Fundação Educacional do Município de Assis
Instituto Municipal de Ensino Superior de Assis
Campus "José Santilli Sobrinho"

RAFAEL ALESSANDRO CASACHI

APLICAÇÃO DO PARADIGMA DE PROGRAMAÇÃO ORIENTADA A ASPECTO NO DESENVOLVIMENTO DE SISTEMAS DE INFORMAÇÃO

Assis
2011

RAFAEL ALESSANDRO CASACHI

APLICAÇÃO DO PARADIGMA DE PROGRAMAÇÃO ORIENTADA A ASPECTO NO DESENVOLVIMENTO DE SISTEMAS DE INFORMAÇÃO

Trabalho de Conclusão de Curso apresentado ao Curso de Bacharelado em Ciência da Computação do Instituto Municipal do Ensino Superior de Assis – IMESA e Fundação Educacional do Município de Assis – FEMA, como requisito para a obtenção do Certificado de Conclusão.

Orientador: Drº Almir Rogério Camolesi

Área de Concentração: Informática

Assis
2011

FICHA CATALOGRÁFICA

CASACHI, Rafael Alessandro

Aplicação do Paradigma de Programação Orientada a Aspecto no Desenvolvimento de Sistemas de Informação / Rafael Alessandro Casachi. Fundação Educacional do Município de Assis – FEMA – Assis, 2011.

80p.

Orientador: Drº Almir Rogério Camolesi

Trabalho de Conclusão de Curso – Instituto Municipal de Ensino Superior de Assis - IMESA

1. AOP. 2. POA. 3. Programação Orientada a Aspecto. 4. *Aspect-Oriented Programming.*

CDD: 001.6

Biblioteca da FEMA

APLICAÇÃO DO PARADIGMA DE PROGRAMAÇÃO ORIENTADA A ASPECTO NO DESENVOLVIMENTO DE SISTEMAS DE INFORMAÇÃO

RAFAEL ALESSANDRO CASACHI

Trabalho de Conclusão de Curso apresentado ao Instituto Municipal de Ensino Superior de Assis, como requisito do Curso de Graduação, analisado pela seguinte comissão examinadora:

Orientador: Drº Almir Rogério Camolesi
Analisador: Drº Alex Sandro Romeo de Souza Poletto

Assis
2011

DEDICATÓRIA

Dedico este trabalho a minha família, que sempre apoiou as minhas decisões.

AGRADECIMENTOS

Agradeço, primeiramente, a Deus pela força e pela serenidade nos momentos mais difíceis.

Ao meu orientador, Dr. Almir Rogério Camolesi, pela confiança depositada em mim e pelo grande apoio, interesse e auxilio, sem ele, este trabalho não teria sido concluído.

A todos os professores da FEMA que criaram a base para que os meus conhecimentos pudessem ter algum fundamento e, em especial, ao professor Dr. Luiz Carlos Begosso que me apresentou o assunto e incentivou a sua elaboração; ao professor Fernando Cesar de Lima, pelo suporte e entusiasmo durante o desenvolvimento deste trabalho; e ao professor e coordenador de curso Dr. Alex Sandro Romeo de Souza Poletto pelo apoio no desenvolvimento do trabalho e pelas úteis sugestões.

A minha família pelo apoio e por entenderem à minha ausência em alguns momentos, aos amigos que incentivaram os meus esforços e a minha dedicação e a todos que colaboraram direta ou indiretamente, na execução deste trabalho.

E por último, a mim mesmo, que dediquei o meu tempo em favor do estudo de uma nova tecnologia para que esta possa ser uma gota da minha colaboração no mar que é a ciência.

“There is a pleasure in the pathless Woods;
There is rapture on the lonely shore;
There is society, where none intrudes,
By the deep sea, and music in its roar:
I love not the man less, but Nature more […]”

Lord Byron (1788-1824)

RESUMO

Este trabalho apresenta o conceito de separação de interesses e como utilizá-lo para desenvolver sistemas de informações utilizando o paradigma de Programação Orientada a Aspecto. O objetivo principal é demonstrar como este paradigma pode solucionar problemas como reutilização de códigos, visibilidade e organização que são comuns em outros paradigmas de programação. O trabalho também possui como objetivo exemplificar, através de um estudo de caso, a construção de aspectos que apliquem os conceitos de Programação Orientada a Aspecto na otimização do desenvolvimento de interesses ortogonais de um software. Deste modo é possível concluir que a utilização de um método mais eficiente de programação, como a programação orientada a aspecto, pode-se melhorar o desenvolvimento de softwares, diminuírem custos, prazos e recursos, além de aumentar a qualidade dos códigos produzidos e diminuir o tempo gasto com manutenções de sistemas e implantações de funcionalidades.

Palavras-chave: POA; Programação Orientada a Aspecto; Separação de Interesses

ABSTRACT

This paper presents the concept of concerns separation e how to use it to develop information systems using the paradigm of Aspect-Oriented Programming. The main goal is to demonstrate how this paradigm can solve problems as code reuse, visibility and organization that are common in other programming paradigms. The work also has as an objective to illustrate, through a case study, the construction of aspects that apply the concepts of Aspect-Oriented Programming in optimizing the development of crosscutting concerns of software. Thus it is possible to conclude that the use of a more efficient method of programming, such as Aspect-Oriented Programming, can improve the software development, reduce costs, time and resources while increasing the quality of code produced and decrease the time spent on maintenance of systems and implantation of new features.

Keywords: AOP; Aspect-Oriented Programming; Concerns Separations

LISTA DE ILUSTRAÇÕES

Figura 1 – Processo de Combinação 16
Figura 2 – Código espalhado por diversas Classes 23
Figura 3 – Código com interesses emaranhados no interesse funcional........ 23
Figura 4 – Representação do processo de combinação com uma Rede de Petri 34
Figura 5 – Autômato da ordem de execução de um adendo 45
Figura 6 – Representação da execução dos adendos a partir dos designadores 52
Figura 7 – Diagrama Entidade-Relacionamento 55
Figura 8 – Estrutura Hierárquica do Sistema 56
Figura 9 – Arquitetura do Sistema Bibliotecário 56
Figura 10 – Diagrama de Classes 58
Figura 11 – Diagrama de Casos de Uso 59
Figura 12 – Agrupamento dos Interesses Sistêmicos em Aspectos 60
Figura 13 – Diagrama de Sequência: AspectoAutentificacao 63
Figura 14 – Diagrama de Sequência: AspectoControleAcesso 64
Figura 15 – Diagrama de Sequência: AspectoControleExcecao 67
Figura 16 – Diagrama de Sequência: AspectoTransactionHibernate finalizado com sucesso 69
Figura 17 – Diagrama de Sequência: AspectoTransactionHibernate finalizado com falha 69
Figura 18 – Diagrama de Sequência: AspectoTransactionHibernate inicialização do Hibernate 70
Figura 19 – Diagrama de Sequência: AspectoLog 72
Figura 20 – Diagrama de Sequência: AspectoDevolucao 73
Figura 21 – Diagrama de Sequência: AspectoEmprestimo 74
Figura 22 – Diagrama de Sequência: AspectoStatusAluno 75

LISTA DE TABELAS

Tabela 1 - Lista de designadores de um ponto de atuação 36
Tabela 2 – Listas de assinaturas por designador 37
Tabela 3 - Comparação entre adendos e métodos 42
Tabela 4 – Separação de Interesses do Sistema Bibliotecário................................. 57
Tabela 5 – Pontos de Junção por aspectos do sistema bibliotecário 61

LISTA DE CÓDIGOS

Código 1 - Classe implementando um interesse funcional 24
Código 2 - Interesse Sistêmico implementado junto ao Interesse Funcional............ 25
Código 3 – Implementação de um aspecto para separação de interesses 26
Código 4 - Implementação de registro de identificação de chamada de método sem separação de interesses.. 27
Código 5 – Implementação do aspecto para identificar chamadas de métodos....... 28
Código 6 – Exemplo e estrutura de uma assinatura de ponto de junção.................. 38
Código 7 - Exemplo de utilização do operador asterisco... 39
Código 8 - Exemplo da utilização do operador ponto ponto 39
Código 9 - Utilização dos curingas para reduzir o Código 5 40
Código 10 – Exemplo da utilização do Adendo before()... 43
Código 11 – Exemplo da utilização do Adendo after 43
Código 12 – Exemplo da utilização do Adendo around()... 44
Código 13 – Exemplo da utilização do objeto especial thisEnclosingJoinPointStaticPart 48
Código 14 – Exemplo de utilização do objeto especial thisJoinPointStaticPart 49
Código 15 – Aspecto de autenticação do usuário 63
Código 16 - Código Fonte do aspecto AspectoControleAcesso 65
Código 17 - Código Fonte do método checarPermissao 66
Código 18 - Bloco de código do Botão Salvar da classe CadastrarAluno 66
Código 19 – Aspecto de Controle de Exceções 67
Código 20 – Aspecto de Controle para Manipulação de Transação do Hibernate ... 70
Código 21 – Código Fonte do botão Salvar após aplicação do conceito de aspecto 71
Código 22 – Adendo *atfer*() *returning*() do Aspecto de Registro de Ações............... 72

SUMÁRIO

1 INTRODUÇÃO 15
1.1 OBJETIVO 16
1.2 JUSTIFICATIVAS 17
1.3 MOTIVAÇÕES 17
1.4 PERSPECTIVAS DE CONTRIBUIÇÃO 17
1.5 METODOLOGIA DE PESQUISA 18
1.6 ESTRUTURA DO TRABALHO 18
2 PROGRAMAÇÃO ORIENTADA A OBJETOS 19
2.1 ELEMENTOS DA PROGRAMAÇÃO ORIENTADA A OBJETOS 19
2.2 LINGUAGEM DE PROGRAMAÇÃO JAVA 20
3 SEPARAÇÃO DE INTERESSES 22
3.1 INTERESSES FUNCIONAIS 24
3.2 INTERESSES SISTÊMICOS 25
4 PROGRAMAÇÃO ORIENTADA A ASPECTO 31
4.1 ASPECTJ E O PROCESSO DE COMBINAÇÃO 33
4.2 PONTOS DE JUNÇÃO (JOIN POINTS) 34
4.3 PONTOS DE ATUAÇÃO (POINTCUTS) 35
4.3.1 Assinatura de um Ponto de Junção 37
4.4 ADENDOS (ADVICE) 42
4.5 ASPECTOS 45
4.6 REFLEXÃO 46
4.6.1 thisJoinPoint 46
4.6.2 thisJoinPointStaticPart 47
4.6.3 thisEnclosingJoinPointStaticPart 48
4.7 INTERTIPOS 49
4.7.1 Precedências 51
5 ESTUDO DE CASO 53
5.1 O SISTEMA BIBLIOTECÁRIO 53
5.2 MÉTODOS DE DESENVOLVIMENTO 54

5.3 IDENTIFICAÇÃO E SEPARAÇÃO DOS INTERESSES 57
5.4 IMPLEMENTAÇÃO DOS ASPECTOS 60
6 CONCLUSÃO 76
6.1 TRABALHOS FUTUROS.. 77
6.2 RESULTADOS 78
REFERÊNCIAS... 79

1 INTRODUÇÃO

Os desenvolvedores de softwares sempre buscaram uma forma de unificar os códigos fontes a fim de facilitar a programação, deixar o programa mais claro e reutilizar os códigos.

Com o surgimento do paradigma de Programação Orientada a Objetos (POO), tornou-se possível separar em classes todos os atributos e métodos para cumprir os requisitos da aplicação. O paradigma POO é atualmente o método mais utilizado no desenvolvimento de aplicativos (GOETTEN; WINCK, 2006), porém a programação orientada a objetos não possui suporte para a implantação de códigos que não fazem parte da função principal do programa.

Com este foco, Gregor Kiczales e um grupo de cientistas da *Xerox Palo Alto Research Center* (Parc) criaram em 1997 a teoria de um novo paradigma de programação chamado de *Aspect-Oriented Programming* (Programação Orientada a Aspecto ou POA).

O novo paradigma tinha como objetivo solucionar a falta de suporte do paradigma Orientado a Objeto (OO) com relação à unificação dos códigos secundários, chamados de *crosscutting concerns* (interesses transversais). Na POA, os programas são desenvolvidos de uma forma mais simples. Conforme descrito no artigo sobre POA da FEUP, a engenharia consiste em separar os interesses transversais das classes em módulos bem definidos e centralizados, assim o analista pode focar cada interesse de forma independente do resto da aplicação.

A Programação Orientada a Aspecto estende a Programação Orientada a Objeto, pois a forma mais comum de programar os componentes (ou os códigos funcionais) é utilizando POO. Segundo Krupa (2010), a implementação dos componentes não está limitada a linguagens orientadas a objetos. Porém a POO, possui um nível de abstração maior que as linguagens procedurais e consegue trabalhar paralelamente com a POA reaproveitando o código e aplicando os conceitos essenciais para o sucesso de uma aplicação, como a modularização e o encapsulamento.

O processo de implementação é simples, é programado de acordo com a separação de interesses (*concerns separation*), os aspectos são codificados em módulos na linguagem suportada pela POA e os componentes são criados em uma linguagem orientada a objetos, como por exemplo, Java. Após esta etapa, existe uma combinação entre componentes e aspectos por meio de um combinador, como mostrado na Figura 1 oriunda dos conceitos apresentados no livro *AspectJ – Programação Orientada a Aspecto com Java* (GOETTEN; WINCK, 2006).

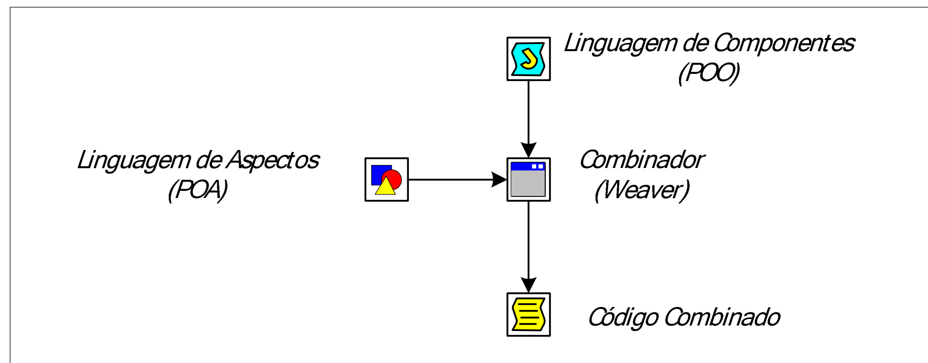

Figura 1 – Processo de Combinação

Como a orientação a aspecto possui a separação de interesses como principal recurso, existe uma facilidade muito grande na implantação de novos aspectos sem alterar o código existente, além de prover uma manutenção mais eficaz, já que torna o código muito mais legível.

1.1 OBJETIVO

Os objetivos deste trabalho é apresentar os conceitos do paradigma de programação a aspectos, desenvolver um estudo de caso e mostrar como é realizada a identificação de interesses sistêmicos em sistemas orientados a objetos. O estudo de caso também tem por objetivo demonstrar a construção de aspectos conforme a necessidade e ordem de execução do código transversal ao ser combinado com o código principal através dos pontos de junção, a fim de auxiliar novas pesquisas e incentivar a utilização corporativa deste novo paradigma.

1.2 JUSTIFICATIVAS

A tecnologia evolui e surgem novas formas de resolver velhos problemas. O paradigma POA possui técnicas ágeis e eficientes para estruturar o código fonte de forma a permitir que as unidades de códigos secundários (relativos ao sistema) se comuniquem com o código principal. Assim, a programação orientada a aspecto possui um nível de reutilização de código muito alta conseguindo captar comportamentos idênticos, de diferentes classes e pacotes, o que facilita a implementação, a análise e a manutenção. Deste modo, é um paradigma completo que aumenta a produtividade desde o inicio do processo de desenvolvimento de software até o complexo processo de manutenção.

1.3 MOTIVAÇÕES

O impacto da POA no futuro do desenvolvimento de softwares é inevitável tendo em vista que é um paradigma confiável e intuitivo que utiliza de recursos do paradigma de programação orientada a objetos para garantir o correto funcionamento do sistema. O estudo de tais assuntos mais detalhadamente, ajuda a fortificação do paradigma e a sua aceitação pelas empresas de desenvolvimento, porque "*um risco considerável em adotar qualquer nova tecnologia é ir muito longe, muito rápido*" (*AspectJ Programming Guide*, 2011).

1.4 PERSPECTIVAS DE CONTRIBUIÇÃO

A perspectiva é de que este trabalho divulgue as vantagens do paradigma de programação orientado a aspecto e estimule novas pesquisas e a implantação de novas tecnologias. Espera-se também que este trabalho contribua para a disseminação, o desenvolvimento e a criação de métodos mais produtivos e interativos de programação a fim de melhorar o ciclo de elaboração de software.

