**FUNDAÇÃO EDUCACIONAL JAYME DE ALTAVILA - FEJAL**

**CENTRO DE ESTUDOS SUPERIORES DE MACEIÓ - CESMAC**

**FACULDADE DE CIÊNCIAS EXATAS E TECNOLÓGICAS – FACET**

Daniel Künzler Benevides de Mendonça

Rafael Oliveira Gaia Duarte

**ASPECTJ:**

Desenvolvendo uma aplicação orientada a aspecto utilizando Java.

MACEIÓ-AL

2008

**DANIEL KÜNZLER BENEVIDES DE MENDONÇA**

**RAFAEL OLIVEIRA GAIA DUARTE**

**ASPECTJ:**

Desenvolvendo uma aplicação orientada a aspecto utilizando Java.

Trabalho final para disciplina de TFG para obtenção da avaliação final, lecionada pelo professor Alexandre Paes, no 8º período do curso de Análise de Sistemas do turno matutino, sob a orientação do professor Mozart de Melo.

Orientador: Esp. Mozart de Melo

MACEIÓ-AL

2008

**DANIEL KÜNZLER BENEVIDES DE MENDONÇA**

**RAFAEL OLIVEIRA GAIA DUARTE**

**ASPECTJ:**

Desenvolvendo uma aplicação orientada a aspecto utilizando Java.

Monografia aprovada como requisito parcial para obtenção do Título de Bacharel em Análise de Sistemas - Administração da Faculdade de Ciências Exatas e Tecnológicas - FACET, do Centro de Estudos Superiores Maceió, pela seguinte banca examinadora:

**BANCA EXAMINADORA**

**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

**Profº. Esp . MOZART DE MELO ALVES JÚNIOR**

**Orientador**

**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

**Examinador (a)**

**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

**Examinador (b)**

**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

**Profº Msc. Alexandre Paes**

**Avaliador Metodológico**

**DEDICATÓRIA**

À Deus, pois sem ele, nada seria possível.

Aos meus pais Beraldo e Delane, pelo esforço.

A minha namorada Joana, pela compreensão.

A minha avó Cecinha, por tudo que me ofereceu.

Daniel Künzler Benevides de Mendonça

À Deus, por me dar saúde e força.  
 Aos meus pais Fernando e Rosa, pela dedicação e esforço.   
Aos meus irmãos Bruno e Fernanda, por serem exemplos.  
A minha namorada Bruna, pela compreensão e paciência.  
**Rafael Oliveira Gaia Duarte**

**AGRADECIMENTO**

Aos meus pais Beraldo Braga e Delane Künzler, pessoas que sempre foram exemplos de coragem, amor, determinação e perseverança.

 Aos meus tios Lourdes Braga e Ivanilton Silva, exemplos de pessoas com uma forma toda especial de ser e incentivar, mesmo sem estarem sempre presentes.

A toda minha família, que representam, para mim, a união nos momentos importantes.

A minha namorada Joana Vieira pessoa importante que cerca minha vida.

Ao meu amigo Rafael, pessoa que participou comigo em vários momentos importantes que superamos com determinação.

Ao Prof. Mozart Junior, mais que um professor, um amigo com quem interagi todo esse ano e que participou das lutas que me fizeram cada vez mais experiente e amadurecido.

Ao Prof. Ricardo Rubens, mais que um professor, um homem de valor, com grande senso de amizade.

Daniel Künzler Benevides de Mendonça

Aos meus pais Fernando e Rosa, que para mim são exemplos de pessoas e que sem eles eu não seria quem sou.

A minha namorada Bruna, que mesmo nos momentos que estive ausente soube entender.

Ao meu parceiro e amigo Daniel Künzler Benevides de Mendonça, por vários dias de estudo e por realizar este trabalho comigo.

Ao orientador, Prof. Mozart de Melo Alves Junior por orientar e mostrar que caminho seguir para a realização deste trabalho e por ser um grande amigo em todos os momentos.

Ao Prof. Ricardo Rubens, que contribuiu bastante com sua inteligência e sabedoria.

**Rafael Oliveira Gaia Duarte**

**RESUMO**

Este trabalho tem como objetivo detalhar as vantagens e o ganho que a programação orientada a aspecto (POA) possibilitou no desenvolvimento de sistemas completos em orientação a objeto (OO), além de demonstrar o que é e como funciona a programação orientada a aspecto. A Programação Orientada a Aspecto é uma evolução da Programação Orientada a Objeto, que por sua vez, foi um melhoramento da Modular e da Estruturada. Quando se programa em OO, e implementa interesses transversais, há o surgimento de códigos espalhados ou emaranhados, quebrando assim o seu paradigma, entretanto na POA este problema é resolvido com a separação estes interesses transversais ou ortogonais em aspecto. A Programação Orientada a Aspecto é divida em linguagem de componente e linguagem de aspecto. O AspectJ é uma linguagem de aspecto que tem como sua linguagem de componente o Java, que será utilizada para desenvolver uma aplicação.

**Palavras-chaves:** Programação Orientada a Aspecto, POA, AspectJ, Programação Orientada a Objeto, POO, JAVA.

**ABSTRACT**

The purpose of this report is to highlight the advantages of the Aspect Oriented Programming (AOP), which made possible the development of complete system using the Object Oriented Programming (OOP), besides demonstrating what it is and how the AOP works.  The AOP is an evolution of the OOP, which in turn was an improvement of the modular as well as the structured.  When programming in OO, and cross interests are implemented, there will be scattered codes, thus breaking the paradigm.  In AOP, however, this problem is solved by separating these cross interests.  The AOP is divided in component and aspect languages.  The aspect one is a language that has the Java language as its component, the one that will be utilized to develop an application.

**Key-words:** Aspect Oriented Programming, AOP, AspectJ, Object Oriented Programming, OOP, JAVA.

**LISTA DE FIGURAS**

|  |  |  |
| --- | --- | --- |
| Figura 1 | Fluxograma Seqüência............................................................................ | 18 |
| Figura 2 | Fluxograma Seleção................................................................................ | 19 |
| Figura 3 | Fluxograma Iteração................................................................................ | 20 |
| Figura 4 | Exemplo de Módulos.............................................................................. | 22 |
| Figura 5 | Procedimento........................................................................................... | 23 |
| Figura 6 | Funções.................................................................................................... | 24 |
| Figura 7 | Conceito de Objeto.................................................................................. | 26 |
| Figura 8 | Classes e Objetos..................................................................................... | 28 |
| Figura 9 | Exemplo de herança................................................................................ | 31 |
| Figura 10 | Exemplo de Polimorfismo....................................................................... | 33 |
| Figura11 | Ocorrência de interesses transversais para atualização da imagem........ | 37 |
| Figura 12 | Implementação do parser XML no Tomcat............................................ | 38 |
| Figura 13 | Implementação da funcionalidade de auditoria (log) no Tomcat............ | 39 |
| Figura 14 | Separação de interesses........................................................................... | 45 |
| Figura 15 | Combinação entre classe Produto e os demais interesses....................... | 45 |
| Figura 16 | Combinação aspectual............................................................................. | 47 |
| Figura 17 | Composição de um sistema orientado a aspecto..................................... | 48 |
| Figura 18 | Pontos de junção encontrados em uma classe......................................... | 55 |
| Figura 19 | Esquema de chamada e execução de métodos, funções ou rotinas......... | 55 |
| Figura 20 | Estrutura de um aspecto.......................................................................... | 70 |
| Figura 21 | Diagrama de caso de uso......................................................................... | 72 |
| Figura 22 | Diagrama de classe.................................................................................. | 73 |
| Figura 23 | Diagrama de seqüência........................................................................... | 73 |
| Figura 24 | UML do aspecto.................................................................................... | 76 |

**LISTA DE QUADROS**

|  |  |  |
| --- | --- | --- |
| Quadro 1 - | Comparativo entre código espalhado e código emaranhado............... | 43 |
| Quadro 2 - | Tipos de ponto de junção..................................................................... | 49 |
| Quadro 3 - | Listagem dos designadores em aspectj................................................ | 50 |
| Quadro 4 - | Declaração de pontos de atuação de Chamada de Métodos e Construtores........................................................................................ | 60 |
| Quadro 5 - | Declaração de pontos de atuação de Execução de Métodos e Construtores........................................................................................ | 60 |
| Quadro 6 - | Declaração de pontos de atuação de Métodos de Acesso a Atributos............................................................................................. | 61 |
| Quadro 7 - | Tipos de adendo disponíveis no AspectJ............................................ | 65 |
| Quadro 8 - | Descrição dos atores............................................................................ | 71 |
| Quadro 9 - | Descrição dos casos de uso.................................................................. | 71 |

**LISTA DE LISTAGEM**

|  |  |  |
| --- | --- | --- |
| Listagem 1 - | Classe Log......................................................................................... | 74 |
| Listagem 2 - | Classe ClienteDAO com interesses sistêmicos................................. | 75 |
| Listagem 3 - | Pointcut auditoria1 do Aspecto Logger............................................. | 77 |
| Listagem 4 - | Pointcut auditoria2 do Aspecto Logger............................................. | 77 |
| Listagem 5 - | Classe ClienteDAO sem os códigos espalhados e emaranhados...... | 78 |

**LISTA DE ABREVIATURAS E SIGLAS**

|  |  |
| --- | --- |
| OO - | Orientação a Objeto |
| POO - | Programação Orientada a Objeto |
| OA - | Orientação a Aspecto |
| POA - | Programação Orientada a Aspecto |
| IDE - | Integrated Development Environment |

**SUMÁRIO**

**1 INTRODUÇÃO** .........................................................................................................14

**2 EVOLUÇÃO DA PROGRAMAÇÃO**.....................................................................17

2.1 Visão Geral................................................................................................................17

2.2 Paradigma Estruturada...............................................................................................17

2.2.1 Técnica Utilizada....................................................................................................18

2.2.2 Vantagens...............................................................................................................21

2.2.3 Desvantagens..........................................................................................................21

2.3 Paradigma Modular...................................................................................................22

2.3.1 Técnica Utilizada....................................................................................................23

2.3.2 Vantagens...............................................................................................................24

2.3.3 Desvantagens..........................................................................................................25

2.4 Paradigma Orientada a Objeto...................................................................................25

2.4.1 Técnica Utilizada....................................................................................................27

2.4.1.1 Abstração.............................................................................................................27

2.4.1.2 Encapsulamento...................................................................................................29

2.4.1.3 Herança................................................................................................................30

2.4.1.4 Polimorfismo.......................................................................................................33

2.4.2 Vantagens...............................................................................................................34

**3 LIMITAÇÃO DA ORIENTAÇÃO A OBJETO E PROGRAMAÇÃO ORIENTADA A ASPECTO** ........................................................................................35

3.1 Visão Geral................................................................................................................35

3.2 Limitação da Orientação a Objeto.............................................................................36

3.2.1 Interesses................................................................................................................37

3.2.2 Códigos Espalhados (Scattering Code)..................................................................39

3.2.3 Códigos Emaranhados (Tangled Code).................................................................41

3.3 Paradigma Orientado a Aspecto................................................................................43

3.3.1 Componentes da Orientação a Aspecto..................................................................46

3.3.2 Conceitos Fundamentais.........................................................................................48

**4 ASPECTJ E APLICAÇÃO ORIENTADA A ASPECTO**......................................52

4.1 AspectJ ......................................................................................................................52

4.2 Divisão do AspectJ ...................................................................................................53

4.3 Elementos do AspectJ ...............................................................................................54

4.3.1 Ponto de Junção (Join Point)..................................................................................54

4.3.2 Ponto de Atuação (Pontcuts)..................................................................................58

4.3.3 Adendo (Advice)....................................................................................................64

4.3.4 Inserção (Introduction)...........................................................................................68

4.3.5 Aspecto (Aspect)....................................................................................................68

4.4 Aplicação Orientada a Aspecto.................................................................................70

4.4.1 Análise do sistema de cadastro de cliente..............................................................71

4.4.1.1 Diagramas do sistema..........................................................................................72

4.4.2 Códigos do sistema.................................................................................................74

4.4.3 Sistema Orientado a Aspecto..................................................................................76

**5** **CONCLUSÃO** ............................................................................................................79

**6 REFERÊNCIAS**..........................................................................................................82

**7 APÊNDICE**.................................................................................................................84

**INTRODUÇÃO**

Este trabalho se propõe a realizar um estudo da evolução dos paradigmas da programação, desde a estruturada até a orientada a aspecto (POA) identificando o que a orientação a aspecto pode contribuir na construção de softwares de qualidade e desenvolver uma aplicação utilizando-se da ferramenta ASPECTJ (aspecto para Java).

O objetivo geral deste trabalho é detalhar as vantagens e o ganho que a programação orientada a aspecto (POA) possibilitou no desenvolvimento de sistemas completos em OO. Além de demonstrar o que é e como funciona a programação orientada a aspecto (POA).

Segundo Winck e Goetten a Programação Orientação a Objeto (POO) era a grande esperança para acabar com a crise que existia na produção mundial de software, com a evolução da programação, encontramos na OO um conjunto de abstrações (classes, objeto, interface, métodos, etc.) que facilitariam a idéia do reuso, dizendo-se que resolveria problemas anteriores, possibilita um grau maior de abstração assim comparada às outras técnicas: estruturada que utilizava procedures – estruturas pré-definidas, e modular que dividia em módulos combinando procedimentos e dados.

Os paradigmas orientados a objeto são os mais utilizados na programação atualmente, mas são incapazes de gerir separações de interesses comuns que devem ser implementados de forma que, a técnica utilizada não seja quebrada. Na ocorrência de implementar interesses adicionais(persistência de dados, autenticação) dentro de determinadas classes, há uma tentativa de impor a junção de um interesse transversal a outro interesse, resultando assim códigos espalhados(scattering) e emaranhados(tangled) por todo o sistema, quebrando conceitos próprios da Orientação a Objeto.

Há alguns problemas encontrados na falha de OO ao modularizar os interesses sistêmicos como, por exemplo, na duplicação de código que é necessário um mesmo tratamento em classes diferentes, uma maior dificuldade na compreensão e no reuso dos códigos. Por fim esses interesses estando em locais bem definidos e separados pode-se melhorar toda essa complexidade, mas existem algumas limitações, porque as abstrações deste paradigma não conseguem lidar com comportamentos que se espalham.

Contudo, a intenção de mostrar mecanismos capazes de suprir essa falha, foi desenvolvida uma nova técnica de programação. A Orientação a Aspecto (OA) estende a metodologia da Orientação a Objeto e outras, inserindo novas abstrações com conceitos de interesses, possibilitando suprir a carência destas, gerando perspectivas de separação avançada. Retrata uma técnica emergente que desenvolve um novo paradigma de engenharia de software fornecendo um caminho completo, após uma década de importante pesquisa foi vista com ampla relevância, apesar de não ser o paradigma ideal, possui uma ferramenta que a suporte.

A Orientação a Aspecto (OA) é uma nova metodologia de programação que veio complementar o atual paradigma da orientação a objetos melhorando os recursos de persistência, auditoria (log) e tratamento de exceções diminuindo os códigos espalhados e extinguindo os códigos emaranhados. Seu principal foco consiste em dividir o código referente ao domínio do sistema dos códigos referentes aos interesses transversais aumentando assim a modularidade, gerando um código de uma forma bem definida e centralizada, possibilitando assim um nível maior de abstração no desenvolvimento de software.

A programação orientada a aspecto é dividida em linguagem de componentes, que é o programa básico escrito pelo desenvolvedor, e linguagem de aspecto. A escolha da ferramenta AspectJ foi devido que ela é uma linguagem de aspecto que suporta componente JAVA, que é bastante familiarizada por diversos desenvolvedores. Essas duas abordagens são gratuitas, e por isso são mais acessíveis para o programador. A perspectiva de se consagrar é relativamente grande, bastando apenas que a comunidade desenvolvedora e a indústria de software, e principalmente, nas faculdades e universidades começarem a desenvolver projetos e até mesmo comercializar.

A programação Orientada a Aspecto foi criada para resolver a implementação destes interesses transversais para os sistemas modelados com orientação a objetos. Usando-a é possível separar estes interesses em unidades chamadas aspectos. Monteiro e Piveta (2003, p 315) descrevem: “Onde o principal objetivo é possibilitar a separação clara entre componentes e aspectos e aumentar a modularidade, facilitando então a compreensão, o desenvolvimento e a manutenção do sistema.”

Existem alguns elementos que são encontrados na programação a aspecto, que são linguagem de componentes, linguagem de aspectos, combinador de aspecto, programa de componentes e programa de aspecto.

A ferramenta AspectJ contêm alguns elementos, como o ponto de junção (join point), ponto de atuação (pointcuts), adendo (advice), inserção (introduction), declaração em tempo de execução (compile-time declaration) e aspecto (aspect), que são utilizados para especificar regras de interesses dinâmicos e estáticos. “O AspectJ utiliza extensões para a linguagem de programação Java, especificando regras de combinações para interesses estáticos e dinâmicos”(WINCK; GOETTEN, 2006, p55).

O primeiro capítulo trata-se do estudo da evolução dos paradigmas da programação, passado pela programação estruturada, depois a modular e por fim a programação orientada a objeto. No segundo capítulo serão mostradas as limitações existentes na programação orientada a objeto e serão analisados os paradigmas emergentes da programação orientada a aspecto. Já o terceiro capítulo, irá apresentar a ferramenta AspectJ, que será explorada e uma aplicação irá ser desenvolvida utilizando esta ferramenta.

**2 EVOLUÇÃO DA PROGRAMAÇÃO**

Nesse capitulo será abordado à visão geral, objetivos, vantagens como também as desvantagens da evolução da programação, iniciando-se pela estruturada, modular e por fim a objeto, demonstrando técnicas, formas e linhas de códigos para uma melhor visualização de suas características.

2.1 Visão Geral

As primeiras técnicas de programação percorreram um longo percurso desde a aparição do computador no mundo, onde vieram realizar o trabalho e resolver problemas com conjuntos de instruções entendíveis pelo PC, ou seja, tinham que ter inicio e fim ao inserir um dado e solucionar uma saída.

Ao surgimento na década de 70 iniciados por Dijkstra, os princípios da programação estruturada tiveram a necessidade de uma linguagem de programação com idéias de implementação mais simples, porem com uma estrutura de dados mais poderosa, assim foi chamada a linguagem Pascal em homenagem a Blaise Pascal, filósofo e matemático francês, derivada da linguagem ALGOL dos anos 60. (COAD, Peter; YOURDON, Edward; 1992)

2.2 Paradigma Estruturada

Essa programação orienta os programadores para a criação de estruturas simples, por meio de um conjunto de técnicas que aumenta a produtividade, reduz o tempo e limita as estruturas de controle, eliminando os erros, visando facilitar o entendimento, a escrita e manutenção de programas.

Contudo, faz respeito à forma do programa e do processo da codificação com um conjunto de mudanças que o programador pode seguir para produzir o código estruturado, impondo limitações sobre o uso das estruturas básicas de controle como documentação e composição.

“A programação estruturada é composta por um conjunto de técnicas que foram se evoluindo aumentando consideravelmente a produtividade do programa reduzindo o tempo de depuração e de manutenção do mesmo.” (LANGA, Sara; 2006)

As características mais notáveis de um programa estruturado são sua forma

hierárquica e o conjunto reduzido de suas estruturas básicas de controle mencionadas acima, há também algumas propriedades encontradas num programa estruturado, no qual é dividido em um conjunto de módulos, o fluxo de execução entre módulos é fácil e rápido, a construção dos módulos é padronizada e limitada as estruturas básicas de controle que será vista a seguir e por fim uma documentação.

2.2.1 Técnica Utilizada

Utiliza-se na programação estruturada mecanismos básicos compostos por blocos que se inter relacionam, são elas seqüência, seleção e iteração que cada um destes modos possui um ponto de inicio e um ponto de termino na execução.

Seqüência onde cada ação entra ao termino da outra de um modo seqüencial implementando os passos necessários para descrever um programa.

[fluxograma]

\includegraphics[scale=.5]{sequc.eps}

[pseudo-linguagem]

\begin{algorithm}{Sequencia}{}
\mbox{Tarefa} \; a \\
\mbox{Tarefa} \; b
\end{algorithm}

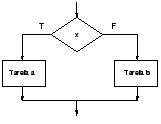
**FIGURA 1 –** Fluxograma seqüência

No exemplo anterior a “tarefa a” é executada e em seguida a “tarefa b”, que ao se analisar vê-se um processo de execução de modo seqüencial.

Seleção tem possibilidade de escolher o caminho da execução do processo com a utilização de expressões lógicas com a inserção do IF, permitindo decidir o caminho de uma forma que se a expressão lógica for verdadeira execute a "tarefa a" senão, se for falsa execute a "tarefa b".

[fluxograma]

|  |
| --- |
|  |



[pseudo-linguagem]

\begin{algorithm}{SelecaoIf}{}
\begin{IF}{x}
\mbox{Tarefa} \; a
\ELSE
\mbox{Tarefa} \; b
\end{IF} \end{algorithm}

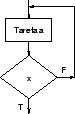
**FIGURA 2** – Fluxograma Seleção

No exemplo anterior há a verificação da expressão que implicara no processo a percorrer onde T é denominado o bloco then da construção, enquanto o outro é denominado bloco else.

Interação permite seqüências de ações que se repetem um número determinado de vezes, utilizando a expressão lógica WHILE que enquanto for falsa o bloco é repetidamente executado até então torna-se verdadeiro segue seu processo de execução.

[fluxograma]

|  |
| --- |
|  |



[pseudo-linguagem]

\begin{algorithm}{IteracaoRepeat}{}
\begin{REPEAT}
{}\mbox{Tarefa} \; a
\end{REPEAT}{x}
\end{algorithm}

**FIGURA 3 –** Fluxograma Interação

No exemplo anterior há a verificação da expressão que implicara no processo a percorrer onde a “tarefa a” avaliada, se falsa retorna F e caso não segue T seu processo de execução.

