USO DA PROGRAMAÇÃO ORIENTADA A ASPECTOS EM PERSISTÊNCIA DE DADOS UTILIZANDO ASPECTJ

Leonardo Franco de Almeida[1], Osvaldo Pereira Duarte Júnior[2], Sérgio Teixeira de Carvalho[3]

Resumo
Esse artigo tem objetivo apresentar um modelo de como a programação orientada a aspectos pode auxiliar em soluções para a persistência de dados utilizando a linguagem *AspectJ*. É apresentada uma visão geral sobre orientação a aspectos, os diversos contextos em que estão inseridas as linguagens de programação orientada a aspectos e o uso de aspectos no gerenciamento de transações envolvendo persistência de dados.

Palavras-chave: mapeamento, armazenamento de dados, recuperação.

USE OF ASPECT ORIENTED PROGRAMMING ON DATA PERSISTENCE USING ASPECTJ

Abstract
This article has the objective to present a model of how aspect oriented programming can assist in solutions for data persistence using the language AspectJ. A general vision of aspect orientation is presented, the contexts in which aspect oriented programming languages are inserted and the use of aspects in the management of transactions involving data persistence.

Key words: mapping, data storage, recovery.

Introdução

A orientação a aspectos (OA) surgiu dos crescentes e cada vez mais complexos interesses dispersos no desenvolvimento de software que a orientação a objetos não consegue tratar, tais como distribuição, segurança, persistência de dados, auditoria e portabilidade (RESENDE e SILVA, 2005). Esse modelo de programação diminui as dificuldades encontradas entre a integração das

[1]Analista de Sistemas; leonardo@fenix.com.br
[2]Engenheiro Eletricista; osvaldop@brturbo.com.br
[3]Mestre em Computação. Professor da Pós-graduação do Centro Universitário de Goiás - Uni-ANHANGÜERA, sergiocarvalho@anhanguera.edu.br

aplicações e as regras de negócio, reduz a complexidade permitindo que as preocupações possam ser componentizadas independentemente de sua natureza funcional e aumenta a produtividade no desenvolvimento de software.

Nesse artigo, será apresentada uma forma de tratar o gerenciamento de armazenamento e recuperação de dados e de conexões com banco de dados utilizando a programação orientada a aspectos (POA) com *AspectJ* (KICZALES et al., 2001). Para esse propósito, apresenta-se uma explanação sucinta sobre a orientação a aspectos, as linguagens de programação orientada a aspectos de propósitos gerais e específicos, usos da programação orientada a aspectos no gerenciamento da persistência de dados.

Orientação a Aspectos

Na fase de análise, os requisitos são agrupados por tópicos visando a modularização do sistema. Nas primeiras atividades de desenvolvimento de software os requisitos são divididos segundo suas funcionalidades, responsabilidades e entidades relacionadas.

O trabalho de organizar os requisitos de software em nichos de funcionalidades, preocupações e responsabilidades, denomina-se de "*separation of concerns*" (separação de interesses) (RESENDE e SILVA, 2005). A persistência de dados é considerada um interesse nos requisitos de um software.

A existência de interesses que ficam dispersos por várias unidades de software independe da técnica de programação utilizada. Três pontos foram observados por pesquisadores: o espalhamento de código que acontece quando um interesse encontra-se em diversos módulos do software, o emaranhamento que acontece na implementação de diversos interesses dentro de um módulo e o entrelaçamento que acontece quando linhas de código repetidas são inseridas em diversos módulos.

Estes tipos de interesses são, geralmente, integrantes de uma segunda dimensão e transversal àqueles módulos que implementam os requisitos funcionais. A persistência de dados (ELRAD et al., 2001) é um exemplo de

interesse transversal que aparece espalhado e entrelaçado nos módulos que implementam os requisitos funcionais.