1.5 METODOLOGIA DE PESQUISA

Foi realizada uma pesquisa sobre o assunto e o desenvolvido um estudo de caso dos conceitos abordados a fim de demonstrar a separação de interesses sistêmicos. No estudo de caso, foi implementado um software utilizando como base a linguagem de programação JAVA e os conceitos de Orientação a Aspectos. O combinador *AspectJ,* que desenvolvido para ser acoplado a ferramenta *Eclipse*, será utilizado para a realização do estudo de caso.

1.6 ESTRUTURA DO TRABALHO

Este trabalho é constituído pelo capítulo de Introdução, seguido pelo capítulo Programação Orientado a Objetos, que mostra os conceitos básicos deste estilo de programação; o capítulo seguinte Separação de Interesses, explana os motivos da aplicação deste método; o capítulo Programação Orientada a Aspecto descreve a história e os conceitos da programação orientada a aspecto; o Estudo de Caso demonstra a aplicação do paradigma POA e da separação de interesses sistêmicos; e por fim, o capítulo Conclusão, que tece observações sobre os estudos realizados e elenca alguns trabalhos futuros.

2 PROGRAMAÇÃO ORIENTADA A OBJETOS

Até o inicio dos anos 60, a programação estruturada era a forma mais comum de desenvolvimento de sistemas, porém não era eficiente. "Essa abordagem apresenta dificuldades para manter o controle da qualidade do software, a legibilidade do código gerado e o planejamento da solução de um problema" (GOETTEN; WINCK, 2006). Existe uma limitação no reuso de software o que torna o código extenso e difícil de ser implementado.

A necessidade de um paradigma mais confiável foi incentivando os desenvolvedores a criar soluções para a programação estruturada. Em 1966, surgiu a linguagem de programação *Simula* que era a extensão do *Argol 60*. *Simula* possuía os conceitos fundamentais da Orientação a Objetos (OO) (GOETTEN; WINCK, 2006), (HEILEMAN, 1996).

Pouco tempo depois do surgimento do *Simula*, a *Xerox* desenvolveu a linguagem *Smalltalk* que possuía algumas idéias do *Simula*, porém com novas técnicas que foram aperfeiçoadas na década de 80 com o desenvolvimento do *C++,* derivada do *C* (HORSTMANN, 2003), (HEILEMAN, 1996).

Diversas linguagens foram criadas desenvolvendo os conceitos da orientação até o desenvolvimento da linguagem de programação Java, em 1995 pela *Sun Microsystems*. A linguagem Java disseminou o desenvolvimento de software orientado a objetos e enriqueceu os conceitos do paradigma, construindo uma estrutura funcional e simplificada (HEILEMAN, 1996), (GOETTEN; WINCK, 2006).

2.1 ELEMENTOS DA PROGRAMAÇÃO ORIENTADA A OBJETOS

"A OO surgiu da necessidade de simular a realidade" (GOETTEN; WINCK, 2006) e para tanto utiliza de métricas como o objeto e a classe.

Um objeto é um elemento da orientação a objetos que representa elementos do mundo real ou entidades abstratas. Os objetos possuem atributos (dados) e

métodos (serviços) na mesma estrutura, que por si é instanciada de uma classe (ORACLE, 2010), (HORSTMANN, 2003).

A classe é um projeto do objeto. Conforme Goetten; Winck (2006), "as classes possuem todas as informações necessárias para construir instâncias separadas dela mesma", ou seja, a classe agrupa as características de uma entidade e se mantém disponível para criar quantos objetos forem necessários a partir do seu modelo.

A classe e o objeto possuem artifícios para trabalharem com flexibilidade dentro de um sistema orientado a objetos. Podemos desmembrar estes artifícios em três mecanismos: Herança, Polimorfismo e Encapsulamento.

A herança é um mecanismo que fornece a possibilidade de uma classe ser criada com base em outra classe representando a relação "é um", ou seja, "a herança é a forma que os objetos de uma classe podem utilizar para acessar os atributos e métodos contidos em outra classe previamente definida" (HEILEMAN, 2003) considerando que a primeira classe deve possuir características similares a segunda classe.

O Polimorfismo é a capacidade de um método de uma mesma classe possuir o mesmo nome ou referência. Assim, um método pode possuir diferentes implementações, porém é acessado conforme os parâmetros que recebe (HEILEMAN, 2003), (GOETTEN; WINCK, 2006).

A forma de proteger os dados contidos nos objetos é o encapsulamento. Este mecanismo possui a capacidade de restringir o acesso aos atributos e métodos de um objeto por outro objeto de outra classe ou acessos diretos do corpo do código fonte. Assim, para acessar os dados de um objeto encapsulado, a classe deste objeto deve programar os métodos *getters* (leitura) e *setters* (escrita).

2.2 LINGUAGEM DE PROGRAMAÇÃO JAVA

Como dito anteriormente, o paradigma de programação orientado a objetos tornou-se popular com a criação da Linguagem de Programação Java em 1995 por um grupo de desenvolvimento da *Sun Microsystems* criado em 1991, denominados como *The Green Team*, liderados pelo engenheiro James Gosling (ORACLE, 2010).

O Java foi desenvolvido com o intuito de unir aparelhos digitais aos computadores, porém apenas no ano de 2000, a linguagem começou a ter aceitação comercial e tornou-se símbolo das aplicações móveis. (ORACLE, 2010).

No ano de 2010, a *Oracle* adquire a *Sun Microsystems* assinando a clausula de que Java continuaria sendo uma linguagem aberta e grátis aos desenvolvedores (ORACLE, 2010).

Assim, a linguagem de programação Java expandiu-se e tornou-se uma plataforma eficiente de programação. Porém, o paradigma OO não trata algumas características dos sistemas de forma correta, fazendo com que o paradigma não aja conforme seus conceitos.

Com esta deficiência, a análise e o desenvolvimento de um software que possui requisitos sistêmicos, como por exemplo, o controle de acesso - *logging*, acaba tornando-se difícil de ser interpretada e implementada. Os objetos manipulam informações que não fazem parte de seu escopo, as classes possuem informações entrelaçadas e não possuem nenhuma unidade centralizada dos códigos sistêmicos para auxiliar as alterações. Para evitar estas incoerências conceituais, uma técnica que pode ser utilizada é a separação de interesses, que será estudada no próximo capítulo.

3 SEPARAÇÃO DE INTERESSES

Durante a programação ou análise de um sistema podemos perceber similaridades entre os requisitos desta aplicação, a estas similaridades damos o nome de interesses. Em um sistema são possíveis diversos tipos de interesses tais como idéias, princípios, restrições, algoritmos, entre outros. Portanto, toda implementação em um sistema possui características ou objetivos similares que a define como um interesse (HEILEMAN, 2003), (GOETTEN; WINCK, 2006) e (SAFONOV, 2008).

"A estratégia de dividir para conquistar é bastante comum para a solução de complexos de qualquer natureza" (GOETTEN; WINCK, 2006). 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 (HEILEMAN, 2003).

A separação de interesses é uma forma de dividir os requisitos de um sistema e agrupá-los de forma ordenada de acordo com suas características e funcionalidades. Portanto, a utilização desta divisão pode evitar à ocorrência de código espalhado (*scattering code*) e de código emaranhado (*tangled code*).

O código espalhado é um termo que denomina que um interesse foi implementado em várias classes. Conforme Figura 2 abaixo, na orientação a objetos um interesse sistêmico, como o *logging*, fica fragmentado em diversas classes.

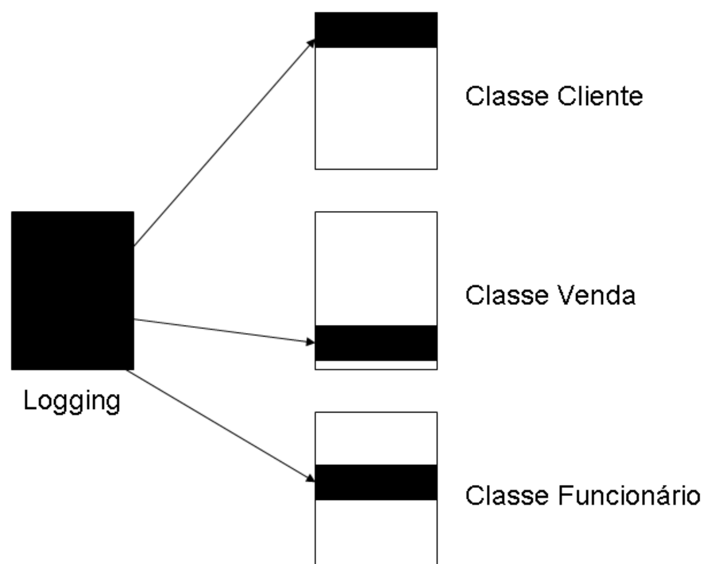

Figura 2 – Código espalhado por diversas Classes

Há a ocorrência de código emaranhado, quando possuímos uma classe ou método que possuem vários interesses programados, ou seja, quando você espalha os códigos ortogonais pelas classes funcionais o código de uma classe fica emaranhado com diversos interesses diferentes, conforme exemplificado na Figura 3.

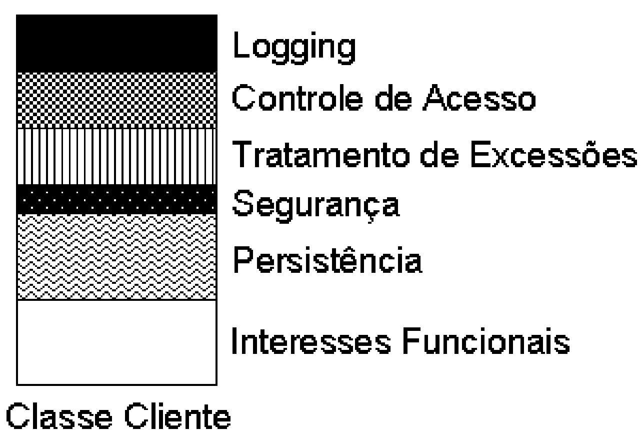

Figura 3 – Código com interesses emaranhados no interesse funcional

O resultado do código emaranhado e espalhado é uma redução significativa na produtividade, dificuldade na manutenção e um código de difícil entendimento e de

baixa qualidade, pois possui ambigüidades e inconsistências no código (BODKIN; LADDAD, 2004).

Podemos agrupar os interesses em dois tipos: Interesses Funcionais e Interesses Sistêmicos que serão explorados nas próximas sessões.

3.1 INTERESSES FUNCIONAIS

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

Quando um interesse funcional encontra-se desorganizado, a aplicação pode ser comprometida. A probabilidade de erro é muito grande, pois o código está difícil de ser compreendido e ele pode ser replicado durante a implementação.

Além dos problemas relacionados à programação, se o esqueleto da aplicação estiver ilegível, todo o desenvolvimento do projeto pode ser impactado, incluindo o não cumprimento de prazos, custos e recursos, além de dificultar as futuras implantações e alterações.

Um exemplo de uma classe implementada corretamente é a classe Cliente implementada em Java ilustrada no Código 1. Esta classe possui um atributo nome e seu respectivo método *get* e *set*, ambos os interesses funcionais.

Código 1 - Classe implementando um interesse funcional

```
 1: // Classe Cliente em Java - Interesse Funcional
 2:
 3: package com.pacote;
 4:
 5: public class Cliente{
 6:   private String nome;
 7:   public String getNome(){
 8:         return nome;
 9:   }
10:   public void setNome (String vNome){
11:         nome = vNome;
12:   }
13:
14:   public static void main (String args[]){
15:         Cliente cliente1 = new Cliente();
```

```
16:          System.out.println(cliente1.getNome());
17:    }
18: }
```

3.2 INTERESSES SISTÊMICOS

Os interesses sistêmicos são normalmente chamados de interesses transversais, ortogonais ou *crosscutting concerns* e estão relacionados com as características de suporte dos interesses funcionais (GOETTEN; WINCK, 2006).

Os interesses sistêmicos são à base do paradigma de programação orientada a aspecto (POA). Eles não podem ser implementados espalhados nos códigos de outros interesses, pois necessitam de organização e de modularização para que possam ser mais bem controlados (SAFONOV, 2008).

Para identificar os interesses sistêmicos é necessário analisar o esqueleto do programa (em um sistema que não foi implementado em POA) ou localizar requisitos que serão suportes às características funcionais do software. Qualquer tipo de código que interage com a aplicação, porém não faz parte do escopo da funcionalidade que está sendo desenvolvida é um interesse sistêmico. Outra forma de identificar um interesse sistêmico é encontrar possíveis variações e replicações de código na aplicação.

Na classe Cliente do Código 1, possuímos um interesse funcional corretamente implementado. Porém, se desejar adicionar um registro para imprimir todas as vezes que o método *getNome* for usado, o código será como no Código 2:

Código 2 - Interesse Sistêmico implementado junto ao Interesse Funcional

```
 1: // Implementação de Interesse Sistêmico nos Interesses Funcionais
 2:
 3: package com.pacote;
 4:
 5: public class Cliente{
 6:   private String nome;
 7:   public String getNome(){
 8:          System.out.println("Chamado um método get");
 9:          return nome;
10:   }
11:   public void setNome (String vNome){
12:          nome = vNome;
13:   }
```

```
14:
15:   public static void main (String args[]){
16:          Cliente cliente1 = new Cliente();
17:          System.out.println(cliente1.getNome());
18:   }
19: }
```

Na linha 8 é exemplificado o uso de um interesse transversal implementado dentro dos interesses funcionais afim de imprimir um aviso toda vez que for chamado o método *getNome*. No Código 3, é utilizado o conceito de aspectos (serão apresentados nos próximos capítulos) para separar os interesses:

Código 3 – Implementação de um aspecto para separação de interesses

```
 1: // Aplicação de Aspecto para separação de interesses
 2:
 3: package com.pacote;
 4:
 5: public class Cliente{
 6:   private String nome;
 7:   public String getNome(){
 8:          return nome;
 9:   }
10:   public void setNome (String vNome){
11:          nome = vNome;
12:   }
13:
14:   public static void main (String args[]){
15:          Cliente cliente1 = new Cliente();
16:          System.out.println(cliente1.getNome());
17:   }
18: }
19:
20: package com.pacote;
21: public aspect AspectoCliente{
22:   pointcut registra() : call(public String Cliente.getNome());
23:   before() : registra(){
24:          System.out.println("Chamado um método get");
25:   }
26: }
```

O Código 3 faz a separação de interesses, porém não parece muito útil substituir uma única linha de código pelas linhas 20 a 26. Aparenta que o código está maior e acaba aumentando a mão de obra do programador. Porém, supondo que haja a necessidade de que esta impressão de chamada do método *getNome* deva ser

utilizada em todos os métodos que serão implementados na classe Clientes, conforme mostrado no código 4 e 5:

Código 4 - Implementação de registro de identificação de chamada de método sem separação de interesses

```
 1: package com.pacote;
 2:
 3: public class ExibeNome{
 4:   private String nome;
 5:   private String endereco;
 6:   private String cidade;
 7:   private String estado;
 8:   private String pais;
 9:
10:   public String getNome(){
11:         System.out.println("Chamado um método");
12:         return nome;
13:   }
14:
15:   public String getEndereco(){
16:         System.out.println("Chamado um método");
17:         return endereco;
18:   }
19:
20:   public String getCidade(){
21:         System.out.println("Chamado um método");
22:         return cidade;
23:   }
24:
25:   public String getEstado(){
26:         System.out.println("Chamado um método");
27:         return estado;
28:   }
29:
30:   public String getPais(){
31:         System.out.println("Chamado um método");
32:         return pais;
33:   }
34:
35:   public void setNome(vNome){
36:         System.out.println("Chamado um método");
37:         nome = vNome;
38:   }
39:
40:   public void setEndereco(vEndereco){
41:         System.out.println("Chamado um método");
42:         endereco = vEndereco;
43:   }
44:
45:   public void setCidade(vCidade){
46:         System.out.println("Chamado um método");
47:         cidade = vCidade;
48:   }
```

```
49:
50:   public void setEstado(vEstado){
51:         System.out.println("Chamado um método");
52:         estado = vEstado;
53:   }
54:
55:   public void setPais(vPais){
56:         System.out.println("Chamado um método");
57:         pais = vPais;
58:   }
59:
60:
61:   public static void main (String args[]){
62:         Cliente cliente1 = new Cliente();
63:         setNome("Rafael");
64:         setEndereco("Rua xyz");
65:         setCidade("XXX");
66:         setEstado("SP");
67:         setPais("Brasil");
68:         System.out.println(cliente1.getNome());
69:         System.out.println(cliente1.getEndereco());
70:         System.out.println(cliente1.getCidade());
71:         System.out.println(cliente1.getEstado());
72:         System.out.println(cliente1.getPais());
73:   }
74: }
```