As três estruturas básicas de controle para a construção de programas estruturados : **Seqüência** - é usada para controlar a execução do programa, os comandos são executados na mesma ordem em que aparecem no código fonte, **Seleção** - é usada para testar uma condição e, então, dependendo de ser o teste verdadeiro ou falso, um dos dois conjuntos alternativos de instruções é executado e por fim **Iteração** - é usada para executar um conjunto de instruções em um número inteiro de vezes - isto é, para construir um laço (loop). Deve-se observar que existem duas formas básicas para a estrutura de iteração: DO UNTIL e WHILE (MATOS, Luiz; 2004, p. 4).

Na programação estruturada pretende diminuir a complexidade na implementação do programa em diferentes fases por meio do refinamento sucessivo, assim decompondo o problema em etapas de subprogramas chamada de desenvolvimento top-down, funciona num sentido de cima para baixo analisando-se cada uma dessas fases, obtendo um maior nível de detalhamento e um programa modularizado, podendo ser implementados separadamente em outros programas.

Na programação top-down, a ordem das etapas de projeto, codificação e teste é diferente daquela do desenvolvimento tradicional de programa. No desenvolvimento tradicional, primeiramente o programa inteiro é projetado, depois é codificado e, finalmente, testado. As três etapas são executadas separada e seqüencialmente, chamada de abordagem por fases ou abordagem tudo em um. (MATOS, Luiz; 2004, p. 9).

Princípios Básicos da Programação Top-Down

* As decisões de projeto devem ser feitas de tal forma que dividirão o problema em partes cujos componentes estarão relacionados a ele.
* Os detalhes de implantação não devem ser considerados até o processo de desenvolvimento estar avançado, mas em cada etapa devem ser exploradas alternativas.
* As etapas de refinamento devem ser simples e explícitas

2.2.2 Vantagens

Por fim algumas das principais vantagens dessa programação são:

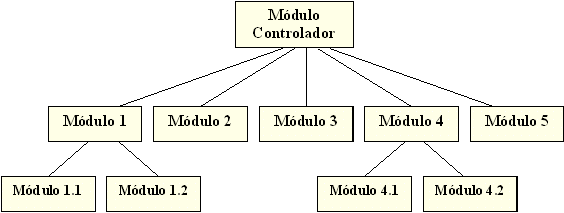
* Os programas são mais fáceis de entender, pois o principal objetivo da programação estruturada é criar um sistema de alta qualidade e baixo custo, resultando em uma programação disciplinada e organizada, reduzindo a complexidade;
* Ganho na produtividade do programador obtendo uma maior confiabilidade e legibilidade;
* Os códigos são mais documentados internamente;
* A seqüência de operações é fácil de acompanhar, já que as estruturas elementares são simples;
* Torna-se fácil testar e corrigir um programa concebido utilizando estas estruturas.

2.2.3 Desvantagens

Como os conceitos de programação modular surgindo, foi percebido algumas das desvantagens da programação estruturada, que são a limitação na reutilização dos códigos, as dificuldades na manutenção e depuração dos códigos, afetando principalmente os procedimentos, que por sua vez impossibilita a criação dos mesmos mais específicos e a principal que seria renomear os dados e cada local onde são acessados, trazendo dificuldades enormes para o programador.

2.3 Paradigma Modular

A programação modular foi criada para tentar melhorar os problemas existentes na programação estruturada. Utiliza-se da filosofia dividir para conquistar para resolver os problemas do sistema. Quando o mesmo é dividido em partes distintas, fáceis de serem entendidas, suas complexidades são gradativamente reduzidas. Como também a programação estruturada utiliza-se do dividir para conquistar, mas não amplia a filosofia de modularização acrescentando os conceitos de organização hierárquica e níveis de abstração importantes para ter controle nas inter-relações rmodulares.



**FIGURA 4** – Exemplo de Módulos

No exemplo a cima visualiza-se ao tentar que os módulos sejam, tanto quanto possível independentes uns dos outros, de maneira que pode testá-los individualmente.

2.3.1 Técnica Utilizada

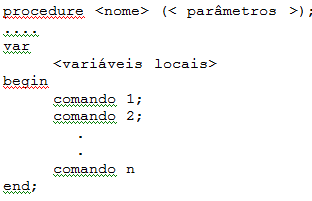
A orientação a módulos é uma técnica de programação que cria programas como um conjunto de unidades individuais que são interligadas através de uma interface comum. O programa se decompõe em vários subprogramas, assim as complexidades são divididas em pequenos problemas em módulos separados.

Para ajudar o programador na escolha de um esquema de modularização eficiente, foram desenvolvidas normas ao que se diz nos tamanho dos módulos. Segundo (Machado, 2001) A IBM, por exemplo, recomenda que se definam módulos com, no máximo, 50 linhas de código de programa ou a uma página de listagem do programa fonte, pois permite que o programador que fará uma leitura ou uma manutenção futura, compreenda-o todo código de uma só vez.

De uma maneira geral, o tamanho dos módulos deve estar entre 10 e 100 instruções, pois módulos com mais de 100 instruções são mais complexos em relação a testes e manutenções, enquanto módulos com menos de 10 instruções dividem o sistema em muitas partes, assim pondo em risco a eficácia do mesmo.

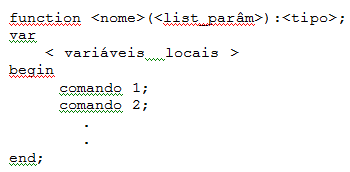
Os subprogramas não detalham sua implementação e quando um precisa do outro, faz-se a chamada pela sua interface. Existem dois tipos de subprogramas, os que retornam um valor, que são as funções e os que executam uma ação, que são os procedimentos.

Exemplo de procedimento na linguagem Pascal:



**FIGURA 5 –** Procedimento

Exemplo de função na linguagem Pascal:



**FIGURA 6 –** Função

2.3.2 Vantagens

São algumas das vantagens a seguir que permitirão a programação a módulos ser bastante utilizada, de bastante importância e possibilitar novos conceitos e abstrações.

* Os programas modulares são mais fáceis de entender, devida análise do programa ser feita através da análise dos módulos, um de cada vez;
* O teste do programa é mais simples;
* Os erros do programa são mais fáceis de serem isolados e corrigidos;
* As mudanças no programa podem ser limitadas a apenas alguns módulos, em vez de percutirem pela maior parte do código do programa;
* Os módulos do programa podem ser reutilizados como blocos de construção de

outros programas;

* O tempo de desenvolvimento do programa pode ser diminuído, porque diferentes módulos podem ser implementados por diferentes programadores.

2.3.2 Desvantagens

Existem duas falhas na programação orientada a módulos, uma delas é no momento de estender os módulos, que não é possível, esses módulos só podem ser reutilizados usando agregação de módulos ou colaboração. A outra é que torna-se limitada por falta de visibilidade, não possibilitando controles ao acesso de determinados recursos, podendo ser alterados.

2.4 Paradigma Orientada a Objeto

Com a evolução da computação bastante rápida, acabou levando a comunidade desse meioa se deparar com a “*crise do software*”, sabendo-se que a linguagem estruturada não dava mais suporte às necessidades, resultando na busca de soluções alternativas, pois a complexidade dos sistemas já tinha atingido tal ponto que nem os seus programadores conseguiam dominá-la mais.

Winck e Goetten (2006) comentam que novos usuários de sistemas eram leigos e não eram especialistas em computação, devido à popularização dos PC que a Apple estimulou. A Orientação a Objeto (OO) surgiu exatamente para facilitar a interface de sistemas controlada por essas pessoas.

Foi criada para tentar aproximar o mundo real do mundo virtual, sua idéia fundamental era tentar simular o mundo real dentro do computador. Com isso, nada mais natural do que utilizar objetos, afinal, nosso mundo é composto dele, que são conjuntos de dados bem estruturados podendo ser visíveis ou não dependendo da escolha do programador e as ações a serem tomadas no programa.

A OO é um paradigma atual que procura re-definir a programação estruturada e modular de uma maneira diferente, pois adiciona o conceito de visibilidade restringindo o acesso aos atributos e métodos, bem mais compreensivos ao raciocínio do homem, pretendendo descrever a solução através de objetos – como no mundo real, trazendo novos paradigmas como o encapsulamento, herança e polimorfismos.

Com a inserção de herança e polimorfismo na orientação modular surge a OO, apenas modificando os procedimentos por objetos e módulos por classes, trazendo benefícios como reuso de códigos e modularização. “O reuso pode ser obtido por derivação de classes e também por composição de objetos (meio disponível na programação modular)” (WINCK; GOETTEN, 2006, p18).

Para entender o conceito de objeto, costuma-se dizer que é alguma coisa que existe, que possui características e que faz algo, assim visto no mundo real como um indivíduo que possui atributos (estado) e métodos (comportamento) imaginado de tal forma um mundo no seu programa através de objetos. Já as classes podem ser definidas como uma união de objetos que possuem características e comportamentos semelhantes. Uma equivalência bastante exemplificada em relação a classe x objeto é a fôrma e o tijolo.

Segundo (DAVID, 2007) “Uma classeé uma abstração que define um tipo de objeto e o que objetos deste determinado tipo têm dentro deles (seus atributos) e também define que tipo de ações esse tipo de objeto é capaz de realizar ( **métodos** ).”

**Objeto**

Características

Comportamentos

**FIGURA 7** - Conceito de Objeto

2.4.1 Técnica Utilizada

Cada objeto tem suas próprias características, moldadas a partir de uma matriz para ser considerada uma linguagem OO, a mesma precisa implementar e suportar os quatro conceitos básicos: abstração, encapsulamento, herança e polimorfismo que são os alicerces da POO, onde uma linguagem que não suporta qualquer um desses conceitos impossibilita a mesma de ser orientada a objeto.

2.4.1.1 Abstração

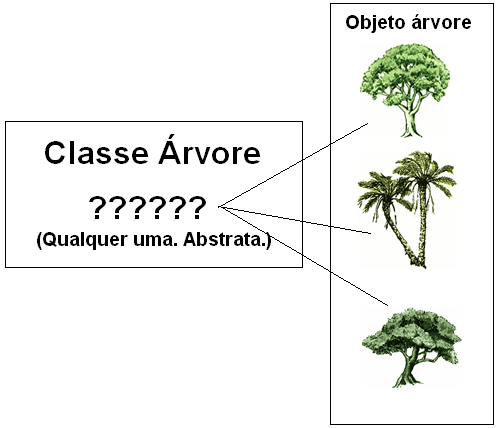
É definida como a maneira de representar cenários complexos usando termos simples. Nesse contexto a abstração refere-se à capacidade de modelar o mundo real, que assim podemos considerá-la como um mecanismo no qual restringimos o nosso universo de análise e as variáveis e constantes que compõem esse universo, desprezando os dados que não nos interessa na análise.

Podemos demonstrar o uso de abstração facilmente, um exemplo da vida real que possa ilustrar o conceito seria um carro é uma abstração de um veículo que um indivíduo pode utilizar com o objetivo de mover-se de um ponto a outro. No dia-a-dia, ninguém dirá: “Vou abrir a porta daquele veículo movido a combustível, entrarei, me sentarei, darei a partida no motor, pisarei na embreagem, engatarei a primeira marcha, acelerarei, controlarei a direção em que o carro irá se mover utilizando o volante”. Tamanha explicação não se faz necessária pois todo o conceito daquilo foi abstraído para algo que conhecemos como “carro”. Apesar de um carro ser algo bastante complexo, basta dizer “vou usar o meu carro para ir ao trabalho amanhã”, e qualquer pessoa entenderá o recado.

Além de objetos como aqueles que possuem representação visual (TextField, Button, Form, etc.), existem também objetos que são criados em muitas aplicações com o intuito de abstraírem objetos complexos da vida real, como por exemplo Pedido, Cliente, Produto, entre outros.

Nesse contexto a abstração refere-se à capacidade de modelar o mundo real, e por outro lado, podemos considerá-la como um mecanismo pelo qual restringimos o nosso universo de análise e as variáveis e constantes que compõem esse universo, desprezando os dados que não nos interessa na análise. Podemos demonstrar o uso de abstração facilmente, quando fechamos os olhos e pensamos em uma mesa; esta mesa imaginária provavelmente não vai ser igual à uma outra imaginada por outras pessoas, mas o que importa é que todos as pessoas que imaginaram uma mesa, colocaram nessa as informações que para elas são necessárias para a sua função (de ser uma mesa). Não importa se a mesa é de três pés ou quatro, ou se o tampão é de vidro, madeira ou mármore; o que importa é que a imagem que idealizamos em nossa cabeça é de uma mesa e tenha as informações necessárias para cumprir sua função. (LEITE, Mário; RAHAL, Nelson Abu, 2005).

Na figura 8 pode-se vê um exemplo de abstração na programação orientada a objeto, imaginando o objeto árvore com as suas características ( caule, folha, raiz ) e que não importa se tem dois caules ou grupamentos de folhas ou folhas únicas, o importante é que se imagine as necessidades de ter os atributos e os métodos.



**FIGURA 8** - Classes e Objetos  
**FONTE:** http://www.linhadecodigo.com.br/ArtigoImpressao.aspx?id=851

2.4.1.2 Encapsulamento

É definido como a tarefa de tornar um objeto o mais auto-suficiente possível, desta maneira, um dado que está encapsulado quando envolvido por código de forma que só é visível na classe onde foi criado. E até mesmo em linguagens consideradas não OO, como é o caso do Clipper que segundo FERREIRA e JARABECK, podem-se observar certo encapsulamento nas rotinas em que as variáveis são declaradas como local. Nesses casos tais variáveis são visíveis somente dentro dessas rotinas onde foi declaradas, o que permite ao programador certa segurança quanto aos acessos indevidos por parte de outras rotinas, o que não acontece com variáveis public, no contexto dessa linguagem.

Os conceitos de Abstração e Encapsulamento andam sempre juntos, com a abstração definimos a entidade que representa um objeto complexo, e o encapsulamento esconde detalhes do objeto, resultando na ocultação de detalhes do seu funcionamento que poderiam inibir qualquer pessoa tentando utilizá-lo.

Exemplo de Encapsulamento:

**public** **class** Carro {

**private** String cor;

**private** String tipo;

**public** Pessoa() {

}

**public** Pessoa(String cor, String tipo) {

**super**();

**this**.cor = cor;

**this**.tipo = tipo;

}

**public** String getCor() {

**return** cor;

}

**public** **void** setCor(String cor) {

**this**.cor = cor;

}

**public** String getTipo() {

**return** tipo;

}

**public** **void** setTipo(String Tipo) {

**this**.tipo = tipo;

}

**public** String toString(){

**return** tipo+" "+cor;

}

O exemplo acima mostra uma classe carro com os seus atributos cor e tipo do tipo string, o termo private demonstra que os seus atributos só são visíveis na classe onde foi criada.

2.4.1.3 Herança

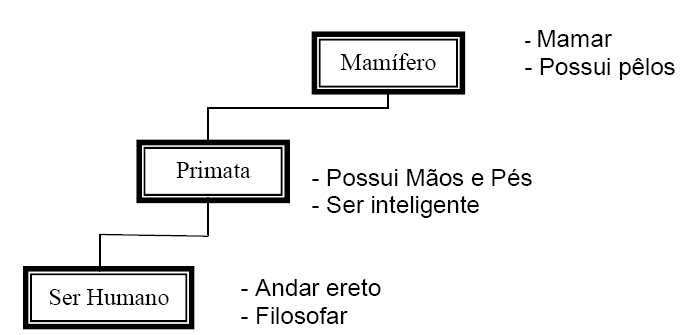
É um mecanismo que se herda ou se transmite por hereditariedade permitindo altos graus de reutilização de código, pode ser entendido como sendo um conjunto de instâncias criadas a partir de um outro conjunto de instâncias com características semelhantes, e os elementos desse subconjunto herdam todas as características do conjunto original. A idéia é fornecer um mecanismo simples (mas muito poderoso) para que se defina novas classes a partir de uma já existente.

Fernanda Farinelli (2007) diz que “Um exemplo simples de explicar a herança é a própria genética. Ou seja, um filho herda características genéticas dos pais, e por sua vez repassa essas características aos seus filhos.”. Na Orientação a Objetos a herança é um mecanismo muito inteligente de reutilização de códigos, isso porque ele é utilizado para criar uma classe através de outra classe já existente.

A classe que é criada através de outra, é chama de subclasse ou classe filho, e a classe que transmite seus atributos e métodos para criar a subclasse é chamada de superclasse ou classe pai. As subclasses herdam todos os comportamentos e as características da superclasse, como os seus atributos e métodos, podendo adicionar novas características e novos comportamentos, além de poder ter comportamentos diferentes da classe pai, isso se os métodos forem modificados.

“Na Orientação a Objeto, Herança é o mecanismo pelo qual uma classe obtém as características e métodos de outra para expandi-la ou especializá-la de alguma forma, ou seja, uma classe pode ‘herdar’ características, métodos e atributos de outra classe. Da mesma maneira uma classe transmite suas características para as outras classes, tornando aquelas que recebem suas características suas herdeiras.” (FARINELLI, 2007, p18)

Na figura a baixo, tem um exemplo simples de herança. A classe Primata é subclasse ou filha da classe Mamífero e herda os métodos Mamar e Possui pêlos, que também é superclasse ou pai da classe Ser humano, que por sua vez herda os métodos tanto da classe Primata como da classe Mamífero.



**FIGURA 9** – Exemplo de herança. **FONTE:** FARINELLI, 2007

Em Java quando se pensa em criar uma classe que herde características e comportamentos de outra classe se usa o termo extends, como se pode vê no exemplo a seguir:

**public** **class** Ferrari **extends** Carro{

**private** String modelo;

**private** String velocidade;

**public** Ferrari(String cor, String tipo, String modelo, String velocidade ) {

**super**(cor, tipo);

**this**.modelo = modelo;

**this**.velocidade = velocidade;

}

**public** String getModelo (){

**return** modelo;

}

**public** **void** setModelo(String modelo){

**this**.modelo = modelo;

}

**public** String getVelocidade (){

**return** velocidade;

}

**public** **void** setVelocidade(String velocidade){

**this**.velocidade = velocidade;

}

**public** String toString(){

**return** **super**.toString()+" "+**this**.modelo+" "+ **this**.velocidade;

}

}

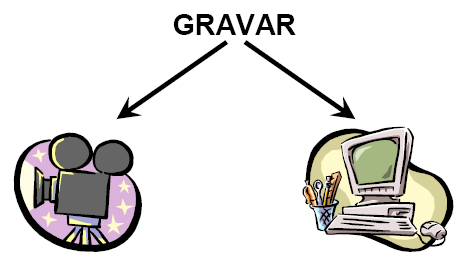
2.4.1.4 Polimorfismo

Do grego quer dizer "várias formas"; é um dos conceitos mais complicados de se compreender, é definido como um código que possui vários comportamentos em outras palavras, é um código que pode ser aplicado à várias classes de objetos de uma maneira prática.

Os conceitos de herança e polimorfismo quando usados em conjunto permitem ganhos extraordinários na codificação de rotinas, evitando repetições desnecessárias de estruturas do tipo If..Then, Select, aumentando a produtividade do desenvolvedor.

Um exemplo bem didático para o polimorfismo é dado por um simples moedor de carne. Esse equipamento tem a função de moer carne, produzindo carne moída para fazer bolinhos. Desse modo, não importa o tipo (classe) de carne alimentada; o resultado será sempre carne moída, não importa se de boi, de frango ou de qualquer outro tipo. As restrições impostas pelo processo estão no próprio objeto, definidas pelo seu fabricante e não pelo usuário do produto.( FERREIRA; JARABECK, 2007)

No exemplo a seguir, verificamos que o computador e a filmadora possuem a mesma função de gravar, porem esta função é realizada de formas diferentes.



**FIGURA 10 –** Exemplo de Polimorfismo  
 **FONTE:** FARINELLI, 2007

Como diz FARINELLI (2007): “O polimorfismo ocorre quando um método que já foi no ancestral é redefinido no descendente com um comportamento diferente”.

2.4.2 Vantagens

O paradigma da programação orientada ao objeto (POA) vem possibilitando a criação de sistemas parecidos com teias de aranhas, onde cada nó dessa teia representa uma classe, mas cada classe pode ser aproveitada para a composição de outras classes, bem como a reutilização dessas classes em outros sistemas, diminuindo o tempo de codificação e depuração de códigos para o programador. (FARINELLI, 2007)

Passamos a analisar a possibilidade de abstração que as classes podem ter, pois quanto maior for a capacidade de abstração, maior será a projeção de reutilização de classes no sistema.

O procedimento de manutenção restrito ao objeto reduz custo, tempo e assegura a certeza de uma manutenção eficaz, pois as variáveis envolvidas estão restritas aos objetos modificados (atributos e métodos), e mais em nenhuma outra parte do sistema, que uma vez esses objetos alterados não necessitarão de qualquer outra alteração. Essa flexibilidade e facilidade permitem que novos objetos sejam adicionados ou excluídos com igual agilidade.

Outro ponto muito produtivo no desenvolvimento de sistemas orientados a objeto é a capacidade de reutilização de códigos e o aumento da produtividade, portanto é muito mais do que copiar funções, é o reaproveitamento de classes em sistemas anteriores em novos sem modificar suas estruturas ou atribuir pequenas modificações.

A técnica orientada a objeto facilita muito a vida do programador, alguns desses benefícios fazem com que houvesse reduções de custo e tempo da manutenção dos sistemas, já que alguns erros podem ser detectados facilmente no momento de desenvolvimento; o reuso de códigos otimiza a produtividade dos desenvolvedores para construção de um software, um exemplo bastante utilizado é a herança e polimorfismo; Os sistemas se tornam mais fáceis de serem entendidos pois a OO chega mais perto da realidade do que qualquer outro técnica de programação.

**3 LIMITAÇÃO DA PROGRAMAÇÃO ORIENTADA A OBJETO E PROGRAMAÇÃO ORIENTADA A ASPECTO**

Neste capítulo será visto a limitação da Programação Orientada a Objeto, ou seja, onde ela falha e o que ocorre para que seu paradigma seja quebrado ao ponto de ter necessidade de criar uma nova técnica de programação. E por fim o conceito desse novo paradigma de programação utilizado atualmente, a Programação Orientada a Aspecto, será apresentada, demonstrando seu conceito e suas características.

