaves estrangeiras da tabela e `cols` contém a quantidade de colunas da tabela.

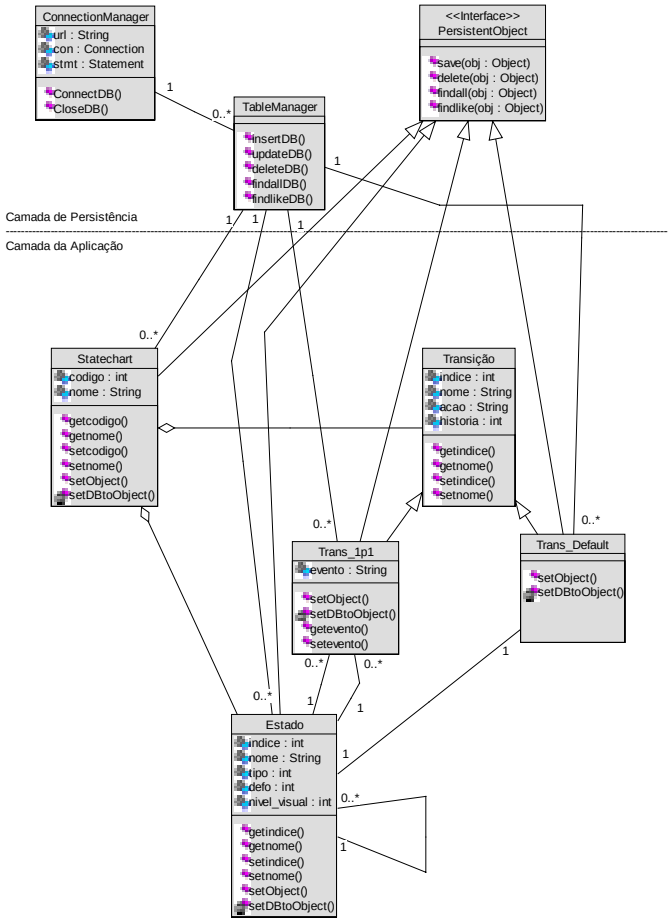

Figura 9 - Classes com atributos parciais e métodos específicos da camada da aplicação e de persistência terceiro modo)

A Figura 10 somente a assinatura e os parâmetros dos métodos da classe *Broker* `TableManager`. Para cada operação no banco de dados cria-se um método genérico para mapear qualquer tipo de objeto para o banco de dados relacional. Os métodos são chamados `insertDB`, `updateDB`, `deleteDB`, `findAllDB`, `findlikeDB`. Esses métodos criam comandos SQL's em tempo de execução, a partir das informações que são passadas como parâmetro. Nos modos anteriores esses parâmetros não foram necessários, pois os comandos SQL's são criados na implementação e não em tempo de execução.O atributo `colNamesException`, adicionado nas classes persistentes da aplicação, nesse terceiro modo, é utilizado na construção dos comandos SQL's que atualizam informações no banco de dados. O atributo `colNamesException` é necessário quando é feita uma atualização, pois os atributos que são chaves (primárias ou estrangeiras) não podem ser atualizados. Além disso, o método `updateDB` contém os parâmetros `clause` e `parameter`, que contêm as cláusulas do comando WHERE, da SQL, e o valor dessas cláusulas, respectivamente.

A implementação do método público `save`, criado na classe persistente da aplicação `Estado`, herdado da classe abstrata `PersistentObject`,invoca um dos métodos da classe

TableManager, insertDB se o objeto não existe no banco ou updateDB se o objeto existe. O método save preenche os vetores clause e parameter, e atualiza o vetor colValues. Os demais métodos públicos (delete, findlike, findall), que implementam as operações do padrão CRUD, possuem no corpo do método apenas invocações aos métodos da classe TableManager (deleteDB, findlikeDB, findallDB), como o método save. A única diferença entre os métodos públicos das classes persistentes da aplicação é a atualização ou a criação dos parâmetros para serem passados na invocação dos métodos da classe TableManager.

```
public class TableManager
{
  private TableManager tableManager;

  // construtor da classe
  public TableManager(){
   tableManager = null;
  }

  public boolean insertDB(String tableName,
  Vector colNames, Vector colValues,
  int cols){
  ...
  }

  public boolean deleteDB(String tableName,
  Vector clause, Vector parameter){ ...
  }

  public boolean updateDB(String tableName,
  Vector colNames, Vector colValues, int cols,
  Vector clause, Vector parameter,
  Vector colNamesException){
  ...
  }

  public ResultSet findallDB(String
  tableName){
  ...
  }

  public ResultSet findlikeDB(String
  tableName, Vector clause, Vector parameter){
  ...
}
```

Figura 10 – Assinatura dos métodos da classe TableManager que implementam o padrão *Persistent Layer* (terceiro modo)

O modelo exibido na Figura 9 possui a classe TableManager, criada na implementação do padrão *Persistent Object*; os relacionamentos de herança da classe PersistentObject e os de associação da classe TableManager com as classes persistentes da aplicação. Há também um relacionamento de associação da classe TableManager com a classe ConnectionManager, para realizar as operações no banco de dados relacional.

