Uso de Programação Orientada a Aspecto no Desenvolvimento de Aplicações que utilizam conceitos de Tecnologia Adaptativa

R. A. Casachi, A. R. Camolesi

Rafael.Casachi@raizen.com, camolesi@femanet.com.br
Coordenadoria de Informática, Fundação Educacional do Município de Assis, Brasil

***Resumo* — Este trabalho tem por objetivo apresentar os conceitos de Tecnologia Adaptativa, de Programação Orientada a Aspecto e o uso da Programação Orientada a Aspectos no desenvolvimento de aplicações que utilizam-se dos conceitos de Tecnologia Adaptativa.**

***Palavras Chaves*— Tecnologia Adaptativa, Programação Orientada a Aspectos, Sistemas de Informação.**

I. INTRODUÇÃO

SIMPLICIDADE e eficiência sempre foram necessidades primordiais no desenvolvimento de software. As linguagens estão se desenvolvendo através destes fatores para facilitar as tarefas de implementação e otimizar os projetos.

As evoluções nas linguagens de programação chegaram até ao Paradigma de Programação Orientada a Objetos (POO) que é o método mais utilizado em desenvolvimento atualmente. A POO possui a capacidade de abstrair os objetos reais e abstratos em classes garantindo melhor visibilidade ao código. Porém, mesmo com seus recursos, a POO não consegue tratar algumas características dos sistemas corretamente, como por exemplo, o controle de acesso (logging).

Em 1997, Gregor Kiczales e alguns cientistas do PARC (Xerox Palo Alto Research Center) desenvolveram o Paradigma de Programação Orientada a Aspecto (POA) [1]. Tal paradigma tem por objetivo solucionar as falhas da Programação Orientada a Objeto com relação à unificação dos códigos secundários (crosscutting concerns) ao código principal. Segundo [2] o mecanismo da Programação Orientada a Aspecto busca separar os interesses transversais das classes em módulos bem definidos e centralizados, nos quais um analista pode dedicar-se a um interesse de forma independente.

Mesmo com a POA, a necessidade de uma linguagem de componentes eficiente é fundamental. Esta linguagem de componentes pode ser um paradigma orientado a objetos ou pode ser uma linguagem estruturada, conforme descrito em [3]. Porém a POO, possui um melhor nível de abstração e possui um melhor suporte ao paradigma de Kiczales [1].

Uma outra característica importante nas aplicações complexas é a possibilidade das mesmas automodificar seu comportamento. Neste contexto o uso da tecnologia adaptativa tem apresentado bons resultados e permite aos desenvolvedores criar aplicações que se automodificam durante a sua execução. Neste trabalho será apresentado um estudo que visa a aplicação da Programação Orientada a Aspecto na implementação de aplicações adaptativas.

Este trabalho está organizado da seguinte forma, a seção II traz os principais conceitos da Programação Orientada a Aspectos e a sua forma de utilização. Na sequencia, na seção III são apresentados os conceitos de Tecnologia Adaptativa e uma comparação entre os seus conceitos e a orientação a aspectos será descrito. Um estudo de caso, com o objetivo de ilustrar a utilização da Programação Orientada a Aspectos no desenvolvimento de aplicações que utilizam os conceitos de Tecnologia Adaptativa será realizado na seção IV. Por fim, na seção V serão tecidas algumas conclusões e trabalhos futuros.

II. SEPARAÇÃO DE INTERESSES

Um sistema possui características, princípios, requisitos, algoritmos ou idéias similares. Este conjunto de similaridades chama-se interesses.

A separação destes interesses constitui-se de um método para dividir estes requisitos a fim de melhor solucionar o problema. "A estratégia de dividir para conquistar é bastante comum para a solução de problemas de qualquer natureza" [4].

A proposta de dividir os problemas de um sistema para melhorar a solução tem sido utilizada desde a programação estruturada. Existiam diversos algoritmos conhecidos como *Divide-and-conquer algorithms* (do inglês, Algoritmos Dividir-e-Conquistar) que dividiam o problema em subproblemas para tornar mais fácil a sua resolução. Estes subproblemas eram solucionados independentemente e combinados para solucionar o problema original [5].

Em um sistema comum, os interesses estão espalhados em todo o código (*scattering code*). Isto dificulta a manutenção do sistema, além de deixar o código ilegível. Na Figura 1, podemos verificar como um interesse se espalha em diversas classes:

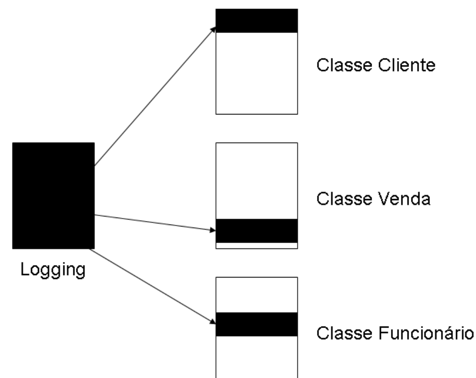

Figura 1. Código Espalhado por Diversas Classes

Além do código espalhado, pode-se verificar que uma classe acumula vários interesses, ocorrendo outro fenômeno chamado código emaranhado (*tangled code*), conforme exemplo da Figura 2.