Os pesquisadores desenvolveram métodos, técnicas e ferramentas para componentizar o desenvolvimento de software facilitando sua evolução e integração. Dentre as técnicas investigadas estão a orientação a aspectos (OA), componentização, padrões de projeto que procuram implementar a teoria de "*Separation of Concerns*".

Por ser um tema de pesquisa recente, os pesquisadores da área de OA preocuparam-se, inicialmente, em estabelecer os conceitos e técnicas básicas da linguagem orientada a aspectos (KICZALES et al., 1997; ELRAD et al., 2001). Em um segundo plano que ocorre atualmente, as pesquisas se voltam para o uso de métodos de projeto, análise de requisitos e abordagem de testes específicos.

Quando as propriedades do programa devem ser compostas de forma distinta e coordenadas, elas se entrecortam (KICZALES et al., 1997). A orientação a aspectos é uma técnica para trabalhar com este problema. Ela tenta separar os níveis de programação durante o desenvolvimento do software (RESENDE e SILVA, 2005).

A proposta é desenvolver determinadas partes sem se preocupar com as demais. A cada interação da integração insere-se um novo nível de preocupação sem ser necessário alterar o que já está pronto.

Baseado nessa proposta, pode-se desenvolver o gerenciamento do armazenamento de dados, da recuperação, da integridade e das conexões com banco de dados utilizando a técnica de orientação a aspectos.

Programação Orientada a Aspectos

A programação orientada a aspectos (POA) é resultado de vários estudos de programação orientada a assuntos, filtros de composição, programação adaptativa e outros. O termo 'aspecto' refere-se aos interesses

que cortam a aplicação e podem ser implementados em módulos separados (KICZALES et al., 2001).

As linguagens de POA podem ser de domínio específico e de propósitos gerais (LEMOS, 2005). As de domínio específico tratam de determinados aspectos ligados ao domínio. Alguns exemplos podem ser vistos na Tabela 1.

Tabela 1 – Exemplos de linguagens de POA no domínio específico

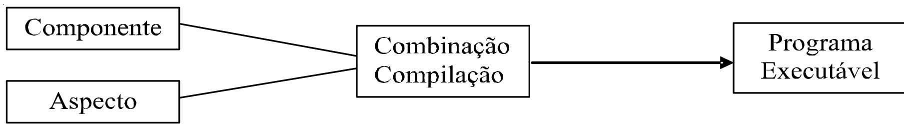
LINGUAGEM DE PROGRAMAÇÃO	INTERESSE
COLL (LOPES, 1997)	*Locking* e exclusão contínua
RIDL (LOPES, 1997)	Invocação remota
D²AL (BECKER, 1998)	Computação Distribuída
QuO (KARR et al., 2001)	Política para o Transporte em Rede.

Fonte: Lemos (2005)

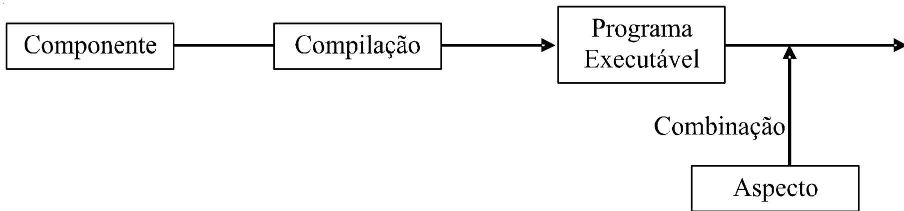
As linguagens de propósito geral permitem codificar qualquer tipo de aspecto em unidades separadas e capturar locais onde os aspectos afetam os módulos básicos do programa (KICZALES et al., 2001). Para que os aspectos possam interagir com os módulos do programa é necessário que o desenvolvedor determine em quais pontos da execução do programa serão definidos comportamentos. Estes pontos são chamados de pontos de junção (*join points*).