3.1 Visão Geral

A engenharia de software ao passar dos anos vem ajudando os desenvolvedores de software, possibilitando várias técnicas, métodos e ferramentas a produzirem sistemas com maior qualidade, no entanto as características mais importantes foram sendo levadas adiante em novos paradigmas em busca do aumento da produtividade, da reusabilidade e da manutenibilidade.

Apesar de todo esse avanço que relativamente aumenta todo dia, houve a preocupação da parte dos desenvolvedores em determinadas funções dos sistemas onde esperava - se um nível de conhecimento mínimo dos usuários requerendo a entrada, o processamento e a saída dos dados o mais simples possível

Atualmente, com o avanço das tecnologias, suas aplicabilidades, e a popularização dos sistemas de informação, desenvolver um sistema exige dos profissionais pensar em preocupações variadas como, por exemplo, interface amigável, distribuição, paralelismo, persistência e segurança.(RESENDE; DA SILVA, 2005).

Outrora, observa-se também que o aumento da complexidade inseriu novos problemas na Programação Orientada a Objetos limitando a resolver. Desenvolver um sistema com apenas um programador que domine todas as fases do desenvolvimento tornou-se uma tarefa impossível. Desta maneira atualmente as tarefas são divididas em grupos, onde cada um dedica-se as fases de especificação de requisitos, analise, implementação, teste, preparação do ambiente, entrega e por fim o treinamento.

Com essa evolução, tem-se forcado e possibilitando a divisão das áreas de desenvolvimento em partes distintas e independentes, obtendo assim grupos divididos especializados no desenvolvimento de soluções de interface, distribuição, persistência e segurança, nos quais ao término integram - se, compondo um sistema completo.

3.1 Limitações da programação orientada a objeto

A Programação Orientada a Objeto tida como a grande promessa para resolver a crise que existia, pois com suas novas abstrações facilitariam o reuso dos códigos - como foi citado no capítulo anterior - contêm algumas limitações. Essas limitações são encontradas no momento de tratar comportamentos que se espalham por vários objetos.

Com todos os benefícios que a orientação a objeto permitiu, no entanto ela também possui limitações e não é capaz de reparar alguns problemas. Falhas na OO acontecem quando algum conjunto de elementos é encontrado junto com informações que fazem parte do domínio. Um conceito muito importante para quem programa em OO é o domínio do problema, pois ele representa a solução de um problema, onde se encontrar todas as informações do mundo real.

Podemos definir que há conjuntos de elementos inerentes a solução, pois servirão como suporte a esta, mas que não fazem parte do agrupamento original de informações que compõem o domínio, como, por exemplo, o tratamento de exceções. Neste ponto surgem os problemas da orientação a objetos. (WINCK; GOETTEN, 2006, p.30).

Na figura seguinte pode ser visto um exemplo da falha da OO. Um diagrama de classe de um editor de imagens que contêm as classes Line e Point. Ambas deveriam conter apenas códigos para implementar linhas e pontos, mas elas também tem interesses que representam a atualização do Display, também conhecido como interesses transversais.

**FIGURA 11** – Ocorrência de interesses transversais para atualização da imagem.   
**FONTE:** WINCK; GOETTEN, 2006.

Neste mesmo exemplo se o desenvolvedor quisesse implementar outros interesses como persistência de dados, autenticação ou distribuição, aconteceria dentro das classes Line e Point. Esses interesses transversais acabam sendo implementados junto com outro interesse, ocasionando a quebra do paradigma da OO.

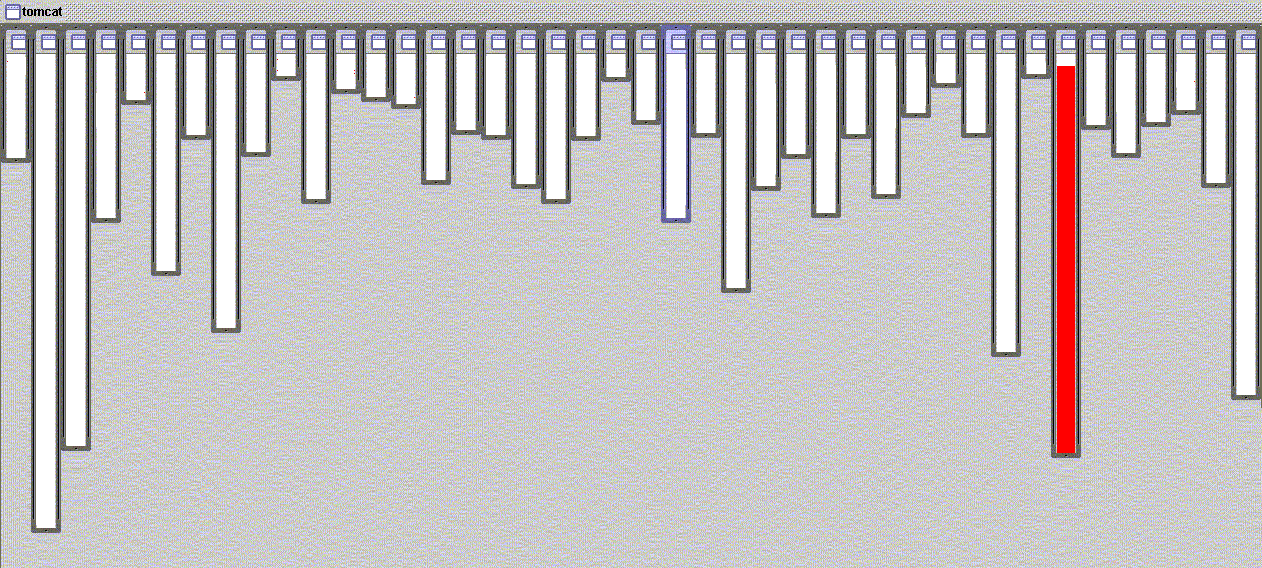
3.1.1 Interesses

Visando resolver futuros problemas, uma técnica comum é dividir em parte menores e tratá-las separadamente, essa técnica chama-se separação de interesses. Existem dois tipos de interesses, os funcionais e os sistêmicos. Segundo Winck e Goetten (2006) os interesses funcionais são os interesses que fazem parte do domínio da aplicação, e os interesses sistêmicos são os interesses que dão suporte a aplicação.

Como diz Winck e Goetten (2006) os interesses sistêmicos também são conhecidos como interesses transversais ou ortogonais e quando se implementa esses interesses em uma aplicação, acaba ocorrendo códigos espalhados, podendo também ocasionar em códigos emaranhados. Alguns exemplos de interesse sistêmico:

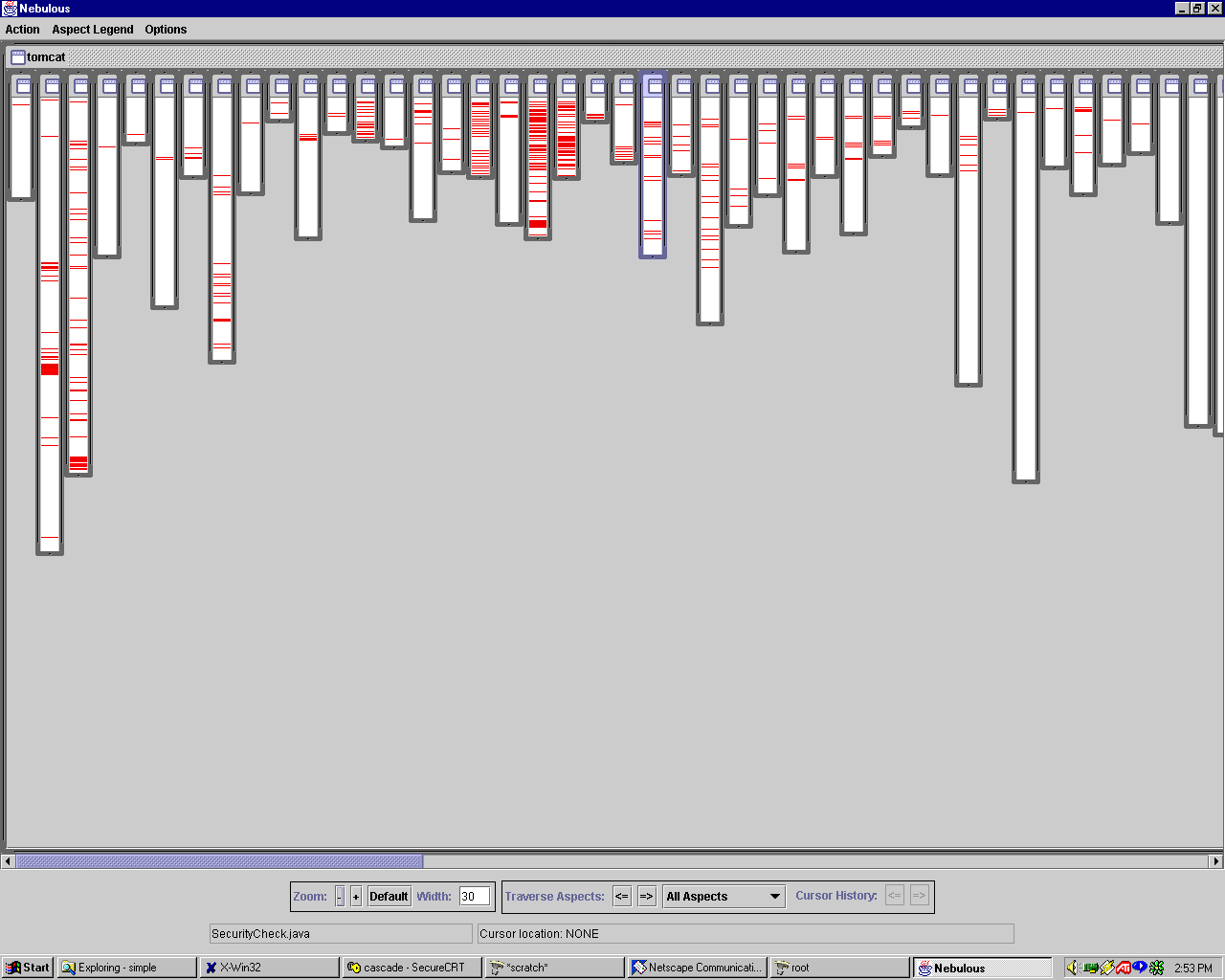
* A sincronização de objeto concorrente permite o uso de múltiplos fluxos de execução, threads, e os dados compartilhados tem de ser sincronizados guardando a integridade da informação;
* Na distribuição, as linguagens atuais deixam a desejar ao dar suporte para implementar essas diferentes perspectivas e propriedades, onde na verdade seria preciso modificar a perspectiva como os componentes especificados deveriam ser;
* No tratamento de exceções, os códigos try, catch e finally acabam se espalhando por todo o programa, que são condições que poderiam ser implementadas separadamente;
* A persistência de dados é muito utilizado e uma parte muito importante para um sistema qualquer. Sendo muito utilizado em um sistema, acaba se espalhando pelo o código e implementado em diversas classes. O desenvolver também pode criar classes especificas para persistência de dado, que são chamadas de camada de persistência;
* A auditoria (Log) detecta e registra, no sistema, qual ação que foi executada. O log ajuda para detectar quando e onde que ocorreram erros na sua aplicação, facilitando futuras manutenções. A implementação de auditoria em um sistema ocasiona códigos espalhados.

Nas figuras x as barras representam as classes e a linha vermelha representa os códigos. Neste primeiro exemplo pode-se vê que não ocorre código espalhado e o código se encontra em uma única unidade.



**FIGURA 12** – Implementação do parser XML no Tomcat.   
 **FONTE:** WINCK; GOETTEN, 2006.

Já no segundo exemplo pode-se notar que na implementação de auditoria (log) os códigos ficaram espalhados por diversas classes do sistema.



**FIGURA 13** – Implementação da funcionalidade de auditoria (log) no Tomcat.  
 **FONTE:** WINCK; GOETTEN, 2006.

3.1.2 Códigos Espalhados (Scattering Code)

O que se diz de códigos espalhados é que são códigos que se espalham por varias classes diferentes em um sistema, e ocorrem quando interesses transversais são implementados juntos com outro interesse que fazem parte do domínio.

No exemplo abaixo observa-se que o serviço das classes conta corrente e conta poupança precisa fazer o login e ao final o logout, caso ocorra algum problema ou manutenção, as duas classes precisarão ser alteradas. Já utilizando a POA seria menos trabalhoso.

class ContaCorrente{  
 public void saldo(){  
 **login (usuario, senha);**  
 …  
 **logout();**  
 }  
 public void sacar(){  
 **login (usuario,senha);**  
 …  
 **logout();**  
 }  
 public void depositar(){  
 **login (usuario,senha);**  
 …  
 **logout();**  
 }  
 public void extrato(){  
 **login (usuario,senha);**  
 …  
 **logout();**  
 }  
 }

class ContaPoupanca{  
 public void saldo(){  
 **login (usuario, senha);**  
 …  
 **logout();**  
 }  
 public void sacar(){  
 **login (usuario,senha);**  
 …  
 **logout();**  
 }  
 public void depositar(){  
 **login (usuario,senha);**  
 …  
 **logout();**  
 }  
 public void extrato(){  
 **login (usuario,senha);**  
 …  
 **logout();** }}

Este simples exemplo não demonstra qual será o real benefício da metodologia da programação orientada a aspectos. Mas, ao se imaginar essas repetições de linhas de código em diversas classes em diferentes pacotes, os benefícios que a AOP irá proporcionar se torna mais evidente. Segundo Resende e da Silva (2005, p22): “Em termo de produtividade e manutenibilidade isto significa retrabalho e necessidade de tempo extra para alterar todos os pontos necessários”.

Contudo, seu principal objetivo consiste em separar o código referente ao negócio do sistema dos códigos referentes aos interesses transversais, de uma forma bem definida e centralizada, possibilitando assim um nível maior de abstração no desenvolvimento de software.” Além do mais, estando bem separados e em locais bem definidos, os componentes podem ser melhor reutilizados e a sua manutenção e legibilidade torna-se mais agradável.

3.1.3 Códigos Emaranhados (Tangled Code)

O entrelaçamento de código ocorre quando torna-se necessário inserir chamadas de responsabilidades de uma classe em outra, ocasionando código invasivo. O espalhamento de código ocorre quando há várias chamadas de métodos de uma instância de uma classe em diversas outras classes, resultando na manutenibilidade e a produtividade mais trabalhosa, como também a reusabilidade é dificultada.

Quando o trabalho de preenchimento dos métodos se inicia, faz-se necessário inserir as chamadas de responsabilidades de uma classe em outra, realizando o entrelaçamento que integra os componentes ao software final. Estas linhas de códigos inseridas em outro componente para integração são denominadas código intrusivo ou invasivo. (RESENDE; DA SILVA, 2005).

Para visualizar esses problemas, abaixo vê-se exemplos de códigos relacionados ao entrelaçamento de código.

**public** **class** Ponto {

int x;  
int y;

**public** Ponto(int x, int y) {

**this**.x = x;

**this**.y = y;

}

**public** int getx() {

**return** this.x;

}

**public** int gety() {

**return** this.y;

}

}

**public** **class** Reta {

int x1,x2;  
int y1,y2;

**public** Reta(Pont x, Ponto y) {

**this**.x1 = x.getX();

**this**.x2 = x.getX();

**this**.y1 = y.getY();

**this**.y1 = y.getY();

}

}

As linhas de código acima representam chamadas de responsabilidades da classe Ponto na classe Reta.

|  |  |  |
| --- | --- | --- |
| **Termo original** | **Tradução** | **Descrição** |
| Scattering Code | Código espalhado | Código necessário para cumprir um interesse propagado em classes que precisam cumprir outros interesses. |
| O Código espalhado pode ser dividido em duas categorias distintas: Bloco de Código Duplicado e Bloco de Código Complementar. |
| Tangled Code | Código emaranhado | Utilização de uma única classe ou método para implementar múltiplos interesses. |
| 0 código emaranhado ocorre quando um módulo é implementado com múltiplos interesses simultaneamente. Um desenvolvedor freqüentemente considera os interesses como lógica do negócio, performance, sincronização, segurança, dentre outros, durante a implementação do módulo, Isso resulta na implementação de vários interesses em um mesmo módulo. |

**QUADRO 1** – Comparativo entre código espalhado e código emaranhado.   
**FONTE**: WINCK; GOETTEN (2006)

Como pôde ser visto no capítulo anterior, a OO trouxe muitos benefícios para vida do desenvolvedor, pois possibilitou uma maior flexibilidade e facilidade no reuso de códigos e na manutenção dos softwares. Com tudo isso o paradigma orientado a objeto ainda não era a técnica de programação ideal, pois seu paradigma era quebrado em determinadas implementações. Para supri essas limitações, surgiu a Programação Orienta a Aspecto (POA).

3.2 Paradigma Orientado a Aspecto

O paradigma orientado a aspectos é uma atual metodologia de programação que vem complementar o estável paradigma da orientação a objetos, com o propósito de tentar resolver as limitações encontradas no desenvolvimento de software orientado a objetos, como persistência, auditoria, entrelaçamento de código e o espalhamento de código que ocasionam muitas dificuldades aos desenvolvedores.

A Programação Orientada a Aspecto tem esse objetivo. Tentar separar os níveis de preocupação durante o desenvolvimento de software. Logo, seria possível pensar separadamente nos problemas referentes a persistência de dados, modelagem de negócios, segurança, distribuição, auditoria, registro de logs etc. A proposta é poder desenvolver as partes do sistemas sem se preocupar com as demais partes. A cada iteração da integração insere-se um novo nível de preocupação sem ser necessário alterar o que já está pronto.(RESENDE; DA SILVA, 2005, p12).

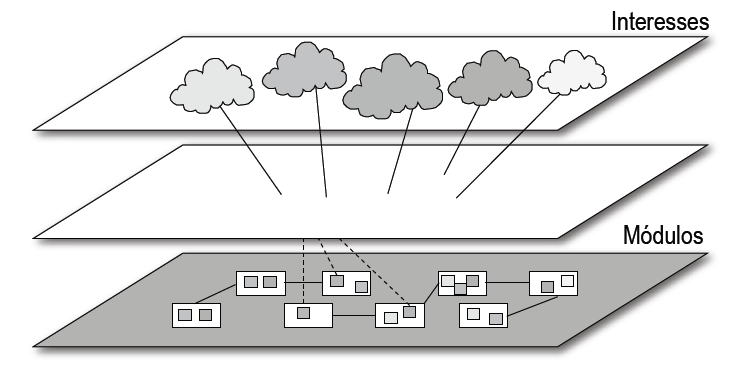
Segundo Winck e Goetten (2006) a programação orientada a aspecto foi desenvolvida por Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Videira Lopes, Jean-Marc Loingtier e John Irwin no laboratório da Xerox, situada em Palo Alto nos Estados Unidos, em meados dos anos 90, mais exato em1997.

Gregor Kiczales e seus companheiros tiveram essa iniciativa, pois idealizaram e criaram uma forma de separar as atividades implementando as características ortogonais fora do escopo da linguagem de componentes, para que fosse possível pensar separadamente e manter o foco no módulo responsável por cada desenvolvedor, que após a finalização ser realizada a integração requerida no software.

Para resolver um tipo de problema, na Orientação a Aspectos propõe-se a separação do código referente o negócio do sistema dos interesses transversais, tornando o código mais centralizado e, portanto, mais fácil de dar manutenção, pois se houver alteração em um interesse você só irá alterar código em um lugar. Interesse como já foi dito anteriormente são as características de extrema importância na aplicação, podendo ser divididos em diversos aspectos cujo são os requisitos do sistema.

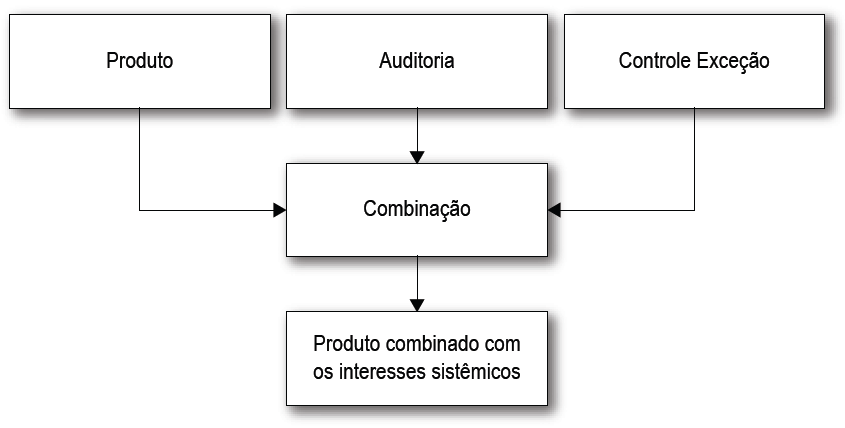
Como diz Resende e da Silva (2005), o nome Programação Orientada a Aspecto surgiu através da separação de trabalhos dentro de um sistema, fazendo com que problemas fossem implementados em diferentes unidades e isolados chamados de aspecto como, por exemplo, um aspecto de segurança, aspecto de persistência, entre outros.

Na figura 14, as nuvens representam os interesses transversais como auditoria (log), persistência, tratamento de exceção e etc. Também se pode vê a separação e a centralização que a POA trouxe, melhorando o reaproveitamento e a inteligibilidade dos códigos:

  
 **FIGURA 14** – Separação de interesses.   
 **FONTE:** WINCK; GOETTEN (2006)

Com a separação de interesse estes componentes podem ser melhor utilizados pelo o desenvolvedor, obtendo um entendimento mais fácil do sistema e fazendo com que a manutenção seja mais agradável.

Outro exemplo que a programação orientada a aspecto propicia a separação dos interesses pode ser visto na figura x, onde são implementados os interesses em módulos separados:

**FIGURA 15** – Combinação entre classe Produto e os demais interesses.   
**FONTE:** WINCK; GOETTEN (2006).

A classe Produto está separada dos interesses, que neste exemplo são Auditoria e Controle de Exceção. Depois da programação ocorre a combinação entre a classe e os aspectos.