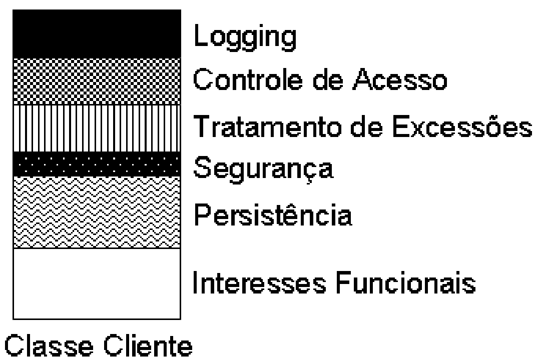

Figura 2. Código com interesses emaranhados no interesse funcional

A fim de evitar que ocorrências como estas aconteçam, separamos os nossos interesses em dois: Interesses Funcionais e Interesses Sistêmicos.

A. INTERESSES FUNCIONAIS

Os interesses funcionais (ou *core concerns*) são as características do domínio da aplicação, ou seja, são requisitos para que a aplicação atenda aos requisitos de funcionamento do sistema.

Quando um interesse funcional está desorganizado e ilegível, toda a aplicação pode ser comprometida e a probabilidade de erro pode ser muito grande.

B. INTERESSES SISTÊMICOS

Os interesses sistêmicos são os requisitos que dão suporte aos interesses funcionais. Estes interesses sistêmicos são à base do paradigma de programação orientado a aspecto, assim, necessitam ser implementados separadamente para serem controlados e organizados [6].

Para identificar os interesses sistêmicos é necessário analisar o esqueleto do programa (em um sistema que não foi implementado em POA) ou localizar as características que não fazem parte do escopo da funcionalidade.

C. ASPECTJ E O PROCESSO DE COMBINAÇÃO

Os códigos funcionais, implementados em uma linguagem de componentes, e os códigos sistêmicos, implementados em uma linguagem aspectual, são unidos através de um processo chamado combinação (ou *Weaving*, em inglês).

O processo de combinação utiliza de um combinador, ou Weaver, para unir os interesses funcionais e sistêmicos em um código intermediário. Após gerar este código intermediário, o arquivo é compilado por um compilador, conforme mostra o Diagrama de Petri da Figura 3.

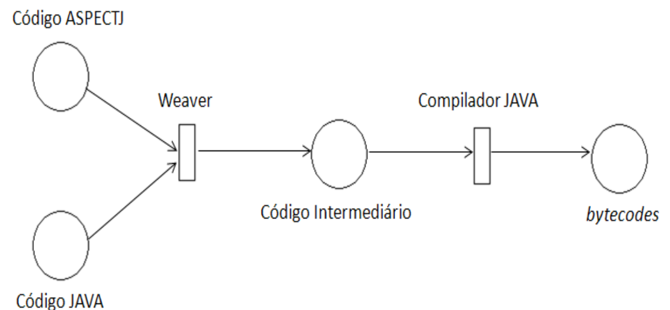

Figura 3. Representação do processo de combinação com uma Rede de Petri

O combinador pioneiro é o AspectJ, criado em 1997 junto com o paradigma POA. O AspectJ é utilizado para ser o combinador aspectual da linguagem de componentes Java e a linguagem de aspectos AspectJ. Em 2002, a IBM agregou o projeto AspectJ à IDE Eclipse, melhorando o suporte ao combinador.

D. PONTOS DE JUNÇÃO (JOIN POINTS)

Os interesses separados devem ser unidos no processo de combinação através de um ponto de junção.

Conforme descrito em [6] *"O ponto de junção é um ponto bem definido na execução de um sistema"*, ou seja, eles são pontos selecionados para controlar o fluxo do programa.

Para que esta união seja bem-sucedida a definição de dois elementos no aspecto é essencial: o ponto de atuação e o adendo.

E. PONTOS DE ATUAÇÃO (POINTCUTS)

O ponto de atuação é o componente da POA que identifica em quais pontos de junção do código principal o aspecto será combinado.

A estrutura de um ponto de atuação é composta pela palavra-chave "*pointcut*" seguido pelo tipo de acesso que o ponto de atuação irá possuir (privado ou público), o nome do ponto de junção com seus possíveis parâmetros e as assinaturas dos pontos de junções capturados com os seus designadores. No exemplo abaixo, o ponto de atuação chamado registra será executado quando o método *public String getNome(),* da classe Cliente, for chamado (*call*).

```
pointcut registra() : call(public String
          Cliente.getNome());
```

Um designador é um elemento encontrado no ponto de atuação que identifica o tipo de comportamento que um ponto de junção deve ter para que possa ser capturado. A Figura 4 apresenta a lista de todos os designadores de um ponto de atuação.

Designador	Descrição
Execution	Corresponde à execução de um método ou de um construtor
Call	Corresponde à chamada para um método ou para um construtor
Initialization	Corresponde à inicialização de um objeto, representada pela execução do primeiro construtor para uma classe
Handler	Corresponde à manipulação das excessões
Get	Corresponde à referência para um atributo de uma classe
Set	Corresponde à definição de um atributo de um classe
This	Retorna o objeto associado com o ponto de junção em particular ou limita o escopo de um ponto de junção utilizando um tipo de classe
Target	Retorna o objeto alvo de um ponto de junção ou limita o escopo do mesmo
Args	Expõe os argumentos para o ponto de junção ou limita o escopo de um ponto de atuação
Cflow	Retorna os pontos de junção na execução do fluxo de outro ponto de atuação
Cflowbelow	Retorna os pontos de junção na execução do fluxo de outro ponto de atuação, exceto o ponto corrente
Staticinitialization	Corresponde à inicialização dos elementos estáticos de um objeto
Withincode	Corresponde aos pontos contidos em um método ou construtor
Within	Corresponde aos pontos de junção contidos em um tipo específico
If	Permite que uma condição dinâmica faça parte de um ponto de atuação
Adviceexecution	Corresponde aoo adendo (advice) do ponto de junção
Preinitialization	Corresponde à pré-inicialização de um ponto de junção

Figura 4. Lista de designadores de um ponto de atuação (In: GOETTEN; WINCK, 2006, P87)

F. ASSINATURA DE UM PONTO DE JUNÇÃO

A função da assinatura é descrever em qual ponto de junção, o aspecto deve ser combinado. Esta assinatura pode variar conforme o tipo de designador utilizado. A Figura 5 descreve a lista das assinaturas por designador.