Depois da codificação dos aspectos e dos componentes com pontos de junção, estes devem ser unidos em um programa executável chamado de *weaving* (KICZALES et al., 2001). Esta combinação pode ser feita de forma estática ou dinâmica, ou seja, em tempo de compilação ou de execução. A Figura 1 mostra a representação destas combinações.

a) Combinação Estática



b) Combinação Dinâmica



Fonte: Lemos (2005), adaptado pelos autores.

Figura 1. Representação de combinação estática e combinação dinâmica

Alguns exemplos de linguagens de propósito geral podem ser vistos na Tabela 2.

Tabela 2 – Exemplos de linguagens de POA de propósitos gerais

LINGUAGEM DE PROGRAMAÇÃO	INTERESSE
AspectC++ (SPINCZYK; GAL; SCHRODER-PREIKSCHET, 2002)	Acrescenta aspectos a linguagem C++.
AspectS (HIRSCHFELD, 2003)	Acrescenta aspectos a linguagem Smalltalk.
AspectC# (KIM, 2002)	Acrescenta aspectos a linguagem C#.
AspectJ (KICZALES et al., 2001)	Acrescenta aspectos a linguagem Java.
JBoos AOP(BURKE, 2003)	Acrescenta aspectos ao servidor JBoss
AspectWerks(BONÉR; VASSEUR, 2002)	Acrescenta aspectos a linguagem Java.

Fonte: Rocha (2005), adaptado pelos autores

Uma linguagem de orientação a aspectos deve determinar: um modelo de pontos de junção em que os comportamentos sejam definidos; um mecanismo de identificação de pontos de junção; as unidades que encapsulam tanto as especificações dos pontos de junção quanto as definições dos comportamentos desejados; e um combinador para unir as unidades de um programa (ELRAD et al., 2001).

Nestas linguagens de propósitos gerais que implementam aspectos em Java, observa-se que o *AspectJ* (KICZALES et al., 2001) possui compilador que implementa construções que consistem em conjuntos de pontos de junção (*pointcut*), adendos (*advices*), funções que afetam a estrutura estática do programa e a hierarquia de classes além dos aspectos (*aspects*). Já o *JBoss* AOP e *AspectWerks*, os aspectos são implementados em java e o processo de combinação é por meio de arquivos XML (Extensible Markup Language) dinamicamente, não sendo necessário compilador (ROCHA, 2005). Neste trabalho, será dada ênfase à linguagem *AspectJ* (KICZALES et al, 2001).

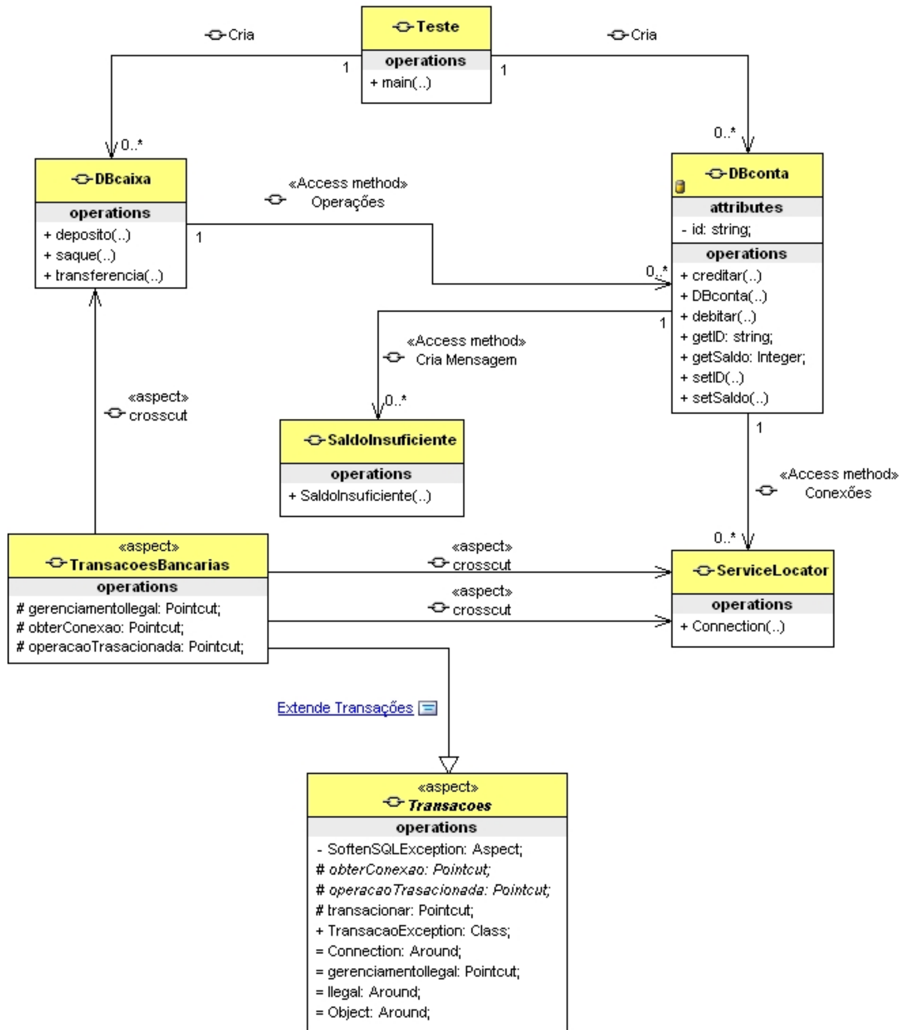
Gerenciamento da Persistência de Dados com Aspectos