Código 5 – Implementação do aspecto para identificar chamadas de métodos

```
 1:package com.pacote;
 2:
 3: public class ExibeNome{
 4:   private String nome;
 5:   private String endereco;
 6:   private String cidade;
 7:   private String estado;
 8:   private String pais;
 9:
10:   public String getNome(){
11:         return nome;
12:   }
13:
14:   public String getEndereco(){
15:         return endereco;
16:   }
17:
18:   public String getCidade(){
19:         return cidade;
20:   }
21:
22:   public String getEstado(){
23:         return estado;
24:   }
25:
26:   public String getPais(){
27:         return pais;
```

```
28:   }
29:
30:   public void setNome(vNome){
31:         nome = vNome;
32:   }
33:
34:   public void setEndereco(vEndereco){
35:         endereco = vEndereco;
36:   }
37:
38:   public void setCidade(vCidade){
39:         cidade = vCidade;
40:   }
41:
42:   public void setEstado(vEstado){
43:         estado = vEstado;
44:   }
45:
46:   public void setPais(vPais){
47:         pais = vPais;
48:   }
49:
50:
51:   public static void main (String args[]){
52:         ExibeNome exibeNome = new ExibeNome();
53:
54:         setNome("Rafael");
55:         setEndereco("Rua xyz");
56:         setCidade("XXX");
57:         setEstado("SP");
58:         setPais("Brasil");
59:         System.out.println(cliente1.getNome());
60:         System.out.println(cliente1.getEndereco());
61:         System.out.println(cliente1.getCidade());
62:         System.out.println(cliente1.getEstado());
63:         System.out.println(cliente1.getPais());
64:   }
65: }
66:
67: package com.pacote;
68:
69: public aspect AspectoCliente{
70:   pointcut registra() :
71:               call(public String cliente1.getNome())||
72:               call(public String cliente1.getEndereco())||
73:               call(public String cliente1.getCidade())||
74:               call(public String cliente1.getEstado())||
75:               call(public String cliente1.getPais())||
76:               call(public void cliente1.setNome(String))||
77:               call(public void cliente1.setEndereco(String))||
78:               call(public void cliente1.setCidade(String))||
79:               call(public void cliente1.setEstado(String))||
80:               call(public void cliente1.setPais(String));
81:
82:   before() : registra(){
83:         System.out.println("Chamado um método");
```

```
84:   }
85: }
```

Conforme descrito no Código 4, os interesses ficam dispersos em todo o código dificultando uma futura alteração, enquanto que o Código 5 possui um único método de impressão que age em todo o código funcional. Se necessitar alterar o texto do comando de impressão de "*Chamado um Método*" para "*Um método da classe cliente foi chamado*", o Código 5 seria mais prático para efetuar a alteração do que no Código 4. Afinal, o Código 5 possui apenas um método para ser modificado, enquanto que o Código 4 possui diversos métodos espalhados pelo corpo da aplicação.

Outros exemplos onde pode ser aplicada a separação de interesses é na programação de tratamento de exceções, segurança e controle de usuários e acessos, sincronização de objetos concorrentes, distribuição, coordenação de múltiplos objetos, persistência, entre outros.

4 PROGRAMAÇÃO ORIENTADA A ASPECTO

Conforme visto anteriormente, o controle de interesses sistêmicos é vital para o desempenho da produção de um software. Mas como separar estes interesses de modo satisfatório e que o mesmo possa interagir com os interesses funcionais? Este questionamento não é um problema atual, em 1970, alguns cientistas reuniram-se na tentativa de solucionar os complicados problemas relacionados ao reuso e manutenção de software. Estes cientistas foram motivados pelo conceito criado por Dijkstra em 1968. Dijkstra tentava criar métricas para desenvolver um sistema operacional mais simples, quando desenvolveu o conceito de nível de abstração (ou *Abstration Level*, em inglês) que consiste na visualização de um problema de modo geral abstraindo os detalhes (SAFONOV, 2008).

Em 1979, Adolph Fouxman, professor da Universidade de Rostov, publicou uma monografia apresentando o conceito *vertical cut* (corte vertical) que foi o predecessor do paradigma orientado a aspecto. Ele sugeriu implementar uma ferramenta que localizasse e visualizasse todos os interesses que não fossem funcionais, porém não foi dada continuidade a idéia (SAFONOV, 2008).

O Paradigma de Programação Orientado a Aspecto – POA (*Aspect-Oriented Programming – AOP*, em inglês) foi criado em 1997, por um grupo de pesquisadores do PARC (*Palo Alto Research Center*) da Xerox e da Universidade de Columbia, liderados por Gregor Kiczales. O objetivo principal era construir um paradigma que consistisse na utilização de dois conceitos: Linguagem Central (linguagem para programar os interesses funcionais) e Linguagem de Aspectos (linguagem para programar os interesses ortogonais), assim a orientação a aspecto trabalha em paralelo com outro paradigma, sendo o mais utilizado a orientação a objetos (SAFONOV, 2008), (GOETTEN; WINCK, 2004).

A POA consegue separar os interesses de forma organizada e definida desde o nível de análise e modelagem até o nível de implementação, incluindo a manutenção (GOETTEN; WINCK, 2004).

A programação orientada a aspecto auxilia a fase de modelagem, pois organiza e separa o trabalho do analista de software. Quando o analista estiver modelando o sistema, ele poderá ser mais preciso nos dados funcionais porque qualquer outro interesse que o usuário necessitar futuramente pode ser facilmente implementado. Por exemplo, se o usuário necessitar um controle de acesso dos usuários e um registro das ações dos usuários no sistema, o programador apenas irá implementar um aspecto para controle de acesso e outro para o registro das ações, sem modificar o código funcional. Isto propicia produtos de melhor qualidade, adaptáveis a necessidade do cliente e com prazos e custos menores (BODKIN; LADDAD, 2004).

Porém a grande facilidade da POA é na implementação, pois possui uma legibilidade dos códigos implementados, o que evita os códigos espalhados e emaranhados, além de facilitar a programação sendo fatal aos códigos de difícil compreensão. Após a programação do software qualquer alteração é muito prática, pois os códigos sistêmicos estão centralizados nos aspectos e os códigos funcionais estão claros, assim não são necessárias amplas mudanças para a manutenção ou implantação de novos recursos (BODKIN; LADDAD, 2004).

A POA encontra-se em nível de maturidade alto, possui ferramentas eficientes, como o *AspectJ* e outros compiladores mantidos por empresas importantes na história da computação ou os IDE (*Integrated Development Environment*) como o *Eclipse* que é um dos IDE mais utilizados no mundo, uma comunidade de usuários que auxiliam o desenvolvimento da tecnologia (*www.eclipse.org/aspectJ*), além de inúmeras aplicações iniciando no mercado (GOETTEN; WINCK, 2006). O futuro do paradigma orientado a aspecto é promissor, pois conforme Kiczales (1997), um paradigma confiável deve conseguir ampla aceitação e precisa ser expressivo, eficiente, intuitivo, compatível e possuir uma boa ferramenta que o suporte.

Portanto, além de atender a todos os requisitos de um paradigma confiável, a POA soluciona um antigo problema na programação de sistemas, a separação de interesses, de forma eficaz, resultando em aplicações de fácil entendimento e em manutenções sem alterar o esqueleto da aplicação.

4.1 ASPECTJ E O PROCESSO DE COMBINAÇÃO

A separação de um sistema em requisitos funcionais e não funcionais aumenta a produtividade no desenvolvimento e análise e a facilidade na manutenção, porém como os interesses ortogonais se comunicam com o esqueleto da aplicação?

Como será visto nas próximas sessões, a orientação a aspecto possui técnicas para manter a comunicação entre os interesses. Os mecanismos utilizados são combinados com um compilador da linguagem POA, chamado combinador aspectual (ou *Weaver*, em inglês).

O Combinador Aspectual é um compilador que constrói um código intermediário combinando os códigos POA e os códigos funcionais. O *Weaver* compatível com a linguagem de programação Java é o *AspectJ* (GOETTEN; WINCK, 2006).

O *AspectJ* foi criado em 1997 pela *Xerox Palo Alto Research Center*. *AspectJ* é a primeira linguagem de desenvolvimento POA, tanto quanto o seu compilador. Em 2002, a IBM agregou o projeto *AspectJ* à IDE *Eclipse*, mantendo um maior suporte à nova linguagem e melhorando o combinador aspectual (*AspectJ Programming Guide*, 2011).

Como o *AspectJ* é uma extensão do Java, qualquer programa que é válido em um compilador Java, também é válido à um compilador *AspectJ*. Os programas em POA implementados pelo *AspectJ* podem ser executados em uma JVM (*Java Virtual Machine*), ou seja, são convertidos em *bytecodes*. Assim, o *AspectJ* é completamente intuitivo, qualquer programador da linguagem de programação Java não terá problemas em desenvolver POA (GOETTEN; WINCK, 2006), (*AspectJ Programming Guide*, 2011).

O processo de compilação do paradigma de programação orientada a aspecto consiste na união da linguagem de aspectos (interesses transversais) com a linguagem de componentes (interesses funcionais) através do combinador aspectual. O combinador aspectual mantém os dois interesses intactos e cria um novo arquivo (chamado arquivo intermediário) unindo os aspectos ao corpo da aplicação nos pontos de junções relacionados em cada ponto de atuação definido no aspecto. Após a criação do código intermediário, o arquivo é compilado pelo

compilador Java e transformado em *bytecodes* para ser executado em uma JVM, conforme a Figura 4 mostra abaixo (GOETTEN; WINCK, 2006), (*AspectJ Programming Guide*, 2011).

A Figura 4 constitui-se de uma Rede de Petri que representa o processo de combinação dos interesses. Esta rede possui como lugares iniciais, o Código *AspectJ* e o Código Java que serão combinados através de um combinador, representado pela transição *Weaver*. Após a combinação de ambos os códigos é gerado outro lugar, chamado Código Intermediário. O Código Intermediário gerado será compilado por um compilador Java e possuirá como saída os *bytescodes* prontos para serem executados em uma JVM.

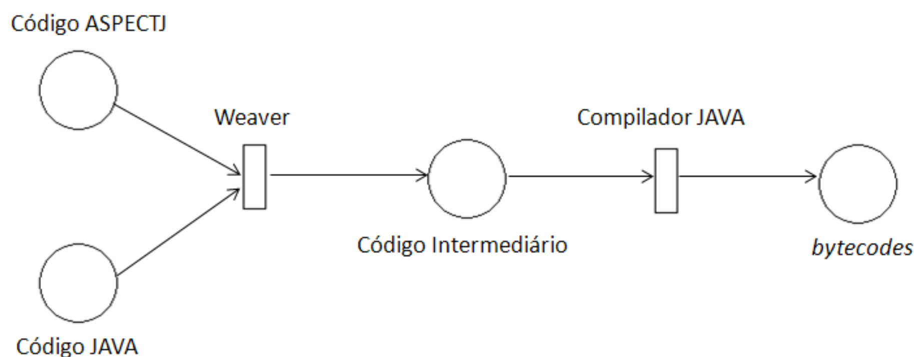

Figura 4 – Representação do processo de combinação com uma Rede de Petri

Portanto, este processo é extremamente eficiente, pois mantém os códigos originais separados por tipo de interesse, facilitando a manutenção e legibilidade do software depois de desenvolvido.

4.2 PONTOS DE JUNÇÃO (*JOIN POINTS*)

De acordo com o capítulo acima, a separação de interesses é utilizado em conjunto com diferentes linguagens de programação para programar cada tipo de interesse. Para unir os interesses necessita-se de um ponto de junção.

"O ponto de junção é um ponto bem definido na execução de um sistema" (SAFONOV, 2008), ou seja, eles são pontos selecionados para controlar o fluxo do programa.

Normalmente um ponto de junção é uma chamada ou execução de um método, a criação de objeto, exceções, métodos *getters* e *setters*, enfim, qualquer ação que possa haver um interesse ortogonal relacionado (BODKIN; LADDA, 2004).

A linha 68 do Código 4 (Sessão 3.2) descreve a impressão dos dados do atributo nome do objeto cliente1 (*System.out.println(cliente1.getNome())*). O método *getNome* é um método que possui um interesse ortogonal vinculado, ou seja, o comando de impressão que identifica a chamada de um método. Assim, este método é um ponto de junção.

Como os pontos de junção são pontos onde o aspecto irá unir (combinar) com o código principal, eles são definidos no programa funcional e unidos por meio de outros elementos que serão apresentados nas próximas sessões.

4.3 PONTOS DE ATUAÇÃO (*POINTCUTS*)

Após separar os interesses e identificar um ponto de junção, é necessário agrupa-los em um ponto de atuação (GOETTEN; WINCK, 2006). Um ponto de atuação é um componente da POA que seleciona os pontos de junção e extrai o contexto baseados em suas características, como nome de classes, métodos, argumentos, exceções, entre outros, ou seja, o ponto de atuação é uma parte do aspecto que define especificamente como um adendo irá ser combinado no código funcional (SAFONOV, 2008), (BODKIN; LADDAD, 2004).

No Código 3 (Sessão 3.2), a linha 22 (*pointcut registra() : call(public String Cliente.getNome());*) representa um *pointcut*. Esta linha indica que o código especificado no adendo do aspecto *AspectoCliente* será combinado em qualquer chamada do método *getNome()* da classe *Cliente*.

A estrutura de um ponto de atuação é bem simples, possibilitando que um ponto de atuação possa agrupar diversos pontos de junção de diferentes partes do sistema,

“sem que seja necessário definir um ponto individualmente (o que tornaria a programação orientada a aspecto quase sem sentido)” (GOETTEN; WINCK, 2006).

```
pointcut <tipo_acesso> <nome_de_referencia>({<lista_de_parametros>}) :
        {<designador> (<assinatura_ponto_de_atuacao>) [&& | ||]};
```

Para criar um ponto de atuação é necessária a palavra-chave “*pointcut*” seguido do tipo de acesso que o ponto de atuação irá possuir (público ou privado), o nome do ponto de atuação com seus possíveis parâmetros e a lista de pontos de junção capturados. Existe uma peculiaridade na forma de descriminar um ponto de junção em um ponto de atuação, que é através de um designador (*designator*, em inglês). Um designador é um mecanismo que identifica o tipo de comportamento de um ponto de junção que o ponto de atuação deverá capturar. Na Tabela 1 existe uma listagem de todos os designadores de um ponto de atuação.

Tabela 1 - Lista de designadores de um ponto de atuação (In: GOETTEN; WINCK, 2006, p87)

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 exceções
Get	Corresponde à referência para um atributo de uma classe
Set	Corresponde à definição de um atributo de uma 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/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 ao adendo (*advice*) do ponto de junção
Preinitialization	Corresponde à pré-inicialização de um ponto de junção

O ponto de atuação exige uma assinatura para identificar o ponto de junção. A função da assinatura é descrever as informações dos interesses funcionais que o designador necessita para o ponto de atuação interferir e executar o bloco de código especificado no adendo (Sessão 4.4). Esta assinatura pode variar conforme o tipo de designador utilizado.

4.3.1 Assinatura de um Ponto de Junção

Conforme visto, a assinatura é um elemento essencial para que um ponto de atuação saiba o ponto de junção que ele deve atuar. A assinatura pode possuir diversas formas de acordo com o designador. Abaixo a Tabela 2 com a lista das assinaturas por designador.

Tabela 2 – Listas de 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>)`
Get	`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>)

Para demonstrar a utilização de um designador, vamos considerar o designador *call* que é utilizado para capturar chamada de métodos, possui uma estrutura bem detalhada conforme descrito no exemplo abaixo (Código 6):

Código 6 – Exemplo e estrutura de uma assinatura de ponto de junção

```
call(<tipo_acesso> <valor_retorno> <classe>.<nome_metodo>({<lista_parametro}>))

call(public void Cliente.setNome(String));
```

No exemplo acima, o ponto de atuação irá capturar toda chamada do método *setNome* da classe *Cliente* que seja público (*public*), não retorne valor (*void*) e possua um parâmetro do tipo *String*. Deste modo, o paradigma de programação orientado a aspecto consegue ser específico em alguns casos e possuir métodos de filtragem para outros casos conforme apresentado abaixo.

Não existiria muita função ao paradigma POA no caso de necessitar escrever um ponto de atuação para cada ponto de junção, pois não haveria reaproveitamento de código. Por este motivo, existem dois operadores chamados curingas na linguagem POA: ***** (asterisco) e **..** (ponto ponto). Tais operadores permitem criar uma regra de padrão para a identificação de pontos de junção (GOETTEN; WINCK, 2006).

A utilização do asterisco é similar ao seu uso em expressões regulares (ER), possui a função de substituir um parâmetro (curinga), elemento ou complemento de uma referência, conforme demonstrado no aspecto abaixo (Código 7).