Designador	Assinatura
Execution	execution(<tipo_acesso> <valor_retorno> <classe>.<nome_metodo>({<lista_parametros>}))
Call	call(<tipo_acesso> <valor_retorno> <classe>.<nome_metodo>({<lista_parametros>}))
Initialization	initialization(<tipo_acesso> <classe>.new({<lista_parametros>}))
Handler	handler(<tipo_excessao>)
Set	get(<tipo_campo> <classe>.<nome_campo>)
Set	set(<tipo_campo> <classe>.<nome_campo>)
This	this(<tipo ou identificador>)
Target	target(<tipo ou identificador>)
Args	args(<tipo ou identificador>,..)
Cflow	cflow(<pointcut>)
Cflowbelow	cflowbelow(<pointcut>)
staticinitialization	staticinitialization(<typePattern>)***
Withincode	withincode(<tipo_acesso> <valor_retorno> <classe>.<nome_metodo>({<lista_parametros>}))
Within	within (<typePattern>)***
If	if(<Expressão Booleana>)
Adviceexecution	adviceexecution()
preinitialization	preinitializantion(<tipo_acesso> <classe>.new({<lista_parametros>}))
Métodos que geram exceções	<designador>(<tipo_acesso> <classe>.<nome_metodo>({<lista_parametros>}) throws <excessao_a_ser_tratada>)

Figura 5. Lista de Assinaturas por designador

G. ADENDO (ADVICE)

Quando o ponto de atuação une o aspecto ao ponto de junção um adendo entra em operação. Um adendo é um bloco de código similar ao método que é executado num momento auto-configurado.

O adendo possui todo o código que será combinado à aplicação. Um aspecto pode ter vários adendos de diversos tipos, porém eles não possuem nome de referência, sua única referência é o ponto de atuação.

Um adendo varia conforme o momento de sua execução. Este pode ter um modificador *before*, *after* ou *around* [7].

O adendo before é mais simples. Este é executado antes da ocorrência do ponto de junção e não possui nenhum critério (como no adendo *after*) ou possibilidade de seu contexto ser alterado (como no adendo *around*) [4].

O adendo *after* é o bloco de código executado no fim da computação do ponto de junção e é divido em três subtipos de acordo com o critério da execução. Se o objetivo é utilizar o adendo apenas após a computação correta do ponto de junção, então se deve utilizar o adendo after returning. Se há a necessidade de execução do adendo depois da computação sem sucesso de um ponto de junção, então se deve utilizar o adendo after throwing. Porém se o resultado da computação não influenciar, pode-se utilizar apenas *after* [4].

O último tipo de adendo é o around que é executado durante a execução do ponto de junção ou bloco de código, ou seja, o adendo around cerca toda a execução do ponto de junção. Para executar o ponto de junção é necessário discriminar a palavra-chave *proceed()* com os mesmos argumentos coletados no corpo do adendo, caso este comando não for encontrado no adendo, o ponto de junção é contornado [4].

A sequência de execução de um adendo é descrita no autômato da Figura 6, o estado inicial *q0* é o momento em que o ponto de atuação encontra um ponto de junção. A partir deste momento, os adendos do aspecto seguem a estrutura descrita. A sequência é simples, qualquer adendo do tipo *before* no aspecto é executado primeiro e antes do ponto de junção. Um adendo *around* é executado após de um adendo *before* e quando houver um comando *proceed* o ponto de junção é executado, e depois, o adendo *around* é finalizado. Após a execução do ponto de junção ou do adendo *around* pode haver um adendo *after* que pode ser de três tipos: quando o ponto de junção foi executado com sucesso (*after returning()*), quando o ponto de junção retornou erro (*after throwing()*) ou quando o estado final do ponto de junção não importar (*after()*).

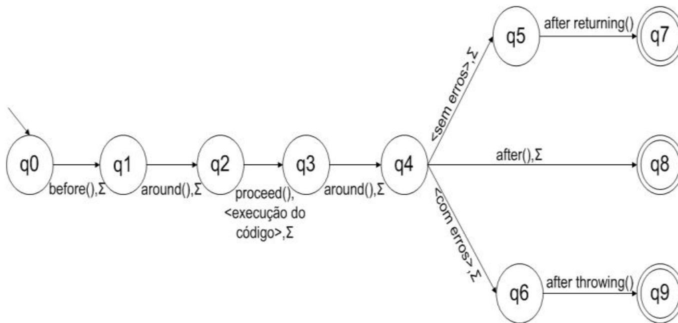

Figura 6. Autômato da ordem de execução de um adendo

H. ASPECTO

Um aspecto é a unidade básica do POA, assim como o objeto é a unidade básica do POO. *"Aspecto é uma parte de um programa POA que separa os interesses ortogonais dos interesses principais"* [6].

Um aspecto é como uma classe. Ela possui os atributos modificadores de acessos (*private*, *protected* e *public*), a palavra-chave *aspect* e o nome de referência. E seu bloco de código abriga todos os elementos da programação orientada a aspecto.

