UNIVERSIDADE FEDERAL DO MARANHÃO
CENTRO DE CIÊNCIAS EXATAS E TECNOLOGIA
CURSO DE CIÊNCIA DA COMPUTAÇÃO

ALINE PORFIRO TEIXEIRA

DESENVOLVIMENTO DE SOFTWARE COM ASPECTOS:
REUTILIZAÇÃO ATRAVÉS DA SEPARAÇÃO DE INTERESSES

São Luís
2015

ALINE PORFIRO TEIXEIRA

DENVOLVIMENTO DE SOFTWARE COM ASPECTOS:
REUTILIZAÇÃO ATRAVÉS DA SEPARAÇÃO DE INTERESSES

Monografia apresentada ao curso de Ciência da Computação da Universidade Federal do Maranhão, como parte dos requisitos necessários para obtenção do grau de Bacharel em Ciência da Computação.

Orientador: Prof. Msc. Maria Auxiliadora Freire

São Luís
2015

Teixeira, Aline Porfiro.

Desenvolvimento e Software com Aspectos: Reutilização através da Separação de Interesses / Aline Porfiro Teixeira. – São Luís, 2015.

51 f.

Impresso por computador (Fotocópia).

Orientadora: Profª Msc. Maria Auxiliadora Freire.

Monografia (Graduação) – Universidade Federal do Maranhão, Curso de Ciência da Computação, 2015.

1. Programação Orientada a Aspecto. 2. Separação de interesses 3. software I. Título.

CDU 004.43

ALINE PORFIRO TEIXEIRA

DESENVOLVIMENTO DE SOFTWARE COM ASPECTOS: REUTILIZAÇÃO ATRAVÉS DA SEPARAÇÃO DE INTERESSES

Monografia apresentada ao curso de Ciência da Computação da Universidade Federal do Maranhão, como parte dos requisitos necessários para obtenção do grau de Bacharel em Ciência da Computação.

Aprovada em: 17/04/2015

BANCA EXAMINADORA

Profª. Ma. Maria Auxiliadora Freire (Orientadora)
Mestra em Ciência de Engenharia
Universidade Federal do Maranhão

Prof. Me. Carlos Eduardo Portela Serra de Castro
Mestre em Informática
Universidade Federal do Maranhão

Prof. Dr. Samyr Beliche Valle
PhD em Ciência da Computação
Universidade Federal do Maranhão

AGRADECIMENTOS

Agradeço primeiramente à Deus, por me conceder forças para chegar até aqui, podendo reconhecer as oportunidades de crescimento pessoal e profissional.

Ao meu filho, Benjamin Porfiro Ribeiro, por me proporcionar as horas mais alegres e de grande aprendizado, por ser luz e motivação nesta conquista e em todas as outras.

Aos meus pais, Maria José Porfiro Teixeira e Luiz Durval Ribeiro Teixeira, por todos esses anos de dedicação, pelos ensinamentos e pelo amor incondicional a mim dispensado.

Aos meus irmãos, Tâmara Porfiro Teixeira e Luiz Durval Ribeiro Teixeira Júnior, por sempre me apoiarem e por acreditarem em mim.

Ao meu amor Rafael Borges Medeiros, por tudo que me ensinou em tão pouco tempo, pelo incentivo e principalmente o profundo sentimento que nos une.

Aos professores do curso de Ciência da Computação, em especial à minha orientadora Maria Auxiliadora Freire pela paciência, compreensão e conhecimentos passados para o desenvolvimento deste trabalho.

Aos meus amigos, em especial à Danuelle Cristine dos Santos Almeida, Raila Silva Maciel, Tassiava Vituriano Leão e Arethusa Rosa Soares, pela convivência, risos diários e anos de parceria.

Por fim, agradeço a todos que acreditaram em mim e que contribuíram, direta ou indiretamente, para conclusão deste trabalho.

“Na vida, não vale tanto o que temos, nem tanto importa o que somos. Vale o que realizamos com aquilo que possuímos e, acima de tudo, importa o que fazemos de nós”. (Chico Xavier)

RESUMO

A constante evolução dos paradigmas de programação tem por objetivo facilitar a programação e corrigir antigos problemas dos paradigmas anteriores. Além disso, em virtude da grande complexidade dos sistemas atuais, há uma preocupação maior também na facilidade que se terá diante da necessidade de se efetuar manutenção nos sistemas construídos. O paradigma Orientado a Objetos se tornou o principal paradigma de desenvolvimento atual, porém existem pontos que podem ser melhorados no sentido da modularização que a POO por vezes não é capaz de fazer. Neste sentido, surge o paradigma Orientado a Aspectos, podendo ser usado em conjunto com a Orientação a Objetos para resolver essas questões. Assim, este trabalho apresenta o conceito de separação de interesses e como utilizá-lo para desenvolver sistemas de informações utilizando o paradigma de Programação Orientada a Aspectos. O objetivo principal é demonstrar como este paradigma pode solucionar problemas como reutilização de códigos, visibilidade e organização que são comuns em outros paradigmas de programação. O trabalho também possui como objetivo exemplificar, através de um estudo de caso, a construção de aspectos que apliquem os conceitos de Programação Orientada a Aspecto na otimização do desenvolvimento de interesses ortogonais de um software. Deste modo é possível concluir que a utilização de um método mais eficiente de programação, como a programação orientada a aspecto, pode-se melhorar o desenvolvimento de softwares, diminuírem custos, prazos e recursos, além de aumentar a qualidade dos códigos produzidos e diminuir o tempo gasto com manutenções de sistemas e implantações de funcionalidades.

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

ABSTRACT

The constant evolution of programming paradigms aims to facilitate programming and fix old problems of previous paradigms. In addition, because of the complexity of today's systems, there is a greater concern also at the facility that will have on the need to perform maintenance on systems built. The Object-Oriented paradigm has become the main current development paradigm, but there are points that can be improved towards modularization that OOP is sometimes not able to do. In this sense, the Aspect-Oriented paradigm emerges and can be used in conjunction with Object Orientation to address these issues. This work presents the concept of separation of concerns and how to use it to develop information systems using the paradigm of Aspect Oriented Programming. The main objective is to demonstrate how this paradigm can solve problems such as code reuse; visibility and organization are common in other programming paradigms. The work also has as objective to illustrate, through a case study, building aspects applying the concepts of Aspect Oriented Programming in optimizing the development of orthogonal interests of software. Thus we conclude that the use of a more efficient method of programming, such as aspect-oriented programming, you can improve software development, decrease cost, time and resources, and increase the quality of the produced codes and decrease time spent on system maintenance and functionality deployments.

Palavras-chave: AOP; Aspect-Oriented Programming; Concerns Separations.

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE ABREVIATURAS

AOP – *Aspect Oriented Programming*

IDE – *Integrated Development Enviroments*

JVM – *Java Virtual Machine*

MOA – Modelagem Orientada a Aspectos

OO – Orientação a Objetos

OOP – *Object Oriented Programming*

POO – Programação Orientada a Objetos

POA – Programação Orientada a Aspectos

SMS – *Short Message Service*

UML – *Unified Modeling Language*

SUMÁRIO

1 INTRODUÇÃO

Programação Orientada a Objetos(POO) teve seu surgimento a partir da necessidade de tornar o produto de software melhor modularizado, a medida que se baseava em um maior nível de abstração, onde modelos do mundo real são representados em classes e seus atributos.

Apesar de se tornar o principal paradigma de desenvolvimento atual, e de ter trazido grandes benefícios neste processo, existem alguns pontos sobre modularização que a POO não é capaz de resolver. Quando se fala de interesses principais do sistema, a Orientação a Objetos resolve muito bem, porém quando utilizada para implementar interesses transversais, tem-se um software com código entrelaçado e espalhado. A consequência disso é a dificuldade na manutenção, reutilização entre outros pontos, que tornam o produto final de baixa qualidade.

Nesta proposta de melhorar a modularização, surge a Programação Orientada a Aspectos (POA), na década de 1990. O que se propõe é que interesses transversais sejam implementados em módulos separados do sistema, os chamados aspectos. Assim, este novo paradigma é utilizado especificamente onde modelos tradicionais não são suficientes para sanar problemas de espalhamento e entrelaçamento de código, sendo assim, os interesses principais do sistema continuam sendo implementados por estes modelos.

Deste modo, visando alcançar os benefícios deste novo paradigma, será proposto a refatoração de um sistema de POO para POA. Assim, os interesses transversais antes espalhados e entrelaçados no projeto inicial serão organizados em *aspectos,* onde será feita uma análise dos benefícios desta abordagem.

1.1 Objetivos

Este trabalho tem como objetivo geral possibilitar a construção de programas mais modulares, de modo que favoreça o aumento da produtividade na análise e no desenvolvimento de sistemas.