A separação de interesses visa tornar o sistema mais modularizado, proporcionando três pontos chaves e essenciais no desenvolvimento de sistemas, pontos esses que são a reusabilidade, manutenibilidade e escalabilidade. E esses pontos chaves fazem parte justamente dos objetivos da programação orientada a aspectos.

3.2.1 Componentes da Orientação a Aspecto

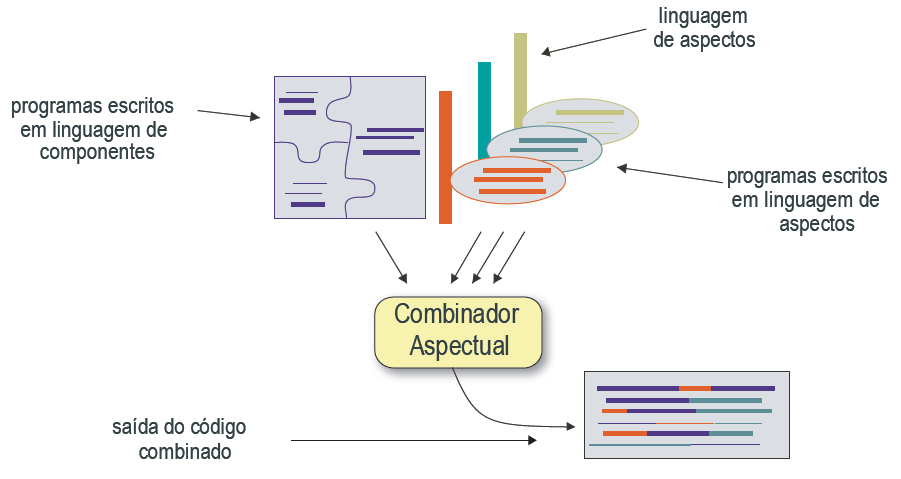
Para desenvolver uma aplicação orientada a aspecto são encontrados cinco componentes, que são: linguagem de componentes, linguagem de aspectos, combinador de aspectos, programa de componentes e programa de aspectos.

A implementação AOP é baseada em uma aplicação que consiste de: uma linguagem de componente com que se programa os componentes, um ou mais linguagens de aspecto com que se programa os aspectos, combinador de aspecto para combinar as linguagens, um programa de componente, isso implementa os componentes que usam a linguagem de componente, e um ou mais programas de aspecto que implementam os aspectos que usam a linguagem de aspecto.(KICZALES *et al*. 1997, p10, tradução nossa)

* Linguagem de componentes: permite aos desenvolvedores escrever programas que implementam as funcionalidades básicas do sistema, ou seja, os módulos relacionados as regras de negócio. Exemplo: Java, C++, C#, PHP etc.
* Linguagem de aspecto: permite aos desenvolvedores implementar os interesses sistêmico de forma que descrevam o comportamento dos aspectos e definam onde eles ocorrem.
* Programa de componentes: são os programas com as funcionalidades do sistema que são compilados pela linguagem de componentes.
* Programa de aspecto: são os programas com os interesses sistêmicos que são compilados pela linguagem de aspecto.
* Combinador de aspectos: faz a junção de códigos implementados na linguagem de componentes, com códigos implementados na linguagem de aspectos. Winck e Goetten descrevem que o combinador de aspecto é um processo antes da compilação que cria um código intermediário na linguagem de componente e pode ser dinâmica ou estática. Ela é estática quando o aspecto não precisa existir no momento da compilação e no momento da execução. Já a dinâmica os aspectos precisam existir no momento da compilação e da execução. Na figura a baixo pode-se vê a recompilação aspectual.

 **FIGURA 16** - Combinação aspectual.   
 **FONTE:** SOARES; BORBA.

Na figura 17 mostra a composição de um sistema desenvolvido em orientação a aspecto. Primeiro desenvolve os programas escritos em linguagem de componentes, depois implementa as linguagens de aspectos para criar os programas escritos em linguagem de aspecto e por fim o combinador gera um código com a combinação entre os programas.



**FIGURA 17** – Composição de um sistema orientado a aspecto   
**FONTE:** WINCK; GOETTEN (2006).

Segundo o Resende e Silva (2005) a POA permite essa separações de problemas e preocupações também chamado de separation of concerns que é o conceito de focar em apenas uma parte do problema (componentes e módulos) de cada vez, programando para unir os componentes e os requisitos não funcionais obtendo uma integração no software final.

3.3.2 Conceitos Fundamentais

A Programação Orientada a Aspecto possui quatro conceitos fundamentais para sua funcionalidade que são o Ponto de junção, Ponto de Atuação, Adendo e Aspecto.

* Pontos de junção (join points) são locais bem definidos no sistema, é onde ocorre a “junção” ou “composição” com os interesses transversais, como:
  + Chamada ou execução de métodos;
  + Acesso a determinados atributos (leitura e escrita);
  + Construção de objetos (new)
  + Lançamento de exceções

Que no quadro abaixo apresenta-se uma breve descrição de cada um deles:

|  |  |
| --- | --- |
| Pontos de Junção | Descrição |
| Chamada e execução de métodos | Pode-se interromper um programa na chamada ou execução do método.  Palavra reservada para isto: call e execution |
| Chamada e execução de construtores | Pode-se interroper um programa na chamada ou execução de construtores  Palavra reservada para isto: new |
| Métodos e acesso a atributos da classe (get e set) | Pode-se interromper um programa na chamada ou execução de métodos getters e setters (gets e sets) responsáveis por alterar ou obter o valor de um atributo.  Palavra reservada para isto: get e set |
| Execução de exceções | Pode-se interromper um programa na chamada ou execução do método  Palavra reservada para isto: handler |
| Execução de inicialização de classe e objeto | Pode-se interromper um programa na chamada ou execução do método  Palavra reservada para isto: staticinitialization |

**QUADRO 2 -** Tipos de ponto de junão **FONTE:** REZENDE; SILVA (2005)

* Pontos de atuação (pointcuts) são como padrões ou regras que definem quais são os pontos de junção que irão ser executados. Tendo o objetivo de criar regras diferentes onde vários pontos de junção serão identificados pelo mesmo ponto de atuação. Que segundo Winck e Goetten (2006) pontos de atuação (pointcuts) são as regras criadas pelo programador para especificar eventos que serão atribuídos aos pontos de junção, ou seja, definem os comportamentos do aspecto. É no ponto de atuação que os pontos de junção para um determinado interesse são agrupados.

Alguns designadores que são utilizados, definidos pelos pointcuts são listados na tabela abaixo.

|  |  |
| --- | --- |
| **Designador** | **Características** |
| Call (Signature) | Invocação do método / construtor identificado por assinatura |
| Execution (Signature) | Execução do método / construtor identificado por assinatura |
| Get (Signature) | Acesso a atributo identificado por assinatura |
| Set (Signature) | Atribuição do atributo identificado por assinatura |
| This (Type pattern) | Objeto em execução é instancia do padrão tipo |
| Target (Type pattern) | Objeto de destino é instancia do padrão tipo |
| Args (Type pattern) | Os argumentos são instancia do padrão tipo |

**QUADRO 3 -** Listagem dos designadores em aspectj  
 **FONTE:** MONTEIRO; PIVETA(2003).

* Como diz Monteiro e Piveta (2003) o Adendo (advice) é todo o código que se inicia quando um ponto de junção, definido pelo ponto de atuação for executado.

Tendo 3 tipos básicos:

* + - * Before() – quando ocorre antes do ponto de junção
      * After() – quando ocorre depois do ponto de junção
      * Around() – quando o código do adendo torna o ponto de junção sob controle.
* Aspecto é a parte central do aspectJ onde irão concentrar todos os pontos de atuação, declarações, instruções e adendos. Imaginando-se uma classe normal Java, porém ao invés de declará-la como public class cliente ela é declarada como “public aspect cliente”.

Aspects encapsulam pointcuts, advices e declarações inter-types em uma unidade modular de implementação. São definidos de maneira semelhante às classes, enquanto estas encapsulam o código que encaixa dentro de classes hierárquicas, aspects encapsulam o código que é ortogonal, transversal ou sobreposto a essas classes hierárquicas. (MONTEIRO; PIVETA 2003, p 320).

Como visto neste capítulo a programação orientada a aspecto necessita de dois tipos de linguagem, a de componente e a de aspecto. Seus benefícios são evidentes no desenvolvimento de sistemas complexos, além de que nesses tipos de sistemas, geralmente ocorrer a implementação de muitos interesses sistêmicos, que ao ser utilizado a orientação a aspectos, tais interesses serão agrupados em suas respectivas unidades funcionais, demonstrando o devido valor da orientação a aspectos. No próximo capítulo será conhecida a ferramenta AspectJ, voltada a linguagem de aspecto.

**4 ASPECTJ E APLICAÇÃO ORIENTADA A ASPECTO**

Neste capítulo será visto o que é a ferramenta AspectJ, conhecer os seus elementos mostrando exemplos de cada. Por fim será mostrada uma aplicação utilizando essas técnicas.

4.1 AspectJ

Existem várias linguagem de aspecto como hyperJ, aspectC++ e etc. Porém a mais interessante é a AspectJ, devido que sua linguagem de componente ser a linguagem de programação Java. O AspectJ é uma extensão do Java e ambas são distribuídas gratuitamente e podem ser executadas em diversas plataformas.

Esta ferramenta realiza a combinação aspectual estaticamente, assim diga, para executar o programa, antes deverá ser feita uma compilação. Durante a compilação, os códigos escritos em linguagem de componentes são combinados com os códigos escritos em linguagem de aspectos.

O AspectJ é um conjunto de compilador, combinador e utilitários associados a AOP com Java. A sintaxe utilizada no AspectJ para a construção dos aspectos é semelhante a Java, um dos motivos de sua ampla aceitação.(DO VALLE, 2006).

Um aspecto, como uma classe Java, pode definir atributos, métodos e uma hierarquia de aspectos, através da definição de aspectos especializados. Além de poder alterar toda a estrutura estática do sistema com a possibilidade de adicionar novos atributos, métodos e construtores a uma classe, alterando a hierarquia do sistema, esta característica de alterar a estrutura estática de um programa é chamada *static crosscutting.*

A ferramenta AspectJ contêm alguns elementos, como o ponto de junção (join point), ponto de atuação (pointcuts), adendo (advice), inserção (introduction), declaração em tempo de execução (compile-time declaration) e aspecto (AspectJ), que são utilizados para especificar regras de interesses dinâmicos e estáticos. “O AspectJ utiliza extensões para a linguagem de programação Java, especificando regras de combinações para interesses estáticos e dinâmicos”(WINCK; GOETTEN, 2006, p55).

Como foi falado o AspectJ é uma extensão da linguagem Java, com isso, como diz Winck e Goetten(2006), há uma preocupação com a compatibilidade de alguns itens, como compatibilidade de todo programa Java é também um programa AspectJ válido; compatibilidade das ferramentas para ser possível estender ferramentas existente para suportar o AspectJ; compatibilidade para o programador para que o desenvolvedor sinta-se como estivesse utilizando uma extensão do Java no momento que estiver desenvolvendo com o AspectJ.

Essas compatibilidades estão divididas entre:

* Compatibilidade total**:** onde todo programa Java válido é também um programa AspectJ válido.
* Compatibilidade de plataforma:todo programa AspectJ pode ser executado em uma máquina virtual Java (JVM).
* Compatibilidade de ferramentas: deve ser possível estender ferramentas existentes para suportar o AspectJ de uma forma natural; isso inclui IDEs (Integrated Development Environments),ferramentas de documentação e ferramentas de projeto.
* Compatibilidade para o programador:Ao programar com AspectJ, oprograma­dor deve sentir-se como se estivesse utilizando uma extensão da linguagem Java.

4.2 Divisão do AspectJ

No AspectJ há uma divisão em duas partes, a linguagem de especificação que designa a linguagem que o código com escrito com AspectJ, com a implementação em Java dos interesses funcionais, e a utilização de extensões na implementação das combinações dos interesses sistêmicos e a linguagem de implementação onde utiliza ferramentas para compilação, debug e integração.

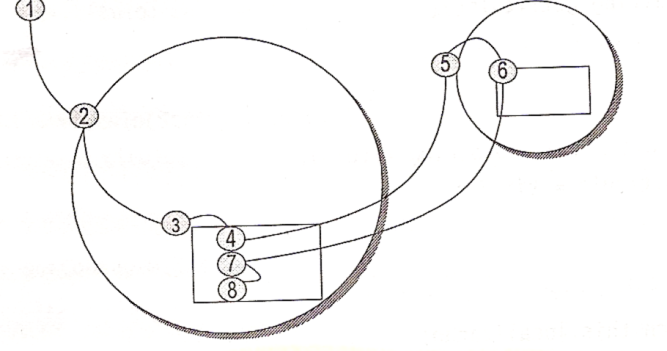
4.3 Elementos do AspectJ

No AspectJ são utilizadas extensões para a linguagem Java, determinando regras de combinações para os interesses, de um modo que facilite e confortabilize sua utilização. Divididos em ponto de junção, ponto de atuação, adendo e aspecto necessários para especificar as regras de combinação.

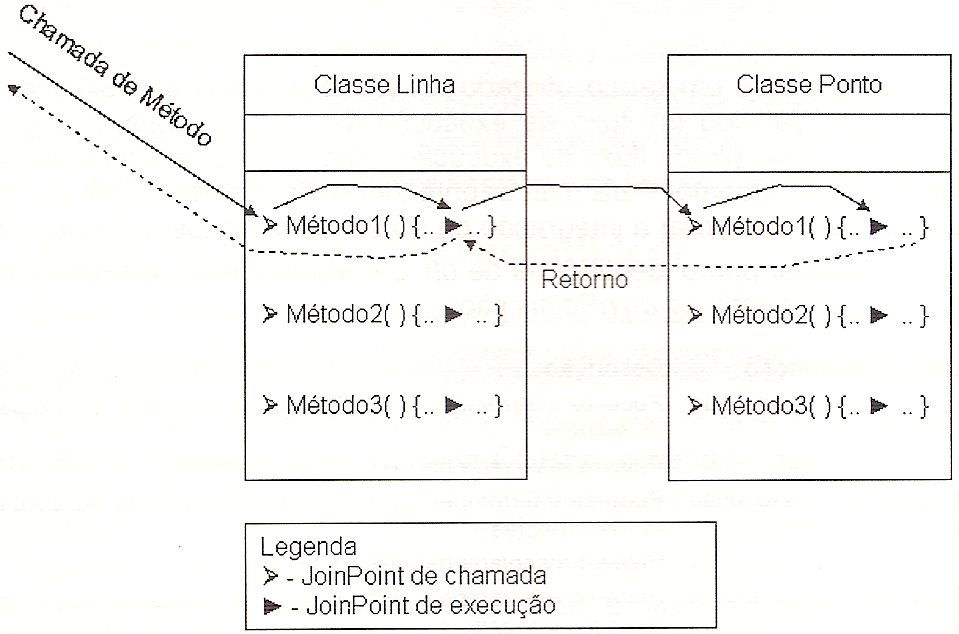
4.3.1 Ponto de Junção (Join Point)

O ponto de junção funciona no AspectJ como um invocador e executor de métodos, definindo-os quais serão interceptados pela sua chama ou execução. O desenvolvedor não programa o ponto de junção na linguagem de aspecto, pois esses pontos de junção já são encontrados no sistema codificados em linguagem de componente.

Na figura a baixo pode-se vê um exemplo de pontos de junção encontrados em duas classes diferentes. No ponto 1 ocorre a chamada do construtor para criação de um objeto, já no 2 há uma entrega da chamada do construtor , no 3 o ponto de junção ocorre a execução do código do construtor que foi invocado, o 4 representa a chamada de um vetor dentro de uma classe que recebe no ponto 5 o comando para criação de um objeto de veto, que esse objeto vetor é executado no ponto 6, já no ponto 7 e 8 se encontrar um ponto de junção que define um valor de um atributo qualquer.

   
**FIGURA 18** – Pontos de junção encontrados em uma classe.  
**FONTE:** WINCK; GOETTEN, 2006.

Como visto também na figura 19, Resende e da Silva (2005) comentam que na execução no método do objeto na classe Linha é chamado um método no Objeto na classe Ponto, as operações relativas a invocação e execução de métodos são ponto de junção(join points), da mesma maneira que um método do objeto da classe linha pode retorna com sucesso ou retornar uma exceção.

**FIGURA 19** – Esquema de chamada e execução de métodos, funções ou rotinas.   
**FONTE:** RESENDE; DA SILVA, 2005.

Os eventos que fazem parte de um modelo de ponto de junção são:

* Chamadas de um método ou um construtor;
* Execução de um método ou um construtor;
* Iniciação de um objeto ou uma classe;
* Acesso ou atualização de um campo;
* Tratamento de exceções.

O ponto de junção chamada de métodos pode ser definido a parti de qualquer chamada para qualquer método e ele é definido no objeto chamado. O código a seguir, pego como exemplo do Winck e Goetten (2006), mostra este tipo de ponto de junção.

...

**public static void** main( String args[]){

DVDBox box = **new** DVDBox(“Lost – 1ª Temporada”)

box.setTitulo(“ Lost”);

}

…

Em uma classe ExecutorLojaDVD é invocado um método do objeto box. Como pode ser visto na linha 3, foi chamado o método setTitulo.

O ponto de junção chamada de construtor é definido quando é chamado no momento de criar um novo objeto, inserindo a palavra new no código. Utilizando o mesmo código da classe ExecutorLojaDVD, encontrar-se um exemplo deste ponto de junção.

**public static void** main( String args[]){

DVDBox box = **new** DVDBox(“Lost – 1ª Temporada”)

}

Já o ponto de junção execução da chamada de métodos ocorre antes que qualquer código no objeto chamado seja executado. Utilizando a mesma classe que os exemplos anteriores podemos encontrar este ponto de junção na linha 3, onde ocorre a chamada do método setTitulo e passa como parâmetro uma String. Esse ponto de junção irá pegar antes da execução do método.

**public static void** main( String args[]){

DVDBox box = **new** DVDBox(“Lost – 1ª Temporada”)

box.setTitulo(“ Lost”);

}

O ponto de junção de acesso a atributos, conhecido mais com get, acontece quando um atributo de um objeto é lido. Um exemplo da utilização deste ponto de junção seria em um classe DVD, executar o método getTitulo.

No ponto de junção de definição de valor para atributos, mais conhecido como o set, ocorre quando o valor de um atributo do objeto é modificado ou definido. Um exemplo utilizando a mesma classe DVD seria no momento da execução do método setTitulo.

O ponto de junção que indica o tratamento de exceção é encontrado no bloco de códigos try e catch, mais especifico no catch. Um exemplo deste ponto de junção é quando se tenta conectar com o banco de dados e necessita tratar utilizando o bloco try e catch.

**try** {

conexaoBD.conectar(ConexaoBD.*DRIVER\_JDBC\_FIREBIRD*, url, "sysdba", "masterkey");

} **catch** (Exception e) {

e.printStackTrace();

}

Na inicialização de classe ocorre a execução do ponto de junção quando se inicializa uma parte estática da classe, caso não possua um elemento estático não existe ponto de junção. Na inicialização de objetos ocorre a execução do ponto de junção quando se inicializa uma parte dinâmica para uma classe. Winck e Goeten (2006) falam:”Esse ponto de junção ocorre depois da chamada para um objeto construtor.”

Somente identificar os pontos de junção e agrupá-los em um ponto de atuação, o aspecto não terá uma funcionalidade que irá ser executada quando o(s) ponto(s) de junção for(em) alcançado(s). Para dá funcionalidade ao aspecto, é preciso declarar um advice, que é semelhante a um método de uma classe java

4.3.2 Ponto de Atuação (Pointcuts)

Sua função é de declarar os pontos onde se deseja interromper a execução de um sistema, porem para se compreender ponto de atuação, necessita-se saber um outro conceito mencionado acima, que é a idéia de um ponto de junção ocorrido em lugares bem definidos durante a execução do sistema, assim o ponto de atuação identifica ocorrências especificas na qual as recomendações devem ser associadas, mostrando os dados do contexto de execuções de cada ponto de junção, assim utilizados pela ocorrência do ponto de junção e mapeado pelo ponto de atuação.

A inserção dessas recomendações é de responsabilidade de um compositor de aspecto que são extensões para os compiladores que processam a definição dos aspectos, das classes de objetos e métodos no sistema, que por sua vez gera um novo sistema incluindo os aspectos nos pontos de junção especificados.

São elementos da Orientação a Aspecto e do AspectJ usados para definir um ponto de junção, como uma espécie de regra criada pelo programador para especificar eventos que serão atribuídos a esses pontos. Os pontos de atuação existem para que possam ser criadas regras genéricas para definir os eventos que serão considerados como ponto de junção.(WINCK; GORTTEN, 2006).

O ligamento dos pontos de junção e atuação é de fato evidente na ocorrência dos interesses sistêmicos.

Uma simples visualização de um ponto de atuação na interrupção do sistema com uma chamada call no método somar da classe contas.

**import** java.util.\*

**public aspect** exemplo\_basico\_pointcuts{

**pointcut** PointCut\_Um(): **call**(int COntas.Somar(int, int));

}

As palavras reservadas, execution, call, get, set, staticinicialization, handler, initialization, preinitialization, adviceexecution, são designadores do ponto de atuação que provêem uma definição ao redor do ponto de junção. Existem, além destes, outros designadores, como o this, target, args, within e o if.

Ponto de atuação para chamadas de métodos e construtores usa-se a palavra reservada call para interceptar os pontos de execução antes da mudança do método a ser chamado e após ter avaliado os parâmetros da chamada, como no código abaixo que exemplifica o ponto de atuação numa chamada de um método e um construtor no texto em negrito.

import java.util.\*

**public aspect** exemplo\_basico\_pointcuts{

**pointcut** ChamadaMétodo();

**call**(void Vector.addElement(Object));

**pointcut** ChamadaConstrutor();

**call**(void Vector.new ());

**pointcut** ExecucaoMétodo();

**execution**(void Vector.add.Element(Object));

**pointcut** ExecucaoConstrutor();

**execution**(void Vector.new ());

}

Aproveitando o ensejo do código já no ponto de atuação para execução de métodos e construtores se utiliza a palavra reservada execution, que interceptam os pontos antes de iniciar a execução do primeira linha do método e posterior a instrução da chamada do método. Resultando os quatro pontos de atuação que interceptam o método (addElement) e seu construtor (Vector) permitindo a interrupção normal do sistema.