Código 7 - Exemplo de utilização do operador asterisco

```
 1: public aspect AspectoCliente{
 2:   pointcut registra() : call(public * Cliente.set*(*));
 3:   before() : registra(){
 4:         System.out.println("Chamado um método set");
 5:   }
 6: }
```

A assinatura do ponto de atuação da linha 2 do aspecto do Código 7, representa três exemplos de utilização do curinga asterisco. O primeiro asterisco substitui o elemento "*Valor de Retorno*" da assinatura; o segundo substitui o complemento do nome de referência do método; e o terceiro, substitui um ou nenhum parâmetro. Neste caso, existe um ponto de atuação chamado *registra* que será combinado ao esqueleto principal do sistema na chamada de um método público da classe *Cliente* quando o seu nome de referência iniciar com "*set*". Tal método pode possuir qualquer tipo de retorno e poderá ter ou não um parâmetro de qualquer tipo de dados.

Desta forma, é possível montar um padrão para agrupar mais facilmente os pontos de junção em um ponto de atuação. Porém, o operador asterisco não consegue atender quando se necessita agrupar dois métodos que possuem diferentes quantidades de parâmetros. Um exemplo deste efeito é o aspecto acima que faz referência à classe *Cliente*. Se tal classe possuir dois métodos, um chamado *setNome* e o outro *setEndereco*; o método *setEndereco* receberá dois parâmetros, um do tipo *String*, representando o endereço e outro do tipo *int*, representando o *número da casa*, conforme mostrado no Código 8, ilustrado abaixo.

Código 8 - Exemplo da utilização do operador *ponto ponto*

```
 1: package com.pacote;
 2:
 3: public class Cliente{
 4:   private String nome;
 5:   private String endereco;
 6:   private int numeroCasa;
 7:
 8:   public void setNome(String vNome){
 9:         nome = vNome
10:   }
11:   public void setEndereco(String vEndereco, int vNumero){
12:         endereco = vEndereco;
```

```
13:         numeroCasa = vNumero;
14:   }
15:
16:   public static void main (String args[]){
17:         Cliente cliente1 = new Cliente();
18:         cliente1.setNome("Rafael");
19:         cliente1.setEndereco("Rua xxx", 320);
20:   }
21: }
22:
23: package com.pacote;
24:
25: public aspect AspectoCliente{
26:   pointcut registra() : call(public void Cliente.set*(..));
27:   before() : registra(){
28:         System.out.println("Chamado um método set");
29:   }
30: }
```

No aspecto apresentado acima, foi utilizado o curinga *ponto ponto* (..) para poder agrupar os dois métodos (pontos de junção) no mesmo ponto de atuação sem a utilização de nenhum operador lógico. Assim, o curinga *ponto ponto* aceita nenhum, um ou mais parâmetros em uma única assinatura. Portanto, respeita o conceito de polimorfismo da orientação a objetos e extingue a possibilidade de criar adaptações no código de componentes para a utilização dos aspectos.

Se aplicar a teoria dos operadores curinga no Código 5 da Sessão 3.2, pode-se reduzir significativamente o tamanho e a abrangência do aspecto, conforme descrito no Código 9.

Código 9 - Utilização dos curingas para reduzir o Código 5

```
 1:package com.pacote;
 2:
 3: public class ExibeNome{
 4:   private String nome;
 5:   private String endereco;
 6:   private String cidade;
 7:   private String estado;
 8:   private String pais;
 9:
10:   public String getNome(){
11:         return nome;
12:   }
13:
14:   public String getEndereco(){
15:         return endereco;
16:   }
```

```
17:
18:   public String getCidade(){
19:         return cidade;
20:   }
21:
22:   public String getEstado(){
23:         return estado;
24:   }
25:
26:   public String getPais(){
27:         return pais;
28:   }
29:
30:   public void setNome(vNome){
31:         nome = vNome;
32:   }
33:
34:   public void setEndereco(vEndereco){
35:         endereco = vEndereco;
36:   }
37:
38:   public void setCidade(vCidade){
39:         cidade = vCidade;
40:   }
41:
42:   public void setEstado(vEstado){
43:         estado = vEstado;
44:   }
45:
46:   public void setPais(vPais){
47:         pais = vPais;
48:   }
49:
50:
51:   public static void main (String args[]){
52:         ExibeNome exibeNome = new ExibeNome();
53:
54:         setNome("Rafael");
55:         setEndereco("Rua xyz");
56:         setCidade("XXX");
57:         setEstado("SP");
58:         setPais("Brasil");
59:         System.out.println(cliente1.getNome());
60:         System.out.println(cliente1.getEndereco());
61:         System.out.println(cliente1.getCidade());
62:         System.out.println(cliente1.getEstado());
63:         System.out.println(cliente1.getPais());
64:   }
65: }
66:
67: package com.pacote;
68:
69: public aspect AspectoCliente{
70:   pointcut registra() :
71:               call(public * cliente1.*(*));
72:
```

```
73:   before() : registra(){
74:         System.out.println("Chamado um método");
75:   }
76: }
```

4.4 ADENDOS (*ADVICE*)

Como dito na sessão anterior, os *pointcuts* definem os *join points* onde serão aplicados os códigos do aspecto. Os códigos que são executados pelo aspecto chamam-se adendos. Em resumo, um adendo é um bloco de código que é combinado ao código principal por intermédio dos pontos de junção definidos no ponto de atuação do aspecto (BODKIN; LADDAD, 2004). No Código 9 da Sessão 4.3.1, as linhas 73 a 75 exemplificam um adendo. A linha 74 é executada antes (*before()*) que todos os pontos de junção definidos no ponto de atuação *registra()* (linhas 70 e 71).

Um adendo é muito parecido com um método da orientação a objeto, porém um adendo não possui nome de referência próprio, sua única referência é o ponto de atuação. Um ponto de atuação pode ter vários adendos executando em diferentes momentos. A Tabela 3 compara os métodos da POO e os adendos.

Tabela 3 - Comparação entre adendos e métodos (In: GOETTEN; WINCK, 2006, p102)

Características	**Adendo**	**Métodos**
Identifição por um nome	Não	Sim
Segue conjunto de regras para acesso aos membros de outros tipos e aspectos	Sim	Sim
Declara que pode tratar exceções checadas (*checked exceptions*)	Sim	Sim
Referência a um aspecto (ou objeto) utilizando *this*	Sim	Sim
Pode ser chamado diretamente (como um método do tipo *static*, por exemplo)	Não	Sim
Identificador de especificação de acesso	Não	Sim
Acesso a variáveis especiais juntamente com o comando *this*, para verificação de pontos de junção capturados (*thisJointPoint*, *thisJoinPointStaticPart*, e *thisEnclosingJoinPointStaticPart*)	Sim	Não

"Um adendo possui três tipos: *before*, *after* e *around*" (BODKIN; LADDAD, 2004). A fim de tornar flexível a execução de um adendo, a POA possui estes três tipos de

adendos. Cada tipo de adendo é executado em um momento diferente quando o ponto de atuação alcança um ponto de junção.

O adendo *before* é o 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*) (GOETTEN; WINCK, 2006), conforme o exemplo do Código 10:

Código 10 – Exemplo da utilização do Adendo *before()*

```
before() : ponto1(){
    system.out.println(“Impresso antes do ponto de junção”);
}
```

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* (GOETTEN; WINCK, 2006), como mostrado no exemplo do Código 11 abaixo:

Código 11 – Exemplo da utilização do Adendo *after*

```
after() : ponto1(){
      system.out.println(“Impresso depois do ponto de junção”);
}

after() returning() : ponto1(){
      system.out.println(“Impresso depois do ponto de junção se não tiver
erros”);
}

after() throwing() : ponto1(){
      system.out.println(“Impresso depois do ponto de junção se encontrar
erro”);
}
```

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 (GOETTE; WINCK, 2006). O Código 12 mostra como um adendo *around* é estruturado:

Código 12 – Exemplo da utilização do Adendo *around()*

```
around() : ponto1(){
      system.out.println("Impresso antes do ponto de junção");

         proceed();

      system.out.println("Impresso depois do ponto de junção");
}
```

A sequência de execução de um adendo é descrita no autômato da Figura 5, o estado inicial *q0* é o momento em que o ponto de atuação encontra um ponto de junção. A partir deste momento, os adendos dentro do aspecto seguem a estrutura abaixo (Figura 5). 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 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 é dividido em 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()*).

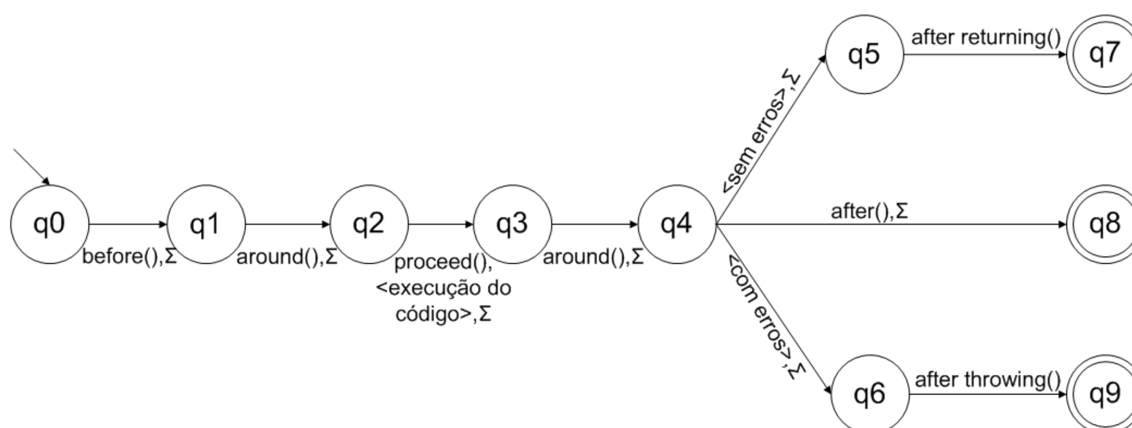

Figura 5 – Autômato da ordem de execução de um adendo

4.5 ASPECTOS

Um aspecto é a unidade básica do paradigma orientado a aspecto, assim como o objeto é a unidade básica do paradigma orientado a objetos. “Aspecto é uma parte de um programa POA que separa os interesses ortogonais dos interesses principais” (SAFONOV, 2008), assim, o aspecto agrupa um conjunto de adendos e seus respectivos pontos de atuação, conforme as linhas 69 a 76 do Código 9 (Sessão 4.3.1).

Assim, um aspecto contém todos os comportamentos que serão aplicados ao esqueleto principal do sistema com a intenção de alterar a semântica dos componentes funcionais. Para melhor organizar estes aspectos, a melhor forma é modularizar os conjuntos de aspectos necessários para programar um interesse.

O aspecto é uma estrutura simples, pois apenas possui um identificador que identifica o tipo de acesso possível (*public*, *protected* ou *private*), a palavra-chave *aspect* e o nome de referência. No bloco de código de um aspecto, deve haver um ponto de junção e os adendos correspondentes a função do aspecto, como também pode agrupar métodos e atributos.

Com a utilização de um aspecto, a programação é encarada por um ângulo mais prático, podendo dedicar mais tempo ao correto desenvolvimento dos códigos funcionais e de seus respectivos aspectos.

4.6 REFLEXÃO

Reflexão é uma característica de um paradigma que permite ao programa acesso às informações sobre si próprio (*AspectJ Programming Guide*, 2011). Uma linguagem de programação deve oferecer algum meio para estas informações sejam recuperadas.

No *AspectJ*, existem três classes que fazem possível a reflexão: *thisJoinPoint*, *thisJoinPointStaticPart* e *thisEnclosingJoinPointStaticPart*.

O *thisJoinPoint* é um objeto especial que identifica o ponto de junção capturado, o objeto *thisJoinPointStaticPart* identifica apenas a parte estática do ponto de junção, onde não precisa alocação de memória quando for utilizada e o objeto *thisEnclosingJoinPointStaticPart* disponibiliza informações sobre o contexto que contém o ponto de junção.

4.6.1 thisJoinPoint

Como dito anteriormente, este objeto contém informações reflexivas sobre o Ponto de Junção que estiver sendo utilizado pelo adendo. O *thisJoinPoint* apenas pode ser utilizado no contexto do adendo, ou seja, não pode acessar métodos ou atributos estáticos (*Aspect Programming Guide*, 2011).

A classe especial *thisJoinPoint* é um objeto da classe ***org.aspectj.lang.JoinPoint*** dentro do adendo, similar ao objeto *this* em Java. Deste modo, o *thisJoinPoint* disponibiliza diversos métodos que retornam as informações dinâmicas sobre o ponto de junção, conforme mostrado abaixo:

- ***String toString()*****:** Este método retorna um texto representando a assinatura completa do ponto de junção.

- ***String toShortString()*****:** Este método retorna um texto representando resumidamente a assinatura do ponto de junção.

- ***String toLongString()***: Este método retorna um texto representando a forma estendida da assinatura do ponto de junção.

- ***Signature getSignature()***: Este método retorna a assinatura do ponto de junção, sendo este um objeto e não um texto. O objeto signature (assinatura) possui alguns métodos próprios:

  - ***String getName():*** Retorna uma string com a identificação da assinatura.

  - ***String getModifiers():*** Retorna uma string que identifica qual é o modificador de acesso da assinatura do ponto de junção.

  - ***String getKind():*** Retorna o tipo qual é o tipo do ponto de junção recuperado.

- ***Object[] getArgs()***: Retorna os argumentos do ponto de junção.

- ***Object getTarget()***: Retorna o objeto que recebeu a chamada do método do ponto de atuação.

- ***Object getThis()***: Retorna o objeto que fez a chamada do método do ponto de atuação.

- ***getStaticPart()***: retorna o objeto *thisJoinPointStaticPart* que será estudado abaixo.

4.6.2 thisJoinPointStaticPart

Este objeto especial trabalha de forma similar a *thisJoinPoint*, porém o *thisJoinPointStaticPart* apenas disponibiliza a parte estática do ponto de junção.

O *thisJoinPointStaticPart* também é um objeto e possui métodos similares ao *thisJoinPoint* como *getSignature()*, *toString()*, *toShortString()* e *toLongString()*, porém possui métodos mais específicos, como:

- *SourceLocation getSourceLocation()*: retorna a localização do ponto de junção no código fonte.

- *String getKind()*: Este método é utilizado para retornar o tipo do ponto de junção.

4.6.3 thisEnclosingJoinPointStaticPart

O objeto especial *thisEnclosingJoinPointStaticPart* possuí dados estáticos do contexto aonde o ponto de junção está, ou seja, neste caso, o contexto seria a execução do método aonde foi feito a chamada.

Este objeto é do mesmo tipo do objeto *thisJoinPointStaticPart*, assim possuí o mesmos métodos.

O Código 13 abaixo se trata de uma classe *Pais* que possui um método *adicionaEstado*. O aspecto *NomearMetodo* atua em dois pontos de junção: *adicionarPais()*, que está dentro do bloco de código *main*, e o método *add*, que está dentro do bloco *acidionarPais()*. O resultado deste aspecto são retornos dos nomes dos métodos do contexto do ponto de junção.

Código 13 – Exemplo da utilização do objeto especial *thisEnclosingJoinPointStaticPart*

```
 1: public class Pais {
 2:   private List<Estado> estados;
 3:   private String nome;
 4:
 5:   public void adicionaEstado(Estado e){
 6:      estados.add(e);
 7:   }
 8:
 9:   public static void main (String[] args){
10:      Pais p = new Pais();
11:      p.adicionaPais(new Estado("São Paulo"));
12:   }
13: }
14:
```

```
15: public aspect NomearMetodo{
16:    pointcut nomear() : call (* *.*(..)) && !within(NomearMetodo);
17:
18:   before() : nomear() {
19:       System.out.println("MÉTODO: " +
thisEnclosingJoinPointStaticPart.getSignature.getName());
20:    }
21: }
```

```
Resultado:
            MÉTODO: main
            MÉTODO: adicionaEstado
```

Se substituir o objeto *thisEnclosingJoinPointStaticPart* pelo *thisJoinPointStaticPart*, como descrito no Código 14, o resultado seria o nome do próprio ponto de junção.

Código 14 – Exemplo de utilização do objeto especial *thisJoinPointStaticPart*

```
15: public aspect NomearMetodo{
16:    pointcut nomear() : call (* *.*(..)) && !within(NomearMetodo);
17:
18:   before() : nomear() {
19:       System.out.println("MÉTODO: " +
thisJoinPointStaticPart.getSignature.getName());
20:    }
21: }
```

```
Resultado:
            MÉTODO: adicionaEstado
            MÉTODO: add
```

4.7 INTERTIPOS

Declarações de Intertipos (*Intertype Declarations*), também conhecidas como Introduções (*Introductions*), são recursos que permitem modificações no sistema, como a inserção de métodos e atributos, também implementa interfaces ou estende classes (BODKIN; LADDA, 2004).

Segundo WINCK e GOETTEN (2006), as declarações podem ter diversas formas, como: inclusão de métodos abstratos e/ou concretos, construtores, campos, hierarquia de classes, precedência entre os aspectos, alterar exceções e exibir mensagens de erro e alerta.