Os objetivos específicos são listados:

a) Verificar a eficiência do uso de conceitos da Orientação à Aspectos no processo de separação de interesses de um sistema;

b) Analisar questões sobre a utilização de uma linguagem de programação para implementação desses interesses de software;

c) Verificar a possibilidade de construção de um sistema mais reutilizável, a partir desta maior modularização.

1.2 Organização do trabalho

Este capítulo apresentou a motivação e os objetivos de desenvolver um sistema baseado no paradigma de Aspectos.

O Capítulo 2 traz a base teórica sobre Linguagens de Programação, bem como Programação Orientada a Objetos(POO) e Programação Orientada a Aspectos(POA). Para o entendimento dos pontos explorados, é necessário que o leitor já possua um conhecimento sobre o paradigma Orientado a Objetos. A seção 2.1.1 e 2.1.2 explora duas características do uso de POO, o Entrelaçamento de código e o Espalhamento de código, respectivamente.

O Capítulo 3 traz a base teórica sobre AspectJ, linguagem implementada baseada em POA, que se trata da linguagem utilizada no estudo de caso.

O Capítulo 4 apresenta um estudo de caso de um mesmo sistema desenvolvido em POO e o mesmo tendo seus interesses transversais modelados em Aspectos.

Conclusão e trabalhos futuros são apresentados no Capítulo 5.

2 LINGUAGEM DE PROGRAMAÇÃO

A engenharia de Software e a Linguagem de Programação exercem mútua dependência entre si. Um bom desenvolvimento de software está intimamente ligado às definições do sistema, bem como a base teórica que a Engenharia de Software provê, e também, a Linguagem de Programação é que possibilitará sua composição de inúmeras formas de modo que o sistema como um todo seja bem definido.

Antigamente, a complexidade no desenvolvimento de software estava reduzida a complexidade do algoritmo utilizado para determinada funcionalidade. Deste modo, entrada, processamento e saída de dados eram procedimentos simples se comparado aos processos que se tem atualmente.

Hoje, se tem sistemas quase que ingerenciáveis, devido a alta complexidade. Portanto, desenvolver software atualmente requer dos profissionais uma maior preocupação em utilizar novas tecnologias que possam garantir uma maior qualidade no produto final.

2.1 Programação orientada a objetos

Atualmente, a Programação Orientada a Objetos (POO) é o paradigma dominante no desenvolvimento de sistemas de software, teve seu surgimento por volta da década de 70 e com seu advento, proporcionou grande melhora para o desenvolvimento de software pois possibilitou a modularização do sistema (classes) e isso permitiu um salto no que tange o reuso de software e manutenibilidade.

Surgindo da necessidade de simular a realidade através de uma abstração do cotidiano, surgiu a Orientação a Objetos. Onde tem-se, como ideia básica, a representação de características inerentes dos objetos envolvidos no sistema computacional que se está desenvolvendo.

Assim, esse novo paradigma (OO) transportou o estado do sistema para dentro dos objetos, ou seja, o estado momentâneo do sistema está armazenado em um objeto, na tentativa de simular a realidade. Diferentemente da orientação a procedimentos, em POO não se tem acessos diretos a atributos e, por conseguinte, ao seu estado, podendo restringir o acesso a métodos. Isto introduz um conceito até então não visto, o da visibilidade. Onde se entende como visibilidade o nível de acesso de propriedades e métodos de uma classe, que possibilitou maior controle às operações.

Este foi um dos pontos que tornou a Programação Orientada a Objetos mais poderosa que a Programação Orientada a Procedimentos, e por consequência, mais utilizada. Assim, tornou-se possível modelar e projetar software de maneira análoga à análise de um problema no mundo real.

Pode-se considerar que a POO trouxe ganhos para o processo de desenvolvimento de software, no sentido que faz o fluxo do programa ser ditado pela interação entre os objetos e não apenas pela chamada sequencial de procedimentos. Trouxe conceitos como herança e polimorfismo à programação modular. Muda a concepção de programas divididos em procedimentos/módulos para objetos/classes. Além de permitir o reuso por derivação de classes e composição de objetos.

Apesar dessas contribuições, existem também pontos negativos. Dentre os quais pode-se citar: (SOMMERVILLE, 2007)

1. Entrelaçamento do Código (*Tangled Code*);
2. Espalhamento do Código (*Spread Code*).

Deve-se, portanto, detalhar estes dois pontos negativos que são consequência da utilização da Programação Orientada à Objetos.

2.1.1 Entrelaçamento de Código

No contexto do desenvolvimento de software Orientado a Objetos, o mundo é representado através de objetos e classes. Para poder funcionar corretamente, um software é composto de atributos e métodos para compor as classes que irão representar uma entidade do mundo real.

Porém, para este funcionamento, é necessário que os diversos módulos do sistema se integrem, de modo a garantir que o sistema como um todo seja eficaz. Ao ser feita esta integração, ocorre o fenômeno conhecido como entrelaçamento do código. Isto ocorre porque, diversos módulos e métodos necessitam de chamadas de responsabilidade de uma outra classe, assim, são inseridas linhas de código de um outro componente, e estes, são chamados de código intruso ou invasivo.

Pode-se citar como exemplos comuns de entrelaçamento de código, implementações com tratamento de exceções, registros de *logs*, persistência de dados, entre outros. Na figura 2.1 pode-se verificar que em uma mesma classe são implementados vários interesses, tendo como consequência uma lógica do software fica *"embaralhada"*, ou seja,

aspectos que deveriam ser tratados somente em suas classes de origem, ficam a cargo de outra classe.

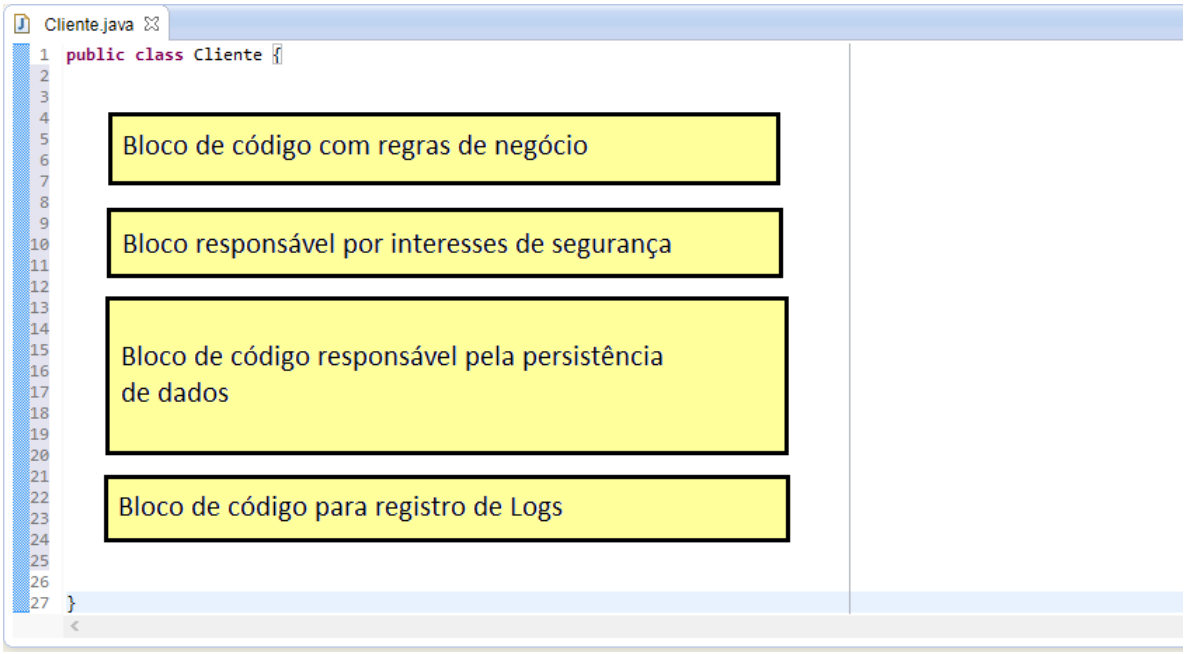

Figura 2.1 - Código entrelaçado causado pela implementação de múltiplos interesses.

Como consequência tem-se a dificuldade no entendimento do funcionamento de uma classe, ao analisar seu código, torna-se difícil saber exatamente o que ocorre com a mesma, isto porque, as chamadas referentes ao entrelaçamento estão fora da classe em questão.

2.1.2 Espalhamento de código (*Scattering Code*)

Além do problema de ter classes compostas de códigos invasivos e tratando de aspectos que não pertences aos seus objetos, tem-se o problema da repetição destes códigos. Isto acontece quando um interesse é implementado em vários módulos. Segue o exemplo de um sistema onde o mecanismo de registro de Log, é implementado em diversas classes, caracterizando o espalhamento de código deste interesse. Na figura 2.2 as linhas grifadas horizontalmente em vermelho representam o espalhamento decorrente da chamada em cada classe para o registro de Logs.