Nos quadros abaixo alguns exemplos de declarações e suas respectivas descrições.

|  |  |
| --- | --- |
| **PointCut** | **Descrição** |
| Call(public void Classe.método(String)) | Intercepta as chamadas a métodos públicos denominados, “método”, da classe “Classe”, que retornam “void” e passam argumento “String”. |
| Call(void.Classe.método(..)) | Intercepta as chamadas a qualquer método denominado “método” da classe “Classe” que retornam “void”, não importando o tipo do argumento. |
| Call(\*Classe.método(..)) | Intercepta as chamadas a qualquer método denominado “método” da classe “Classe” não importando o tipo de retorno e de argumento |

**QUADRO 4 –** Declaração de pontos de atuação de Chamada de Métodos e Construtores  
 **FONTE:** RESENDE; DA SILVA, 2005

|  |  |
| --- | --- |
| **PointCut** | **Descrição** |
| execution(public void Classe.método(String)) | Intercepta a execução a métodos públicos denominados, “método”, da classe “Classe”, que retornam “void” e passam argumento “String”. |
| execution(void.Classe.método(..)) | Intercepta a execução a qualquer método denominado “método” da classe “Classe” que retornam “void”, não importando o tipo do argumento. |
| execution(\*Classe.método(..)) | Intercepta a execução a qualquer método denominado “método” da classe “Classe” não importando o tipo de retorno e de argumento |

**QUADRO 5 –** Declaração de pontos de atuação de Execução de Métodos e Construtores **FONTE:** RESENDE; DA SILVA, 2005

Os pontos de atuação de métodos de Acesso a Campos utiliza os conhecidos métodos de acesso a campos (getters e setters) sua aparição vem na necessidade de interceptar as chamadas de métodos de alteração do valor do atributo (set) e na obtenção desse mesmo valor(get).

A seguir no quadro 6 as declarações do Get e Set.

|  |  |
| --- | --- |
| **PointCut** | **Descrição** |
| Get(tipo Classe.atributo) | Intercepta as chamadas ao método “get” que retornam o tipo “tipo” e que acesso o atributo “atributo” que pertence à classe “Classe” |
| Get(int Classe.\*) | Intercepta as chamadas ao método “get” que retornam inteiros e pertencem à classe “Classe”, independente do atributo que ele acessa. |
| Get(int Classe.se\*) | Intercepta as chamadas dos métodos “get” cujo nome comece por “se”, retornam inteiro e pertençam à classe “Classe”, independente do atributo que ele acessa |
| Get(\* Classe \*) | Intercepta todas as chamadas aos métodos “get” classe “Classe” |
| Set(tipo Classe.atributo) | Intercepta as chamadas ao método “set” que retornam o tipo “tipo” e que acesso o atributo “atributo” que pertence à classe “Classe” |
| Set(int Classe.\*) | Intercepta as chamadas ao método “set” que retornam inteiros e pertencem à classe “Classe”, independente do atributo que ele acessa. |
| Set(int Classe.se\*) | Intercepta as chamadas dos métodos “set” cujo nome comece por “se”, retornam inteiro e pertençam à classe “Classe”, independente do atributo que ele acessa |
| Set(\* Classe \*) | Intercepta todas as chamadas aos métodos “set” classe “Classe” |

**QUADRO 6 –** Declaração de pontos de atuação de Métodos de Acesso a Atributos   
**FONTE:** RESENDE; DA SILVA, 2005

Portanto no quadro acima as declarações interceptam os métodos de um único, de alguns e todos os atributos, como mencionado “A POA permite capturar os métodos getters e setters de um atributo especifico, de alguns atributos ou de todos atributos”.(RESENDE; DA SILVA, 2005).

No exemplo abaixo será mostrado algumas declarações aplicadas ao código e sua funcionalidade é o comentário de Resende e da Silva (2005) Além de apresentar algumas declarações, será mostrado um trecho de código identificando a ação dos metodos, que para capturar apenas um get de um campo é preciso uma declaração pointcut one\_get, já um set dos campos do tipo String implenta-se a declaração pointcut String\_set e no caso de uma captura de todos os campos da classe utiliza-se o pointcut all\_set.

package lib.entidade;

**public aspect** Exemplo\_GettersAndSetters{

**pointcut** one\_**get**(): get(String Cliente.nome);

**pointcut** String\_**set**(): set (String Cliente.\*);

**pointcut** all\_**set**(): set (\* Produto.\*);

before() : String\_**set**(){

System.out.println("Capturado: " + **thisJoinPoint**);

}

}

Com a aparição da orientação a objeto tornou muito conhecido o bloco de capturar e tratar os caminhos alternativos do sistema chamados try.. catch, em orientação a aspecto no ponto de atuação de chamada de exceções sua função seria capturar o bloco catch das exceções utilizando a forma handler, que uma vez havendo erro seu caminho será desviado podendo-se executar um advice que será abordado posteriormente.

Algumas declarações do ponto de atuação de Chamadas de Exceções seriam handler(RemoteException) interceptando o bloco catch junto a sua exceção remota, handler(IOException+) que intercepta tambem o bloco associado a sua excecao e suas subclasses (+).

Por fim surgi os pontos de atuação de Inicializacao de Classe, que captura o ponto de junção referente a inicialização de um objeto, utiliza como palavra reservada a assinatura initialization, preinitialization, bom salientar a importância da ordem em que os pontos de atuação tanto de inicialização e pré-inicialização onde um age na chamada dos construtores resultando numa ordem hierárquica decrescente(do baixo para o alto), e o outro age após a execução dos construtores da classe seguindo uma ordem hierárquica crescente(do alto para o baixo).

O pointcut pré-inicializacao ocorres quando os construtores estão se preparando para criar uma classe, ou seja, na chamada dos construtores. O pointcut inicialização ocorre depois que o objeto é inicializado, ou seja, após a execução do construtor.(RESENDE; DA SILVA, 2005)

Como exemplo o trecho de um código

**public class** base0 {

**public** int x=0;

**public** Base0() {

System.out.println(" Executando o construtor de Base0()!");

}

**public** Base0(int x){

System.out.println(" Executando o construtor de Base0(int)!");

}

}

**public class** base1 {

**public** int x=1;

**public** Base1() {

System.out.println(" Executando o construtor de Base1()!");

}

**public** Base1(int x){

System.out.println(" Executando o construtor de Base1(int)!");

}

}

**public class** base2 {

**public** int x=2;

**public** Base2() {

System.out.println(" Executando o construtor de Base2()!");

}

**public** Base2(int x){

System.out.println(" Executando o construtor de Base2(int)!");

}

}

4.3.3 Adendo (Advice)

Um dos elementos fundamentais na programação orientada a aspecto como também ao aspectj, muito similar e quase equivalente a métodos das classes Java, declarando um conjunto de códigos que serão executados na hora em que o ponto de junção definido no ponto de atuação for capturado, ressaltando ainda que é peça indispensável na implementação de conceitos de aspecto.

“Advices equivalem a métodos (ou funções) e são formados por um conjunto de instruções que poderão ser executados no momento em que um JoinPoit for alcançado” (RESENDE; DA SILVA, 2005, p79).

Todo adendo é ligado a um ponto de junção pelo fato, no momento da interceptação ser inserida novas rotinas, utiliza-se de três palavras reservadas que são before, aftere a around.

“O adendo e sempre definido para um ponto de atuação. O ponto de atuação e responsável por selecionar os pontos de junção onde o código contido no adendo devera ser executado. O corpo do adendo atua exatamente como o corpo de um método”(WINCK; GORTTEN, 2006, p102).

A seguir um breve exemplo apresentando o adendo em um método sacar() com registro de auditoria.

before() : execution(void Conta.credito(double)){

System.out.println("Operação de saque da classe Conta");

}

Com os tres diferentes tipos de adendo no AspectJ será demonstrado no quadro 7 a atuação de cada um deles, na utilização correta e obtendo os melhores resultados.

|  |  |
| --- | --- |
| Tipo de Adendo | Descrição |
| Before | Determina que o adendo execute imediatamente antes de um ponto de atuação ser alcançado. |
| After | Determina que o adendo execute imediatamente após um ponto de atuação ser alcançado. |
| after returning | Executa após ocorrer um retorno normal do método com sucesso. |
| after throwing | Executa após ocorrer uma exceção . |
| around | Executa quando o ponto de junção é alcançado e toma controle do fluxo de execução do sistema |

**QUADRO 7 –** Tipos de adendo disponíveis no AspectJ

No código abaixo da classe Cliente são demonstrados os pontos exatos de interceptação das declarações do adendo no aspecto Teste.

public aspect Teste{

pointcut allcall(): call(\* Cliente.\*(..));

before(): allcall(){

System.out.println("[Before] " + thisJoinPoint);

}

after(): allcall(){

System.out.println("[After] " + thisJoinPoint);

}

before() returning : allcall(){

System.out.println("[Before returning] " + thisJoinPoint);

}

after() returning : allcall(){

System.out.println("[After returning] " + thisJoinPoint);

}

public static void main(String[] args) {

Cliente c1 = new Cliente();

c1.setNome("Cliente 1");

c1.setEndereco("rua das arvores");

c1.setIdade(32);

c1.setSalario((float)100.0);

c1.imprimir();

}

}

public class Cliente{

private int idade;

private float salario;

private String nome, endereco;

//setters

public void setNome(String var){

this.nome = var;

}

public void setEndereco(String var){

this.endereco = var;

}

public void setIdade(int var){

this.idade = var;

}

public void setSalario(float var){

this.salario = var;

}

//getters

public String getNome() {

return this.nome;

}

public String getEndereco() {

return this.endereco;

}

public int getIdade() {

return this.idade;

}

public float getSalario() {

return this.salario;

}

public int getSoma() {

return this.idade + 10;

}

public void imprimir(){

System.out.println("IMPRIMINDO CLIENTE");

System.out.println("Nome: "+this.getNome());

System.out.println("End: "+this.getEndereco());

System.out.println("Idade: "+this.getIdade());

System.out.println("Salario: "+this.getSalario());

System.out.println("Soma: "+ this.getSoma);

}

}

4.3.4 Inserção (Introduction)

Sua função é basicamente alterar a estrutura da classe, introduzindo novos métodos, atributos, construtores, getters, setters e herança, como mostra o exemplo abaixo onde tanto na inserção de um atributo ou método, se utiliza o nome da classe, . (ponto) o nome do método, assim também para os atributos, já no caso de construtores pode-se a palavra new.

public aspect exemplo\_introdution{

private String Cliente.nome;

private int Cliente.idade;

}

public Cliente.new(String nome, int idade){

this.nome = nome;

this.idade = idade;

}

4.3.5 Aspecto (Aspect)

O Aspecto do AspectJ é como se fosse a Classe para o Java. A forma de declaração geral de um aspecto é formada por:

[privileged] [Modifiers] aspect Id [extends Type] [implements TypeList] [

PerClause]

{

Body

}

O privileged indica a permissão, ao aspecto, para o acesso das partes privadas das classes. Resende e da Silva (2005) recomendam a não utilização dele, pois em maior parte se usa os gets e os sets para manipulação dos atributos.

O Modifiers são para indicar regras de visibilidade, que nem as classes de Java, usando public para declarar o aspecto publico, private para o aspecto private e protect para o aspecto só ser visto dentro do pacote que se encontrar ou pelos subaspectos.

Exemplo de uma declaração de aspecto:

**public aspect** Aspecto\_01{

**Public static void** main(String [] **args**){

}

}

O extends é utilizado se o aspecto for herdar atributos e métodos de outro aspecto e o implements se o aspecto for utilizar implementação de uma interface. O próximo exemplo mostra a utilização dos dois.

**public aspect** Aspecto\_04 **extends** Aspecto\_03 **implements** Runnable{

//Implementação do método RUN da interface Runnable

**public void** run(){}

**public static void** main(String[] **args**){

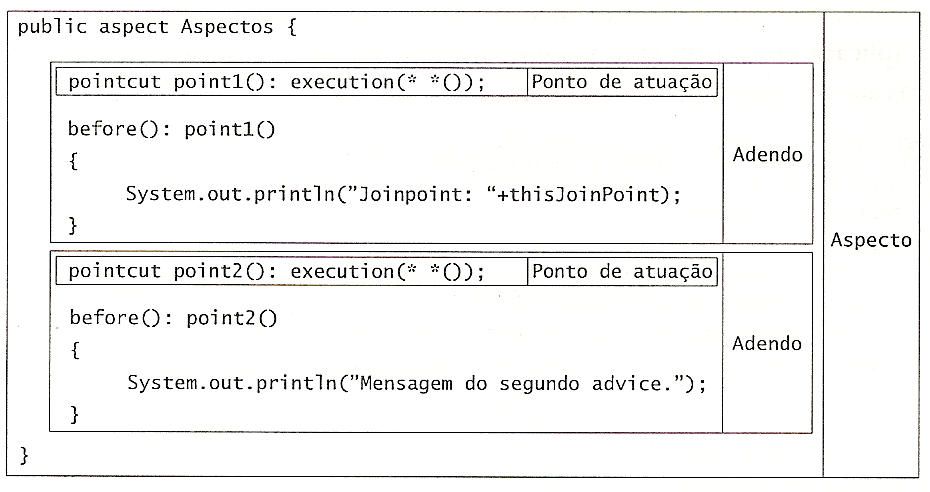
}

}

Aspectos são completamente diferentes de outras abstrações de programas, sendo que os aspectos em si incluem uma especificação de onde devem ser executados. Em outras abstrações, como métodos, há uma separação clara entre a definição de uma abstração e o seu uso. Você pode chamar um método com base em qualquer lugar no escopo e não é possível saber de onde o método deve ser chamado apenas olhando para ele. Aspectos, de outro modo, incluem um ponto de junção – uma declaração que define onde o aspecto será composto no programa. (SOMMERVILLE, Ian; 2007, p.514)

O aspecto encapsula os outros elementos do AspectJ como o ponto de junção, ponto de atuação e adendo. Além disso, o aspecto pode conter dados e métodos próprios. Winck e Goeten (2006, p.116) comentam: “O aspecto é bastante similar a uma classe Java. O aspecto provê um contêiner para encapsulamento do código referente a pontos de atuação, ponto de junção e adendos. O aspecto pode conter ainda seus próprios atributos e métodos”.

Na figura a baixo pode-se vê a estrutura de um aspecto que contem dois adendos e que cada adendo contêm um ponto de atuação pré-definidos pelo ponto de junção.

**FIGURA 20** - Estrutura de um aspecto  
**FONTE:** WINCK; GOETEN, 2006.

4.2 Aplicação Orientada a Aspecto

Uma aplicação de cadastro de cliente foi desenvolvida, com funcionalidades como cadastrar, excluir, visualizar o cliente, sempre fazendo uma auditoria no momento de cadastrar e excluir, com a metodologia de programação orientada a objeto e orientada a aspecto para demonstrar a diferença entre as duas.

Para o desenvolvimento desta aplicação foi utilizada a IDE eclipse Europa 3.3.2, que é distribuídos gratuitamente, o banco de dados Firebird e o plugin AJDT versão 1.5.2 com a versão do AspectJ 1.6.0 para implementar os aspecto.

4.2.1 Análise do sistema de cadastro de cliente

Na análise do sistema de cadastro de cliente podem-se encontrar dois atores que são Usuário e Cliente, já o caso de uso são quatro, como cadastrar, excluir, visualizar cliente e AuditarSistema.

**Atores:** Na terminologia da UML, qualquer elemento *externo* que *interage* com o sistema é denominado Ator.

|  |  |
| --- | --- |
| **Atores** | **Descrição** |
|  | Usuário é a pessoa responsável para cadastrar, excluir e visualizar os dados dos clientes. |
|  | O cliente é a pessoa responsável por informar seus dados para que seja cadastrado no sistema e para que possam ser realizadas as operações referentes a ele. |

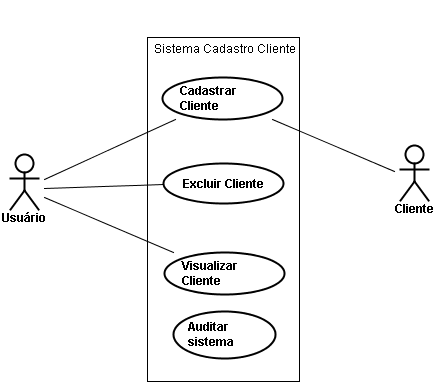
**QUADRO 8 –** Descrição dos atores.

|  |  |
| --- | --- |
| **Caso de uso** | **Descrição** |
| Cadastrar Cliente | Módulo do sistema que o usuário cadastra os dados passados pelo cliente. |
| Excluir Cliente | Módulo do sistema que o usuário exclui os dados do cliente caso seja necessário. |
| Visualizar Cliente | Módulo do sistema para visualizar os dados do cliente que o usuário escolher. |
| Auditar Sistema | Módulo do sistema para criar um log no momento em que o usuário cadastrar e exclui o cliente. |

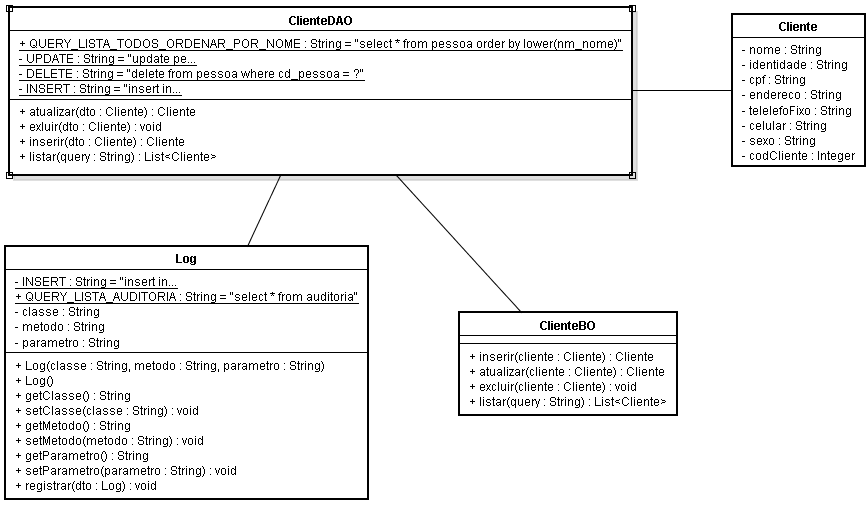
**QUADRO 9** – Descrição dos casos de uso.

4.2.1.1 Diagramas do sistema

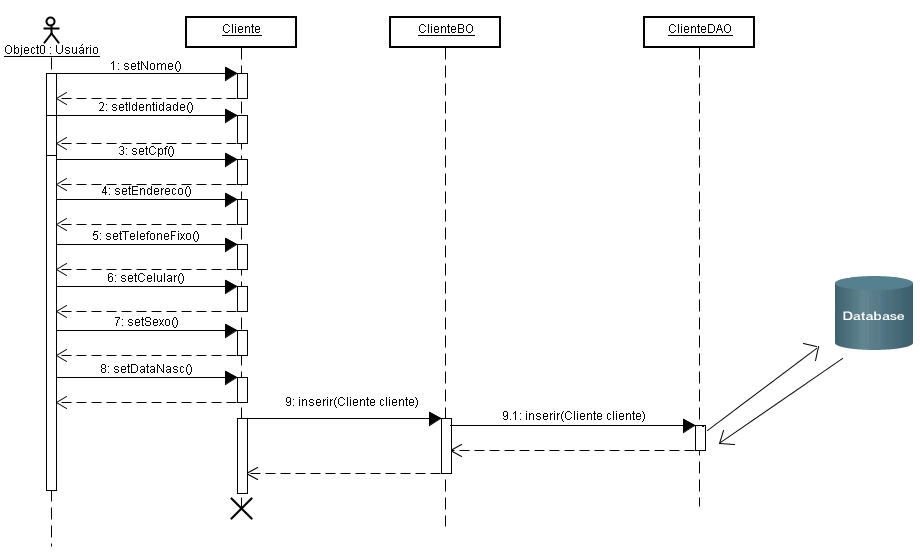
Como pode ser visto o diagrama de caso de uso abaixo o ator usuário cadastra o cliente após o ator cliente passar os dados necessários para o cadastro. Além disso, o ator usuário excluir os dados do cliente quando necessário e visualiza esses dados de qualquer outro cliente cadastrado. O caso de uso Auditar Sistema cadastra um log no banco de dados do momento da inserção e exclusão dos clientes.

  
 **FIGURA 21 –** Diagrama de caso de uso.

No diagrama de classe mostra a ligação entre as classes Cliente, ClienteDAO, ClienteBO e Log com seus atributos e métodos.