Quando se desenvolve aplicações envolvendo persistência de dados, o recurso de transações torna-se importante com o objetivo de garantir a integridade dos dados em determinadas regras de negócio (RESENDE e SILVA, 2005).

Nesse item, é abordado um exemplo que implementa persistência de dados envolvendo banco de dados relacional, onde são utilizados aspectos para gerenciar as transações de conexões, armazenamento e a recuperação de dados armazenados no banco de dados relacional.

A Figura 2 mostra o diagrama de classes de domínio de uma aplicação bancária que realiza operações de depósito, saque e transferência entre contas

bancárias. Este diagrama foi adaptado de Resende e Silva (2005) e nele foram incluídas as classes Teste e Saldo Insuficiente, os relacionamentos, os métodos de todas as classes e os aspectos com os *pointcuts* e *advices*..



Fonte: Resende e Silva (2005), adaptado pelos autores

Figura 2 – Diagrama de classes aplicação bancária

Neste exemplo da Figura 2, aspectos são utilizados para gerenciar os métodos depósito( ), saque( ) e transferência( ) da classe DBcaixa e do método getConnection( ) da classe ServiceLocator.

Nos itens 4.1 a 4.6, serão mostrados os códigos fontes das classes que compõe o exemplo da Figura 2, codificados em java, com os comentários necessários para elucidar as diversas etapas de criação.

Classe Dbcaixa

Realiza as operações de saque, depósito e transferência entre contas.

```
import java.sql.*;
public class DBCaixa {
//EXECUTA A TRANSACAO TRANSFERENCIA
    public void transferencia(DBConta origem, DBConta
destino, int valor) throws SaldoInsuficiente,
SQLException {
        destino.creditar(valor);
        origem.debitar(valor);
    }
//EXECUTA A TRANSACAO DEBITAR
    public void saque(DBConta conta, int valor)
throws SaldoInsuficiente, SQLException {
        conta.debitar(valor);
    }
//EXECUTA A TRANSACAO DEPOSITO
    public void deposito(DBConta conta, int valor)
throws SQLException {
        conta.creditar(valor);
    }
}
```

Fonte: Resende e Silva (2005), adaptado pelos autores.