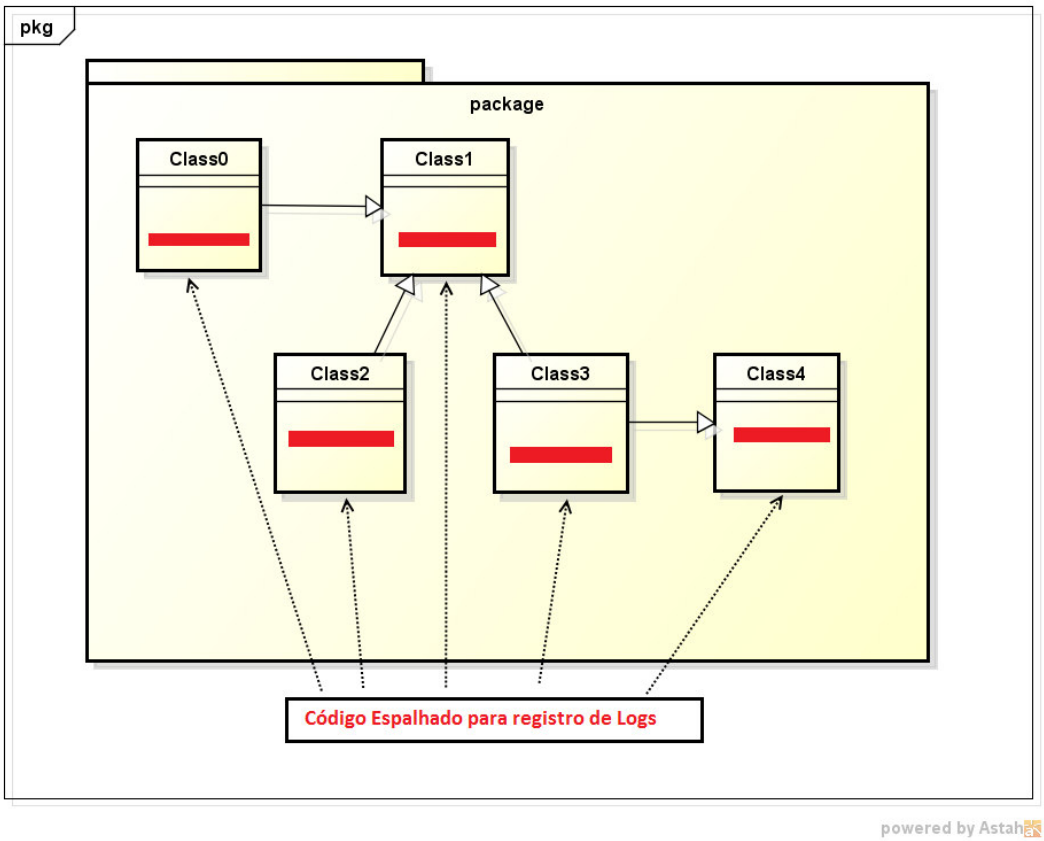

Figura 2.2 - Interesse de registro de Log espalhado por diversas classes.

Portanto, a implementação de códigos entrelaçados e espalhados afetam o desenvolvimento do software de diversas maneiras:

1. Forte acoplamento - métodos das classes primárias precisam conhecer métodos das classes que implementam interesses transversais (c*rosscutting concerns*);
2. Fraca coesão - classes contêm instruções que não estão relacionadas diretamente com as funcionalidades que deveriam implementar;
3. Redundância – códigos semelhantes espalhados por vários módulos;
4. Dificuldades de compreender, manter e reusar – todos esses fatores levam à dificuldade de entender e dar manutenção ao software com essas características.

2.2 Programação orientada a aspectos

Um aspecto é uma unidade básica da POA, assim como o objeto é a unidade básica da POO. Surge com o intuito de se separar os interesses transversais dos interesses principais do sistema.

Desta forma, o aspecto funciona como uma classe, e assim como esta, possui modificadores de acesso (*protected, private, public*), a palavra chave *aspect* e o nome de referência. Seu bloco de código abriga todos os elementos da Programação Orientada à Aspectos.

Assim, em POA é introduzido um novo mecanismo para abstração e composição, que facilita a modularização dos interesses transversais, o aspecto (*aspect*). Desta forma, os sistemas de software são decompostos em componentes e aspectos. Assim, os requisitos funcionais normalmente são organizados em componentes através de uma linguagem POO, como Java, e os requisitos não funcionais como aspectos relacionados as propriedades que afetam o comportamento do sistema.

2.2.1 Interesses de software

Neste contexto, surge um conceito importante no processo de desenvolvimento de um software. Trata-se da identificação dos interesses que compõem o domínio do sistema a ser desenvolvido. Através disso, será possível organizar cada elemento do programa, de modo que seja possível seu entendimento sem necessitar entender outros elementos e, além disso, quando forem necessária mudanças, serão feitas em um número bem menor de elementos.

Como definição do que seriam os interesses em um software, pode-se dizer que são reflexões de requisitos do sistema e das prioridades dos *stakeholders* do sistema (SOMMERVILLE, 2007). Assim, interesse é algo significativo para um *stakeholder* ou para um grupo de *stakeholders*.

Existe, porém, uma diferença quanto a natureza desses interesses. Existem interesses centrais do software, que estão relacionados à requisitos funcionais e primários dentre as funcionalidades identificadas. Além destes, existem os chamados interesses sistêmicos, que se referem à características de suporte à aplicação, relacionando com os requisitos, são os requisitos não funcionais de um sistema.

Mesmo não sendo inerentes ao negócio, os interesses sistêmicos são, na maioria das vezes, de fundamental importância para o sucesso da aplicação. Segue alguns dos mais importantes interesses sistêmicos. (LADDAD, 2010).

1. Persistência – Trata-se da parte da aplicação responsável por se comunicar com o banco de dados, utilizando ou não *framework* de persistência. É, portanto, a comunicação da aplicação com o banco de dados, a fim de extrair, inserir e

atualizar as informações. É também responsável por transformar modelo de Objetos em modelos Relacionais, já que em muitos casos se lida com banco de dados relacionais;

2. Auditoria - De maneira geral, o processo de auditoria deve identificar problemas potenciais de segurança da entidade, com base na legislação vigente, atividades e transações da empresa de forma a propiciar o cumprimento dos serviços contratados com entidade dentro dos prazos e de forma segura, estabelecendo a natureza, oportunidade e extensão dos exames a serem efetuados em conjunto com os termos constantes na sua proposta de serviços para a realização do trabalho;
3. Autenticação e autorização - É comum que sistemas tenham mais de um ponto de entrada possível. Por exemplo, em sistemas web, o usuário pode criar um "*bookmark*" que aponta diretamente para uma página que lhe é útil, de forma que ele não precise navegar sempre através da página principal. No entanto, muitos desses sistemas exigem que o usuário se autentique com sucesso para ter acesso aos dados. Cria-se então rotinas de autenticação e autorização para garantir segurança da aplicação;
4. *Logging, tracing e profiling* : *Logging* - consiste em guardar registros das ações executadas por um programa para fins de análise posterior. Em geral não se sabe se o log será necessário, mas ele pode vir a ser importante para encontrar defeitos no sistema. *Tracing* é basicamente o mesmo que *logging*, mas com objetivos diferentes: o resultado é usado somente para depuração de programas. Em geral o resultado é descartado após o uso e as rotinas de *tracing* nunca fazem parte do produto final que vai para o cliente. *Profiling* é semelhante ao *tracing*, mas com o objetivo de juntar informações sobre o tempo de execução dos métodos para fins de otimização do sistema;
5. Tratamento de exceções - O tratamento de exceções permite capturar erros ocorridos durante a execução de um programa. Para tanto, um programa pode “lançar” e “capturar” exceções;
6. Sincronização de objetos concorrentes - Em sistemas com várias *threads* ou processos paralelos que acessam memória compartilhada, é essencial que se implemente uma política para impedir condições de corrida e *deadlocks* no acesso à memória;

7. Acesso transacional - Uma transação é uma operação que envolve várias operações menores mas onde se garante que ou todas as operações serão executadas com sucesso, ou nenhuma delas será. Por exemplo, em um sistema de comércio eletrônico, ao se efetuar uma compra, é feito um pedido de remessa ao cliente e um pedido de cobrança. Se houver algum problema em apenas uma das operações, o usuário poderá receber algo pelo qual não pagou, ou ser cobrado por algo que não receberá;
8. Garantia do cumprimento das regras arquiteturais - Toda arquitetura define restrições, ou seja, ela define não só os subsistemas que se comunicam mas também os subsistemas que não devem se comunicar. No entanto, é comum que programadores violem regras arquiteturais, incluindo comunicações entre subsistemas que deveriam ser independentes, muitas vezes para contornar falhas ou dificuldades. No entanto, aquilo que torna o código correto também torna a arquitetura e o projeto incorretos, pois passam a não refletir a realidade do código;
9. Otimização: *pooling* e *caching* - Muitas vezes a criação de um recurso é uma operação cara em termos de tempo de execução. Por exemplo, isso acontece com conexões ao banco de dados - leva tempo estabelecer uma conexão autenticada com um banco remoto. Se isso for feito a cada consulta, o programa será muito lento. A solução mais comum é manter um "pool" de conexões - um conjunto de conexões que permanecem abertas e que partes do programa requisitam quando precisam delas, depois as liberam quando não são mais necessárias.