Essa maneira de implementar o padrão *Persistent Layer* tem a vantagem de utilizá-lo com mais facilidade, pois uma vez implementada a classe *Broker* (TableManager) os programadores podem reusá-la e não precisam se preocupar com a persistência de qualquer tipo de objeto no banco, é necessário apenas conhecer os parâmetros dos métodos dessa classe. A classe TableManager deve ser construída de modo genérico, ou seja, deve ser capaz de retratar todas as possibilidades das operações no banco de dados, uma vez que seus métodos devem mapear qualquer tipo de objeto nesse banco.

4. Avaliação do Trabalho Realizado

As ferramentas utilizadas para a implementação do padrão *Persistent Layer* nos três modos são: Visual Café for Java [16] e o Sistema de Banco de Dados Relacional Sybase [1]. A parte do ambiente StatSim que foi submetida a reengenharia com mudança de linguagem de programação e da forma de armazenamento possui a mesma funcionalidade que a do sistema legado. Isso foi comprovado por testes que criaram exemplos de *statecharts* tanto no sistema legado como no submetido à reengenharia.

O primeiro modo de implementação não utiliza todos os benefícios do padrão *Persistent Layer* pois o padrão *Table Manager* não é usado. Há necessidade de um pequeno trecho de código, que implementa as operações do padrão CRUD, em cada classe persistente da aplicação. Esse código pode ser reusado por outras classes persistentes da aplicação com algumas modificações nos comandos SQL's.

Para essa implementação o engenheiro de software dispunha somente dos modelos de objetos produzidos no processo de engenharia reversa e gastou entre 120 e 150 horas. O seu desconhecimento da linguagem Java foi a maior dificuldade encontrada.

Ao se comparar o primeiro modo de implementação com a segmentação, realizada anteriormente, observa-se que os métodos de persistência e da aplicação estão associados às mesmas classes. Por exemplo, na segmentação o método, `ler_indice()`, da classe da aplicação `Estado` da Figura 1, tem código semelhante ao do método `get_indice()` da classe `Estado`, da Figura 4. O mesmo ocorre com outros métodos de persistência, por exemplo, `criar()` (Figura 1) e `insertDB()` (Figura 4) da classe `Estado`.

Nesse primeiro modo os métodos da classe abstrata `PersistentObject` são implementados em cada classe persistente da aplicação. Assim, a classe `PersistentObject` da Figura 4, por ser abstrata, deixa a implementação em orientação a objetos com as mesmas características utilizadas na segmentação.

Para a implementação do segundo modo o engenheiro de software dispunha dos modelos de objetos produzidos no processo de engenharia reversa e do código em Java, produzido no modo anterior. Foram gastas, aproximadamente, 30 horas, pois houve a criação das classes de persistência, a migração dos métodos específicos e a preservação do encapsulamento de dados para as respectivas classes. Com isso tem-se aproximadamente 80% de reuso de código do primeiro para o segundo modo de implementação do padrão *Persistent Layer*.

A vantagem do segundo modo de implementação do padrão *Persistent Layer* é que se o engenheiro de software decidir pela mudança da forma de armazenamento somente as classes de persistência, como por exemplo, `Conj_Estado`, serão alteradas. Poupam-se assim esforços de entendimento do código e de manutenção, uma vez que a camada de persistência está desacoplada da camada da aplicação. Os métodos que implementam as operações do padrão CRUD são reescritos para cada classe de persistência que é criada para cada classe persistente da aplicação.

Como ocorreu no primeiro modo, neste segundo modo de implementação, o padrão *Table Manager* também não é utilizado, mas há aumento do número das classes de persistência.

Comparando-se a implementação do segundo modo com a segmentação observa-se que os métodos da camada de persistência estão desacoplados da camada da aplicação, enquanto na segmentação eles encontram-se acoplados nas respectivas classes.

Para a implementação do terceiro modo o engenheiro de software dispunha dos modelos de objetos produzidos no processo de engenharia reversa e do código em Java, produzido nos modos anteriores. Foram gastas entre 120 e 150 horas. A dificuldade encontrada foi tornar a classe *Broker* mais genérica possível com a criação de alguns atributos (`colNames`, `colValues`, `parameter`, `colNamesException`, etc.) nas classes persistentes da aplicação para possibilitar a criação de comandos SQL em tempo de execução.

O terceiro modo de implementação do padrão *Persistent Layer* possui a vantagem de utilizar o padrão *Table Manager*. O programador não tem preocupação com o mapeamento dos objetos no banco de dados. Há necessidade de que ele conheça apenas os parâmetros dos métodos da classe `TableManager` que implementam esse padrão.

Nesse terceiro modo de implementação há redução do número de classes da camada de persistência quando comparado com o segundo. Isso ocorre porque o padrão *Table Manager* cuida da implementação genérica dos métodos do padrão CRUD, da camada de persistência.