Para programar os métodos concretos, ou *Concrete Methods*, em outra classe, deve-se seguir a regra:

```
visibilidade tipoDeDados classeReceptora.nomeDoMetodoConcreto(parâmetros){
      // códigos
}
```

Porém, se o método a ser introduzido é abstrato, então é necessário que o modificador *abstract* esteja descriminado, conforme a regra abaixo:

```
visibilidade abstract tipoDeDados classeReceptora.nomeDoMetodo(parâmetros){
      // códigos
}
```

Para introduzir um construtor em uma classe, a regra é simples, porém necessita da palavra-chave *new*:

```
      visibilidade classeReceptora.new(parâmetros){
            // códigos
      }
```

Como nas demais regras anteriores, a criação de uma declaração intertipos para a introdução de um atributo não é diferente, necessita-se do modificador de visibilidade, o tipo do atributo, a classe receptora, o nome do atributo e o valor (se ele já for inicializado). O modificador de visibilidade é apenas para utilização do aspecto, assim se o atributo for *private*, este apenas será visível ao código do aspecto. Não ocorrerá nenhum erro por duplicidade de nomes, desde que o atributo a ser introduzido e o atributo da classe receptora forem declarados como público.

```
      visibilidade tipoDeDados classeReceptora.nomeDoAtributo = valor;
```

As declarações de intertipos podem fazer com que uma classe herde ou implemente outra classe. Para que isto aconteça, as classes devem manter as regras da linguagem de programação, elas apenas implementam ou herdam de uma classe que não possuem outra superclasse. Esta hierarquia criada é apenas utilizada em nível aspectual, ou seja, estas classes possuirão hierarquia apenas para tratamento dentro do aspecto, enquanto que os códigos componentes não serão modificados.

As regras para declarar uma herança e implementação de outra classe estão, respectivamente, descritas abaixo:

```
declare parents : classeFilho extends classePai;
declare parents : classe implements classeInterface;
```

As exceções também podem ser tratadas com as declarações intertipos. Assim, como as exceções são tratadas pelos aspectos, o compilador necessita de algum artifício para ignorar (*bypass*) o erro da necessidade de tratamento das exceções. Desta forma, utiliza-se o artifício *declare soft* para indicar este desvio, como descrito na regra abaixo:

```
declare soft : Exception : Pointcut;
```

Através de declarações intertipos também é possível forçar o compilador a apresentar erros ou avisos caso algum *joinpoint* for executado, retornando um atributo *String*, conforme abaixo:

```
declare error : Pointcut : String;
declare warning : Pointcut : String;
```

4.7.1 Precedências

Quando utilizamos um aspecto que possui diversos adendos que são aplicados ao mesmo ponto de junção, a precedência entre os mesmo será aplicada conforme o momento de execução do adendo (*before(), after(), around()*), porém se existir dois adendos com o mesmo ponto de execução, os adendos serão executados na ordem que foram criados.

Um fator que pode auxiliar na criação de um aspecto são os tipos de chamadas de um ponto de junção (*call, execution, initialization*) definidos no ponto de atuação. A Figura 6 mostra como estes comandos podem ordenar os adendos através do tipo de chamada dos pontos de junção. O estado inicial *q0* é o momento em que o ponto de atuação encontra um ponto de junção. Sequencialmente, qualquer adendo que possua um ponto de junção com o tipo de chamada *initialization* é executado quando este for instanciado. Se dois adendos *before()* diferentes atuarem em um único

ponto de junção e um deles possuir um ponto de atuação com o designador *call* para este ponto de junção e o outro adendo possuir o designador *execution*; o adendo *before()* com designador *call* é executado primeiro (como na transição de *q1* para *q2*) e depois o *execution* (transição *q2* para *q3*). A ordem de execução de adendos com designadores *call* e *execution* combinados depois do código funcional é alterada, sendo executado primeiro o adendo com o designador *execution*, e depois, o adendo com o designador *call*.

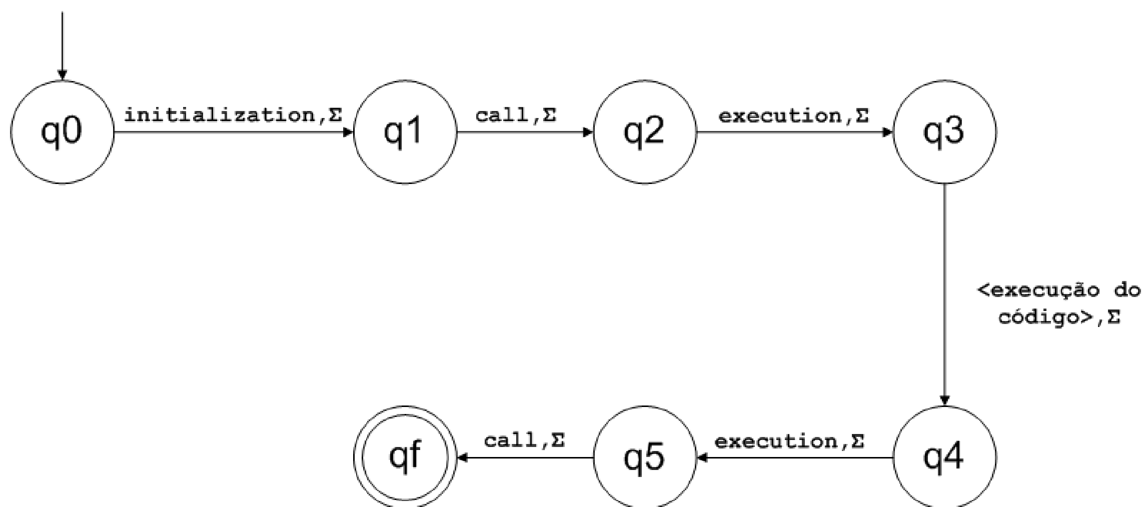

Figura 6 – Representação da execução dos adendos a partir dos designadores

Para ordenar a execução de aspectos de um mesmo ponto de junção que possui momentos de execução e tipos de chamadas idênticos deve-se utilizar uma declaração de intertipos chamada *precedence*, que ordena a execução dos aspectos conforme a ordem inserida, conforme regra abaixo:

```
declare precedence : aspecto1, aspecto2, aspectoN;
```

5 ESTUDO DE CASO

O paradigma de Programação Orientado a Aspecto pode ser aplicado em qualquer tipo de projeto, sendo este projeto implementado inicialmente em POA ou não. A aplicação de aspectos em um projeto programado em outra linguagem é muito simples e o *Eclipse* disponibiliza uma opção de conversão para que as bibliotecas necessárias sejam incluídas ao projeto.

Com a intenção de aplicar os conceitos da POA foi desenvolvido um sistema bibliotecário. O sistema foi inicialmente implementado utilizando o paradigma de programação orientada a objetos e, posteriormente, os aspectos foram combinados ao código principal, assim permitiu a visualização de como seria a conversão de um projeto do tipo OO para um projeto orientado a aspectos.

5.1 O SISTEMA BIBLIOTECÁRIO

Um sistema bibliotecário possui como característica principal manter o controle da movimentação dos exemplares de uma biblioteca. Assim, o software deve manter informações de *alunos*, *funcionários*, *livros* e *exemplares*. O sistema também deve controlar os empréstimos e as devoluções.

Neste sistema, os livros são classificados por tipo e cada tipo possui um período para que o exemplar seja entregue. Caso o usuário não entregue em dia, uma multa diária deverá ser cobrada. Um livro pode possui diversos exemplares que são controlados por um número (ou tombo). Cada exemplar necessita de um atributo que identifique o status do exemplar.

Cada aluno possui um atributo identificador do seu status de pendência na biblioteca. Este status indicará se o aluno pode ou não locar um livro. Durante a devolução, o sistema deve indicar se existe alguma multa, e, em caso positivo, imprimir a fatura.

Este sistema registra as ações que são efetuadas no processo *Manter Exemplar* e *Emprestar Livros*. Assim pode-se verificar o histórico dos valores adicionados aos atributos.

Normalmente, em um sistema qualquer, existem métodos para que os usuários apenas tenham acesso às transações que lhes são permitidas. Deste modo, este sistema partirá do pressuposto que os funcionários sejam classificados por categoria, por exemplo: *Atendentes*, *Bibliotecários*, *Coordenador*, *Administrador*. Estes acessos serão definidos na tela de *Cadastro de Tipo de Usuários*.

5.2 MÉTODOS DE DESENVOLVIMENTO

O Sistema Bibliotecário, explanado na sessão anterior, possui características sistêmicas que seriam facilmente tratadas pela POA, porém para ampliar os resultados, o software foi desenvolvido inicialmente em POO, após o desenvolvimento foi realizada uma análise para a separação dos seus interesses.

A análise realizada permitiu a identificação dos interesses chaves para a aplicação, interesses referentes às regras de negócios e os interesses de suporte ao sistema, conforme será descrito na Sessão 5.3. Assim, os interesses principais foram implementados primeiramente em Java. E os demais interesses foram desenvolvidos depois, em *AspectJ*.

As informações deste sistema serão registradas no banco de dados do SGBD (Sistema Gerenciador de Banco de Dados) HSQLDB, desenvolvido em Java. A estrutura das tabelas deste banco de dados pode ser observada no Diagrama Entidade-Relacionamento da Figura 7.

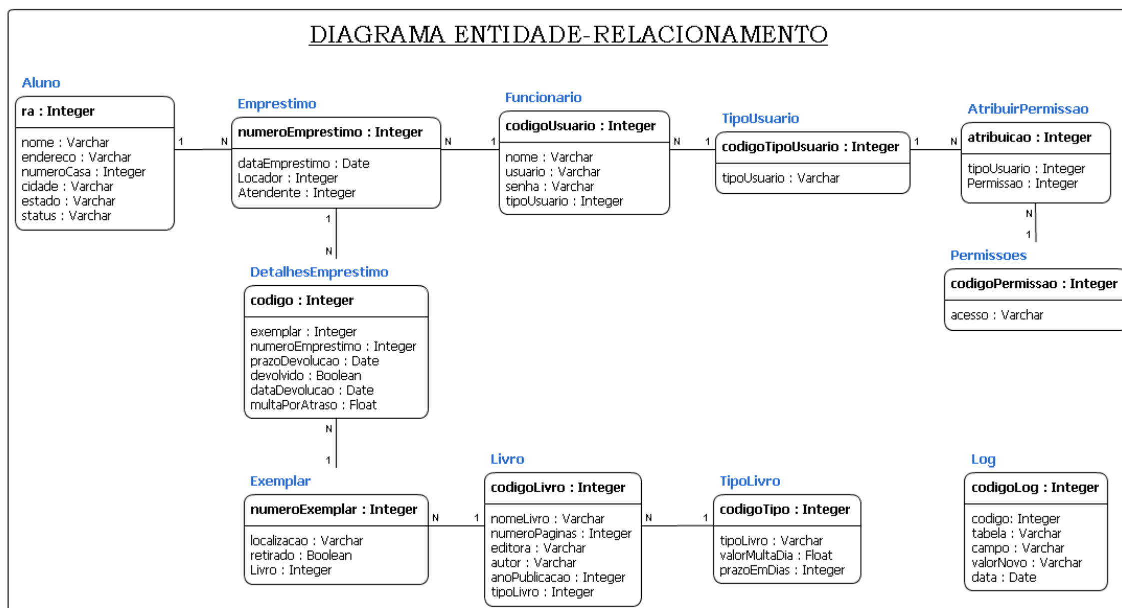

Figura 7 – Diagrama Entidade-Relacionamento

A persistência com o banco de dados HSQLDB foi implementado com o auxilio do *Hibernate*, que é um framework para mapeamento de objetos para a comunicação com o banco. Mesmo a persistência sendo um interesse sistêmico, ela é um pouco mais complexa e merece uma atenção especial, por isto não foi integrada neste trabalho.

O sistema está organizado hierarquicamente em pacotes. O pacote *beans* agrupa as classes que correspondem ao modelo objeto que mapeia o banco relacional. O pacote *dao* contém as classes que fazem comunicação dos dados entre a aplicação e o banco de dados utilizando o *Hibernate*. No pacote *util* está o arquivo de configuração do *Hibernate*. O pacote *aspectos* e o pacote *telas* agrupam respectivamente os interesses sistêmicos e os códigos da tela, conforme descrito na Figura 8.

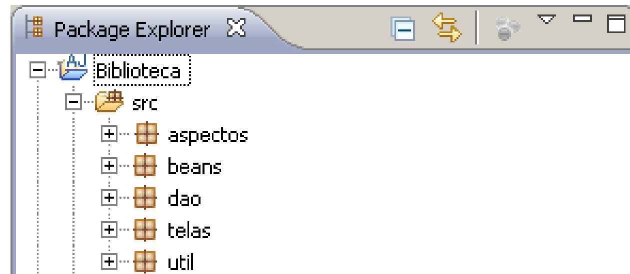

Figura 8 – Estrutura Hierárquica do Sistema

Para o desenvolvimento deste software foi utilizado o conceito de separação de interesses, visto no Capítulo 3. Deste modo, o software foi dividido em duas partes: Componentes e Funcionalidades. Na Figura 9, pode-se visualizar a estrutura do sistema, onde os aspectos atuam diretamente nos objetos instanciados nas classes do pacote *telas* que, como dito anteriormente, possui os componentes que interagem com o usuário.

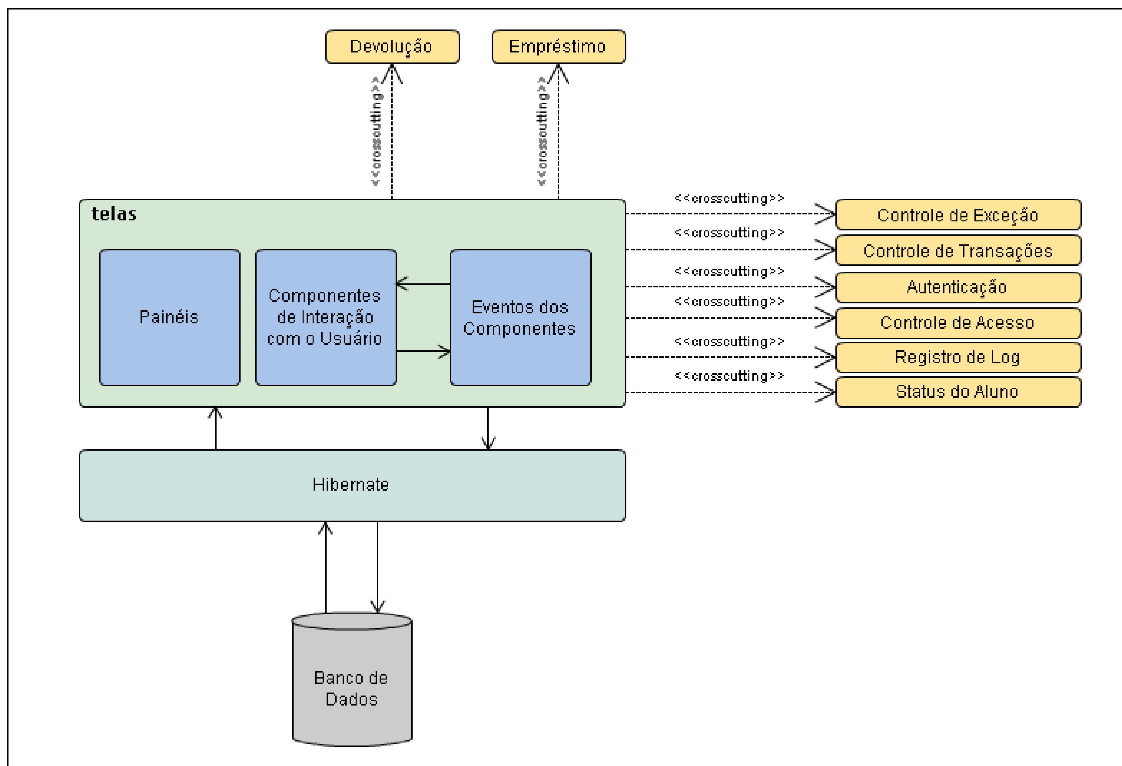

Figura 9 – Arquitetura do Sistema Bibliotecário

5.3 IDENTIFICAÇÃO E SEPARAÇÃO DOS INTERESSES

Conforme visto anteriormente, a separação de interesses é um método muito importante para o desenvolvimento de um software organizado. Na Tabela 4, os interesses do sistema bibliotecário foram separados.

Tabela 4 – Separação de Interesses do Sistema Bibliotecário

SISTEMA BIBLIOTECÁRIO	
INTERESSES FUNCIONAIS	**INTERESSES SISTÊMICOS**
Manter Tipo de Livros	Controlar Acesso
Manter Livros	Autenticar Usuário
Manter Exemplar	Checar Disponibilidade do Livro
Pesquisar Livros	Alterar Status do Exemplar
Manter Aluno	Determinar Prazo de Devolução
Manter Tipo de Usuário	Checar Disponibilidade do Aluno
Manter Funcionário	Impressão da Fatura (Multa)
Atribuir Permissões	Calcular Multa
Efetuar *Login*	Alterar Status do Exemplar
Emprestar Livro	Alterar Status do Aluno
Pesquisar Empréstimos	Registrar Ações
Devolver Livro	Tratamento de Exceções
	Controle de Transações

Na coluna descrita como Interesses Funcionais estão listadas as ações que constituem o corpo do sistema, ou seja, os requisitos essenciais para que o sistema atenda as expectativas do cliente. Podem-se representar os interesses funcionais através de um diagrama de classes, como na Figura 10.