Classe DBconta

Esta classe é responsável por criar a conta, armazenar e recuperar os dados persistentes por intermédio de conexões com o banco de dados. Para simplificar, Resende e Silva (2005) criaram somente os campos Id e Saldo, mas a linha de raciocínio pode ser estendida para outros campos.

```
import java.sql.*;
public class DBConta {
    private String id;
//CRIA UMA CONTA E ARMAZENA OS VALORES DE ID E SALDO
    public DBConta(String novoId, int deposito)
throws SQLException {
        this.setId(novoId);
        try {
            Connection conn =
ServiceLocator.getConnection();
            Statement stmt = conn.createStatement();
            stmt.executeUpdate("INSERT INTO CONTA
(ID, SALDO) VALUES ("novoId+","+deposito+")");

            stmt.close();
            conn.close();
        } catch (SQLException ex) {
            int valor = 0;
            if (deposito > 0) {
                valor = deposito;
            }
            this.setSaldo(this.getSaldo()+valor);
        }
    }
    public String getId() {
```

```
        return id;
    }
    private void setId(String novoId) {
        this.id = novoId;
    }
//RECUPERA O VALOR DO SALDO DA CONTA
    public int getSaldo() throws SQLException {
        Connection conn =
ServiceLocator.getConnection();
        Statement stmt = conn.createStatement();
        ResultSet rs = stmt.executeQuery("SELECT
SALDO FROM CONTA " +"WHERE ID = " + this.getId());
        rs.next();
        int saldo = rs.getInt(1);
        stmt.close();
        conn.close();
        return saldo;
    }
//ATUALIZA O VALOR DO SALDO
    private void setSaldo(int saldo) throws
SQLException {
        Connection conn =
ServiceLocator.getConnection();
        Statement stmt = conn.createStatement();
        stmt.executeUpdate("UPDATE CONTA SET SALDO
= "                          + saldo +
                             " WHERE ID = "
                             + this.getId());
        stmt.close();
        conn.close();
    }
```

```
//IMPLEMENTA O METODO CREDITAR
    public void creditar(int valor) throws
SQLException {
        int saldoTotal = getSaldo() + valor;
        setSaldo(saldoTotal);
    }
//IMPLEMENTA O METODO DEBITAR
    public void debitar(int valor) throws
SQLException, SaldoInsuficiente {
        int saldoAtual = getSaldo();
        if (saldoAtual < valor) {
//INSTANCIA A CLASSE SaldoInsuficiente
            throw new SaldoInsuficiente (“Saldo
insuficiente para efetuar operação”);
        } else {
            int valorTotal = saldoAtual - valor;
            setSaldo(valorTotal);
        }
    }
}
```

Fonte: Resende e Silva (2005), adaptado pelo autor.

Classe ServiceLocator

Esta classe é responsável por realizar as conexões com o banco de dados. Os autores utilizaram drivers do *MySQL* para a API do JDBC (Java Database Connectivity).

```
import java.sql.Connection;
import java.sql.SQLException;
import java.sql.DriverManager;

public class ServiceLocator {
```

```
    public static Connection getConnection() throws
SQLException {
        Connection conn = null;
        try {
          String url = "jdbc:mysql://localhost/poaj";
          Class.forName("com.mysql.jdbc.Driver");
          conn = DriverManager.getConnection(url,
"user", "password");
          conn.setAutoCommit(true);
        } catch (ClassNotFoundException ex) {
          System.err.println(ex);
        }
        return conn;
    }
}
```

Fonte: Adaptado de Resende e Silva (2005, p.158)

Classe Teste

Esta classe é responsável por criar instâncias para as classes DBconta e DBcaixa, além de realizar transações entre as mesmas.

```
import java.sql.*;
public class Teste {
        public static void main(String[] args) {
            try {
            DBConta conta1 = new DBConta("1", 0);
            DBConta conta2 = new DBConta("2", 0);
            DBConta conta3 = new DBConta("3", 50);

                DBCaixa caixa = new DBCaixa();
                caixa.deposito(conta1, 100);
                caixa.deposito(conta2, 100);
```

```
                caixa.saque(conta3,  10);
                caixa.transferencia(conta1, conta3, 90);
                caixa.transferencia(conta1, conta2, 50);
                } catch (SQLException ex) {
                    System.err.println(ex);
                } catch (SaldoInsuficiente ex) {
                    System.err.println(ex);
                }
    }
}
```

Fonte: Resende e Silva (2005), asaptado pelo autor.