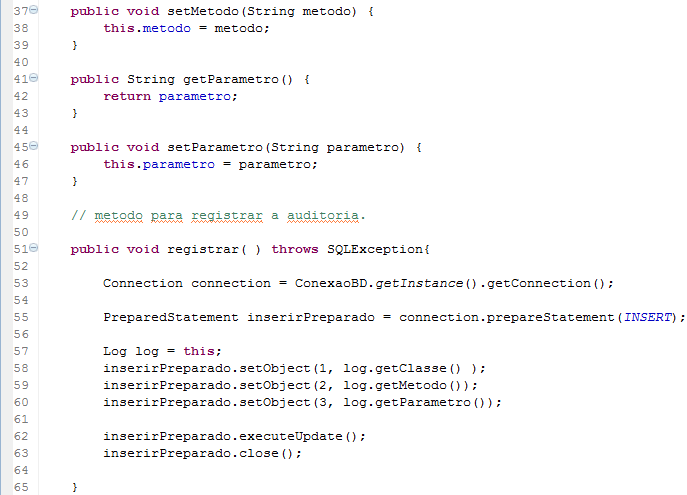
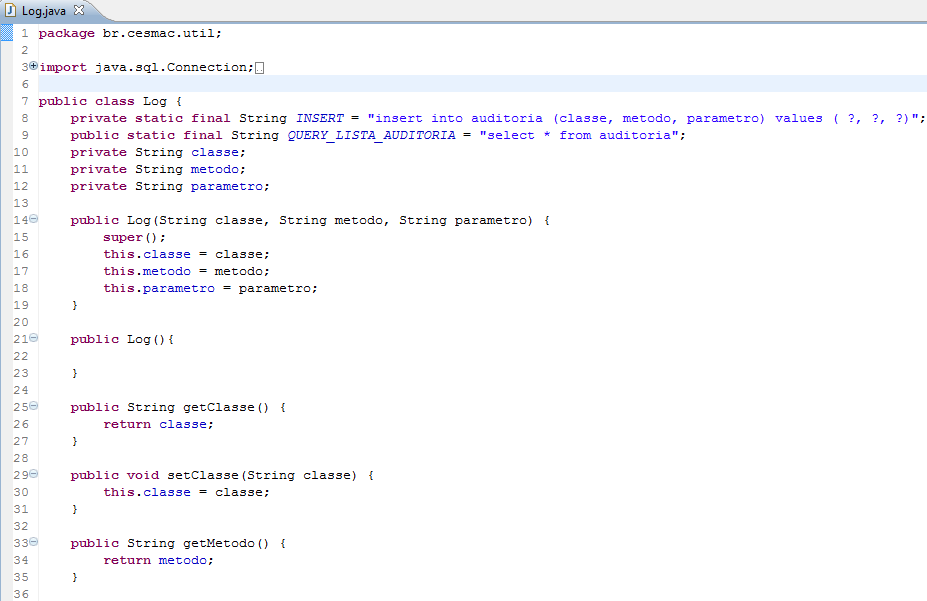
**FIGURA 22** – Diagrama de Classe.

No diagrama de seqüência mostra no momento de cadastrar o cliente, o usuário insere os dados dos seguintes atributos da classe cliente, que chama o método inserir do ClienteBO que valida os campos digitados para chamar o método inserir da classe ClienteDAO, que esta insere os dados no banco de dados.

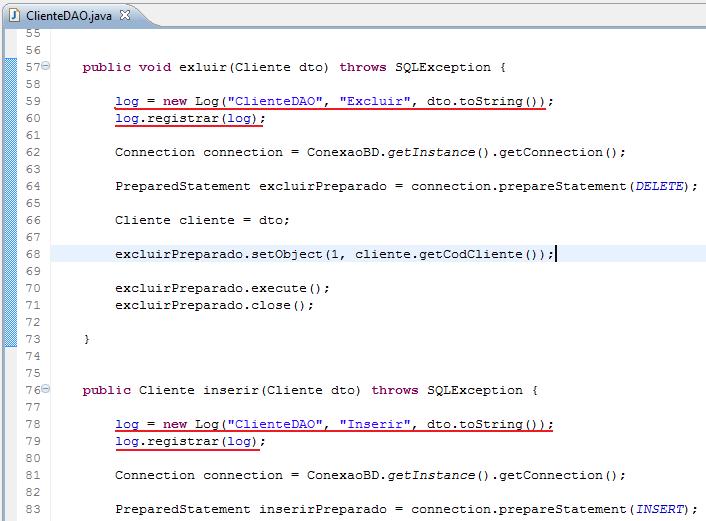
 **FIGURA 23** – Diagrama de Seqüência.

4.2.2 Códigos do sistema

Na aplicação orientada a objeto, foi criada uma classe Log e dentro desta classe um método registrar para fazer a auditoria no momento de excluir e adicionar novos clientes.

**LISTAGEM 1** – Classe Log.

Na classe ClienteDAO, que é a classe que forma a camada de persistência de dados, encontra-se os códigos para fazer a auditoria do sistema, mais exato, nos métodos excluir e inserir desta classe como se pode vê na listagem abaixo:

  
**LISTAGEM 2 –** Classe ClienteDAO com interesses sistêmicos.

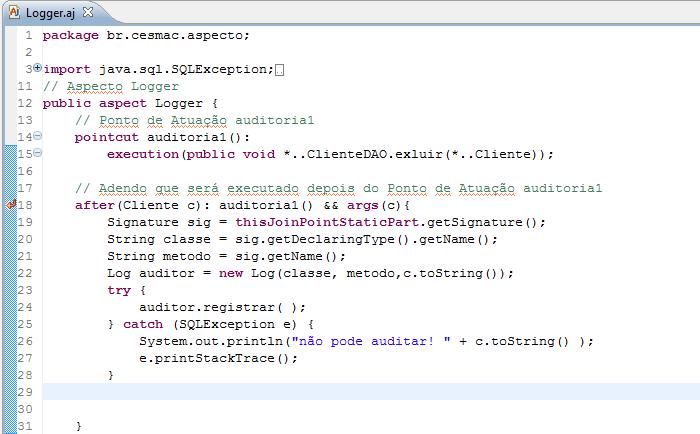
Pode-se vê que se utilizar desta metodologia de programação, o programador terá que espalhar o código de auditoria por todo o sistema como esta marcada nas linhas sublinhadas em vermelho. Se caso fosse fazer uma auditoria mais completa esse código de auditoria espalharia mais ainda pelo sistema. Além de espalhados, os códigos ficam emaranhados, pois na classe ClienteDAO é chama um método de outra classe que seria o registrar da classe Log.

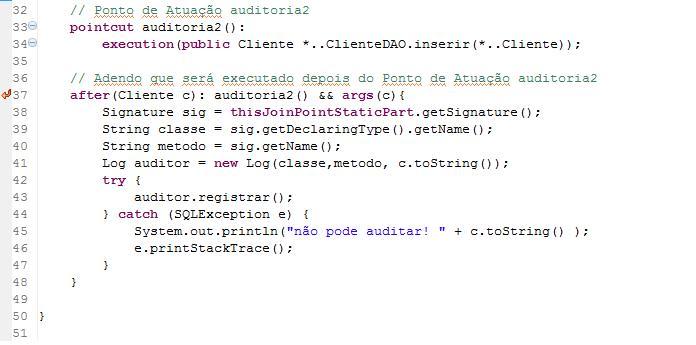
4.2.3 Sistema Orientado a Aspecto

Em virtude de todos os problemas encontrado na aplicação orientada a objeto que resulta nos códigos repetidos e emaranhados, encontrados na classe ClienteDAO, no momento da auditoria, o sistema orientado a aspecto trás beneficio para o desenvolvedor caso queira adicionar futuras modificações na auditoria. Neste caso se o programador queria implementar o aspecto para esse interesse sistêmico ele deve criar um aspecto com seus ponto de atuação e adendos.

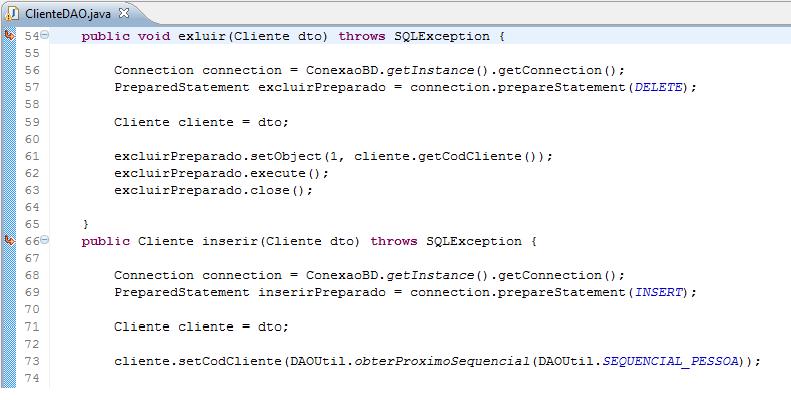
**FIGURA 24** – UML do aspecto.

No código orientado aspecto, foi criado um aspecto chamado Logger com dois pontos de atuação (pointcuts), o auditoria1 e o auditoria2. No primeiro, o pointcuts intercepta no momento de execução do método excluir na classe ClienteDAO, que tenha como retorno public void e que passe como parâmetro um cliente. No seu adendo, que será executado depois da execução do ponto de junção, passa-se como parâmetro o Cliente c. Dentro do adendo será capturado o nome da classe e o nome do método para os atributos classe e método. Instancia-se o Log auditor e passando os atributos classe e método como parâmetro junto ao método toString do Cliente c.

**LISTAGEM 3** – Pointcut auditoria1 do Aspecto Logger.

Observando o segundo ponto de atuação, auditoria2, haverá uma pequena diferença entre o primeiro ponto de atuação. Esta distinção é apenas o método que será o inserir em vez do excluir, e o tipo de retorno será um Cliente em vez de public void.   
  
**LISTAGEM 4** – Pointcut auditoria2 do Aspecto Logger.

Utilizando o aspecto Logger faz-se com que na classe ClienteDAO onde os códigos apresentavam-se espalhados e emaranhados foram totalmente extintos como pode-se vê na listagem abaixo.

**LISTAGEM 5** – Classe ClienteDAO sem os códigos espalhados e emaranhados.

A parti da inserção dos aspectos chega ao um resultado satisfatório de duplicação de códigos, permitindo uma melhor organização no código, obtendo uma manutenção mais simples e facilitada, e com uma maior reutilização de códigos.

**CONCLUSÃO**

A importância desse trabalho foi demonstrar as limitações encontradas em sistemas orientados a objeto como a decomposição dos interesses, causando os códigos emaranhados e espalhados, uma vez que tais interesses adicionais forçam a implementação de novos interesses, quebrando o conceito paradigma Orientado a Objeto, ocasionando a dificuldade na manutenção de modo que cada alteração terá de ser efetuada em todos os locais, a diminuição na reutilização dos códigos e dificuldade na compreensão do programa visto pelo desenvolvedor.

Certamente a idéia principal de uma aplicação orientada a aspecto é fazer com que todo mercado de software progrida com prazos mais curtos, proporcionando uma nova filosofia de desenvolvimento tendo como objetivo, tornar a codificação de sistema mais simples e flexível, resultando uma produtividade de software muito mais ágil.

Como foi visto na aplicação, após a criação dos aspectos, as linhas repetitivas (espalhamento de códigos) foram sendo extinguidas, ate o ponto de ser retiradas por completo e implementadas uma única vez dentro dos aspectos. Tornando os benefícios claros, a diminuição de linhas de código e uma melhor modularização do sistema eliminando com os códigos emaranhados.

Reusabilidade e manutenibilidade, na qual o código fica limpo, enxuto e mais modularizado. Devida à utilização da POA, cada módulo será responsável por certa funcionalidade do sistema, ainda a estrutura do sistema ser alterada, acrescida sem ser preciso mexer no código já implementado possibilitada pela escalabilidade.

É possível dizer, ao que foi exposto nos capítulos anteriores, as chances dessas abordagens juntas, consagrarão a POA no desenvolvimento de sistemas melhor programados especificando separadamente os interesses transversais dos interesses funcionais, bastando apenas que a comunidade desenvolvedora e a indústria de software, e principalmente, nas faculdades e universidades começarem a desenvolver projetos e até mesmo comercializar.

Portanto é justo comentar que a Orientação a Aspectos está longe de ser o paradigma ideal, e acredita-se que surgirão outros paradigmas capazes de melhor abstrair / expressar soluções, assim, possibilitando criar novas soluções. Mas desde já o paradigma orientado a aspecto é um grande salto rumo ao paradigma ideal.

**REFERÊNCIAS**

DAVID, Marcio Frayze. **Programação Orientada a Objetos: uma introdução.** Disponível em: <http://www.guiadohardware.net/artigos/programacao-orientada-objetos/>. Acesso em: 5 abr. 2008.

DO VALLE, Mauro. **Introdução à programação orientada a aspectos – Conceitos.** Disponível em:< http://infomediatv.terra.com.br/infomediatv/?section=11&article=14 > Acesso em: 14 mar. 2008, 16:00.

GOETTEN, Vicente Junior; WINCK, Diogo Vinícius. **AspectJ – Programação Orientada a Aspectos com Java**. 1. ed. São Paulo: Novatec, 2006.

KICZALES, Gregor et al. Aspect Oriented Programming. In: EUROPEN CONFERENCE ON OBJECT-ORIENTED PROGRAMMING, 1997, Palo Alto. **ECOOP 1997-AOP.pdf.** Finland: Springer - Verlag, 1997. p. 0 - 25.

LANGA, Sara Alvarez. **Tipos de programação.** Disponível em: <http://www.criarweb.com/artigos/707. php >. Acesso em: 01 abr. 2008.

LEITE, Mário; RAHAL JÚNIOR, Nelson Abu Sanra. **Programação Orientada ao Objeto:** uma abordagem didática. Disponível em: <http://www.ccuec.unicamp.br/revista/infotec/artigos/leite\_rahal.html>. Acesso em: 5 abr. 2008.

MARTIM, Robert. **O que é programacao orientada a objeto.** Disponível em: <http://www.linhadecodigo.com.br/ArtigoImpressao.aspx?id=851>. Acesso em: 02 maio 2008.

MONTEIRO, Elaine da Silva; PIVETA, Eduardo Kessler. **Programação Orientada a Aspectos em AspectJ**. In: ENCONTRO DE ESTUDANTES DE INFORMÁTICA DO TOCANTINS, 4., 2003, Palmas. Aspectj-encoinfo2003.pdf. Palmas: ., 2003. p. 313 - 322.

RESENDE, Antonio Maria Pereira de; SILVA, Claudiney Calixto da. **Programação Orientada a Aspectos em Java – Desenvolvimento de Software Orientado a Aspectos**. 1 ed. Rio de janeiro : Brasport, 2005.

SOMMERVILLE, Ian. **Engenharia de Software, 8ª.** Tradução:  
Selma Shin Shimizu Melnikoff, Reginaldo Arakaki, Edílson de Andrade Barbosa. São Paulo: Pearson Addison-Wesley, [2007]. 5141 p.

**APÊNDICE**

**APÊNDICE A – Aspecto Logger**

package br.cesmac.aspecto;

import java.sql.SQLException;

import org.aspectj.lang.Signature;

import br.cesmac.util.Log;

import br.cesmac.modelos.vo.Cliente;

// Aspecto Logger

public aspect Logger {

// Ponto de Atuação auditoria1

pointcut auditoria1():

execution(public void \*..ClienteDAO.exluir(\*..Cliente));

// Adendo que será executado depois do Ponto de Atuação auditoria1

after(Cliente c): auditoria1() && args(c){

Signature sig = thisJoinPointStaticPart.getSignature();

String classe = sig.getDeclaringType().getName();

String metodo = sig.getName();

Log auditor = new Log(classe, metodo,c.toString());

try {

auditor.registrar( );

} catch (SQLException e) {

System.out.println("não pode auditar! " + c.toString() );

e.printStackTrace();

}

}

// Ponto de Atuação auditoria2

pointcut auditoria2():

execution(public Cliente \*..ClienteDAO.inserir(\*..Cliente));

// Adendo que será executado depois do Ponto de Atuação auditoria2

after(Cliente c): auditoria2() && args(c){

Signature sig = thisJoinPointStaticPart.getSignature();

String classe = sig.getDeclaringType().getName();

String metodo = sig.getName();

Log auditor = new Log(classe,metodo, c.toString());

try {

auditor.registrar();

} catch (SQLException e) {

System.out.println("não pode auditar! " + c.toString() );

e.printStackTrace();

}

}}

**APÊNDICE B – Classe Cliente**

**package** br.cesmac.modelos.vo;

**import** java.util.Date;

**public** **class** Cliente {

**private** String nome, identidade, cpf, endereco, telelefoFixo, celular;

**private** String sexo;

**private** Integer codCliente, codTitular;

**private** Date dataNasc;

**public** Cliente(String nome, String identidade, String cpf, String endereco,

String telelefoneFixo, String celular, String sexo, Integer codCliente,

Integer codTitular, Date dataNasc) {

**super**();

**this**.nome = nome;

**this**.identidade = identidade;

**this**.cpf = cpf;

**this**.endereco = endereco;

**this**.telelefoFixo = telelefoneFixo;

**this**.celular = celular;

**this**.sexo = sexo;

**this**.codCliente = codCliente;

**this**.codTitular = codTitular;

**this**.dataNasc = dataNasc;

}

**public** Cliente() {

}

**public** String getNome() {

**return** nome;

}

**public** **void** setNome(String nome) {

**this**.nome = nome;

}

**public** String getIdentidade() {

**return** identidade;

}

**public** **void** setIdentidade(String identidade) {

**this**.identidade = identidade;

}

**public** String getCpf() {

**return** cpf;

}

**public** **void** setCpf(String cpf) {

**this**.cpf = cpf;

}

**public** String getEndereco() {

**return** endereco;

}

**public** **void** setEndereco(String endereco) {

**this**.endereco = endereco;

}

**public** String getTelelefoFixo() {

**return** telelefoFixo;

}

**public** **void** setTelelefoFixo(String telelefoFixo) {

**this**.telelefoFixo = telelefoFixo;

}

**public** String getCelular() {

**return** celular;

}

**public** **void** setCelular(String celular) {

**this**.celular = celular;

}

**public** String getSexo() {

**return** sexo;

}

**public** **void** setSexo(String sexo) {

**this**.sexo = sexo;

}

**public** Integer getCodCliente() {

**return** codCliente;

}

**public** **void** setCodCliente(Integer codCliente) {

**this**.codCliente = codCliente;

}

**public** Integer getCodTitular() {

**return** codTitular;

}

**public** **void** setCodTitular(Integer codTitular) {

**this**.codTitular = codTitular;

}

**public** Date getDataNasc() {

**return** dataNasc;

}

**public** **void** setDataNasc(Date dataNasc) {

**this**.dataNasc = dataNasc;

}

@Override

**public** **int** hashCode() {

**final** **int** prime = 31;

**int** result = 1;

result = prime \* result + codCliente;

**return** result;

}

@Override

**public** **boolean** equals(Object obj) {

**if** (**this** == obj)

**return** **true**;

**if** (obj == **null**)

**return** **false**;

**if** (getClass() != obj.getClass())

**return** **false**;

**final** Cliente other = (Cliente) obj;

**if** (codCliente != other.codCliente)

**return** **false**;

**return** **true**;

}

@Override

**public** String toString() {

**return** "Cod.: "+codCliente+" - "+nome+" - Cod. Titular: "+codTitular;

}

}

**APÊNDICE C – Classe ClienteDAO**

package br.cesmac.modelos.dao;

import java.sql.Connection;

import java.sql.Date;

import java.sql.PreparedStatement;

import java.sql.ResultSet;

import java.sql.SQLException;

import java.sql.Statement;

import java.util.ArrayList;

import java.util.List;

import br.cesmac.modelos.vo.Cliente;

import br.cesmac.util.ConexaoBD;

import br.cesmac.util.Log;

public class ClienteDAO {

public static final String QUERY\_LISTA\_TODOS\_ORDENAR\_POR\_NOME = "select \* from pessoa order by lower(nm\_nome)";

private static final String UPDATE = "update pessoa set cd\_titular = ?, set nm\_nome = ?," +

"set dt\_nascimento = ?, set nm\_sexo = ?, set nm\_identidade = ?, set nm\_cpf = ?, set tx\_endereco = ?," +

"set nm\_telefone\_fixo = ?, set nm\_telefone\_movel = ? where cd\_pessoa = ?";

private static final String DELETE = "delete from pessoa where cd\_pessoa = ?";

private static final String INSERT = "insert into pessoa (cd\_pessoa,cd\_titular,nm\_nome," +

"dt\_nascimento,nm\_sexo,nm\_identidade,nm\_cpf,tx\_endereco," +

"nm\_telefone\_fixo,nm\_telefone\_movel) values ( ?, ?, ?, ?, ?, ?, ?, ?, ?, ?)";

private static final String RECUPERAR\_POR\_ID = " select \* from pessoa where cd\_pessoa = ?";

public Cliente atualizar(Cliente dto) throws SQLException {

Connection connection = ConexaoBD.getInstance().getConnection();

PreparedStatement atualizarPreparado = connection.prepareStatement(UPDATE);

Cliente cliente = dto;

atualizarPreparado.setObject(1, cliente.getCodTitular());

atualizarPreparado.setObject(2, cliente.getNome());

Date data = new Date(cliente.getDataNasc().getTime());

atualizarPreparado.setObject(3, data);

atualizarPreparado.setObject(4, cliente.getSexo());

atualizarPreparado.setObject(5, cliente.getIdentidade());

atualizarPreparado.setObject(6, cliente.getCpf());

atualizarPreparado.setObject(7, cliente.getEndereco());

atualizarPreparado.setObject(8, cliente.getTelelefoFixo());

atualizarPreparado.setObject(9, cliente.getCelular());

atualizarPreparado.setObject(10, cliente.getCodCliente());

atualizarPreparado.executeUpdate();

atualizarPreparado.close();

return cliente;

}

public void exluir(Cliente dto) throws SQLException {

Connection connection = ConexaoBD.getInstance().getConnection();

PreparedStatement excluirPreparado = connection.prepareStatement(DELETE);

Cliente cliente = dto;

excluirPreparado.setObject(1, cliente.getCodCliente());

excluirPreparado.execute();

excluirPreparado.close();

}

public Cliente inserir(Cliente dto) throws SQLException {

Connection connection = ConexaoBD.getInstance().getConnection();

PreparedStatement inserirPreparado = connection.prepareStatement(INSERT);

Cliente cliente = dto;

cliente.setCodCliente(DAOUtil.obterProximoSequencial(DAOUtil.SEQUENCIAL\_PESSOA));

inserirPreparado.setObject(1, cliente.getCodCliente());

inserirPreparado.setObject(2, cliente.getCodTitular());

inserirPreparado.setObject(3, cliente.getNome());

Date data = new Date(cliente.getDataNasc().getTime());

inserirPreparado.setObject(4, data);

inserirPreparado.setObject(5, cliente.getSexo());

inserirPreparado.setObject(6, cliente.getIdentidade());

inserirPreparado.setObject(7, cliente.getCpf());

inserirPreparado.setObject(8, cliente.getEndereco());

inserirPreparado.setObject(9, cliente.getTelelefoFixo());

inserirPreparado.setObject(10, cliente.getCelular());

inserirPreparado.executeUpdate();

inserirPreparado.close();

return cliente;