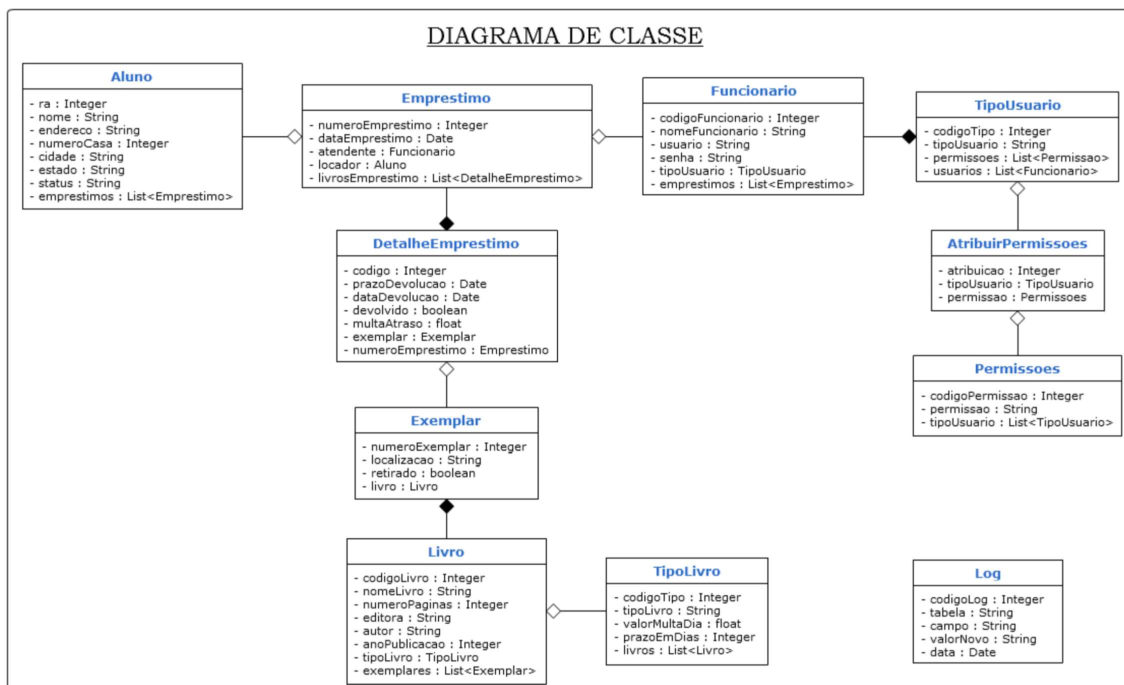

Figura 10 – Diagrama de Classes

A coluna descrita como Interesses Sistêmicos, da Tabela 4, está listada as características sistêmicas que devem constituir o sistema. Devido às divergências nas técnicas de modelagem de um sistema orientado a aspecto, foi utilizado o Diagrama de Casos de Uso (UML) para demonstrar o comportamento dos interesses sistêmicos do sistema. Na Figura 11, pode-se visualizar um caso de uso do sistema no qual são destacados os interesses ortogonais.

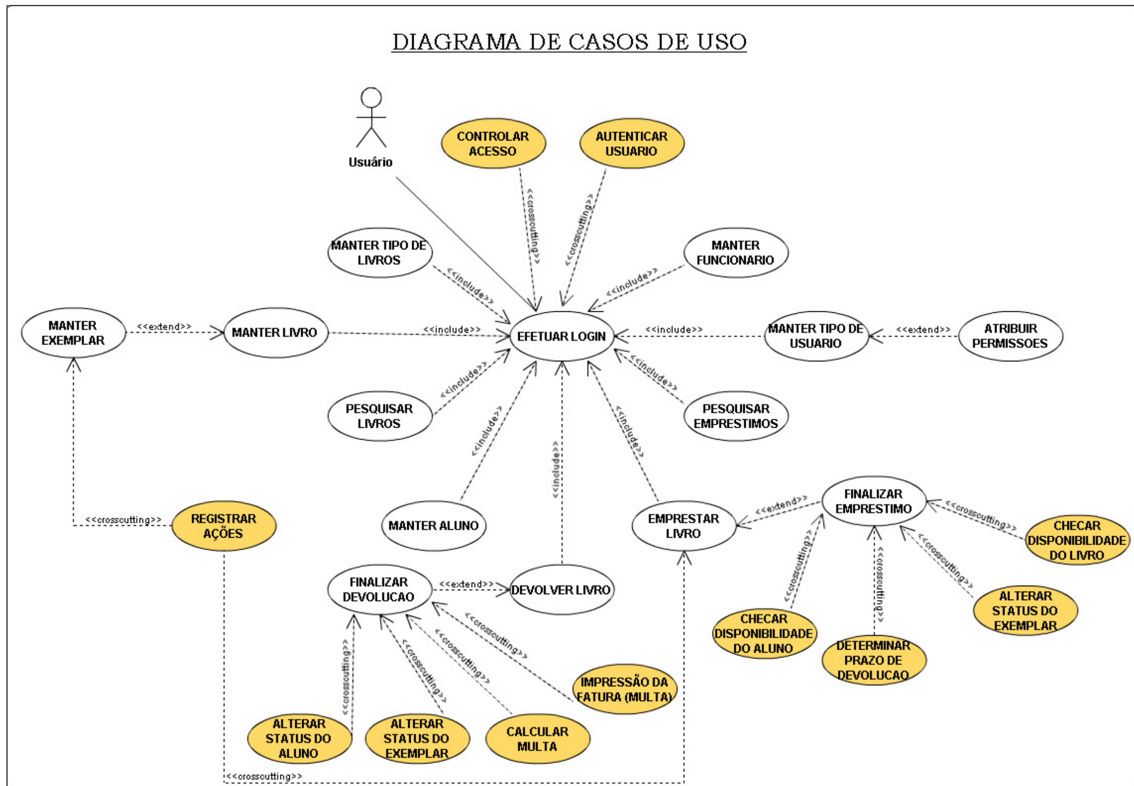

Figura 11 – Diagrama de Casos de Uso

Existem aspectos que não são possíveis de serem representados no caso de uso, como os aspectos que controlam exceções e transações.

Para separar os interesses de forma satisfatória deve-se analisar se existe possibilidade de agrupar as similaridades do software, não se importando com o código, e sim, com sua finalidade.

Deste modo, verifica-se que os interesses *Alterar Status do Exemplar*, *Calcular 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 recurso de agrupamento foi utilizado para unir 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*, conforme demonstrado na Figura 12.

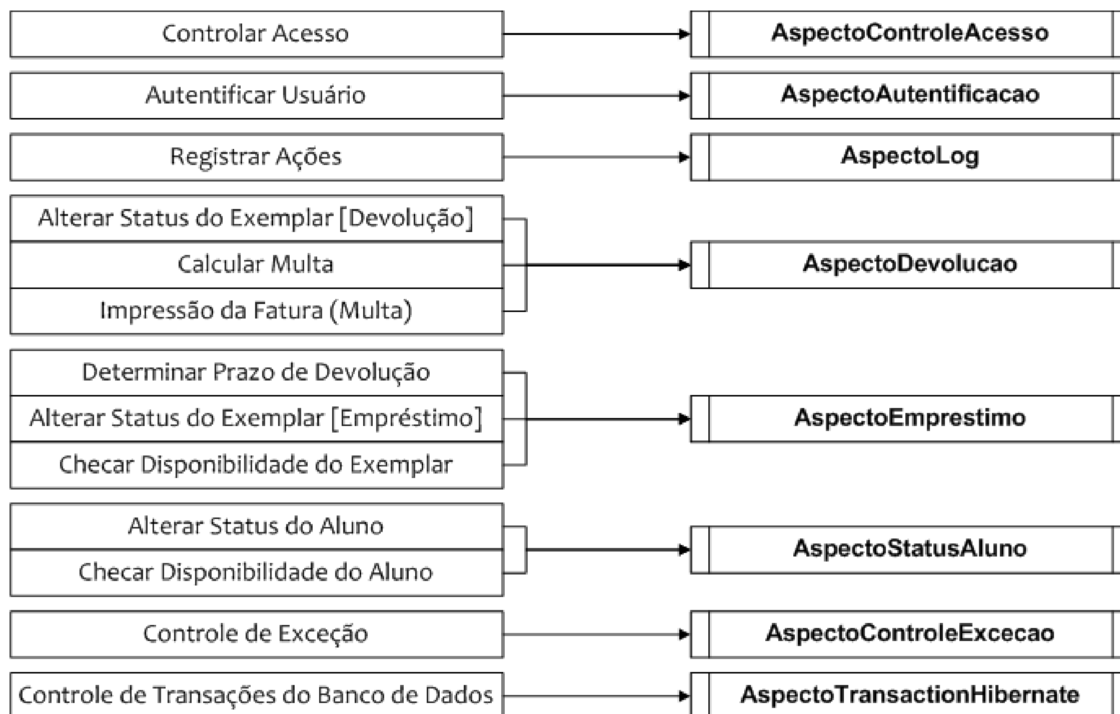

Figura 12 – Agrupamento dos Interesses Sistêmicos em Aspectos

5.4 IMPLEMENTAÇÃO DOS ASPECTOS

Conforme mostrado neste capítulo, em um estudo de caso de uma biblioteca pode haver diversos tipos de aspectos. Estes aspectos podem ser representados pelos comportamentos que atuam sobre um ponto de junção. A Tabela 5 exibe a relação de pontos de junções por aspecto desenvolvido neste estudo de caso, onde a coluna Pontos de Atuação descreve o nome do ponto de atuação, o tipo de chamada e o ponto de junção.

Tabela 5 – Pontos de Junção por aspectos do sistema bibliotecário

ASPECTO	PONTOS DE ATUAÇÃO
AspectoAutentificacao	- autentificacao : **call** : telas.Login.autentificacao
AspectoControleAcesso	- controle : **call** : telas.Principal.checarPermissao
AspectoControleExcecao	- <sem nome> : **call** : telas.CadastrarAluno.botaoSalvar, telas.CadastrarAluno.botaoExcluir, telas.CadastrarFuncionario.botaoSalvar, telas.CadastrarFuncionario.botaoExcluir, telas.CadastrarFuncionario.botaoPesquisar, telas.CadastrarLivros.botaoSalvar, telas.CadastrarLivros.botaoDeletar, telas.CadastrarTipoLivro.botaoSalvar, telas.CadastrarTipoLivro.botaoExcluir, telas.CadastrarTipoLivro.botaoPesquisa, telas.CadastrarTipoUsuario.botaoSalvar, telas.CadastrarTipoUsuario.botaoPesquisar, telas.CadastrarTipoUsuario.botaoRemover, telas.CadastrarTipoUsuario.botaoAdicionar, telas.Exemplar.botaoSalvar, telas.Login.botaoRetirar, telas.Logs.botaoPesquisar, telas.Permissoes.botaoPesquisar, telas.PesquisarEmprestimo.botaoPesquisar, telas.PesquisarLivros.cliqueTabelaLivro, telas.PesquisarLivros.botaoPesquisar, telas.Emprestimos.botaoSalvar, telas.Emprestimos.botaoAdicionar, telas.Emprestimos.botaoDeletar, aspectos.AspectoLog.formataData, aspectos.AspectoDevolucao.setarStatusExemplar, aspectos.AspectoEmprestimo.setarStatusExemplar, aspectos.AspectoEmprestimo.atualizarPrazo, aspectos.AspectoStatusAluno.checarAluno, aspectos.AspectoStatusAluno.alterarStatus
AspectoDevolucao	- devolver : **call** : telas.LivrosDevolver.botaoFinalizar - multa : **initialization** : telas.LivrosDevolver.new
AspectoEmprestimo	- emprestar : **call** : telas.Emprestimo.botaoAdicionar
AspectoLog	- registros : **call** : beans.Exemplar.setNumeroExemplar, beans.Exemplar.setLocalizacao, beans.Exemplar.setRetirado, beans.Exemplar.setLivro, beans.DetalheEmprestimo.setCodigo, beans.DetalheEmprestimo.setPrazoDevolucao, beans.DetalheEmprestimo.setDataDevolucao, beans.DetalheEmprestimo.setDevolvido, beans.DetalheEmprestimo.setMultaAtraso, beans.DetalheEmprestimo.setExemplar, beans.DetalheEmprestimo.setNumeroEmprestimo
AspectoStatusAluno	- status : **call** : telas.LivrosDevolver.botaoFinalizar, telas.Emprestimo.selecionarCbLocador, telas.CadastrarAluno.getValoresTable
AspectoTransactionHibernate	- transactionInicializar : **initialization** : telas.Login.**new** - transaction : **call** : telas.CadastrarAluno.botaoPesquisar, telas.CadastrarAluno.botaoSalvar, telas.CadastrarAluno.botaoExcluir, telas.CadastrarFuncionario.botaoPesquisar, telas.CadastrarFuncionario.botaoSalvar, telas.CadastrarFuncionario.botaoExcluir, telas.CadastrarFuncionario.carregarCombobox, telas.CadastrarLivros.criarJanela, telas.CadastrarLivros.desenharPainelExemplares,

	telas.CadastrarLivros.botaoSalvar, telas.CadastrarLivros.botaoAdicionar, telas.CadastrarLivros.botaoDeletar, telas.CadastrarTipoLivro.botaoSalvar, telas.CadastrarTipoLivro.botaoExcluir, telas.CadastrarTipoLivro.botaoPesquisa, telas.CadastrarTipoUsuario.botaoSalvar, telas.CadastrarTipoUsuario.botaoPesquisar, telas.CadastrarTipoUsuario.botaoRemover, telas.CadastrarTipoUsuario.botaoAdicionar, telas.CadastrarTipoUsuario.cliqueTabelaTipoUsuario, telas.Emprestimos.botaoSalvar, telas.Emprestimos.botaoAdicionar, telas.Emprestimos.botaoDeletar, telas.Emprestimos.carregarAlunos, telas.Emprestimos.carregarFuncionario, telas.Login.buscarUsuario, telas.Logs.botaoPesquisar, telas.PesquisarEmprestimo.botaoPesquisar, telas.PesquisarLivros.cliqueTabelaLivro, telas.PesquisarLivros.botaoPesquisar, telas.Principal.chamarTipoLivro, aspectos.AspectoDevolucao.setarStatusExemplar, aspectos.AspectoEmprestimo.setarStatusExemplar, aspectos.AspectoEmprestimo.atualizarPrazo, aspectos.AspectoStatusAluno.checarAluno, aspectos.AspectoStatusAluno.alterarStatus

A partir destes pontos de junção cada aspecto possui uma característica. Como se pode verificar analisando o aspecto *AspectoAutenticacao* que possui apenas um adendo que é executado antes do ponto de junção *autenticacao*. O ponto de junção *autenticacao* possui três argumentos que foram capturados pelo aspecto no momento de sua intervenção no *join point*, são eles: uma lista de funcionários filtrados pelo nome de usuário e o usuário e senha que foram digitados pelo usuário. O objetivo do aspecto é verificar se as informações que estão armazenadas na lista de funcionários estão coerentes com os dados de usuário e senha fornecidos pelo usuário, a fim de permitir ou não o acesso ao sistema, de acordo com o Código 15.

Código 15 – Aspecto de autenticação do usuário

```
public aspect AspectoAutenticacao {

   pointcut autenticacao(List<Funcionario> func, String user, String password) :
      call(void telas.Login.autenticacao(List<Funcionario>, String, String))
          && args(func, user, password);

   before(List<Funcionario> func, String user, String password) :
                                    autenticacao(func, user, password){

      Funcionario usuarioLogado = new Funcionario();

      if (func.size() == 0){
         JOptionPane.showMessageDialog(null, "Usuário não corresponde");
          return ;
         } else {
                usuarioLogado = func.get(0);
                if ((usuarioLogado.getUsuario().compareTo(user) == 0 )){
                   if((usuarioLogado.getSenha().compareTo(password) == 0)){
                       new Principal(usuarioLogado);
                    }else{
                         JOptionPane.showMessageDialog(null, "Senha errada");
                         return ;
                         }
                  }else{
                       JOptionPane.showMessageDialog(null, "Usuário Incorreto");
                       return ;
                       }
                }
         }
}
```

A Figura 13 é um diagrama de sequência que representa a intervenção do aspecto *AspectoAutenticacao* no código principal.