CLASSE SaldoInsuficiente

Esta classe é responsável por enviar uma mensagem quando uma exceção por saldo insuficiente é detectada na execução do método transferência() da classe DBcaixa.

```
public class SaldoInsuficiente extends Exception {

    public SaldoInsuficiente(String msg) {
        super(msg);
    }

}
```

Fonte: Resende e Silva (2005), adaptado pelos autores

ASPECTOS Transações e TransaçõesBancárias

No aspecto TransacoesBancarias, as instâncias dos *pointcut* operacaoTrasacionada, obterConexao e gerenciamentoIlegal são herdados do aspecto Transações (Figura 2).

Aspecto Transações Bancárias

O aspecto Transacoes Bancarias captura os pontos de execução dos métodos deposito(), saque(), e transferencia() da classe DBcaixa através do *pointcut* operacaoTrasacionada(). A captura ocorre após o sistema ter invocado o método e antes da execução do primeiro comando, além de obedecer a construção lógica "ou" dos pontos de execução.

De outra maneira, o aspecto TransacoesBancarias captura o ponto de chamada do método getConnection() da classe ServiceLocator através do *pointcut* obterConexao(). A chamada acontece antes da mudança do fluxo para o método getConnection() e a avaliação dos argumentos que serão passados.

As intervenções realizadas pelo aspecto TransacoesBancarias têm como objetivo tratar o gerenciamento dos dados que as operações dos métodos da classe DBcaixa estão manipulando além de atuar diretamente nas confirmações das operações persistentes que a classe DBconta realiza com o banco de dados.

As decisões que serão tomadas nas capturas realizadas pelo aspecto TransacoesBancarias estão implementadas no aspecto Transações (Figura 2).

```
package transacoes;

import java.sql.Connection;

public aspect TransacoesBancarias
        extends Transacoes {

//DETECTA CHAMADAS PARA AS OPERAÇÕES TRANSACIONAS

    protected pointcut operacaoTrasacionada()
        : execution(* DBCaixa.deposito(..))
        || execution(* DBCaixa.saque(..))
        || execution(* DBCaixa.transferencia(..));
```

```
//DETECTA CHAMADAS DE CRIAÇÃO DE CONEXÕES AO BANCO
DE DADOS

    protected pointcut obterConexao()
    :call(Connection
ServiceLocator.getConnection(..));
}
```

Fonte: Adaptado de Resende e Silva(2005, p.164)

Aspecto Transações

No aspecto Transações é criada a cláusula percflow(transacionar) com o objetivo de permitir que uma instância do aspecto TransacoesBancarias seja definida para cada entrada de controle de fluxo do ponto de junção (*joinpoint*) do *pointcut* transacionar().

O *pointcut* transacionar() é formado pelo *pointcut* operacaoTrasacionada() e pelos pontos de junção que não estejam no fluxo de controle do *pointcut* operacaoTrasacionada(). Ele intercepta e obtém informações sobre as chamadas das operações que o *pointcut* operacaoTrasacionada() realiza.

Para um cenário do método transferencia() da classe DBcaixa com sucesso, o *pointcut* transacionar() aplica a ação do método commit() no objeto conn e o fluxo das operações continua como antes da interceptação através da cláusula proceed() do *advice* tipo around(). Assim, as operações dos métodos getSaldo(), setSaldo() e do construtor DBconta() são efetivadas. Neste cenário e em outros que serão descritos adiante, o *advice* Ilegal, tipo around() foi adaptado para retornar uma informação do ponto de junção (*jointpoint*) em execução, naquele momento, através da declaração de impressão System.out.println("Ponto de Junção: " + thisJoinPoint). A cláusula proceed() ordena que o fluxo continue como antes da interceptação.