2.2.1.1 Interesses transversais

O processo de separação de interesses consiste em quebrar o esforço de desenvolvimento de um programa em partes, o que geralmente resulta na quebra do programa em módulos que se sobrepõem em funcionalidade o mínimo possível. Este processo é fundamental em engenharia de software, pois reduz a complexidade do processo de desenvolvimento além de tornar mais fácil o entendimento do software desenvolvido. O impacto das mudanças também é minimizado, ao se separar em módulos cada interesse presente no sistema.

Os paradigmas de programação oferecem mecanismos que dão suporte à separação e encapsulamento de interesses. Como exemplo, tem-se procedimentos, pacotes, classes, métodos. No entanto, alguns interesses não se encaixam nas formas de encapsulamento providas por paradigmas tradicionais, como o funcional ou orientação a objetos. Estes são os chamados interesses transversais (*crosscutting concerns*) ou ortogonais.

Tais interesses são difíceis de modularizar, e por esta razão a sua implementação acaba atravessando várias partes do programa, espalhada por entre os módulos que o compõem. São chamados de transversais em relação a outros interesses.

Na maioria dos casos, interesses centrais, podem ser facilmente encapsulados utilizando paradigmas tradicionais de programação. No entanto, interesses sistêmicos, ao serem implementados, resultam em um código espalhado por meio de diversas partes do programa e entrelaçado, tornando entendimento e modificação do código difícil.

Como pode ser observado na figura 2.3, interesses centrais são facilmente modularizados, abrangendo as operações de acordo com a responsabilidade de cada funcionalidade. No entanto, os interesses sistêmicos atravessam os interesses centrais, pois abrangem o sistema como um todo.

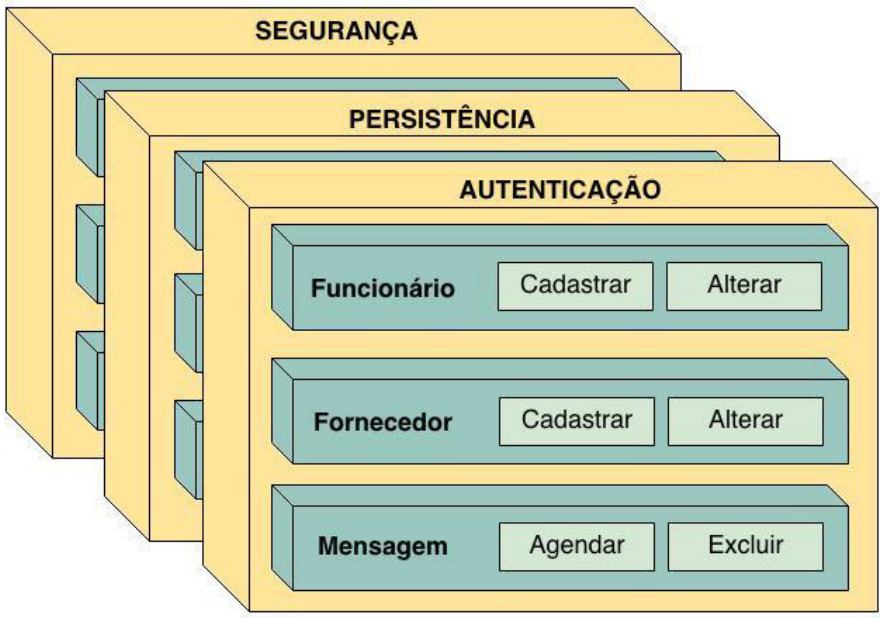

Figura 2.3 - Interesses Transversais “atravessam” interesses centrais.

2.3 Etapas de desenvolvimento

A POA envolve basicamente três etapas distintas de desenvolvimento: (Figura 2.4)

1. Decompor os interesses (*aspectual decomposition*) – identificar e separar os interesses transversais dos interesses do negócio do sistema em questão;
2. Implementar os interesses (*concern implementation*) – implementar cada um dos interesses identificados separadamente;

3. Recompor o aspectual (*aspectual recomposition*) – com a finalidade de integrar aspectos em um processo de junção da codificação dos componentes(POO) e dos aspectos(POA). Esta etapa é denominada combinação (*weaving*).

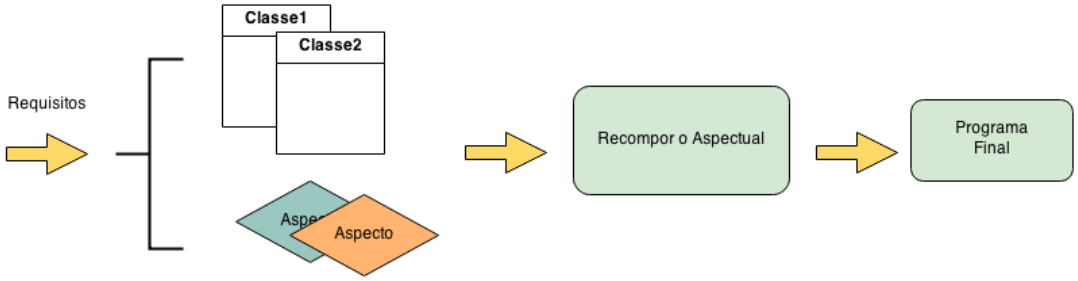

Figura 2.4 - Etapas do desenvolvimento de software orientado à aspectos.

Pode-se então definir os seguintes elementos de um sistema implementado com as técnicas de POA: (MACHADO, WINCK e GOETTEN, 2004)

1. Linguagem de Componentes – Elemento que irá implementar os interesses não transversais do sistema, permite que sejam implementadas as funcionalidades básicas do sistema. Não prevê nada do que deve ser implementado na linguagem de aspectos. Por exemplo: Java, C++, C#, PHP, etc;
2. Linguagem de Aspectos – Irá implementar interesses transversais do sistema, descrevendo o comportamento do aspecto. Deve suportar a implementação das propriedades desejadas de forma clara e concisa, fornecendo construções necessárias para que o programador crie estruturas que descrevam o comportamento dos aspectos e definam em que situações eles ocorrem. Por exemplo: AspectJ, InfraAspect, Aspect #, PHPAspect, etc;
3. Combinador de Aspectos – Responsável por integrar os elementos criados por linguagem de componentes e os elementos usando linguagem de aspectos. Assim, o combinador de aspectos (*aspect weaver*) combina os programas escritos em linguagem de componentes com os escritos em linguagens de aspectos. Funciona como um "motor de fusão". Ou seja, trata-se de um processo responsável por combinar os elementos escritos em linguagem de componentes com os elementos escritos em linguagem de aspectos. Pode ser estática ou dinâmica. Ele antecede à compilação, gerando um código intermediário (combinação estática) na linguagem de componentes ou permite a sua realização durante a execução do programa

(combinação dinâmica). As classes referentes ao código do negócio implementadas não sofrem alteração com a programação orientada a aspectos. Isso é feito no momento da combinação entre componentes e aspectos, como mostra a figura 2.5.

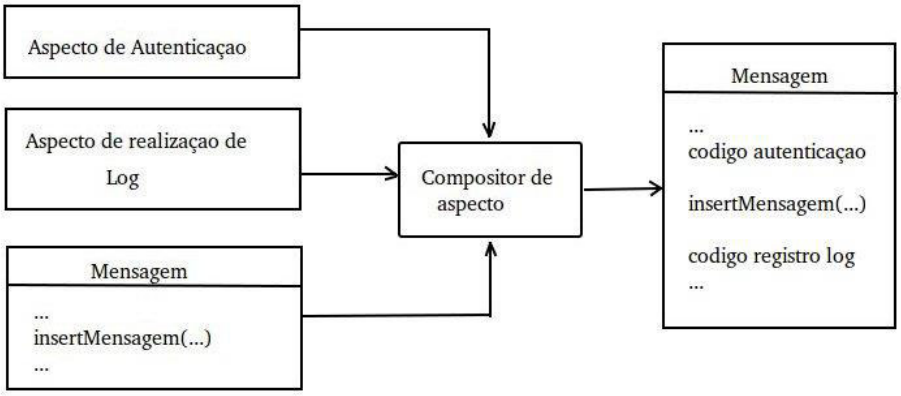

Figura 2.5 - Composição de Aspectos.