III. TECNOLOGIA ADAPTATIVA

Um comportamento é denominado adaptativo sempre que este modifica-se dinamicamente, em resposta aos estímulos de entrada, sem interferência de agentes externos e de seus usuarios. Historicamente, o estudo da tecnologia adaptativa teve início com trabalhos realizados na busca de uma forma eficaz e de baixa complexidade para a resolução do problema da análise sintática de Linguagens Livres de Contexto [8]. Diversos trabalhos foram realizados nesta linha e em [9] foi apresentado o autômato e o transdutor adaptativo como dispositivos de reconhecimento e transdução sintática.

Com base no trabalho desenvolvido em [9], foram realizados outros trabalhos aplicando o uso da tecnologia adaptativa no projeto de sistemas reativos. Em [10] é apresentada uma evolução da notação de *Statecharts* [11], na qual foram acrescentadas algumas características provenientes da teoria de Autômatos Adaptativos.

Dispositivos adaptativos devem ser auto-modificáveis e as possíveis mudanças no comportamento do dispositivo devem ser conhecidas por todas as atividades desempenhadas pelo mesmo, em qualquer passo de operação na qual as mudanças ocorrem. Entretanto, dispositivos adaptativos devem ser capazes de eliminar todas as situações que causam possíveis modificações não desejadas e para adequar reações há imposições que correspondem a mudanças no comportamento do dispositivo. Os dispositivos adaptativos considerados têm seu comportamento baseado na operação de dispositivos subjacentes não adaptativos que são descritos por um conjunto finito de regras. Um dispositivo adaptativo dirigido por regras pode ser obtido anexando ações adaptativas as regras do formalismo subjacente.

Em [12] foi proposta uma representação genérica para formalismos adaptativos que permite representar e manipular dispositivos adaptativos dirigidos por regras e estados. A principal característica desta formulação é que preserva totalmente a natureza do formalismo adaptativo, desde que o dispositivo adaptativo resultante pode ser facilmente entendido por pessoas que possuem conhecimento em relação ao dispositivo subjacente.

A. Dispositivos Adaptativos Dirigidos por Regras

A formalização apresentada em [12] fundamenta-se em um mecanismo adaptativo (AM) que envolve o núcleo de um dispositivo subjacente, não adaptativo (ND). Desta forma, um dispositivo adaptativo (AD) é definido formalmente por um óctuplo AD= (C, AR, S, c_0, A, NA, BA, AA).

Nesta formulação C é o conjunto de todas as possíveis configurações de ND e $c_0 \in C$ é a sua configuração inicial. S é o conjunto de todos os possíveis eventos de que se compõem a cadeia de entrada de AD e o conjunto A representa as configurações de aceitação para ND.

Os conjuntos BA e AA são conjuntos de ações adaptativas. NA é um conjunto de todos os símbolos que podem ser gerados com saídas por AD, em resposta à aplicação de regras adaptativas. AR é o conjunto das regras adaptativas que definem o comportamento adaptativo de AD e é dado por uma

relação Ar ⊆ BA×C×S×C×NA×AA, na qual, ações adaptativas modificam o conjunto corrente de regras adaptativas AR de AD para um novo conjunto AR adicionando e/ou eliminando regras adaptativas em AR.

Com base na formulação geral para dispositivos adaptativos foi apresentado em [13] a formulação geral para o Autômato de Estados Finitos Adaptativo. Um Autômato Finito (dispositivo subjacente) é um dispositivo simples e está intimamente relacionado com a classe das Linguagens Regulares, conforme denominado na hierarquia de Chomsky. Em [14] um Autômato de Estados Finitos (AEF) é constituído por um conjunto de estados e por uma função de transição que informa quais transições o mecanismo deve executar, a partir de um estímulo obtido na sua cadeia de entrada.

Um dispositivo torna-se adaptativo ao agregar uma camada adaptativa envolvendo o seu núcleo subjacente. Nesse contexto define-se um Autômato de Estados Finito Adaptativo (AEF_{Adp}) por meio do acréscimo de uma camada adaptativa envolvendo o seu núcleo subjacente. A Figura 7 ilustra a estrutura de um AEF_{Adp} constituída por uma camada adaptativa contendo funções anteriores e posteriores e suas respectivas ações.

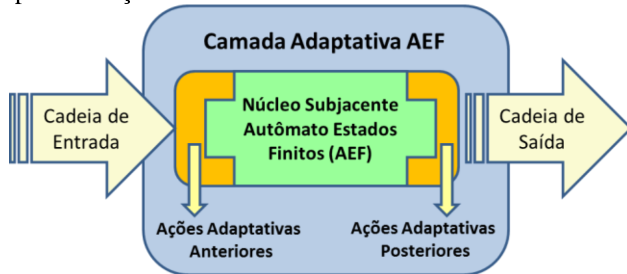

Figura 7. Dispositivo Autômato de Estados Finitos Adaptativo

B. *Tecnologia Adaptativa e Programação Orientada a Aspectos.*

Ao fazer uma relação dos conceitos de Tecnologia Adaptativa aos conceitos de Orientação a Aspectos encontra-se uma forte relação entre estas duas tecnologias. Como dito anteriormente a orientação a aspecto se fundamenta em aspectos e na combinação destes. Ao projetar uma aplicação orientada a aspectos devemos representar os seus pontos de junção (*join points*) e os pontos de atuação (*pointcuts*). Um ponto de atuação une o aspecto ao ponto de junção um adendo entra em operação. Um adendo é um bloco de código similar ao método que é executado num momento auto-configurado. Os adendos podem ser de três tipos; *before*, *after* ou *around*. Ao realizar o mapeamento dos conceitos de Tecnologia Adaptativa aos conceitos de Programação Orientada a Aspecto realiza-se o mapeamento das ações adaptativas anteriores (*before*) aos adendos *before* de programação orientada a aspecto. As funções adaptativas posteriores (*after*) devem ser mapeadas aos adendos *after* , e por fim, os adendos *around* mapeiam os trechos de código de execução normal (tecnologia adaptativa) que são envolvidos pelas funções adaptativas conforme ilustrado na Figura 8.