Para um cenário do método transferencia() da classe DBcaixa sem sucesso onde uma exceção é capturada para um objeto criado, o *pointcut* transacionar()

aplica a ação do método roolback() no objeto conn, interrompe o fluxo de controle e as operações dos métodos getSaldo(), setSaldo() e do construtor DBconta() são reiniciadas para o mesmo objeto. Da mesma forma que no cenário com sucesso, o *advice* Ilegal, tipo around() retorna uma informação do ponto de junção (*jointpoint*) em execução naquele momento.

Se uma nova exceção ocorre após a ação do método rollback() para o mesmo objeto, o *advice* tipo around() cria uma nova instância da classe TransacaoException, aplica o método close() no objeto conn que finaliza as operações dos métodos setSaldo(), getSaldo() e do construtor DBconta() da classe DBconta. O *advice* Connection(), tipo around(), intercepta a conexão com o banco de dados e atribui o estado false para a confirmação automática de transações para o objeto que gerou a exceção após a chamada do método rollback().

```
package transacoes;

import java.sql.*;

public abstract aspect Transacoes

    percflow(transacionar()) {
    private Connection conn;
    protected abstract pointcut
operacaoTrasacionada();
    protected abstract pointcut obterConexao();
    protected pointcut
transacionar():operacaoTrasacionada()
    && !cflowbelow(operacaoTrasacionada());

//DESVIA A EXECUÇÃO DAS OPERAÇÕES TRANSACIONADAS.
IMPLEMENTA O CONTROLE TRANSACIONAL DA APLICAÇÃO.
```

```
        Object around() throws SQLException :
transacionar() {
            Object result;

            try {
                result = proceed();
                if (conn != null) {
                    conn.commit();
                }
            } catch (Exception ex) {
                if (conn != null) {
                    conn.rollback();
                }
                throw new TransacaoException(ex);
            } finally {
                if (conn != null) {
                    conn.close();
                }
            }

            return result;
        }

//ATRIBUI O ESTADO DE FALSO À CONFIRMAÇÃO
AUTOMÁTICA DE TRANSAÇÕES.

    Connection around() throws SQLException
    : obterConexao() && cflow(operacaoTrasacionada
()) {

            if (conn == null) {
```

```
                conn = proceed();
                conn.setAutoCommit(false);
            }
            return conn;
        }

        public static class TransacaoException
            extends RuntimeException {
                public
TransacaoException(Exception cause) {
                    super(cause);
                }
        }

        private static aspect SoftenSQLException {
            declare soft : java.sql.SQLException
            : (call(void Connection.rollback())
            || call(void Connection.close()))
            && within(Transacoes);
        }

//DETECTA CHAMADA DE CONEXÕES

        pointcut gerenciamentoIlegal()
            : (call(void Connection.close())
            || call(void Connection.commit())
            || call(void Connection.rollback())
            || call(void onnection.setAutoCommit
(boolean)))
            && !within(Transacoes);
```

```
//DESVIA O FLUXO DE CHAMADA DE CONEXÕES.
        void Ilegal around(Connection conn) throws
SQLException:
          GerenciamentoIlegal( ) &&
     cflow(Transacionar( ) ) {
            System.out.println("Ponto de Junção:"
+thisJoinPoint);
            Return proceed(conn);
          }

     }
```

Fonte: Resende e Silva (2005), adaptado pelos autores.

Conclusão

Esse artigo mostrou uma forma de como a programação orientada a aspectos pode auxiliar no gerenciamento e controle das transações de armazenamento e recuperação de dados persistentes, além das conexões com o banco de dados utilizando a linguagem *AspectJ* (KICZALES et al., 2001).

A utilização da programação orientada a aspectos (POA) tem auxiliado a modelar dinamicamente softwares. No nível de implementação, é possível programar a dinâmica do software de maneira que seu comportamento possa evoluir, adaptar ou corrigir defeitos segundo as mudanças dos contextos em que estes estão inseridos.