Portanto, é possível observar que neste capitulo o objetivo foi conceder uma visão geral com relação aos pontos negativos da Orientação a Objetos e sobre a Programação Orientada a Aspectos, evidenciando sua utilidade para sanar as lacunas da Orientação a Objetos.

3 ASPECTJ

AspectJ trata-se de uma implementação específica de POA. Foi desenvolvida para a linguagem de programação Java (MACHADO, WINCK e GOETTEN, 2004). Como qualquer outra implementação, deve-se considerar dois pontos principais para seu entendimento: a especificação da linguagem (gramática e semântica) e a implementação da linguagem (compilador). Este compilador será responsável por produzir o código byte compatível com a máquina virtual Java (JVM). Com o interesse de simplificar o desenvolvimento de aplicações POA utilizando AspectJ, a linguagem também oferece suporte para Ambientes de Desenvolvimento Integrado (*Integrated Development Enviroments* - IDEs).

Desta forma, Linguagens de Programação como AspectJ foram desenvolvidas e estenderam a programação orientada a objetos para incluir aspectos [Laddad, 2010]. Assim, após ser feita decomposição do sistema em partes não entrelaçadas e não espalhadas, com o uso de aspectos, há o processo de composição, onde se irá juntar essas partes, de modo a se obter de forma significativa o sistema desejado.

Portanto, há uma preocupação em relação à compatibilidade entre ambas as linguagens. Pode-se destacar :

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

Em suas primeiras implementações, o AspectJ estendia Java através de palavras-chaves adicionais que suportavam conceitos de POA. A versão mais recente do AspectJ inclui suporte para o uso de *annotations,* que são assinaturas que especificam metadados(dados ou informações adicionais) sobre estes elementos.

Existem três questões que devem ser abordadas em qualquer linguagem orientada a aspecto relacionada a este processo de composição: a correspondência, a semântica composicional, e o tempo de ligação (LADDAD, 2010).

1. Correspondência – utilizada para descrever quais entidades serão compostas entre si. Esta correspondência pode ser implícita(determinada por regras da linguagem) ou explícita(descrita pelo próprio programador). Métodos com o mesmo nome por exemplo, possuem uma correspondência implícita entre si.
2. Semântica Composicional – é o que deve acontecer com elementos que correspondem. No caso de linguagens POA, a semântica das chamadas a métodos é modificada. Em linguagens procedurais, chamar a função X significa executar a função X. Em linguagens orientadas a objetos, chamar o método X implica em executar algum método X em uma das subclasses que definem X. Em linguagens orientadas a aspectos, chamar o método X pode ter diversas consequências:
    a. X é executado;
    b. Y é executado(algum outro método);
    c. X + Y são executados, em alguma ordem definida.

    Em POA, diversas semânticas podem ser definidas, e em geral são determinadas pelo próprio programador.

3. Tempo de Ligação – está relacionada ao momento em que a correspondência passa a surtir efeito; pode ser estático(em tempo de compilação) ou dinâmico(em tempo de execução).

3.1 O compilador e IDE Eclipse para AspectJ

A ferramenta ou compilador que faz a composição dos elementos em POA, é o que foi definido como *weaver* (tecelão) (CHAGAS, OLIVEIRA, 2009), tem este nome pois será responsável por "tecer" os vários fragmentos do programa em um programa único.

Programas orientados a aspectos precisam de um compilador específico. No caso da linguagem AspectJ, o compilador "ajc" (*AspectJ Compiler*) transforma um programa escrito em AspectJ em um programa em bytecode Java, que pode ser executado por qualquer máquina virtual Java (JVM). O Eclipse é um IDE gratuito, produzido pela IBM, totalmente desenvolvido usando tecnologia Java. O Eclipse é extensível através de plugins.

3.2 Exemplo de utilização

Para tornar mais fácil o entendimento sobre como utilizar os conceitos de aspectos e a linguagem AspectJ, será demonstrado como adicionar um aspecto em um programa inicialmente em POO. Tendo como exemplo a figura 3.1, primeiramente tem-se uma classe *EntregaMensagem* que contém dois métodos que irá imprimir mensagens. E também a classe Main, na figura 3.2 que utiliza estes dois métodos.

Main.java | EntregaMensagem.java

```
package aspecto.exemplos;

    public class EntregaMensagem {

    public void entrega(String mensagem) {
            System.out.println(mensagem);
    }
    public void entrega(String pessoa, String mensagem) {
        System.out.println(pessoa + ", " + mensagem);
    }
}
```

Figura 3.1 - Classe EntregaMensagem

Main.java | EntregaMensagem.java

```
package aspecto.exemplos;

public class Main {
    public static void main(String[] args) {
        EntregaMensagem entregaMensagem = new EntregaMensagem();
        entregaMensagem.entrega("Aprendendo Aspectj");
        entregaMensagem.entrega("Pessoa", "exemplo AspectJ");
    }
}
```

Figura 3.2 - Classe Main.

Supondo que, no exemplo dado, antes de efetuar a entrega de uma mensagem, seja necessário verificar se o usuário em questão está autenticado. Sem o uso de aspecto, esta funcionalidade, pode ser implementada como segue na figura 3.3.

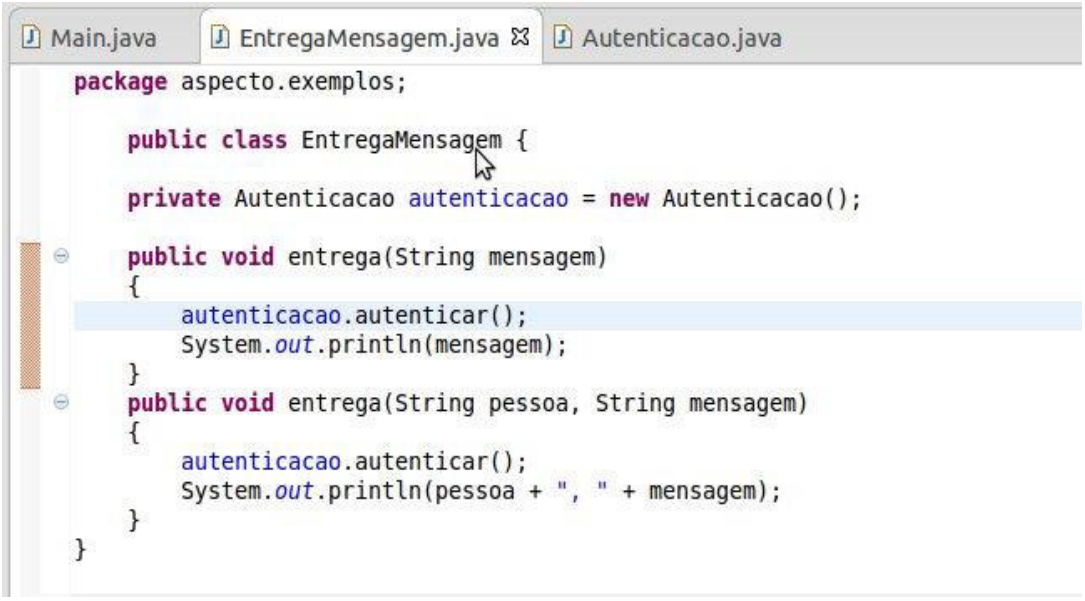

Figura 3.3 - Adicionando método para autenticação.

Torna-se necessário adicionar uma chamada para autenticar() em cada método que precisa de autenticação, o que leva ao código emaranhamento. Código semelhante deve estar presente em todas as classes que requerem a funcionalidade de autenticação levando ao código de espalhamento. Com POA, não se muda uma única linha de código na classe de MessageCommunicator, e adiciona-se a funcionalidade criando um aspecto para o sistema, como mostra a figura 3.4.