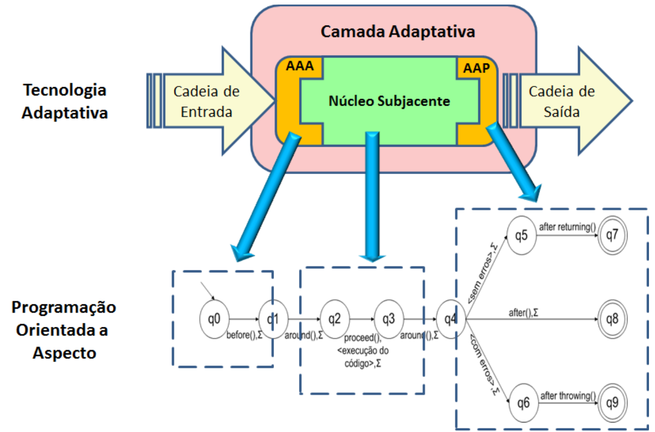

Figura 8. Mapeamento dos conceitos de Tecnologia Adaptativa para Programação Orientada a Aspectos

IV. Estudo de caso

Com o intuito de aplicar os conceitos do Paradigma de Programação Orientado a Aspecto associado aos conceitos de Tecnologia Adaptativa para o desenvolvimento de aplicações foi realizado um estudo de caso com o foco no desenvolvimento de um Sistema para Gestão de Bibliotecas. Em tal sistema os aspectos atuam no código principal para modificar seu comportamento.

A aplicação desenvolvida consiste em um software para gestão que tem como principais funcionalidades manter os cadastros de alunos, de funcionários, de livros e exemplares. Esta aplicação permite o empréstimo de exemplares e sua futura devolução, além de proibir o empréstimo aos alunos inadimplentes. Para manter a integridade, o software deve restringir o acesso de alguns usuários a certas funcionalidades do sistema de acordo com seu cargo. A Figura 9 apresenta o caso de uso geral do referido sistema.

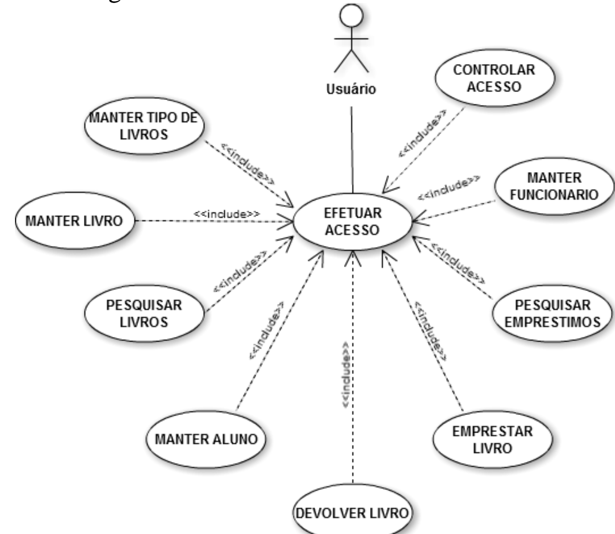

Figura 9. Caso de uso geral Sistema Gestão Biblioteca

O sistema foi projetado para ser implementado na Linguagem Java, para o armazenamento dos dados foi utilizado o banco de dados HSQLDB e o framework de persistência a dados

Hibernate[1] foi utilizado para auxiliar nas operações de manipulação dos dados necessárias ao sistema. Na Figura 10 é apresentado um Diagrama de Classes resumido que ilustra as principais classes que compõe o modelo objeto-relacional da aplicação desenvolvida.

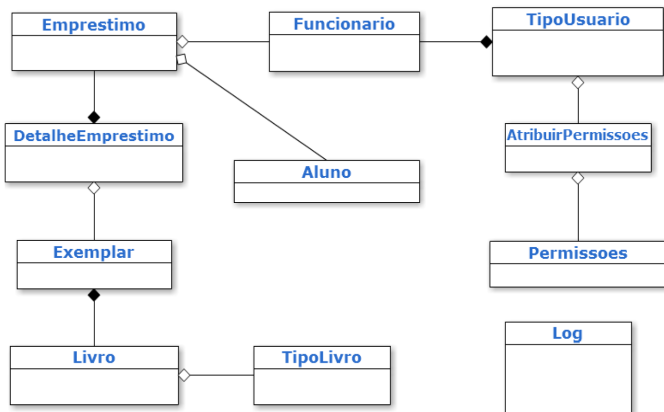

Figura 10. Diagrama de classes resumido

Ao organizar a arquitetura do sistema utilizou-se como base o conceito de separação de interesses, neste contexto o software foi dividido em duas partes: o pacote *Componentes* e o pacote *Funcionalidades*. Na Figura 11, pode-se verificar a estrutura do sistema, onde todos os aspectos estão atuando diretamente nos objetos instanciados nas classes do pacote ***Interfaces***. Este pacote contém os componentes do sistema que interagem com o usuário. As classes do pacote ***Interfaces*** comunicam-se com o banco de dados por meio do framework de persistência *Hibernate* que dá suporte a manipulação de dados no sistema.