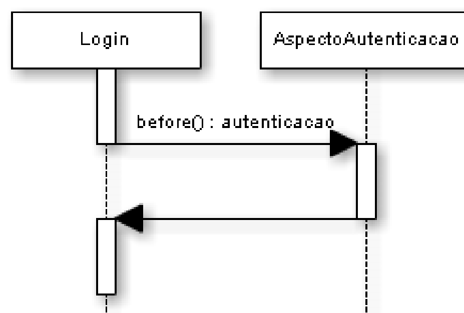

Figura 13 – Diagrama de Sequência: AspectoAutentificacao

O aspecto desvia o curso de execução da classe *Login* para a classe *AspectoAutenticacao* antes da execução do ponto de junção (*before() : autentificacao*).

O comportamento do *AspectoControleAcesso* pode ser definida através da Figura 14, que é um diagrama de sequência do adendo *around* deste aspecto. Quando o ponto de junção é localizado na classe *Principal* a execução é desviada para o aspecto através do adendo *around*. O aspecto continua ser executado até encontrar o comando *proceed* que retorna a execução ao código principal até o término do bloco de código indicado como ponto de junção, após este momento o foco volta ao aspecto, para que este possa concluir a execução.

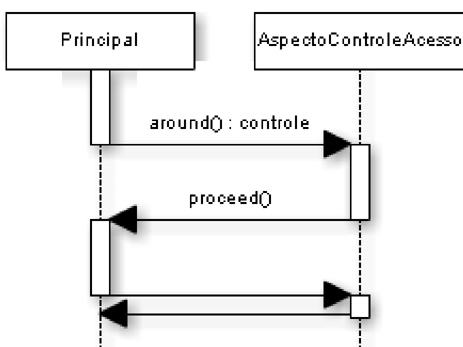

Figura 14 – Diagrama de Sequência: AspectoControleAcesso

Como o *AspectoControleAcesso* possui a função de restringir o acesso do usuário às telas do sistema conforme a atribuição das permissões que está cadastrado no banco de dados, não houve a necessidade executar nenhum código após a execução do ponto de junção. Porém houve a necessidade de utilizar o adendo *around* para que pudesse ser utilizado o comando *proceed* e retornar alguns argumentos alterados ao código principal, como é possível verificar no Código 16.

Código 16 - Código Fonte do aspecto AspectoControleAcesso

```
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().compareTo(perm[cont]) == 0){
              aut = true;
           }
         }
         if(aut){
            at[cont] = true;
         } else {
                 at[cont] = false;
                 }
      }

      proceed(perm,at,atPerm);

      }
}
```

No código acima, os parâmetros *perm*, *at* e *atPerm* são respectivamente, uma lista das permissões existentes no sistema, um vetor do tipo *boolean* que representa a configuração das permissões que serão aplicadas e uma lista de permissões que o usuário possui. O ponto de junção é chamado do método *checarPermissao* (Código 17), que tem por objetivo habilitar ou não os componentes gráficos de acesso às telas do sistema conforme o valor que está no vetor *at*. O aspecto intercepta este método antes que o código seja executado, podendo assim modificar o valor que será atribuído futuramente aos componentes.

Código 17 - Código Fonte do método *checarPermissao*

```
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]);
}
```

Como o sistema foi implementado inicialmente no Paradigma Orientado a Objetos, os interesses sistêmicos estavam espalhados por todo o código principal, elevando a complexidade do software. O Código 18 é um exemplo deste problema encontrado no estudo de caso.

Código 18 - Bloco de código do Botão Salvar da classe CadastrarAluno

```
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);
```

Pode-se observar no código acima que dois interesses sistêmicos, *Controle de Exceções* e *Controle de Transações do Hibernate*, estão misturados ao código do componente.

Para melhorar a visibilidade deste código deve-se identificar o ponto de junção. Neste caso, percebe-se que os interesses ortogonais circundam uma chamada ao evento deste botão (*botaoSalvar()*), assim, concluímos que os interesses sistêmicos estão dando suporte a este ponto do código, identificado como ponto de junção.

A partir deste momento, é necessária a identificação individual de cada interesse sistêmico neste bloco de código, afinal o código deve ser reutilizado em toda a aplicação.

O bloco *try/catch* constitui o interesse de controle de exceções. Para satisfazer este interesse, o ponto de junção deve estar dentro do *try/catch*, fazendo necessária a utilização de um adendo *around* como demonstrado na Figura 15:

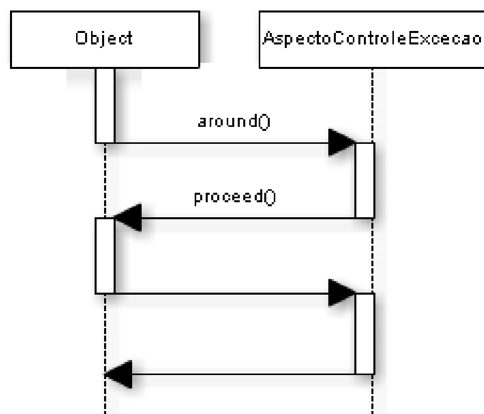

Figura 15 – Diagrama de Sequência: *AspectoControleExcecao*

Para que um aspecto possa combinar o controle de exceção com qualquer código que precise deste controle, é necessário a modularização do código, descrever os pontos de junção que o controle atuará e criar um intertipo *soft* para que o compilador entenda que a exceção está sendo tratada em outro ambiente, conforme mostrado no Código 19:

Código 19 – Aspecto de Controle de Exceções

```
public aspect AspectoControleExcessao {

   declare soft : Exception :
      execution(void telas.CadastrarAluno.botaoSalvar())
      || execution (void telas.CadastrarAluno.botaoExcluir())
      || execution (void telas.CadastrarFuncionario.botaoSalvar())
      || execution (void telas.CadastrarFuncionario.botaoExcluir())
      || execution (void telas.CadastrarFuncionario.botaoPesquisar())
      || execution (void telas.CadastrarLivros.botaoSalvar())
      || execution (void telas.CadastrarLivros.botaoDeletar())
      || execution (void telas.CadastrarTipoLivro.botaoSalvar())
```

```
        || execution (void telas.CadastrarTipoLivro.botaoExcluir())
        || execution (void telas.CadastrarTipoLivro.botaoPesquisa())
        || execution (void telas.CadastrarTipoUsuario.botaoSalvar())
        || execution (void telas.CadastrarTipoUsuario.botaoPesquisar())
        || execution (void telas.CadastrarTipoUsuario.botaoRemover())
        || execution (void telas.CadastrarTipoUsuario.botaoAdicionar())
        || execution (void telas.Exemplar.botaoSalvar())
        || execution (void telas.Login.buscarUsuario())
        || execution (void telas.Logs.botaoPesquisar())
        || execution (void telas.Permissoes.botaoPesquisar())
        || execution (void telas.PesquisarEmprestimo.botaoPesquisar())
        || execution (void telas.PesquisarLivros.cliqueTabelaLivro())
        || execution (void telas.PesquisarLivros.botaoPesquisar())
        || execution (void telas.Emprestimos.botaoSalvar())
        || execution (void telas.Emprestimos.botaoAdicionar())
        || execution (void telas.Emprestimos.botaoDeletar())
        || execution (void aspectos.AspectoLog.formataData(..))
        || execution (void aspectos.AspectoLog.cadastrarLog(..));

    void around(): call(void telas.CadastrarAluno.botaoSalvar())
        || call (void telas.CadastrarAluno.botaoExcluir())
        || call (void telas.CadastrarFuncionario.botaoSalvar())
        || call (void telas.CadastrarFuncionario.botaoExcluir())
        || call (void telas.CadastrarFuncionario.botaoPesquisar())
        || call (void telas.CadastrarLivros.botaoSalvar())
        || call (void telas.CadastrarLivros.botaoDeletar())
        || call (void telas.CadastrarTipoLivro.botaoSalvar())
        || call (void telas.CadastrarTipoLivro.botaoExcluir())
        || call (void telas.CadastrarTipoLivro.botaoPesquisa())
        || call (void telas.CadastrarTipoUsuario.botaoSalvar())
        || call (void telas.CadastrarTipoUsuario.botaoPesquisar())
        || call (void telas.CadastrarTipoUsuario.botaoRemover())
        || call (void telas.CadastrarTipoUsuario.botaoAdicionar())
        || call (void telas.Exemplar.botaoSalvar())
        || call (void telas.Login.botaoRetirar())
        || call (void telas.Logs.botaoPesquisar())
        || call (void telas.Permissoes.botaoPesquisar())
        || call (void telas.PesquisarEmprestimo.botaoPesquisar())
        || call (void telas.PesquisarLivros.cliqueTabelaLivro())
        || call (void telas.PesquisarLivros.botaoPesquisar())
        || call (void telas.Emprestimos.botaoSalvar())
        || call (void telas.Emprestimos.botaoAdicionar())
        || call (void telas.Emprestimos.botaoDeletar())
        || call (void aspectos.AspectoLog.formataData(..))
        {
         try{

             proceed();

         }catch(Exception e){
             e.printStackTrace();
         }
    }
}
```

Após a separação do interesse de controle de exceção, pode-se iniciar a separação do controle de transação do *Hibernate*. Para agrupar os interesses, deve-se pensar que o *Hibernate* possui três tipos de situação: quando a transação é efetuada com sucesso, quando a transação encontra algum erro e a inicialização do Hibernate, representados, respectivamente, nas Figuras 16, 17 e 18.

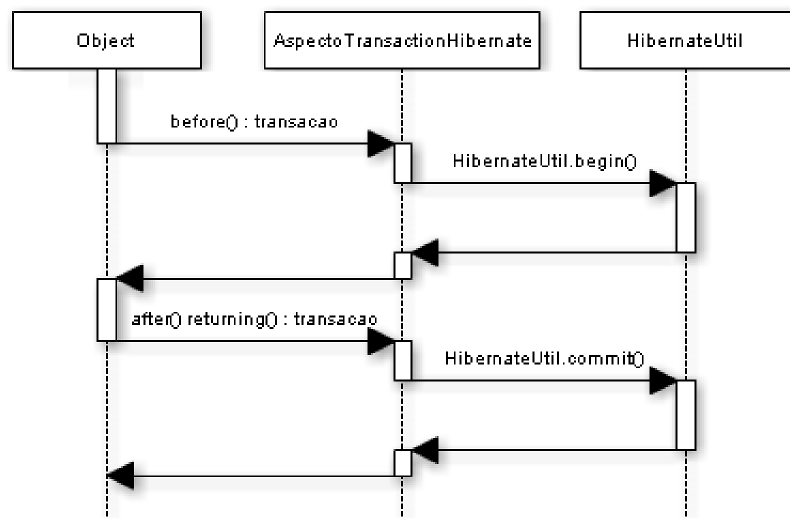

Figura 16 – Diagrama de Sequência: *AspectoTransactionHibernate* finalizado com sucesso

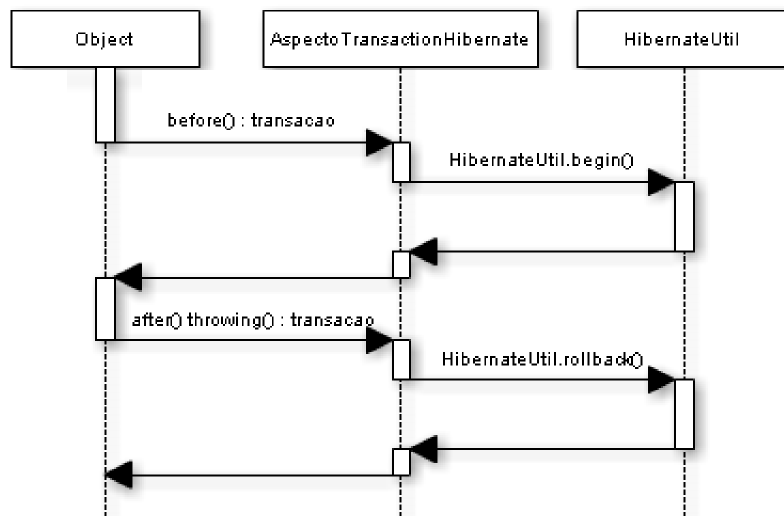

Figura 17 – Diagrama de Sequência: *AspectoTransactionHibernate* finalizado com falha

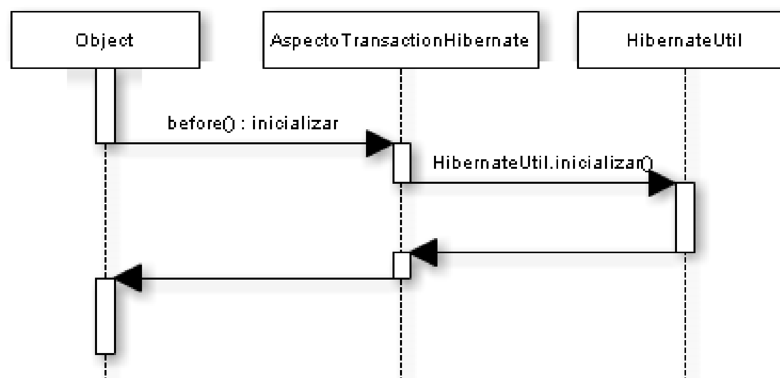

Figura 18 – Diagrama de Sequência: *AspectoTransactionHibernate* inicialização do *Hibernate*

O que controlava as ações de sucesso e erro do sistema era o bloco *try/catch*, deste modo, pode-se usar os adendos *after returning* e *after throwing* para indicar os estados da transação e o *before* para abrir uma transação, conforme o Código 20:

Código 20 – Aspecto de Controle para Manipulação de Transação do Hibernate

```
public aspect AspectoTransactionHibernate {

   pointcut transactionInicializar() : initialization (public telas.Login.new())

   pointcut transaction() : call (void telas.CadastrarAluno.botaoPesquisar())
      || call (void telas.CadastrarAluno.botaoSalvar())
      || call (void telas.CadastrarAluno.botaoExcluir())
      || call (void telas.CadastrarFuncionario.botaoPesquisar())
      || call (void telas.CadastrarFuncionario.botaoSalvar())
      || call (void telas.CadastrarFuncionario.botaoExcluir())
      || call (void telas.CadastrarFuncionario.carregarCombobox())
      || call (void telas.CadastrarLivros.criarJanela(..))
      || call (void telas.CadastrarLivros.desenharPainelExemplares())
      || call (void telas.CadastrarLivros.botaoSalvar())
      || call (void telas.CadastrarLivros.botaoAdicionar())
      || call (void telas.CadastrarLivros.botaoDeletar())
      || call (void telas.CadastrarTipoLivro.botaoSalvar())
      || call (void telas.CadastrarTipoLivro.botaoExcluir())
      || call (void telas.CadastrarTipoLivro.botaoPesquisa())
      || call (void telas.CadastrarTipoUsuario.botaoSalvar())
      || call (void telas.CadastrarTipoUsuario.botaoPesquisar())
      || call (void telas.CadastrarTipoUsuario.botaoRemover())
      || call (void telas.CadastrarTipoUsuario.botaoAdicionar())
      || call (void telas.CadastrarTipoUsuario.cliqueTabelaTipoUsuario())
      || call (void telas.Emprestimos.botaoSalvar())
      || call (void telas.Emprestimos.botaoAdicionar())
      || call (void telas.Emprestimos.botaoDeletar())
      || call (void telas.Emprestimos.carregarAlunos())
      || call (void telas.Emprestimos.carregarFuncionario())
```

```
        || call (void telas.Login.buscarUsuario())
        || call (void telas.Logs.botaoPesquisar())
        || call (void telas.PesquisarEmprestimo.botaoPesquisar())
        || call (void telas.PesquisarLivros.cliqueTabelaLivro())
        || call (void telas.PesquisarLivros.botaoPesquisar())
        || call (void telas.Principal.chamarTipoLivro())

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

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

        after() throwing() : transaction(){
                HibernateUtil.rollback();
        }

        before() : transactionInicializar(){
                HibernateUtil.inicializar();
        }
}
```

Assim, o aspecto *AspectoTransactionHibernate* mantém a integridade do banco de dados pois além de controlar a abertura das transações que comunica o banco de dados com a aplicação, ainda certifica que somente dados completos serão gravados no banco.

Note que estes aspectos específicos devem ser tratados como códigos *plug-and-play*, ou seja, códigos que podem ser acoplados na aplicação sem qualquer adaptação da mesma, conforme abordado no código 21:

Código 21 – Código Fonte do botão Salvar após aplicação do conceito de aspecto

```
// BOTÃO SALVAR
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);
```

Para demonstrar o conceito de reflexão da POA, foi implementado o aspecto *AspectoLog* que captura as chamadas dos métodos *set* das classes *Exemplar* e *DetalheEmprestimo* e salva estas ações na tabela *Log* no banco de dados. O

Diagrama de Sequência da Figura 19, exibe o comportamento do aspecto *AspectoLog* após a captura do ponto de junção *registros*. O *AspectoLog* interage com o banco de dados através de uma classe *LogDao*.