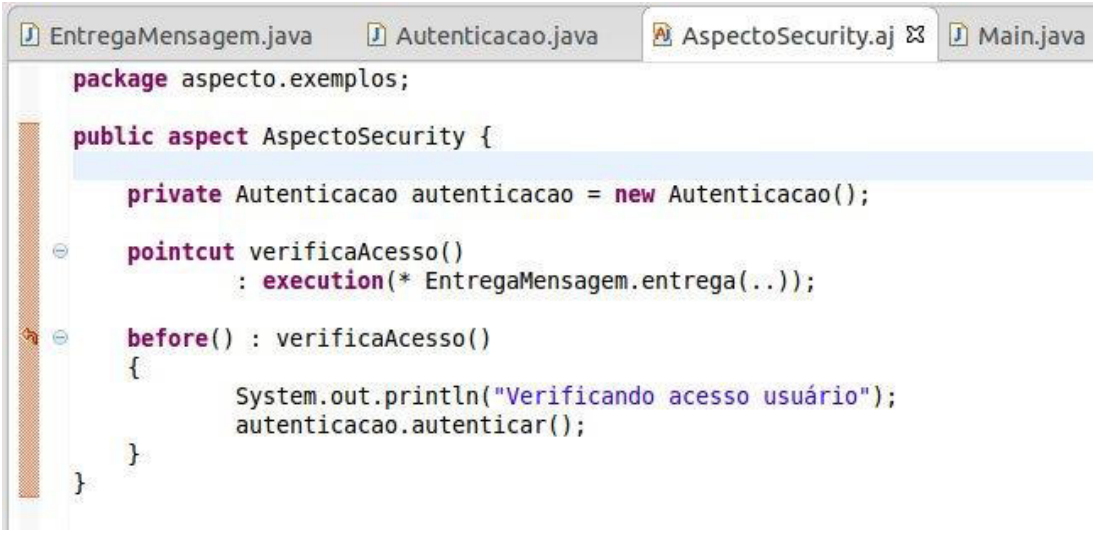

Figura 3.4 - Adicionando Aspecto de Segurança.

Como pode-se perceber, um aspecto é uma unidade de modularização em POA , muito parecido com uma classe, que é a unidade modularizadora em POO. A declaração de um aspecto é semelhante a uma declaração de classe.

O *pointcut* seleciona pontos interessantes de execução em um sistema , chamado de *join points*. O *pointcut* verificaAcesso() que seleciona a execução de todos os métodos chamado "entrega" na classe EntregaMensagem. O curinga * indica que o ponto corte combina com qualquer tipo de retorno, e os dois pontos entre parênteses após "entrega", indica que é independentemente do número de argumentos ou seus tipos. Neste exemplo, o *pointcut* seleciona execução de ambas as versões sobrecarregadas de entrega() na classe EntregaMensagem. No *advice* é definido o código para executar ao atingir o *join points* selecionados pelo *pointcut* criado. Neste exemplo, é definido um *advice* para executar antes de alcançar o *join points* selecionados pelo *pointcut* verificaAcesso(), ou seja, antes da execução de qualquer método EntregaMensagem.entrega() . No *advice*, é feita autenticação do usuário atual. Com o aspecto já presente no sistema , cada vez que EntregaMensagem.entrega() é executado , o código do *advice* realiza a lógica de autenticação antes do método.

3.3 Elementos de POA em AspectJ

Torna-se importante o entendimento de 4 conceitos que norteiam as construção de um programa com Orientação à Aspectos. São eles: (LADDAD, 2010)

1. Pontos de Junção (*join points*);
2. Pontos de Atuação (*pointcuts*);
3. Adendo (*advice*);
4. Aspectos.

3.3.1 Pontos de junção (*join points*)

Um ponto de junção é qualquer ponto identificável pelo AspectJ durante a execução de um programa. Aspectos podem ser associados a pontos de junção e executados antes, depois, ou ao invés deles. Existem diversos pontos de junção reconhecidos pelo AspectJ: (LADDAD, 2010)

1. Métodos - Existem dois tipos de pontos de junção relacionados a métodos: pontos de chamada de métodos e pontos de execução de métodos

a. Chamada de Métodos – são os pontos indicativos de onde, no código, o método é chamado.
b. Execução de métodos – É o corpo dos métodos propriamente ditos.

Há uma diferença entre essas duas definições: como em linguagens orientadas a objetos existe polimorfismo, é possível chamar-se um método de uma classe e executar-se um método de uma de suas subclasses. Assim, é útil fazer a distinção em se tratando de aspectos. Além disso, como a chamada e execução podem estar em arquivos fisicamente diferentes, pode-se gerar pontos de corte compostos onde a diferença entre chamada e execução torna-se mais importante;

2. Construtores - Assim como métodos, construtores têm dois pontos de junção: um para chamadas ao construtor e outro para a execução do construtor. Pontos de junção de chamada de construtores são encontrados em locais onde um objeto é instanciado com comandos *new*. Pontos de junção de execução de construtores são o próprio código do construtor;
3. Acesso a campos - Existe um ponto de junção em cada acesso aos dados de uma classe. Os pontos são divididos em pontos de leitura, onde o dado é usado mas não modificado, e pontos de escrita, onde o dado é modificado. É importante notar que só há pontos de junção em dados declarados no escopo da classe. Não há pontos de junção para variáveis locais a métodos;
4. Tratamento de exceções - Um ponto de junção de tratamento de exceções é o interior de um bloco *catch* que 'captura' algum tipo de exceção;
5. Inicialização de classes - São pontos de junção que representam o instante em que uma classe é carregada na memória e seus dados estáticos são inicializados;

Há outros tipos de pontos de junção menos usados: inicialização de objetos, pré-inicialização de objetos, e execução de advice.

3.3.2 Pontos de corte *(pointcuts)*

Um ponto de corte (*pointcut*) é uma construção sintática do AspectJ para se agrupar um conjunto de pontos de junção. Desta forma, são elementos do programa usados para definir um ponto de junção como uma espécie de regra criada pelo programador para especificar eventos que serão atribuídos aos pontos de junção.

Pointcuts está para atributos assim como *aspects* está para *class*. Um *pointcut* pode ser entendido como uma variável. Esta variável armazena uma lista contendo as assinaturas de métodos que se tem interesse em interceptar. Esta interceptação é feita com o objetivo de alterar o fluxo original do programa e inserir novas chamadas, *advices* (RESENDE; SILVA, 2005).

Como os *pointcuts* podem ser vistos como variáveis contendo assinaturas de métodos, então se disponibilizou os curingas ou *wildcards* como *, + e .. (dois pontos sequênciais) para facilitar o trabalho de escrever os *pointcuts*. Na figura 3.5 segue o exemplo de código de um *pointcuts.*

```
public pointcut verificaAcesso()
        : execution(* EntregaMensagem.entrega(..));
```

Figura 3.5 - Código de um ponto de corte.

- A primeira parte é a declaração de restrição de acesso - nesse caso, *public*, mas *pointcuts* podem ser *private* ou *protected*.
- A palavra-chave *pointcut* denota que se está declarando um ponto de corte.
- Todo *pointcut* tem um nome qualquer, e pode receber parâmetros - nesse caso, o *pointcut* não recebe parâmetros.
- Depois dos dois pontos (:) obrigatórios vem o tipo dos pontos de junção agrupados pelo *pointcut* - nesse caso temos um *pointcut* do tipo *execution*, que indica uma execução de método.
- Finalmente vem a assinatura do *pointcut*, uma especificação dos pontos de junção aos quais o *pointcut* se refere.

Na descrição da assinatura, pode-se usar alguns caracteres especiais para incluir mais de um ponto de junção no pointcut. Conforme tabela 3.1.

Tabela 3.1 - Caracteres especiais.

Caractere	**Significado**
*	Qualquer sequência de caracteres não contendo pontos
..	Qualquer sequência de caracteres, inclusive contendo pontos
+	Qualquer subclasse de uma classe.

Um *pointcut* pode conter várias assinaturas ligadas através de operadores lógicos. Conforme tabela 3.2

Tabela 3.2 - Operadores lógicos em *pointcuts*.

Operador	**Significado**	**Exemplo**	**Interpretação do Exemplo**
!	Negação	!EntregaMensagem	Qualquer classe exceto EntregaMensagem
||	"ou" lógico	EntregaMensagem || EntregaEmail	Classe EntregaMensagem ou classe EntregaEmail
&&	"e" lógico	Cloneable && Runnable	Classes que implementam ambas as interfaces Cloneable e Runnable

Para caracterizar o tipo de ponto de junção agrupados pelo *pointcut*, pode ser usado o *execution*, conforme exemplo dado. Há também outros tipos de ponto de junção, segue tabela 3.3 com os tipos existentes.

Tabela 3.3 - Tipos de ponto de junção.

Categoria do ponto de junção	**Sintaxe do pointcut**
Chamada de método ou construtor	call(AssinaturaDoMetodo)
Execução de método ou construtor	execution(AssinaturaDoMetodo)
Inicialização de classe	staticinitialization(Classe)
Leitura de dado de classe	get(AssinaturaDoCampo)
Escrita de dado de classe	set(AssinaturaDoCampo)
Tratamento de exceção	handler(Exceção)

Dois tipos de *pointcuts* levam em consideração trechos do código do programa, sem levar em consideração a execução. O primeiro é *within*(Tipo). Esse *pointcut* inclui todos os pontos de junção dentro de uma classe ou aspecto passada como parâmetro. O segundo é *withincode* (Método), que inclui todos os pontos de junção dentro de um método ou construtor. Segue detalhamente na tabela 3.4.