Aplicações desta modelagem dinâmica podem ser encontradas no trabalho desenvolvido por Sommerville e Chitchyan (2004) onde uma particular aproximação da orientação a aspectos (OA) é usada para decomposição, reflexão e composição dos módulos de um software.

Nesta aproximação, o software é modelado como uma série de módulos chamados de hyperslices, e cada um representa somente um único interesse e

são compostos de diferentes pontos de junção (joinpoints). A aplicabilidade é mostrada em um cenário onde classes persistentes armazenam e recuperam dados em um banco de dados orientado a objetos.

Referências Bibliográfcas

BECKER, U.; DAL: A design based aspect language for distribution control. In: proceeding of the 12th International Workshop on Aspect Oriented Programming at ECOOP, Bursels, Belgium, 1998.

BONER, J.; VASSEUR, A.; **Aspectwerkz 2002**. Disponível em http://aspectwerkz.codehans.org, acessado em 24 de julho de 2006.

BURKE B.; Jboos. **Aspect Oriented Programming 2003**. Disponível em http://labs.jboss.com/portal/jboosaop/index.html, acessado em 24 de julho de 2006.

ELRAD, T.; KICZALES, T.; AKSIST, M.; LIEBERSSHER, K.; OSSHER, H. Discussing aspects of AOP, **Communications of the ACM**, v.44, n.10, p.33-38, 2001.

HIRSCHFELD, R.; Aspects – Aspect oriented programming with squeak. In: Revised papers from the International Conference Net Object Day on Objects, Components, Architectures, Services and Applications for a Networked World, Erfurt, Alemanha, 2003.

KARR, D. A.; RODRIGUES, C.; LOYALL, J. P.; SCHANTZ, R. E.; KRUSHNAMURTHY, Y.; PYA-RALI, I.; SCHMIDT, D. C.; Application of the QUO- Qos framework to a distributed video application. In: Proceeding of the Third International Symposium on Distributed Objects and Applications, Berlim,Alemanha, 2001.

KICZALES, G.; HILSDALE, E.; HUGUNIN, J.; KERSTEIN, M.; GRISWOLD, W.G.; An overview of AspectJ. **Lecture Notes in Computer Science**, v.2072, p.327-355, 2001.

KICZALES, G.; LAMPING, J.; MENHDHEKAR, A.; MAEDA, C.; LOPES, C.; LOINGTIER, J. M.; IRWIN, J.; Aspect Oriented Programming. **In: Proceedings European Conference on Object Oriented Programming,** v.1241, Berlim, Heidelberg and New York, p.220-242, 1997.

KIM, H. **Aspect C#**: an AOSD implementation for C#. Dissertação de Mestrado, Trinity College, Dublin, Ireland, 2002.

LEMOS, L. A. O.; **Teste de programas orientados à aspectos**: uma abordagem estrutural para AspectJ. Dissertação de Mestrado, São Carlos, Brasil, 2005.

LOPES, C. V. D.; **A language framework for distributed programming**. Tese de Doutorado, College of Computer Science, Notheastern University, Boston, MA, 1997.

RESENDE, A. M. P.; SILVA, C. C.; **Programação Orientada à Aspectos em Java:** desenvolvimento de Software Orientado à Aspectos, Rio de Janeiro, Brasport, 2005.

ROCHA, D. A.; **Uma ferramenta baseada em aspectos para apoio ao teste funcional de programas java**. Dissertação de Mestrado, São Carlos, Brasil, 2005.

SPINCZYK, O.; GAL, A.; SCHRODER-PREIKS CHAT, W.; Aspect C++: An aspect oriented extension to the C++ programming language. **In: proceeding of the 14th International Conference on Technology of Object Oriented Language and Systems** (Tools Pacific 2002), Sydney, Austrália, 2002.