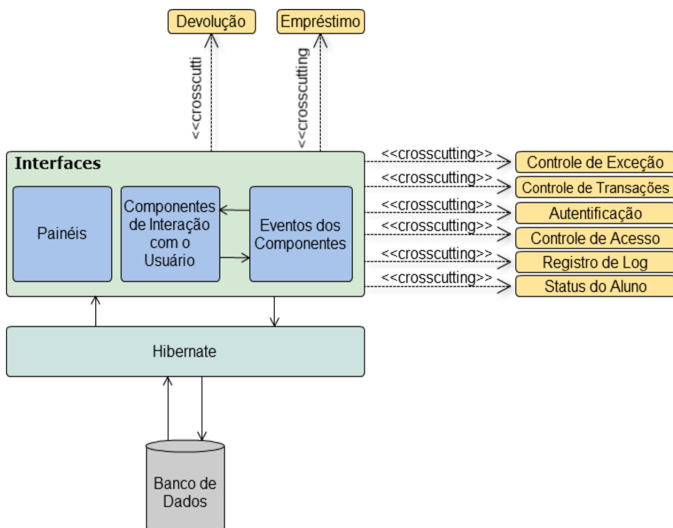

Figura 11. Arquitetura do Sistema de Gestão Bibliotecas

Devido a não existência de métodos e técnicas específicos para a modelagem de sistema que utilizam-se dos conceitos de Orientação a Aspectos, foi utilizado o Diagrama de Casos de Uso (UML) para descrever o comportamento do sistema. Na Figura 12 é apresentado um caso de caso de uso do sistema no qual são destacados os interesses ortogonais (elipses em laranja). Por exemplo, os interesses *Alterar Status do Exemplar*, *Consultar Multa* e *Impressão da Fatura* são interesses que possuem o mesmo objetivo, o de manter a integridade na regra de negócio "devolver livros". Desta forma tais interesses foram agrupados em um único aspecto. Este mesmo critério foi utilizado para agrupar os interesses *Determinar Prazo de Devolução*, *Alterar Status do Exemplar* e *Checar Disponibilidade do Exemplar* no aspecto *Empréstimo*. O aspecto *Status do Aluno* agrupa os interesses *Alterar Status do Aluno* e *Checar Disponibilidade do Aluno*.

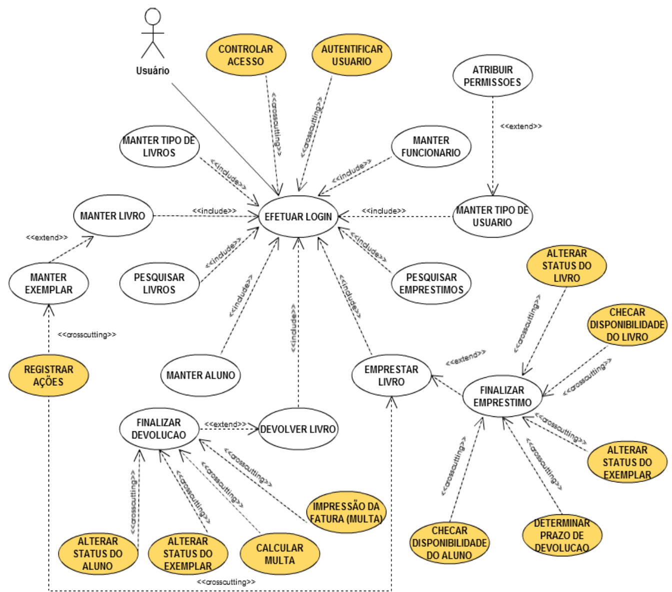

Figura 12 - Diagrama de Casos de Uso

Existem aspectos que não são possíveis de ser representados no caso de uso, como os aspectos que controlam exceções e transações. Os aspectos de controle de exceção capturam os pontos de junção que necessitam de um bloco de tratamento de exceções (*try/catch)* e inserem os mesmos no código no momento da combinação. O mesmo acontece com o controle de transações, o aspecto insere um código de iniciação da transação antes da sua utilização, e conforme o resultado da transação insere um comando de *commit* do banco ou um *rollback*, conforme código abaixo:

```
before() : transaction(){
   HibernateUtil.begin();
}

after() returning() : transaction(){
   HibernateUtil.commit();
}

after() throwing() : transaction(){
   HibernateUtil.rollback();
}
```
Figura 13. Trecho de Código Orientado a Ascpecto

Durante a implementação, os componentes foram programados primeiro utilizando o paradigma de programação orientado a objetos, mais especificamente a linguagem de programação Java. Os aspectos programados por último e

[1] O **Hibernate** é um framework para o mapeamento objeto-relacional escrito na linguagem Java disponível em http://www.hibernate.org/

divididos em dois tipos de acordo com suas funcionalidades: suporte ao sistema e suporte às regras de negócio.