Tabela 3.4 - *Pointcuts within* e *withincode*.

Ponto de corte	**Descrição**
within(Autenticacao+)	Todos os pontos de junção que ocorrem dentro da classe Autenticacao ou de qualquer de suas subclasses.
withincode(* Autenticacao.set*(..)	Todos os pontos de junção que ocorrem dentro de métodos da classe Autenticacao cujos nomes começam com "set".

3.3.3 Adendo *(advice)*

Advice é uma estrutura que denota o que um aspecto deve fazer, ou seja, qual o comportamento do aspecto. Em termos mais formais, o *advice* designa a semântica comportamental do aspecto. São partes da implementação de um aspecto executados em pontos bem definidos do programa principal (pontos de junção). Também denominado de recomendação, segundo (SOMMERVILLE, 2007).

Os adendos são compostos de duas partes: o ponto de atuação que define as regras de captura dos pontos de junção; e o código que será executado quando ocorrer o ponto de junção definido pela primeira parte. O adendo, se comparado à POO, pode ser considerado um método, cuja função é declarar o código que deve ser executado a cada ponto de junção em um ponto de atuação (LADDAD, 2010).

Todo advice está associado a um *pointcut*, que define pontos de junção. Há três tipos de *advice*: (Laddad, 2010).

1. *before*: executa antes do ponto de junção.
   O *advice* do tipo *before* é o mais simples: ele simplesmente executa antes do ponto de junção. O único detalhe a se observar é que se o *advice* levantar uma exceção, o ponto de junção não será executado.

2. *after*: executa depois do ponto de junção.
   O *advice* do tipo *after* executa após o ponto de junção. Existem três variacões desse *advice*, que dependem de como terminou a execução do ponto de junção: se terminou normalmente, com uma exceção, ou de qualquer forma. Segue tabela 3.5 com detalhamento.

Tabela 3.5 - Variações do *advice* do tipo *after*.

after() : ponto_de_corte()	Executa depois do ponto de junção, independente de como ele retornou.
after() returning : ponto_de_corte()	Executa depois do ponto de junção se ele tiver terminado normalmente (sem exceção).
after() throwing : ponto_de_corte()	Executa depois do ponto de junção somente se este tiver saído com uma exceção.

3. *around*: executa "em volta" do ponto de junção; esse tipo de *advice* serve para substituir a execução do ponto de junção pela execução do *advice*, ou executar parte do *advice* antes do ponto e junção e outra parte depois.
   Os advices *around* substituem o comportamento original do ponto de junção. Para isso, a linguagem oferece o comando *proceed*(), o qual permite que o comportamento original seja substituído e, ainda, que comportamentos adicionais sejam realizados antes e após o comportamento original do join point selecionado (CHAVES, 2002).

Todos os tipos de advice são definidos de forma semelhante. Segue exemplo na figura 3.6.

```
before() :                  // declaração do advice
    verificaAcesso()        // ponto de corte associado
    {
                            //corpo do advice
        System.out.println("Verificando acesso usuário");
        autenticacao.autenticar();
    }
}
```

Figura 3.6 - Exemplo de um *advice*.

3.3.4 Aspectos

Um aspecto é o mecanismo disponibilizado pela programação orientada a aspectos para agrupar fragmentos de código referente aos componentes não funcionais em uma unidade no sistema. Propriedades, tais como sincronização, interação entre componentes, distribuição e persistência, são expressas em fragmentos de código espalhados por diversos componentes do sistema, são os interesses sistêmicos.

O aspecto não pode existir isoladamente para implementação dos interesses do sistema (tanto funcionais como sistêmicos). Os objetos continuam existindo e neles são implementados os interesses funcionais. Logo, aspecto-objeto é a unidade principal da POA, assim como o objeto seria o principal da POO (LADDAD, 2010).

4 PROCESSO DE DESENVOLVIMENTO EM POA

Neste capítulo será abordado questões relacionadas à Modelagem Orientada a Aspectos (MOA) com o objetivo de definir os requisitos para ferramentas e linguagens de modelagem orientadas a aspectos. Para isto, será utilizado o estudo de caso do desenvolvimento de um sistema, bem como um comparativo de como seria sua modelagem orientada à objetos e orientada à aspectos.

Sabe-se que a modelagem de software contemporânea adota uma perspectiva orientada a objetos, na qual os principais blocos básicos dos sistemas de software são objetos e classes. Essa perspectiva mostrou-se útil em termos de compreensibilidade, extensibilidade e reusabilidade em vários domínios. Quantos às linguagens de modelagem utilizadas, após serem levantadas uma grande variedade delas, foi introduzida a UML (*Unified Modeling Language*), sendo esta orientada a objetos e utilizada como linguagem de propósito geral padrão, sendo possível seu uso em uma grande variedade de domínios de aplicações.

A UML oferece visões inter-relacionadas, complementares e separadas a fim de descrever um sistema orientado a objetos, desde seus requisitos até sua implementação. Algumas dessas visões podem ainda ser separadas em modelos comportamentais e estruturais. Portanto, UML usa o princípio de separação de *concerns* em vários níveis diferentes, mas a realização desse princípio é ligeiramente diferente daquela promovida pela POA.

A POA embasa-se na ideia de que para qualquer perspectiva e blocos básicos usados, *concerns* transversais podem estar espalhados e entrelaçados em vários elementos. Assim, propõe o uso de aspectos para modularizar esses elementos transversais a fim de melhorar a compreensão além de facilitar a reutilização e a evolução. Portanto, a Modelagem Orientada a Aspectos necessita do uso de outros elementos para exemplificar esses *concerns* transversais e a utilização de outras visões a fim de melhorar a compreensão, além de facilitar a reutilização e a evolução.

Na abordagem orientada a aspectos, o bloco básico adicional é o aspecto, e o mecanismo de composição que relaciona aspectos e blocos básicos é chamado de *crosscutting*. O principal objetivo da MOA é capturar de forma adequada os *concerns* transversais por meio de uma linguagem de modelagem que forneça notação e conceitos dedicados e precisos para a representação dos principais conceitos da POA através de várias visões. A MOA deve permitir que o projetista escolha e capture explicitamente qualquer tipo de aspecto de um domínio de problemas e qualquer tipo de dependência entre aspectos.

4.1 Aspectos e UML

No processo de Modelagem de um sistema que se utiliza de Orientação à Aspectos, deve-se além de compreender e lidar com a estrutura e o comportamento de aspectos individuais, deve-se compreender os relacionamentos entre aspectos e classes, as possíveis interações entre aspectos, as possíveis interferências entre os novos mecanismos de composição e os convencionais, a estrutura e o comportamento do sistema depois da combinação de classes e aspectos, e assim por diante.

Como qualquer outra abordagem moderna ou não para modelagem, a MOA necessita de ferramentas que forneça suporte quanto edição gráfica, à visualização, à análise, à reengenharia etc.

Quanto à UML, sabe-se que esta é uma linguagem uniforme, desde a especificação de requisitos até a fase de implantação: o mesmo conjunto de conceitos e notação podem ser usados em diferentes atividades de desenvolvimento. A UML não depende de um processo de desenvolvimento, linguagem de programação ou ferramenta em particular. No entanto, UML permite (mas não exige) um processo incremental, iterativo, centrado na arquitetura e orientado a casos de uso.

Torna-se necessário investigar a adequabilidade de UML como uma linguagem de modelagem que possa dar suporte à modelagem orientada a aspectos. Embora os elementos da POA ainda não tenham sido incorporados oficialmente à UML, algumas abordagens estão sendo propostas pela comunidade científica e tecnológica, como alternativa para adaptação dos diagramas da UML.

4.2 Definição do sistema

Com o intuito de aplicar os conceitos do Paradigma de Programação Orientada à Aspecto para o desenvolvimento de aplicações, foi realizado um estudo de caso em um sistema de baixa complexidade, o Sistema Despachante de Email e SMS.

A aplicação desenvolvida consiste em um software para gestão que tem como principais funcionalidades de permitir o agendamento do envio em massa de Email e SMS para estes fornecedores. Para manter a integridade, o software deve ter um controle de acesso, de modo que apenas usuários autorizados e cadastrados previamente possam efetuar estas operações. Por segurança, o sistema deve também registrar quando for efetuado alguma operação no sistema, bem como o usuário que a efetuou, o tipo de operação.