}

public List<Cliente> listar(String query) throws SQLException {

Connection connection = ConexaoBD.getInstance().getConnection();

Statement stm = connection.createStatement();

ResultSet rs = stm.executeQuery(query);

List<Cliente> lista = new ArrayList<Cliente>();

while(rs.next()){

Cliente cliente = new Cliente();

cliente.setCelular(rs.getString("nm\_telefone\_movel"));

cliente.setCodCliente(rs.getInt("cd\_pessoa"));

cliente.setCodTitular(rs.getInt("cd\_titular"));

cliente.setCpf(rs.getString("nm\_cpf"));

cliente.setDataNasc(rs.getDate("dt\_nascimento"));

cliente.setEndereco(rs.getString("tx\_endereco"));

cliente.setIdentidade(rs.getString("nm\_identidade"));

cliente.setNome(rs.getString("nm\_nome"));

cliente.setSexo(rs.getString("nm\_sexo"));

cliente.setTelelefoFixo(rs.getString("nm\_telefone\_fixo"));

lista.add(cliente);

}

rs.close();

stm.close();

return lista;

}

public Cliente recuperarPorId(Object id) throws SQLException {

Connection connection = ConexaoBD.getInstance().getConnection();

PreparedStatement recuperarPorIDPreparado = connection.prepareStatement(RECUPERAR\_POR\_ID);

recuperarPorIDPreparado.setObject(1, id);

ResultSet rs = recuperarPorIDPreparado.executeQuery();

Cliente retorno = null;

while(rs.next()){

Cliente cliente = new Cliente();

cliente.setCelular(rs.getString("nm\_telefone\_movel"));

cliente.setCodCliente(rs.getInt("cd\_pessoa"));

cliente.setCodTitular(rs.getInt("cd\_titular"));

cliente.setCpf(rs.getString("nm\_cpf"));

cliente.setDataNasc(rs.getDate("dt\_nascimento"));

cliente.setEndereco(rs.getString("tx\_endereco"));

cliente.setIdentidade(rs.getString("nm\_identidade"));

cliente.setNome(rs.getString("nm\_nome"));

cliente.setSexo(rs.getString("nm\_sexo"));

cliente.setTelelefoFixo(rs.getString("nm\_telefone\_fixo"));

retorno = cliente;

}

return retorno;

}

}

**APÊNDICE D – Classe ClienteBO**

package br.cesmac.modelos.bo;

import java.text.SimpleDateFormat;

import java.util.Date;

import java.util.Formatter;

import java.util.List;

import javax.swing.JOptionPane;

import br.cesmac.modelos.dao.ClienteDAO;

import br.cesmac.modelos.vo.Cliente;

public class ClienteBO {

public Cliente inserir(Cliente cliente) throws Exception{

if(cliente.getNome() == null){

throw new Exception("O Campo nome não pode ser vazio.");

}else{

if(cliente.getNome().trim().equals("")){

throw new Exception("O Campo nome não pode ser vazio.");

}

}

if(cliente.getDataNasc() == null){

throw new Exception("O Campo Data de nascimento não pode ser vazio.");

}

Date dataAtual = new Date();

int comparacao = dataAtual.compareTo(cliente.getDataNasc());

if(comparacao < 0){

throw new Exception("Data de nascimento não pode ser superior a data atual.");

}

if(cliente.getIdentidade() == null){

throw new Exception("O campo identidade não pode ser vazio.");

}else{

if(cliente.getIdentidade().trim().equals("")){

throw new Exception("O campo identidade não pode ser vazio.");

}

}

if(cliente.getCpf() == null){

throw new Exception("O campo CPF não pode ser vazio.");

}else{

if(cliente.getCpf().trim().equals("")){

throw new Exception("O campo CPF não pode ser vazio.");

}

}

ClienteDAO clienteDAO = new ClienteDAO();

Cliente clienteInserido = clienteDAO.inserir(cliente);

return clienteInserido;

}

public Cliente atualizar(Cliente cliente)throws Exception{

if(cliente.getNome() == null){

throw new Exception("O Campo nome não pode ser vazio.");

}else{

if(cliente.getNome().trim().equals("")){

throw new Exception("O Campo nome não pode ser vazio.");

}

}

if(cliente.getDataNasc() == null){

throw new Exception("O Campo Data de nascimento não pode ser vazio.");

}

Date dataAtual = new Date();

int comparacao = dataAtual.compareTo(cliente.getDataNasc());

if(comparacao > 0){

throw new Exception("Data de nascimento não pode ser superior a data atual.");

}

if(cliente.getIdentidade() == null){

throw new Exception("O campo identidade não pode ser vazio.");

}else{

if(cliente.getIdentidade().trim().equals("")){

throw new Exception("O campo identidade não pode ser vazio.");

}

}

if(cliente.getCpf() == null){

throw new Exception("O campo CPF não pode ser vazio.");

}else{

if(cliente.getCpf().trim().equals("")){

throw new Exception("O campo CPF não pode ser vazio.");

}

}

ClienteDAO clienteDAO = new ClienteDAO();

Cliente clienteAtualizado = clienteDAO.atualizar(cliente);

return clienteAtualizado;

}

public void excluir(Cliente cliente)throws Exception {

ClienteDAO clienteDAO = new ClienteDAO();

clienteDAO.exluir(cliente);

}

public List<Cliente> listar(String query)throws Exception{

ClienteDAO clienteDAO = new ClienteDAO();

List<Cliente> clienteListado = clienteDAO.listar(query);

return clienteListado;

}

public Cliente recuperarPorID(Cliente id)throws Exception{

ClienteDAO clienteDAO = new ClienteDAO();

Cliente clienteRecuperado = clienteDAO.recuperarPorId(id);

return clienteRecuperado;

}

}

**APÊNDICE E – Classe DialogEditarCliente**

package br.cesmac.ui.dialogs;

import javax.swing.JFrame;

import javax.swing.SwingUtilities;

import java.awt.event.WindowAdapter;

import java.awt.event.WindowEvent;

import javax.swing.DefaultComboBoxModel;

import javax.swing.JButton;

import javax.swing.JComboBox;

import javax.swing.JDialog;

import javax.swing.JFormattedTextField;

import javax.swing.JLabel;

import javax.swing.JPanel;

import javax.swing.JScrollPane;

import javax.swing.JTextArea;

import javax.swing.JTextField;

import javax.swing.border.EtchedBorder;

import javax.swing.border.TitledBorder;

import br.cesmac.modelos.vo.Cliente;

public class DialogEditarCliente extends JDialog {

private JButton cadastrarButton;

private JFormattedTextField formattedTextFieldCelular;

private JLabel celularLabel;

private JFormattedTextField formattedTextFieldTelefone;

private JLabel telefoneLabel;

private JPanel panel\_3;

private JTextField textFieldCidade;

private JLabel cidadeLabel;

private JComboBox comboBoxEstado;

private JLabel estadoLabel;

private JTextField textFieldBairro;

private JLabel bairroLabel;

private JTextArea textAreaEndereco;

private JScrollPane scrollPane\_1;

private JLabel endLabel;

private JPanel panel\_2;

private JFormattedTextField formattedTextFieldCpf;

private JLabel cpfLabel;

private JTextField textFieldRg;

private JLabel rgLabel;

private JTextField textFieldNome;

private JLabel nomeLabel;

private JPanel panel\_1;

private Cliente cliente;

public DialogEditarCliente(Cliente cliente) {

super();

this.cliente = cliente;

setBounds(100, 100, 500, 408);

initComponents();

//

}

private void initComponents() {

getContentPane().setLayout(null);

setTitle("Editar Cliente");

{

panel\_1 = new JPanel();

panel\_1.setBorder(new TitledBorder(null, "Dados Pessoais", TitledBorder.DEFAULT\_JUSTIFICATION, TitledBorder.DEFAULT\_POSITION, null, null));

panel\_1.setLayout(null);

panel\_1.setBounds(10, 10, 468, 97);

getContentPane().add(panel\_1);

{

nomeLabel = new JLabel();

nomeLabel.setText("Nome:");

nomeLabel.setBounds(10, 30, 36, 16);

panel\_1.add(nomeLabel);

}

{

textFieldNome = new JTextField();

textFieldNome.setBounds(63, 28, 395, 20);

panel\_1.add(textFieldNome);

}

{

rgLabel = new JLabel();

rgLabel.setText("RG:");

rgLabel.setBounds(10, 68, 36, 16);

panel\_1.add(rgLabel);

}

{

textFieldRg = new JTextField();

textFieldRg.setBounds(64, 66, 144, 20);

panel\_1.add(textFieldRg);

}

{

cpfLabel = new JLabel();

cpfLabel.setText("CPF:");

cpfLabel.setBounds(214, 68, 36, 16);

panel\_1.add(cpfLabel);

}

{

formattedTextFieldCpf = new JFormattedTextField();

formattedTextFieldCpf.setBounds(256, 66, 110, 20);

panel\_1.add(formattedTextFieldCpf);

}

}

{

panel\_2 = new JPanel();

panel\_2.setBorder(new TitledBorder(null, "Endereco", TitledBorder.DEFAULT\_JUSTIFICATION, TitledBorder.DEFAULT\_POSITION, null, null));

panel\_2.setLayout(null);

panel\_2.setBounds(10, 126, 468, 148);

getContentPane().add(panel\_2);

{

endLabel = new JLabel();

endLabel.setText("End.:");

endLabel.setBounds(10, 23, 35, 16);

panel\_2.add(endLabel);

}

{

scrollPane\_1 = new JScrollPane();

scrollPane\_1.setBounds(60, 19, 398, 48);

panel\_2.add(scrollPane\_1);

{

textAreaEndereco = new JTextArea();

scrollPane\_1.setViewportView(textAreaEndereco);

}

}

{

bairroLabel = new JLabel();

bairroLabel.setText("Bairro:");

bairroLabel.setBounds(10, 85, 38, 16);

panel\_2.add(bairroLabel);

}

{

textFieldBairro = new JTextField();

textFieldBairro.setBounds(60, 83, 192, 20);

panel\_2.add(textFieldBairro);

}

{

estadoLabel = new JLabel();

estadoLabel.setText("Estado:");

estadoLabel.setBounds(258, 85, 42, 16);

panel\_2.add(estadoLabel);

}

{

comboBoxEstado = new JComboBox();

comboBoxEstado.setModel(new DefaultComboBoxModel(new String[] {"AL", "PE", "PB", "SE", "BA", "RN", "MA", "TO"}));

comboBoxEstado.setBounds(314, 81, 46, 25);

panel\_2.add(comboBoxEstado);

}

{

cidadeLabel = new JLabel();

cidadeLabel.setText("Cidade:");

cidadeLabel.setBounds(10, 115, 46, 16);

panel\_2.add(cidadeLabel);

}

{

textFieldCidade = new JTextField();

textFieldCidade.setBounds(60, 113, 192, 20);

panel\_2.add(textFieldCidade);

}

}

{

panel\_3 = new JPanel();

panel\_3.setBorder(new TitledBorder(null, "Contato", TitledBorder.DEFAULT\_JUSTIFICATION, TitledBorder.DEFAULT\_POSITION, null, null));

panel\_3.setLayout(null);

panel\_3.setBounds(10, 282, 186, 82);

getContentPane().add(panel\_3);

{

telefoneLabel = new JLabel();

telefoneLabel.setText("Telefone:");

telefoneLabel.setBounds(10, 22, 52, 16);

panel\_3.add(telefoneLabel);

}

{

formattedTextFieldTelefone = new JFormattedTextField();

formattedTextFieldTelefone.setBounds(68, 20, 108, 20);

panel\_3.add(formattedTextFieldTelefone);

}

{

celularLabel = new JLabel();

celularLabel.setText("Celular:");

celularLabel.setBounds(10, 56, 52, 16);

panel\_3.add(celularLabel);

}

{

formattedTextFieldCelular = new JFormattedTextField();

formattedTextFieldCelular.setBounds(68, 54, 108, 20);

panel\_3.add(formattedTextFieldCelular);

}

}

{

cadastrarButton = new JButton();

cadastrarButton.setText("Cadastrar");

cadastrarButton.setBounds(372, 338, 106, 26);

getContentPane().add(cadastrarButton);

}

}

}

**APÊNDICE F – Classe DialogVisualizarCliente**

package br.cesmac.ui.dialogs;

import java.awt.event.WindowAdapter;

import java.awt.event.WindowEvent;

import java.sql.Date;

import java.text.SimpleDateFormat;

import javax.swing.JDialog;

import javax.swing.JFormattedTextField;

import javax.swing.JLabel;

import javax.swing.JPanel;

import javax.swing.JTextField;

import javax.swing.border.EtchedBorder;

import javax.swing.border.TitledBorder;

import br.cesmac.modelos.vo.Cliente;

import sun.text.resources.FormatData;

public class DialogVisualizarCliente extends JDialog {

private JTextField textFieldCodTitular;

private JLabel codigoTitularLabel;

private JPanel panel\_3;

private JFormattedTextField formattedTextFieldCelular;

private JFormattedTextField formattedTextFieldResidencial;

private JLabel celularLabel;

private JLabel residencialLabel;

private JPanel panel\_2;

private JTextField textFieldSexo;

private JTextField textFieldEndereco;

private JLabel enderecoLabel;

private JFormattedTextField formattedTextFieldDataNasc;

private JLabel dataNascLabel;

private JLabel sexoLabel;

private JFormattedTextField formattedTextFieldCPF;

private JLabel cpfLabel;

private JTextField textFieldId;

private JLabel indentidadeLabel;

private JTextField textFieldNome;

private JLabel nomeLabel;

private JPanel panel\_1;

private Cliente cliente;

public DialogVisualizarCliente(Cliente cliente) {

super();

this.cliente = cliente;

setBounds(100, 100, 500, 375);

initComponents();

preencherCampos();

//

}

private void preencherCampos() {

textFieldCodTitular.setText(cliente.getCodTitular().toString());

textFieldEndereco.setText(cliente.getEndereco());

textFieldId.setText(cliente.getIdentidade());

textFieldNome.setText(cliente.getNome());

textFieldSexo.setText(cliente.getSexo());

formattedTextFieldCelular.setText(cliente.getCelular());

formattedTextFieldCPF.setText(cliente.getCpf());

SimpleDateFormat dateFormat = new SimpleDateFormat("dd/MM/yyyy");

formattedTextFieldDataNasc.setText(dateFormat.format(cliente.getDataNasc()));

formattedTextFieldResidencial.setText(cliente.getTelelefoFixo());

}

private void initComponents() {

getContentPane().setLayout(null);

setTitle("Cliente");

{

panel\_1 = new JPanel();

panel\_1.setBorder(new TitledBorder(null, "Dados Pessoais", TitledBorder.DEFAULT\_JUSTIFICATION, TitledBorder.DEFAULT\_POSITION, null, null));

panel\_1.setLayout(null);

panel\_1.setBounds(10, 23, 465, 183);

getContentPane().add(panel\_1);

{

nomeLabel = new JLabel();

nomeLabel.setText("Nome:");

nomeLabel.setBounds(10, 35, 66, 16);

panel\_1.add(nomeLabel);

}

{

textFieldNome = new JTextField();

textFieldNome.setEditable(false);

textFieldNome.setBounds(100, 33, 338, 20);

panel\_1.add(textFieldNome);

}

{

indentidadeLabel = new JLabel();

indentidadeLabel.setText("Identidade:");

indentidadeLabel.setBounds(10, 76, 66, 16);

panel\_1.add(indentidadeLabel);

}

{

textFieldId = new JTextField();

textFieldId.setEditable(false);

textFieldId.setBounds(100, 74, 135, 20);

panel\_1.add(textFieldId);

}

{

cpfLabel = new JLabel();

cpfLabel.setText("CPF:");

cpfLabel.setBounds(241, 76, 66, 16);

panel\_1.add(cpfLabel);

}

{

formattedTextFieldCPF = new JFormattedTextField();

formattedTextFieldCPF.setEditable(false);

formattedTextFieldCPF.setBounds(274, 74, 164, 20);

panel\_1.add(formattedTextFieldCPF);

}

{

sexoLabel = new JLabel();

sexoLabel.setText("Sexo:");

sexoLabel.setBounds(228, 113, 39, 16);

panel\_1.add(sexoLabel);

}

{

dataNascLabel = new JLabel();

dataNascLabel.setText("Data Nasc.:");

dataNascLabel.setBounds(10, 113, 66, 16);

panel\_1.add(dataNascLabel);

}

{

formattedTextFieldDataNasc = new JFormattedTextField();

formattedTextFieldDataNasc.setEditable(false);

formattedTextFieldDataNasc.setBounds(100, 111, 86, 20);

panel\_1.add(formattedTextFieldDataNasc);

}

{

enderecoLabel = new JLabel();

enderecoLabel.setText("Endereco:");

enderecoLabel.setBounds(10, 149, 66, 16);

panel\_1.add(enderecoLabel);

}

{

textFieldEndereco = new JTextField();

textFieldEndereco.setEditable(false);

textFieldEndereco.setBounds(100, 147, 338, 20);

panel\_1.add(textFieldEndereco);

}

{

textFieldSexo = new JTextField();

textFieldSexo.setEditable(false);

textFieldSexo.setBounds(274, 111, 87, 20);

panel\_1.add(textFieldSexo);

}

}

{

panel\_2 = new JPanel();

panel\_2.setBorder(new TitledBorder(null, "Telefone", TitledBorder.DEFAULT\_JUSTIFICATION, TitledBorder.DEFAULT\_POSITION, null, null));

panel\_2.setLayout(null);

panel\_2.setBounds(10, 215, 208, 95);

getContentPane().add(panel\_2);

{

residencialLabel = new JLabel();

residencialLabel.setText("Residencial:");

residencialLabel.setBounds(10, 24, 77, 16);

panel\_2.add(residencialLabel);

}

{

celularLabel = new JLabel();

celularLabel.setText("Celular:");

celularLabel.setBounds(10, 58, 66, 16);

panel\_2.add(celularLabel);

}

{

formattedTextFieldResidencial = new JFormattedTextField();

formattedTextFieldResidencial.setEditable(false);

formattedTextFieldResidencial.setBounds(83, 22, 115, 20);

panel\_2.add(formattedTextFieldResidencial);

}

{

formattedTextFieldCelular = new JFormattedTextField();

formattedTextFieldCelular.setEditable(false);

formattedTextFieldCelular.setBounds(83, 56, 115, 20);

panel\_2.add(formattedTextFieldCelular);

}

}

{

panel\_3 = new JPanel();

panel\_3.setBorder(new TitledBorder(null, "Dependente", TitledBorder.DEFAULT\_JUSTIFICATION, TitledBorder.DEFAULT\_POSITION, null, null));

panel\_3.setLayout(null);

panel\_3.setBounds(256, 215, 219, 50);

getContentPane().add(panel\_3);

{

codigoTitularLabel = new JLabel();

codigoTitularLabel.setText("Codigo Titular:");

codigoTitularLabel.setBounds(10, 24, 90, 16);

panel\_3.add(codigoTitularLabel);

}

{

textFieldCodTitular = new JTextField();

textFieldCodTitular.setEditable(false);

textFieldCodTitular.setBounds(106, 22, 87, 20);

panel\_3.add(textFieldCodTitular);

}

}

}

}

**APÊNDICE G – Classe FramePrincipal**

package br.cesmac.ui.frame.external;

import javax.swing.WindowConstants;

import org.jdesktop.application.Application;

import javax.swing.SwingUtilities;

import java.awt.BorderLayout;

import java.awt.event.ActionEvent;

import java.awt.event.ActionListener;

import javax.swing.JButton;

import javax.swing.JDesktopPane;

import javax.swing.JFrame;

import javax.swing.JInternalFrame;

import javax.swing.JMenu;

import javax.swing.JMenuBar;

import javax.swing.JMenuItem;

import javax.swing.JOptionPane;

import javax.swing.JSeparator;

import javax.swing.JToolBar;

import br.cesmac.ui.frame.internal.ClienteProvedor;

import br.cesmac.ui.frame.internal.InternalCliente;

import br.cesmac.ui.frame.internal.InternalGenerico;

public class FramePrincipal extends JFrame {

private JButton buttonCliente;

private JToolBar toolBar;

private JMenuItem listarMenuItem\_1;

private JMenuItem listarMenuItem;

private JDesktopPane desktopPane;

private JMenuItem cadastrarMenuItem\_1;

private JMenuItem cadastrarMenuItem;

private JMenu clienteMenu;

private JMenuItem sairMenuItem;

private JMenu arquivosMenu;

private JMenuBar menuBar;

private static FramePrincipal framePrincipal;

public static FramePrincipal getInstance(){

if(framePrincipal == null){

framePrincipal = new FramePrincipal();

}

return framePrincipal;

}

public FramePrincipal() {

super();

setBounds(100, 100, 500, 375);

setDefaultCloseOperation(JFrame.EXIT\_ON\_CLOSE);

initComponents();

//

}

private void initComponents() {

setTitle("Sistema Cadastra Cliente");

{

menuBar = new JMenuBar();

setJMenuBar(menuBar);

{

arquivosMenu = new JMenu();

arquivosMenu.setText("Arquivos");

menuBar.add(arquivosMenu);

{

sairMenuItem = new JMenuItem();

sairMenuItem.addActionListener(new ActionListener() {

public void actionPerformed(final ActionEvent e) {

int escolha = JOptionPane.showConfirmDialog(null, "Deseja mesmo sair?", "Sair", JOptionPane.YES\_NO\_OPTION);

if(escolha == 0){

System.exit(0);

}

}

});

sairMenuItem.setText("Sair");

arquivosMenu.add(sairMenuItem);

}

}

{

clienteMenu = new JMenu();

clienteMenu.setText("Cliente");

menuBar.add(clienteMenu);

{

cadastrarMenuItem = new JMenuItem();

cadastrarMenuItem.addActionListener(new ActionListener() {

public void actionPerformed(final ActionEvent e) {

InternalCliente cliente = new InternalCliente();

desktopPane.add(cliente);

desktopPane.getDesktopManager().activateFrame(cliente);

cliente.setVisible(true);