A. Implementação Funcionalidade de Acesso ao Sistema

O código abaixo descreve o aspecto de controle de acesso onde o ponto de junção é a chamada do método *checarPermissao* na classe *Principal*. Este método atribui falso para todos os acessos no sistema. Porém quando este método é chamado, o aspecto altera os valores que serão atribuidos ao acesso.

```
public void buscarPermissoes(Funcionario user){
AtribuirPermissaoDao dao = new
              AtribuirPermissaoDao();
listPermissoes = (List<AtribuirPermissoes>)
              dao.pesquisar(user.getTipoUsuario());

String perm[] = new String[9];
perm[0] = "CADASTRO DE LIVROS";
perm[1] = "PESQUISA DE LIVROS";
perm[2] = "CADASTRO TIPO DE LIVRO";
perm[3] = "CADASTRO DE ALUNOS";
perm[4] = "CADASTRO DE FUNCIONÁRIOS";
perm[5] = "CADASTRO DE TIPO DE USUARIO";
perm[6] = "RELATÓRIO DE LOGS";
perm[7] = "CADASTRO DE EMPRÉSTIMOS";
perm[8] = "PESQUISA DE EMPRÉSTIMOS";

boolean autPerm[] = new boolean[9];
for(int i = 0; i < 9; i++)
              autPerm[i] = false;

checarPermissao(perm,autPerm,listPermissoes);
}

public void checarPermissao(String permissao[],
              boolean aut[],List
              <AtribuirPermissoes> atPerm){

  mntmCadastrarLivros.setEnabled(aut[0]);
  mntmPesquisarLivros.setEnabled(aut[1]);
  mntmTipoDeLivros.setEnabled(aut[2]);
  cadastrarAlunoBtn.setEnabled(aut[3]);
  cadastrarFuncionarioBtn.setEnabled(aut[4]);
  mntmTipoDeUsuario.setEnabled(aut[5]);
  mntmLog.setEnabled(aut[6]);
  mntmNovoEmprestimo.setEnabled(aut[7]);
  mntmPesquisarEmprestimos.setEnabled(aut[8]);
}

public aspect AspectoControleAcesso {

  pointcut  controle(String  perm[],  boolean  at[],
List <AtribuirPermissoes> atPerm) :
    call(void
telas.Principal.checarPermissao(String[],boolean[],
List <AtribuirPermissoes>))
      && args(perm[],at[],atPerm);

  void   around(String   perm[],boolean   at[],   List
<AtribuirPermissoes> atPerm) :
controle(perm[],at[], atPerm){
    boolean aut = false;
    for (int cont = 0; cont < 9; cont++){
      aut = false;
      for  (int  x  =  0;  x  <  atPerm.size()  &&  aut  ==
false; x++){
  if(atPerm.get(x).getPermissao().getPermissao().com
pareTo(perm[cont]) == 0){
          aut = true;
        }
      }
      if(aut){
        at[cont] = true;
      } else {
        at[cont] = false;
      }
    }
      proceed(perm,at,atPerm);
  }
}
```

Figura 12. Trecho de Código Orientado a Ascpecto

O aspecto captura os três argumentos do método *checarPermissao*: *perm*[], *at*[] e atPerm[]. O vetor de String *perm*[] possui todas as permissões de acesso, cada posição do vetor booleano *at*[] é uma acesso que foi permitido ao usuário e a lista *atPerm* possui a relação de permissões que foi atribuida ao usuário que obteve acesso. Deste modo o aspecto *AspectoControleAcesso* combina no ponto de junção deste método e executa o adendo *around*(). A adendo *around* checa se usuário possui o acesso, se possuir este acesso ele seta *true* na posição correspondente no vetor *at*[]. O comando proceed(perm,at,atPerm) encerra a primeira fase do adendo *around* enviando os valores modificados ao método *checarPermissao*, para que o mesmo possa alterar o acesso conforme aos novos valores atribuidos. Como após o comando *proceed(perm,at,atPerm)* não existe mais nenhum comando, a instância do aspecto é encerrado.

B. Controle de transação

Considerando o código abaixo que é a forma de implementar um controle de transação sem a utilização da linguagem de programação orientada a aspecto, o código necessitaria ser estruturado individualmente para cada abertura da transação que o sistema possuir. Além de aumentar o número de linhas do seu código, a visibilidade, a complexidade do código e o tempo irão ser muito grandes.

```
btnSalvar.addActionListener(new ActionListener() {
   public void actionPerformed(ActionEvent arg0) {
      try {
           HibernateUtil.begin();
           botaoSalvar();
           HibernateUtil.commit();
           } catch (Exception e) {
           HibernateUtil.rollback();
      }
   }
});
btnSalvar.setFont(new  Font("Segoe  UI",  Font.PLAIN,
11));
btnSalvar.setBounds(698, 462, 89, 23);
getContentPane().add(btnSalvar);
```

Figura 13. Trecho de Código Orientado a Ascpecto Botão *Salvar*

Se utilizar o conceito de aspectos, pode-se criar um único aspecto que controle a transação do *hibernate* para todos os outros pontos que necessitaram deste comando. Assim, a implementação abaixo demonstra o uso de um aspecto para o código do botão *salvar* apresentado na Figura 13.

```
public aspect AspectoTransactionHibernate {
   pointcut    transaction()    :        call    (void
telas.CadastrarAluno.botaoSalvar());
   before() : transaction(){
      HibernateUtil.begin();
   }
   after() returning() : transaction(){
      HibernateUtil.commit();
   }
   after() throwing() : transaction(){
      HibernateUtil.rollback();
   }
}
```

Figura 14. Trecho de Código *AspectoTransactionHibernate*.