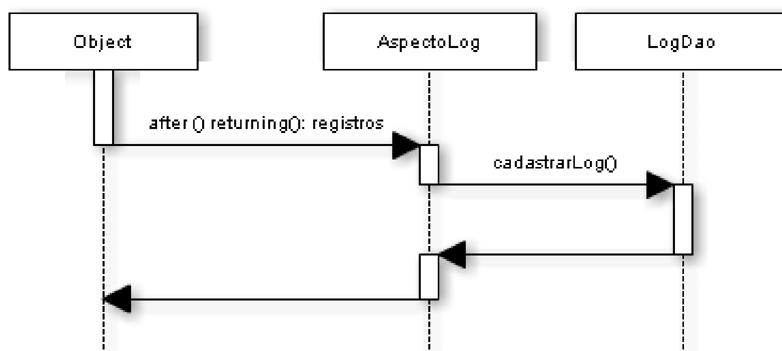

Figura 19 – Diagrama de Sequência: *AspectoLog*

As ações são tratadas a partir dos objetos especiais de reflexão, estudados na Sessão 4.6. O Código 22 mostra como foi feito o tratamento destas informações utilizando o conceito de reflexão.

Código 22 – Adendo *after() returning()* do Aspecto de Registro de Ações

```
after() returning() : registros (){

   dataAtual();
   Log registroLog = new Log();
   registroLog.setTabela(thisJoinPoint.getTarget().getClass().getSimpleName());
   registroLog.setCampo(thisJoinPoint.getSignature().getName());
   registroLog.setValorNovo(thisJoinPoint.getArgs()[0].toString());
   registroLog.setData(data);

   cadastrarLog(registroLog);

}
```

Como explicado nas sessões anteriores, a regra de negócio foi dividido em três aspectos: *AspectoDevolucao*, *AspectoEmprestimo*, *AspectoStatusAluno*. O aspecto *AspectoDevolucao* possui dois adendos: um adendo *around* e um adendo *after returning*. O adendo *around* possui o cálculo de multas por atraso dos livros; o

adendo *after returning* altera o status dos exemplares para *"Disponível"* e exibe a tela de impressão da fatura, conforme o diagrama de sequência da Figura 20.

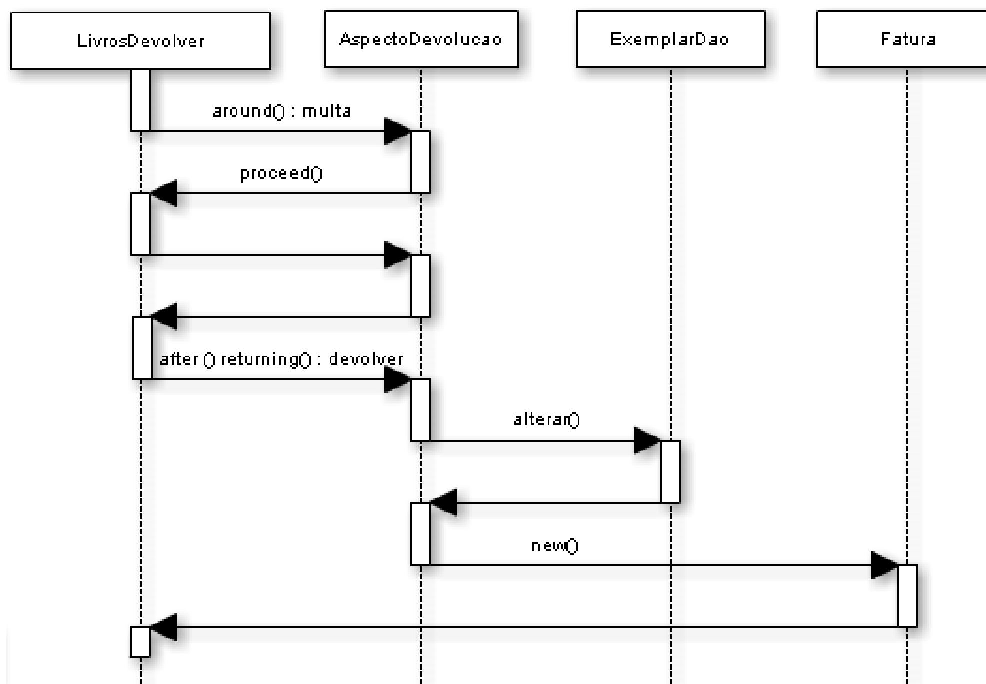

Figura 20 – Diagrama de Sequência: *AspectoDevolucao*

O aspecto *AspectoEmprestimo* é aplicado a classe *telas.Emprestimo* e possui dois adendos, conforme pode ser observado no diagrama de sequência da Figura 21. Neste diagrama pode ser observado que o primeiro adendo (*before*) possui a finalidade de verificar a disponibilidade o exemplar, e que o segundo adendo (*after returning*) interage com duas classes do pacote *dao*, pois será efetuada a alteração do status do exemplar para *"Retirado"* e a alteração do prazo de entrega do livro.

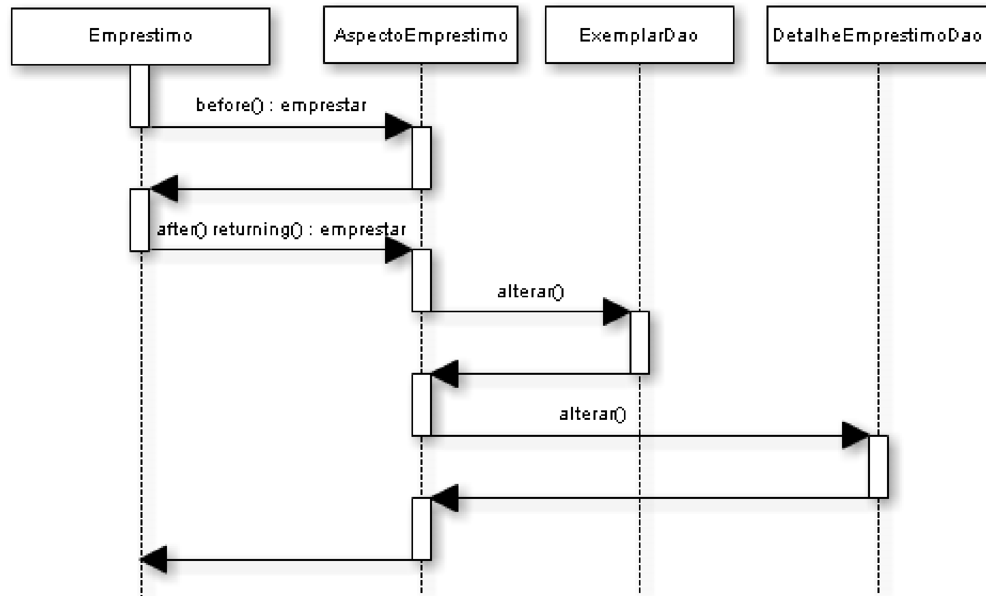

Figura 21 – Diagrama de Sequência: *AspectoEmprestimo*

Uma regra de negócio que é utilizada para diversas classes também pode ser agrupada em um único aspecto, como é o caso do aspecto *AspectoStatusAluno*, que controla o atributo status do objeto *Aluno* conforme padrões pré-determinados dentro do aspecto. Para representar o comportamento deste aspecto, foi desenvolvido o Diagrama de Sequência da Figura 22. Neste Diagrama, nota-se que o ponto de junção pode ser acionado a partir de qualquer classe (*Object*) e possui um processo de checagem do status atual do aluno, através do método *pesquisarAluno* da classe *DetalheEmprestimoDao*. Após esta checagem, inicia-se a alteração do status do aluno, de acordo com parâmetros enviados ao método *alterar* da classe *AlunoDao*.

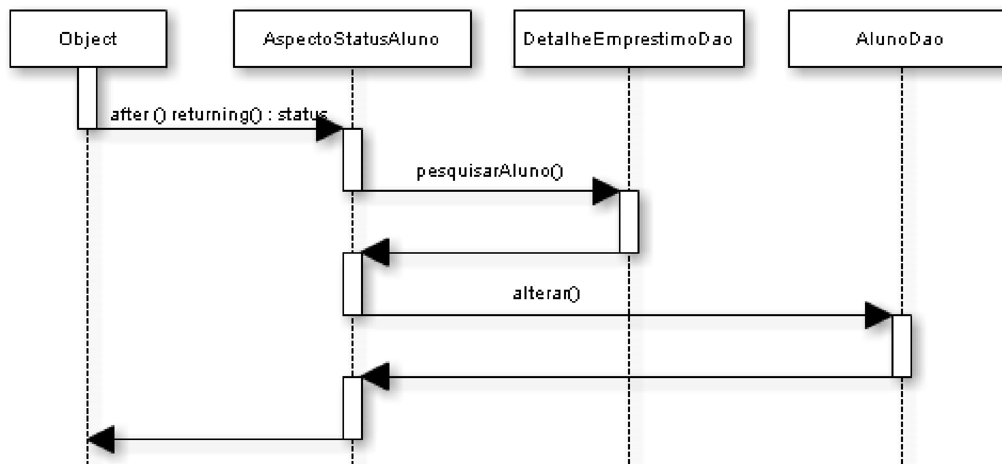

Figura 22 – Diagrama de Sequência: *AspectoStatusAluno*

Deste modo, pode-se verificar que um aspecto garante a modularização não apenas de regras sistêmicas, mas também de regras de negócio. Assim, a manutenção de qualquer alteração organizacional do sistema é simples, pois os códigos estão centralizados.

6 CONCLUSÃO

A programação orientada a aspectos facilita o ciclo de desenvolvimento de software, pois é caracterizada pela implementação de código centralizado e organizado que permite a completa separação de interesses funcionais de interesses ortogonais. Com um código mais centralizado, a estrutura do sistema fica mais clara e compreensível.

Um código espalhado e emaranhado resulta em redução da qualidade do software, além de baixa produtividade, dificuldade de manutenção e códigos extensos e complexos. A separação de interesses, incentivado pela POA, corrige estes problemas de forma simples, além de ampliar o foco dos profissionais para requisitos mais específicos dos clientes. A POO não consegue sanar tais problemas, pois todos os métodos criados pela OO necessitam de um comando que chame a execução destes métodos, continuando a manter rastros no código principal.

Ainda não existem modelos específicos para modelar o paradigma de Programação Orientada a Aspectos. A análise destes sistemas deve ser focada nos interesses da aplicação e do cliente e na organização do código. Sem estes modelos específicos, um projeto grande orientado a aspecto pode possuir problemas para ser gerenciado. Atualmente, a UML é adaptada para tentar sanar estes problemas.

A manutenção e o reuso de código de um aspecto é flexível, porque o mesmo concentra os códigos em apenas uma unidade. Se houver a necessidade de inclusão de um interesse sistêmico depois que o sistema foi desenvolvido, a programação orientada a aspecto consegue implantar este interesse sem alterar uma linha de código da aplicação principal.

Na conversão de um programa que é escrito inicialmente em outro paradigma para POA, necessita-se de uma análise que cruze as informações da documentação do software com o código fonte. Porém, se o programador necessitar apenas incluir uma funcionalidade a um projeto em outra linguagem, o aspecto pode ser escrito

separadamente e combinado com o código principal, pois o *AspectJ* possui uma alta interação com o código Java.

Na programação orientada a objetos, o programador necessita de algum artifício para controlar alguns fluxos de código enquanto que na POA também permite que o código não rompa nenhuma integridade do sistema, pois o código é implementado de forma compacta e natural, além de possuir uma melhor visualização de quais pontos os bloco de códigos serão aplicados, como por exemplo, na implantação de uma transação, onde o programador consegue evitar os conflitos entre aberturas e fechamentos das mesmas. Caso este aspecto que controle transações fosse o aspecto *AspectoTransactionHibernate*, que foi implementado no estudo de caso deste trabalho, necessitasse ser convertido para OO e incorporado ao código, o programa aumentaria bruscamente, pois este aspecto possui três linhas de códigos separadas em três diferentes tempos de execução utilizadas em trinta e seis pontos diferentes no sistema, ou seja, seriam cento e oito linhas de códigos espalhadas no código, replicando o código e correndo o risco de entrarem em conflito.

Concluindo, a utilização de um paradigma como o POA no desenvolvimento de um sistema é viável, pois melhora o processo de software, diminuem os custos de programação e manutenção, melhora a legibilidade do código, organiza o aplicativo, e diminui o tempo de desenvolvimento de um projeto, além de permitir que os comportamentos sejam modificados, adaptados, removidos ou criados para prover a evolução do sistema.

6.1 TRABALHOS FUTUROS

Com base na pesquisa desenvolvida, várias vertentes para futuros trabalhos podem ser identificadas. Como possíveis trabalhos futuros, pode-se apontar:

- Pesquisa para desenvolvimento de padrões para modelagem, análise e programação de Interesses Sistêmicos utilizando o paradigma de Programação Orientada a Aspecto e padrões UML;

- formalização da implementação da camada de Persistência como interesse sistêmico com a utilização de *AspectJ*, *Hibernate* e *Annotation*;

- pesquisar sobre a integração do Paradigma de Programação Orientada a Aspecto no desenvolvimento da Tecnologia Adaptativa;

- outro trabalho futuro relevante é desenvolver métricas e metodologias de conversão de projetos Orientados a Objetos para Orientados a Aspectos.

6.2 RESULTADOS

Este trabalho rendeu a publicação de três artigos:

- Utilização do Paradigma de Programação Orientado a Aspecto na Otimização do Desenvolvimento de Softwares. CASACHI, Rafael Alessandro; CAMOLESI, Almir Rogério. IV Fórum Científico – FEMA (Fundação Educacional do Município de Assis), Assis.

- Aplicação dos Conceitos de Programação Orientada a Aspecto no Desenvolvimento de Sistemas de Informação. CASACHI, Rafael Alessandro; CAMOLESI, Almir Rogério. 14º Encontro de Atividades Científicas UNOPAR, Londrina.

- Uso de Programação Orientada a Aspecto no Desenvolvimento de Aplicações que utilizam conceitos de Tecnologia Adaptativa. CASACHI, Rafael Alessandro; CAMOLESI, Almir Rogério. WTA 2012 (Workshop de Tecnologia Adaptativa), USP, São Paulo.

REFERÊNCIAS

BODKIN, Ron; LADDAD, Ramnivas. **Zen and the art of Aspect-Oriented Programming**. Linux Magazine, April, 2004.

GOETTEN, Vicente J.; WINCK, Diogo V. **AspectJ – Programação Orientada a Aspectos com Java**. São Paulo: Novatec Editora, 2006.

GOETTEN, Vicente J.; WINCK, Diogo V. **AspectJ em 20 minutos**. JavaFree. Disponível em: <“javafree.uol.com.br/artigo/871488/AspectJ-em-20-minutos.html”>. Acessado em: 06 abr. 2011

HEILEMAN, Gregory L. **Data Structures, Algorithms, and Object-Oriented Programming**. Singapore: McGraw-Hill, 1996.

HORSTMANN, Cay. **Conceitos de Computação com o Essencial de C++**. Porto Alegre: Bookman, 2003.

KICZALES, Gregor; LAMPING, John; MENDHEKAR, Anurag; MAEDA, Chris; LOPES, Cristina Videira; LOINGTIER, Jean-Marc. **Aspect-Oriented Programming**. In: European Conference on Object-Oriented Programming (ECOOP), 06,1997. Finlândia. Anais Springer-Verlag LNCS 1241, 06, 1997

KICZALES, Gregor; MEZINI, Mira. **Separation of Concerns with Procedures, Annotations, Advices and Pointcuts**. Vancouver, Canada: University of British Columbia

KRUPA, Artur. **Analyse Aspect-Oriented Software Approach and Its Application**. Athabaska, Alberta, Canadá: Athabaska University, 2010.

LOUREIRO, João Manuel B. P.; COSTA, João Pedro C. S. G.; FONSECA, Rossana Mendes S. B.; LOUREIRO, Virgílio A. N. **Programação Orientada a Aspecto**. Porto, Portugal: FEUP Universidade do Porto, <ANO>

ORACLE. **Object-Oriented Programming Concepts**. The Java Tutorials, 2011. Disponível em: <”download.oracle.com/javase/tutorial/java/concepts/index.html”>. Acessado em: 14 jun. 2011.

SAFONOV, Vladimir O. **Using Aspect-Oriented Programming for Trustwothy Software Development**. New Jersey: Wiley-Interscience, 2008.

THE ASPECT TEAM. **The AspectJ Programming Guide**. Xerox Corporation, Palo Alto Research Center (PARC), Palo Alto, CA, Estados Unidos. Disponível em: <”www.eclipse.org/aspectJ”>. Acessado em: 06 abr. 2011.

UBAYASHI, Naoyasu; MASUHARA, Hidehiko; MORIYAMA, Genki; TAMAI, Tetsuo. **A Parameterized Interpreter for Modeling Different AOP Machanisms**. Fukuoka, Japão: Kyushu Institute of Technology, 2005.