}

});

cadastrarMenuItem.setText("Cadastrar");

clienteMenu.add(cadastrarMenuItem);

}

{

clienteMenu.addSeparator();

}

{

listarMenuItem = new JMenuItem();

listarMenuItem.addActionListener(new ActionListener() {

public void actionPerformed(final ActionEvent arg0) {

InternalGenerico listaCliente = new InternalGenerico(new ClienteProvedor());

desktopPane.add(listaCliente);

desktopPane.getDesktopManager().activateFrame(listaCliente);

listaCliente.setVisible(true);

}

});

listarMenuItem.setText("Gerenciar");

clienteMenu.add(listarMenuItem);

}

{

desktopPane = new JDesktopPane();

desktopPane.setLayout(null);

getContentPane().add(desktopPane, BorderLayout.CENTER);

}

{

toolBar = new JToolBar();

getContentPane().add(toolBar, BorderLayout.NORTH);

toolBar.addSeparator();

{

buttonCliente = new JButton();

buttonCliente.setText("Cliente");

buttonCliente.addActionListener(new ActionListener() {

public void actionPerformed(final ActionEvent e) {

InternalCliente cliente = new InternalCliente();

desktopPane.add(cliente);

desktopPane.getDesktopManager().activateFrame(cliente);

cliente.setVisible(true);

}

});

buttonCliente.setToolTipText("Cadastrar Cliente");

toolBar.add(buttonCliente);

}

}

}

}

Application.getInstance().getContext().getResourceMap(getClass()).injectComponents(getContentPane());

}

public void adicionarFrameDesktop(JInternalFrame internalFrame){

desktopPane.add(internalFrame);

desktopPane.getDesktopManager().activateFrame(internalFrame);

internalFrame.setVisible(true);

}

}

**APÊNDICE H – Classe ClienteProvedor**

package br.cesmac.ui.frame.internal;

import java.util.ArrayList;

import java.util.List;

import javax.swing.JOptionPane;

import br.cesmac.modelos.bo.ClienteBO;

import br.cesmac.modelos.dao.ClienteDAO;

import br.cesmac.modelos.vo.Cliente;

import br.cesmac.ui.dialogs.DialogVisualizarCliente;

public class ClienteProvedor implements ProvedorConteudoAcoesFrameGenerico{

@Override

public void executarAcaoEditarDados(Object obj) {

}

@Override

public void executarAcaoExcluirDados(Object obj) {

int escolha = JOptionPane.showConfirmDialog(null, "Deseja mesmo excluir?", "Excluir", JOptionPane.YES\_NO\_OPTION);

if(escolha == 0){

Cliente cliente = (Cliente) obj;

ClienteBO clienteBO = new ClienteBO();

try {

clienteBO.excluir(cliente);

} catch (Exception e) {

e.printStackTrace();

JOptionPane.showMessageDialog(null,e.getMessage(), "Erro", JOptionPane.ERROR\_MESSAGE);

}

}

}

@Override

public void executarAcaoGerarRelatorio(Object obj) {

// TODO Auto-generated method stub

}

@Override

public void executarAcaoVisualizarDados(Object obj) {

Cliente cliente = (Cliente) obj;

DialogVisualizarCliente internalVisualizarCliente = new DialogVisualizarCliente(cliente);

internalVisualizarCliente.setVisible(true);

}

@Override

public Object[] getListaObjetos() {

ClienteBO clienteBO = new ClienteBO();

List<Cliente> listar = new ArrayList<Cliente>();

try {

listar = clienteBO.listar(ClienteDAO.QUERY\_LISTA\_TODOS\_ORDENAR\_POR\_NOME);

} catch (Exception e) {

e.printStackTrace();

}

return listar.toArray();

}

}

**APÊNDICE I – Classe InternalCliente**

package br.cesmac.ui.frame.internal;

import java.awt.event.ActionEvent;

import java.awt.event.ActionListener;

import java.sql.SQLException;

import java.text.DateFormat;

import java.text.ParseException;

import java.text.SimpleDateFormat;

import java.util.Date;

import javax.swing.ButtonGroup;

import javax.swing.JButton;

import javax.swing.JFormattedTextField;

import javax.swing.JInternalFrame;

import javax.swing.JLabel;

import javax.swing.JOptionPane;

import javax.swing.JPanel;

import javax.swing.JRadioButton;

import javax.swing.JSeparator;

import javax.swing.JTextField;

import javax.swing.border.EtchedBorder;

import javax.swing.border.TitledBorder;

import javax.swing.text.MaskFormatter;

import br.cesmac.modelos.bo.ClienteBO;

import br.cesmac.modelos.dao.DAOUtil;

import br.cesmac.modelos.vo.Cliente;

public class InternalCliente extends JInternalFrame {

private JButton cadastrarButton\_2;

private JButton cadastrarButton\_1;

private JSeparator separator;

private JTextField textFieldCodTitular;

private JLabel codigoTitularLabel;

private JPanel panel\_2;

private JButton cadastrarButton;

private JFormattedTextField formattedTextFieldCelular;

private JFormattedTextField formattedTextFieldResidencial;

private JLabel celularLabel;

private JLabel residencialLabel;

private JPanel panel\_1;

private JTextField textFieldEndereco;

private JLabel enderecoLabel;

private JFormattedTextField formattedTextFieldDataNasc;

private JLabel dataNascLabel;

private ButtonGroup buttonGroup = new ButtonGroup();

private JRadioButton femininoRadioButton;

private JRadioButton masculinoRadioButton;

private JLabel sexoLabel;

private JFormattedTextField formattedTextFieldCPF;

private JLabel cpfLabel;

private JTextField textFieldId;

private JLabel indentidadeLabel;

private JTextField textFieldNome;

private JLabel nomeLabel;

private JPanel panel;

Cliente cliente = new Cliente();

public InternalCliente() {

super();

this.setBounds(0, 0, 491, 422);

initComponents();

//

}

private void initComponents() {

setTitle("Cadastrar Cliente");

setIconifiable(true);

setClosable(true);

getContentPane().setLayout(null);

this.setPreferredSize(new java.awt.Dimension(491, 422));

{

panel = new JPanel();

panel.setBounds(10, 23, 465, 183);

panel.setBorder(new TitledBorder(null, "Dados Pessoais", TitledBorder.DEFAULT\_JUSTIFICATION, TitledBorder.DEFAULT\_POSITION, null, null));

panel.setLayout(null);

getContentPane().add(panel);

{

nomeLabel = new JLabel();

nomeLabel.setText("Nome:");

nomeLabel.setBounds(10, 35, 66, 16);

panel.add(nomeLabel);

}

{

textFieldNome = new JTextField();

textFieldNome.setBounds(100, 33, 338, 20);

panel.add(textFieldNome);

}

{

indentidadeLabel = new JLabel();

indentidadeLabel.setText("Identidade:");

indentidadeLabel.setBounds(10, 76, 66, 16);

panel.add(indentidadeLabel);

}

{

textFieldId = new JTextField();

textFieldId.setBounds(100, 74, 135, 20);

panel.add(textFieldId);

}

{

cpfLabel = new JLabel();

cpfLabel.setText("CPF:");

cpfLabel.setBounds(241, 76, 66, 16);

panel.add(cpfLabel);

}

{ MaskFormatter maskFormatterCPF = null;

try {

maskFormatterCPF = new MaskFormatter("###.###.###-##");

} catch (ParseException e) {

e.printStackTrace();

}

formattedTextFieldCPF = new JFormattedTextField(maskFormatterCPF);

formattedTextFieldCPF.setBounds(274, 74, 164, 20);

panel.add(formattedTextFieldCPF);

}

{

sexoLabel = new JLabel();

sexoLabel.setText("Sexo:");

sexoLabel.setBounds(228, 113, 39, 16);

panel.add(sexoLabel);

}

{

masculinoRadioButton = new JRadioButton();

masculinoRadioButton.setSelected(true);

buttonGroup.add(masculinoRadioButton);

masculinoRadioButton.setText("Masculino");

masculinoRadioButton.setBounds(266, 109, 83, 24);

panel.add(masculinoRadioButton);

masculinoRadioButton.setActionCommand("M");

}

{

femininoRadioButton = new JRadioButton();

buttonGroup.add(femininoRadioButton);

femininoRadioButton.setText("Feminino");

femininoRadioButton.setBounds(355, 109, 83, 24);

panel.add(femininoRadioButton);

femininoRadioButton.setActionCommand("F");

}

{

dataNascLabel = new JLabel();

dataNascLabel.setText("Data Nasc.:");

dataNascLabel.setBounds(10, 113, 66, 16);

panel.add(dataNascLabel);

}

{ MaskFormatter maskFormatterData= null;

try {

maskFormatterData = new MaskFormatter("##/##/####");

} catch (ParseException e) {

e.printStackTrace();

}

formattedTextFieldDataNasc = new JFormattedTextField(maskFormatterData);

formattedTextFieldDataNasc.setBounds(100, 111, 86, 20);

panel.add(formattedTextFieldDataNasc);

}

{

enderecoLabel = new JLabel();

enderecoLabel.setText("Endereco:");

enderecoLabel.setBounds(10, 149, 66, 16);

panel.add(enderecoLabel);

}

{

textFieldEndereco = new JTextField();

textFieldEndereco.setBounds(100, 147, 338, 20);

panel.add(textFieldEndereco);

}

}

{

panel\_1 = new JPanel();

panel\_1.setLayout(null);

panel\_1.setBounds(18, 224, 208, 95);

panel\_1.setBorder(new TitledBorder(null, "Telefone", TitledBorder.DEFAULT\_JUSTIFICATION, TitledBorder.DEFAULT\_POSITION, null, null));

getContentPane().add(panel\_1);

{

residencialLabel = new JLabel();

residencialLabel.setBounds(10, 24, 77, 16);

residencialLabel.setText("Residencial:");

panel\_1.add(residencialLabel);

}

{

celularLabel = new JLabel();

celularLabel.setText("Celular:");

celularLabel.setBounds(10, 58, 66, 16);

panel\_1.add(celularLabel);

}

MaskFormatter maskFormatterTelefone = null;

try {

maskFormatterTelefone = new MaskFormatter("####-####");

} catch (ParseException e) {

e.printStackTrace();

}

{

formattedTextFieldResidencial = new JFormattedTextField(maskFormatterTelefone);

formattedTextFieldResidencial.setBounds(83, 22, 115, 20);

panel\_1.add(formattedTextFieldResidencial);

}

{

formattedTextFieldCelular = new JFormattedTextField(maskFormatterTelefone);

formattedTextFieldCelular.setBounds(83, 56, 115, 20);

panel\_1.add(formattedTextFieldCelular);

}

}

{

panel\_2 = new JPanel();

panel\_2.setBorder(new TitledBorder(null, "Dependente", TitledBorder.DEFAULT\_JUSTIFICATION, TitledBorder.DEFAULT\_POSITION, null, null));

panel\_2.setLayout(null);

panel\_2.setBounds(256, 224, 219, 50);

getContentPane().add(panel\_2);

{

codigoTitularLabel = new JLabel();

codigoTitularLabel.setText("Codigo Titular:");

codigoTitularLabel.setBounds(10, 24, 90, 16);

panel\_2.add(codigoTitularLabel);

}

{

textFieldCodTitular = new JTextField();

textFieldCodTitular.setBounds(106, 22, 87, 20);

panel\_2.add(textFieldCodTitular);

}

}

{

separator = new JSeparator();

separator.setLayout(null);

separator.setBounds(0, 357, 490, 49);

getContentPane().add(separator);

{

cadastrarButton\_2 = new JButton();

cadastrarButton\_2.addActionListener(new ActionListener() {

public void actionPerformed(final ActionEvent e) {

atualizarObjeto();

executarBotao();

limparCampos();

}

});

cadastrarButton\_2.setText("Cadastrar");

cadastrarButton\_2.setBounds(374, 10, 106, 26);

separator.add(cadastrarButton\_2);

}

}

}

private void executarBotao() {

ClienteBO clienteBO = new ClienteBO();

try {

clienteBO.inserir(cliente);

JOptionPane.showMessageDialog(null, "Cliente cadastrado com sucesso.", "Cadastrado", JOptionPane.INFORMATION\_MESSAGE);

} catch (Exception e1) {

e1.printStackTrace();

JOptionPane.showMessageDialog(null, e1.getMessage(),"Erro", JOptionPane.ERROR\_MESSAGE);

}

}

private void atualizarObjeto() {

cliente.setCelular(formattedTextFieldCelular.getText());

try {

cliente.setCodCliente(DAOUtil.obterProximoSequencial(DAOUtil.SEQUENCIAL\_PESSOA));

} catch (SQLException e) {

e.printStackTrace();

}

if(!textFieldCodTitular.getText().trim().equals("")){

try {

cliente.setCodTitular(Integer.parseInt(textFieldCodTitular.getText()));

} catch (NumberFormatException e) {

JOptionPane.showMessageDialog(null, "Codigo do titular esta invalido.", "Erro", JOptionPane.ERROR\_MESSAGE);

}

}

cliente.setCpf(formattedTextFieldCPF.getText());

SimpleDateFormat dateFormat = new SimpleDateFormat("dd/MM/yyyy");

try {

cliente.setDataNasc(dateFormat.parse(formattedTextFieldDataNasc.getText()));

} catch (ParseException e) {

e.printStackTrace();

}

cliente.setEndereco(textFieldEndereco.getText());

cliente.setIdentidade(textFieldId.getText());

cliente.setNome(textFieldNome.getText());

cliente.setSexo(buttonGroup.getSelection().getActionCommand());

cliente.setTelelefoFixo(formattedTextFieldResidencial.getText());

}

private void limparCampos() {

textFieldCodTitular.setText(null);

textFieldEndereco.setText(null);

textFieldId.setText(null);

textFieldNome.setText(null);

formattedTextFieldCelular.setText(null);

formattedTextFieldCPF.setText(null);

formattedTextFieldDataNasc.setText(null);

formattedTextFieldResidencial.setText(null);

}

}

**APÊNDICE J – InternalGenerico**

package br.cesmac.ui.frame.internal;

import java.awt.event.ActionEvent;

import java.awt.event.ActionListener;

import javax.swing.DefaultComboBoxModel;

import javax.swing.JButton;

import javax.swing.JInternalFrame;

import javax.swing.JLabel;

import javax.swing.JList;

import javax.swing.JPanel;

import javax.swing.JScrollPane;

import javax.swing.JTextField;

import javax.swing.border.TitledBorder;

public class InternalGenerico extends JInternalFrame {

private JButton atualizarListaButton;

private JButton excluirClienteButton;

private JButton visualizarDadosButton;

private JTextField textFieldPesquisaCliente;

private JLabel clientLabel;

private JPanel panel\_1;

private JPanel panel;

private JList listCliente;

private JScrollPane scrollPane;

private ProvedorConteudoAcoesFrameGenerico provedor;

public InternalGenerico(ProvedorConteudoAcoesFrameGenerico provedor) {

super();

this.provedor = provedor;

this.setBounds(0, 0, 524, 503);

initComponents();

//

}

public InternalGenerico() {

}

private void initComponents() {

getContentPane().setLayout(null);

setClosable(true);

setIconifiable(true);

setTitle("Lista");

this.setPreferredSize(new java.awt.Dimension(524, 503));

{

scrollPane = new JScrollPane();

scrollPane.setBounds(22, 109, 286, 353);

getContentPane().add(scrollPane);

{

listCliente = new JList();

scrollPane.setViewportView(listCliente);

DefaultComboBoxModel modeloLista = new DefaultComboBoxModel(provedor.getListaObjetos());

listCliente.setModel(modeloLista);

// TODO

}

}

{

panel = new JPanel();

panel.setLayout(null);

panel.setBorder(new TitledBorder(null, "Ação:", TitledBorder.DEFAULT\_JUSTIFICATION, TitledBorder.DEFAULT\_POSITION, null, null));

panel.setBounds(314, 100, 163, 104);

getContentPane().add(panel);

{

visualizarDadosButton = new JButton();

visualizarDadosButton.addActionListener(new ActionListener() {

public void actionPerformed(final ActionEvent e) {

executarAcaoBotaoVisualizar();

}

});

visualizarDadosButton.setText("Visualizar Dados");

visualizarDadosButton.setBounds(10, 33, 137, 26);

panel.add(visualizarDadosButton);

}

{

excluirClienteButton = new JButton();

excluirClienteButton.addActionListener(new ActionListener() {

public void actionPerformed(final ActionEvent e) {

executarAcaoBotaoExcluir();

atualizarLista();

}

});

excluirClienteButton.setText("Excluir");

excluirClienteButton.setBounds(10, 64, 137, 26);

panel.add(excluirClienteButton);

}

}

{

panel\_1 = new JPanel();

panel\_1.setLayout(null);

panel\_1.setBorder(new TitledBorder(null, "Pesquisa:", TitledBorder.DEFAULT\_JUSTIFICATION, TitledBorder.DEFAULT\_POSITION, null, null));

panel\_1.setBounds(22, 18, 455, 76);

getContentPane().add(panel\_1);

{

clientLabel = new JLabel();

clientLabel.setText("Nome/Descrição:");

clientLabel.setBounds(10, 35, 103, 16);

panel\_1.add(clientLabel);

}

{

textFieldPesquisaCliente = new JTextField();

textFieldPesquisaCliente.setBounds(119, 33, 309, 20);

panel\_1.add(textFieldPesquisaCliente);

}

}

{

atualizarListaButton = new JButton();

atualizarListaButton.addActionListener(new ActionListener() {

public void actionPerformed(final ActionEvent e) {

atualizarLista();

}

});

atualizarListaButton.setBounds(340, 436, 137, 26);

getContentPane().add(atualizarListaButton);

atualizarListaButton.setText("Atualizar Lista");

}

}

private void executarAcaoBotaoExcluir() {

provedor.executarAcaoExcluirDados(listCliente.getSelectedValue());

atualizarLista();

}

private void executarAcaoBotaoVisualizar() {

provedor.executarAcaoVisualizarDados(listCliente.getSelectedValue());

}

private void atualizarLista() {

listCliente.setModel(new DefaultComboBoxModel(provedor.getListaObjetos()));

}

}

**APÊNDICE L – Interface ProvedorConteudoAcoesFrameGenerico**

**package** br.cesmac.ui.frame.internal;

**public** **interface** ProvedorConteudoAcoesFrameGenerico {

**public** Object[] getListaObjetos();

**public** **void** executarAcaoVisualizarDados(Object obj);

**public** **void** executarAcaoEditarDados(Object obj);

**public** **void** executarAcaoExcluirDados(Object obj);

**public** **void** executarAcaoGerarRelatorio(Object obj);

}

**APÊNDICE M – Classe ConexaoBD**

package br.cesmac.util;

import java.sql.Connection;

import java.sql.DriverManager;

import java.sql.SQLException;

public class ConexaoBD {

public static final String DRIVER\_JDBC\_FIREBIRD = "org.firebirdsql.jdbc.FBDriver";

private static ConexaoBD conexaoBD;

private Connection connection;

private ConexaoBD(){

}

public static ConexaoBD getInstance(){

if(conexaoBD == null){

conexaoBD = new ConexaoBD();

}

return conexaoBD;

}

public void conectar(String driver, String url, String usuario, String senha)throws Exception{

try {

Class.forName(driver);

} catch (ClassNotFoundException e) {

e.printStackTrace();

throw new Exception();

}

try {

this.connection = DriverManager.getConnection(url,usuario,senha);

} catch (SQLException e) {

e.printStackTrace();

throw new Exception();

}

}

public Connection getConnection(){

return this.connection;

}

}

**APÊNDICE N – Classe Log**

package br.cesmac.util;

import java.sql.Connection;

import java.sql.PreparedStatement;

import java.sql.SQLException;

public class Log {

private static final String INSERT = "insert into auditoria (classe, metodo, parametro) values ( ?, ?, ?)";

public static final String QUERY\_LISTA\_AUDITORIA = "select \* from auditoria";

private String classe;

private String metodo;

private String parametro;

public Log(String classe, String metodo, String parametro) {

super();

this.classe = classe;

this.metodo = metodo;

this.parametro = parametro;

}

public Log(){

}

public String getClasse() {

return classe;

}

public void setClasse(String classe) {

this.classe = classe;

}

public String getMetodo() {

return metodo;

}

public void setMetodo(String metodo) {

this.metodo = metodo;

}

public String getParametro() {

return parametro;

}

public void setParametro(String parametro) {

this.parametro = parametro;

}

// metodo para registrar a auditoria.

public void registrar( ) throws SQLException{

Connection connection = ConexaoBD.getInstance().getConnection();

PreparedStatement inserirPreparado = connection.prepareStatement(INSERT);

Log log = this;

inserirPreparado.setObject(1, log.getClasse() );

inserirPreparado.setObject(2, log.getMetodo());

inserirPreparado.setObject(3, log.getParametro());

inserirPreparado.executeUpdate();

inserirPreparado.close();

}

}

**APÊNDICE O – Classe Main**

package br.cesmac.main;

import javax.swing.JOptionPane;

import javax.swing.LookAndFeel;

import javax.swing.UIManager;

import javax.swing.UnsupportedLookAndFeelException;

import br.cesmac.ui.frame.external.FramePrincipal;

import br.cesmac.util.ConexaoBD;

import com.jgoodies.looks.windows.WindowsLookAndFeel;

public class Main {

public static void main(String[] args) {

ConexaoBD conexaoBD = ConexaoBD.getInstance();

String url = "jdbc:firebirdsql:localhost/3050:/BD/LOCADORA.FDB";

try {

conexaoBD.conectar(ConexaoBD.DRIVER\_JDBC\_FIREBIRD, url, "sysdba", "masterkey");

} catch (Exception e) {

e.printStackTrace();

String message = "Erro na conexão com o banco de dados.\n" +

"O programa será finalizado.";

String title = "Erro";

JOptionPane.showMessageDialog(null, message, title, JOptionPane.ERROR\_MESSAGE);

System.exit(1);

}

try {

UIManager.setLookAndFeel(new WindowsLookAndFeel());

} catch (UnsupportedLookAndFeelException e) {

e.printStackTrace();

}

FramePrincipal framePrincipal = new FramePrincipal();

framePrincipal.setVisible(true);

}

}