5. Conclusão

Este trabalho complementa o anterior [7], que cuidou do processo de reengenharia aplicado anteriormente no ambiente StatSim, conhecido como segmentação, que preservou a linguagem de programação C e adicionou características orientadas a objetos, e permite sua mais objetiva avaliação. A experiência adquirida no processo de segmentação ajudou consideravelmente na realização deste trabalho, embora aqui os métodos tenham sido desenvolvidos em uma linguagem orientada a objetos.

No primeiro modo de implementação do padrão *Persistent Layer*, após a 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.

Com a implementação do padrão *Persistent Layer* foi possível avaliar e validar a segmentação realizada anteriormente. Além do primeiro modo de implementação do padrão *Persistent Layer*, o segundo também pode ser implementado em uma linguagem de programação procedimental. A herança da classe abstrata `PersistentObject` não é indispensável, considerando que os métodos que implementam a persistência podem ser migrados para classes específicas criadas para a camada de persistência.

Comparando-se o terceiro modo de implementação com a segmentação, observa-se que não é possível reunir os métodos de persistência em uma única classe pois a linguagem C não utiliza os conceitos de herança.

Com este trabalho pode-se notar que a implementação utilizando o padrão CRUD com linguagem orientada a objetos, nos dois primeiros modos, é semelhante à realizada no processo de segmentação. Assim evidencia-se a potência desse padrão, que é utilizável tanto em linguagens procedimentais como em orientadas a objetos. Ressalva é feita com relação ao conceito de herança que por não existir nas linguagens procedimentais não permite comparação com a orientação a objetos.

O ambiente StatSim foi submetido a diversos processos de reengenharia resultando em diversos produtos: o código legado [4], o código segmentado [7] e os três códigos Java apresentados neste trabalho. A melhoria da manutenibilidade desses produtos é avaliada por meio de um estudo piloto [17]. Experimentos de manutenção nos produtos acima citados foram realizados por diferentes programadores, com e sem participação nos seus desenvolvimentos. Dessa forma há indícios de que a manutenibilidade do sistema desenvolvido em linguagem orientada a objetos com o padrão *Persistent Layer* é maior que a do sistema segmentado, que por sua vez, é maior que a do sistema legado. Por meio desses experimentos notou-se que o tempo médio de manutenção em Java é inferior ao tempo médio gasto na versão do sistema segmentado. Já o tempo médio para manter a versão em Java que implementa o segundo modo de implementação do padrão *Persistent Layer* é 1/3 menor que o primeiro. O tempo médio de manutenção em Java para o terceiro modo é inferior ao primeiro e ao segundo modo, pois há o reuso dos métodos da classe *Broker*.

Referências Bibliográficas

[1] **Sybase -** Sybase Inc. URL: http://www.sybase.com.

[2] **Java** - JDBC Basics. URL: java.sun.com/docs/books/tutorial/jdbc/basics/index.html.

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

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

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

[6] **Rational Corporation -** Unified Modeling Language. URL: http://www.rational.com/uml/references.

[7] **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, Amsterdan. Proceedings. IEEE, p. 111-119. 1999.

[8] **Cagnin, M. I, Penteado,R. -** Passos para Condução de Reengenharia de Sistemas Procedimentais para Sistemas Orientados a Objetos, Preservando a Linguagem de Implementação Original. Documento de Trabalho – Departamento de Computação/UFSCar. São Carlos, 18 p., Janeiro, 1999.

[9] **Taschwer, M.; Rauner-Reithmayer, D.; Mittermeir, R. –** Gerating Objects from C Code – Features of the CORET Tool-Set. In: European Conference on Software Maintenance and Reengineering (CSMR'99), 3, Amsterdan-The Netherlands. Proceedings. IEEE, p. 91-100. 1999.

[10] **Gall, H., Klösch R.** - Capsule Oriented Reverse Engineering for Software Reuse. In: European Software Engineering Conference, ESEC'93, 4. Garmish-Partenkirchen, Germany, p. 418-33. 1993.

[11] **Gall, H., Klösch R. -** Finding Objects in Procedural Programs: An Alternative Approach. In: Working Conference on Reverse Engineering, 2, Toronto-Canada. Proceedings. IEEE Computer Society Press, p. 208-216. 1995.

[12] **Buschmann, Frank; et al.** - Pattern-Oriented Software Architeture. European Conference on Object-Oriented Programming, 11, Finland. Proceedings. Finland. 1997.

[13] **Gamma, E., Helm, R., Johnson, R., Vlissides, J.** - Design Patterns Elements of Reusable of Object-Oriented Software. Reading-MA, Addison-Wesley. 1995

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

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

[16] **Symantec Corporation.** URL: http://www.symantec.com.

[17] **Cagnin, M. I.** – Avaliação das Vantagens quanto à Facilidade de Manutenção e Expansão de Sistemas Legados Sujeitos a Engenharia Reversa e Reengenharia. Dissertação de Mestrado – Departamento de Computação, Universidade Federal de São Carlos. São Carlos, 101 p. agosto, 1999.