Os métodos *HibernateUtil.begin(), HibernateUtil.commit()* e *HibernateUtil.rollback()* estão encapsulados dentro do aspecto *AspectoTransactionHibernate* (Figura 14), podendo ser utilizados em qualquer outro ponto de junção descrito dentro do corpo do aspecto, não necessitando de nenhuma chamada externa ao aspecto. Deste modo, o código principal fica limpo deste interesses, como observa-se no código do botão salvar após a aplicação do conceito da POA ilustrado na Figura 15.

```
btnSalvar.addActionListener(new ActionListener() {
   public void actionPerformed(ActionEvent arg0) {
           botaoSalvar();
   }
});
btnSalvar.setFont(new  Font("Segoe  UI",  Font.PLAIN,
11));
btnSalvar.setBounds(698, 462, 89, 23);
getContentPane().add(btnSalvar);
```

Figura 15. Trecho de Código Orientado a Ascpecto Botão *Salvar* com o uso de POA.

V. CONCLUSÃO

A forma de implementação dos componentes, inicialmente ter sido realizada Orientado a Objetos, e depois numa segunda fase o projeto foi convertido para POA permitiu a visualização de como seria a transformação de um projeto Orientado a Objetos em um projeto com o *AspectJ*, além de mostrar as dificuldades da implantação de uma nova funcionalidade ao programa e a versatilidade na manutenção dos aspectos.

O uso da POA permitiu a implantação da Tecnologia Adaptativa de uma forma fácil e segura, também permite que novas funcionalidades (aspectos) sejam adicionados ao software sem a necessidade de modificar o código fonte já produzido. O programador deve apenas acrescentar novos aspectos ao software e o mesmo se adapta ao código já existente. A POA foi interessante em relação a implementação de Tecnologia Adaptativa, porém comprova-se que os aspectos de TA não são implementados completamente, uma vez que os programas em POA são combinados antes da compilação e os programas produzidos não são adaptativos em tempo de execução.

REFERÊNCIAS

[1] G. KICZALES, J. LAMPING, A. MENDHEKAR, C. MAEDA, C.V. LOPES, J. LOINGTIER. "Aspect-Oriented Programming. In: *European Conference on Object-Oriented Programming (ECOOP")*, 06,1997. Finlândia. Anais Springer-Verlag LNCS 1241, 06, 1997

[2] J. M. LOUREIRO, J. P. C. S. G. COSTA, R. M. S. B. FONSECA, V. A. N. LOUREIRO, "Programação Orientada a Aspecto". *FEUP Universidade do Porto*, Porto, Portugal.

[3] KRUPA, Artur. Analyse "Aspect-Oriented Software Approach and Its Application". Athabaska, Alberta, Canadá: Athabaska University, 2010.

[4] V. J. GOETTEN, D. WINCK. "AspectJ – Programação Orientada a Aspectos com Java", *Novatec Editora*, São Paulo, 2006.

[5] G. L. HEILEMAN, Data Structures, Algorithms, and Object-Oriented Programming *McGraw-Hill*, Singapore, 1996.V.

[6] R. SAFONOV, "Using Aspect-Oriented Programming for Trustwothy Software Development". *Wiley-Interscience*, New Jersey, 2008.

[7] R. BODKIN, R. LADDAD, Zen and the art of Aspect-Oriented Programming. *Linux Magazine*, April, 2004.

[8] J. J. NETO e M. E. S. MAGALHÃES, "Um Gerador Automático de Reconhecedores Sintáticos para o SPD". *VIII SEMISH - Seminário de Software e Hardware*, pp. 213-228, Florianópolis, 1981.

[9] J. J. NETO. "Contribuições à metodologia de construção de compiladores". Tese de Livre Docência, USP, São Paulo, 1993.

[10] J. R. ALMEIDA "STAD - Uma ferramenta para representação e simulação de sistemas através de statecharts adaptativos". Tese de Doutorado, Escola Politécnica, Universidade de São Paulo, São Paulo, 1995.

[11] D. HAREL et al. "On the formal semantics of statecharts". In: Symposyum on logic in Computer Science, 2°, Ithaca, Proceedings, IEEE Press, pp. 54-64, New York, 1987.

[12] J. J. NETO. "Adaptive Rule-Driven Devices - General Formulation and Case Study". Lecture Notes in Computer Science. Watson, B.W. and Wood, D. (Eds.): Implementation and Application of Automata 6th International Conference, CIAA 2001, Springer-Verlag, Vol.2494, pp. 234-250, Pretoria, South Africa, July 23-25, 2001.

[13] A. R. CAMOLESI. "Proposta de um Gerador de Ambientes para a Modelagem de Aplicações usando Tecnologia Adaptativa". Tese de Doutorado, Escola Politécnica, Universidade de São Paulo, São Paulo, 2007

[14] H. LEWIS; C. H. PAPADIMITRIOUS. "Elements of the theory or computation". Prentice Hall, editor, 1998.

Rafael Casachi nasceu na cidade de Maracaí, São Paulo, Brasil, em 18 de Novembro de 1990. Estudante de Bacharel em Ciência da Computação na Fundação Educacional do Município de Assis (FEMA) em Assis, São Paulo, Brasil. Entre os anos de 2008 e 2011. Suas principais áreas de pesquisas são: Programação Orientada a Aspectos e desenvolvimento de Aplicações Java.

Almir Rogério Camolesi possui graduação em Processamento de Dados pela Fundação Educacional do Município de Assis (1992), mestrado em Ciência da Computação pela Universidade Federal de São Carlos (2000) e doutorado em Engenharia de Computação e Sistemas Digitais pela Universidade de São Paulo (2007). Atualmente é professor titular da Fundação Educacional do Município de Assis. Tem experiência na área de Ciência da Computação, com ênfase em Tecnologias Adaptativas, atuando principalmente nos seguintes temas:. tecnologia adaptativa, computação distribuída, teoria da computação, modelagem abstrata, ensino a distância,, algorítmos e programação para web.