4.2.1 Levantamento de requisitos

Segue nas tabelas 4.1 e 4.2 os requisitos funcionais e não-funcionais, respectivamente, levantados para serem implementados no sistema Despachante.

Tabela 4.1 - Requisitos Funcionais Sistema Despachante.

ID	**Requisito**	**Descrição**
[RF01]	Efetuar login	Usuários que possuam login e senha válidos podem ser autenticados no sistema para acessá-lo.
[RF02]	Agendamento de envio de Sms	Permitir que usuários autenticados efetuem o agendamento de sms para um grupo de fornecedores.
[RF03]	Agendamento de envio de Email	Permitir que usuários autenticados efetuem o agendamento de email para um grupo de fornecedores.
[RF04]	Excluir Agendamento de envio de SMS.	Permitir que usuários autenticados efetuem o a exclusão de um agendamento de sms feito anteriormente.
[RF05]	Excluir Agendamento de envio de Email.	Permitir que usuários autenticados efetuem o a exclusão de um agendamento de email feito anteriormente.

Tabela 4.2 - Requisitos Não-Funcionais Sistema Despachante.

ID	**Descrição**
[RNF01]	Segurança: Será necessário que, antes de qualquer operação de inclusão/exclusão no sistema, haja um controle de erros que podem eventualmente ocorrer
[RNF02]	*Loging*: Será necessário que toda operação realizada seja registrada, para possível necessidade posterior.

4.2.2 Casos de uso

A figura 4.1 mostra o diagrama de caso de uso proposto para a formação dos requisitos funcionais do sistema Despachante proposto neste trabalho, mostrando apenas o que será de mais funcional dentro da modelagem do sistema, como pode-se perceber, requisitos não funcionais como registro de log, autenticação, entre outros, não são mostrados.

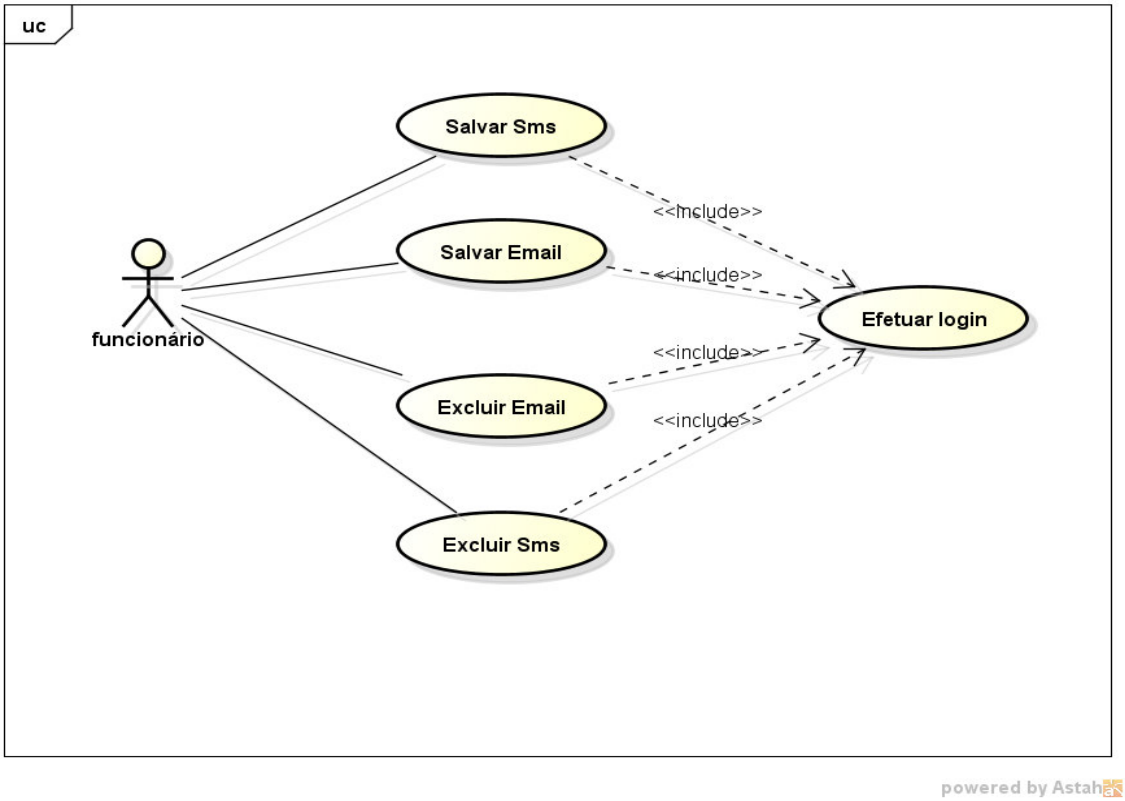

Figura 4.1 - Caso de Uso sem aspectos.

Porém, quando se leva em consideração os requisitos não-funcionais, e também sua modularização e implementação utilizando os conceitos de aspetos, pode-se representá-los em um novo caso de uso, como mostra a figura 4.2. Conforme foi dito anteriormente, os elementos de POA ainda não foram oficialmente incorporados à UML, trata-se portanto de uma proposta, uma alternativa para a representação de aspectos em diagramas UML.

Como pode ser observado, os casos de uso com coloração diferente representam os aspectos abrangem os requisitos não funcionais do sistema.

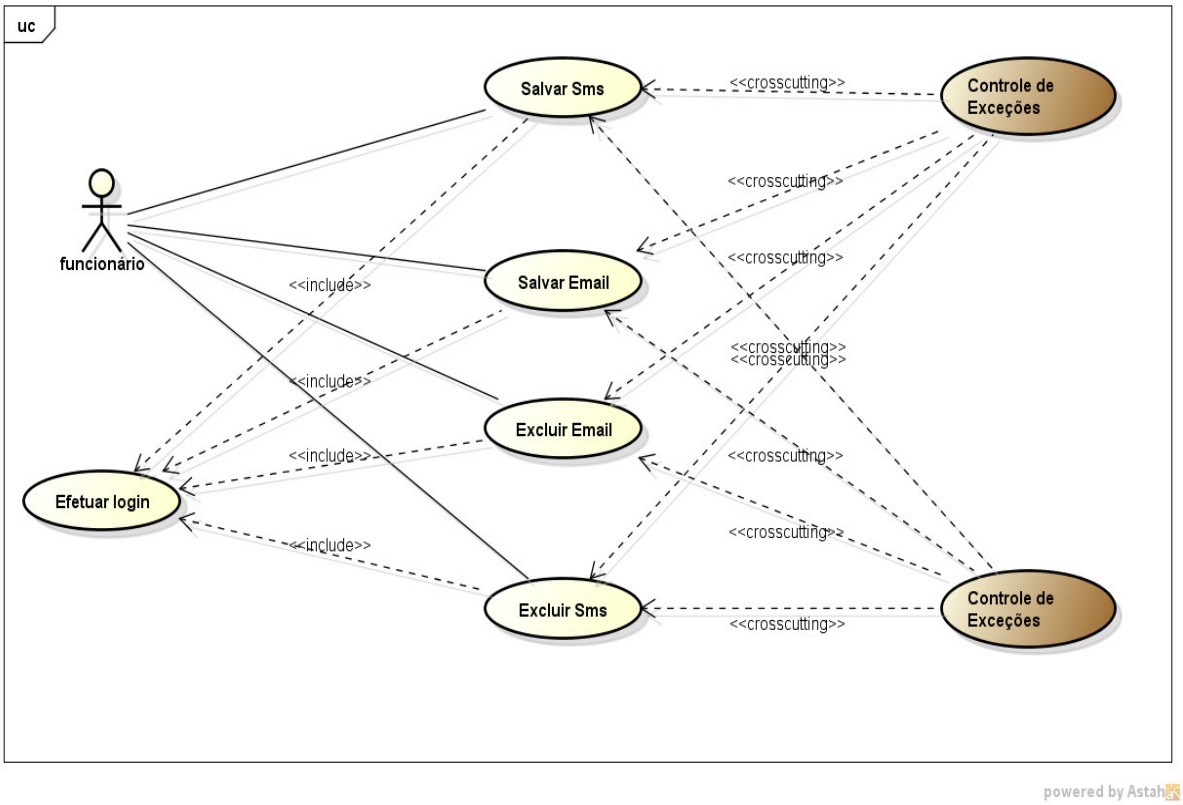

Figura 4.2 - Caso de uso com Aspectos.

4.2.3 Diagrama de classes

A figura 4.3 representa o diagrama de classes de forma tradicional, que modela o domínio da aplicação e traz as classes Funcionario, Fornecedor, EnvioMsg, Mensagem, esta especializada em SMS e Email.

Quando se acrescenta as classes que irão implementar os interesses transversais do sistema, pode-se representar como na figura 4.4, ao adicionar as classes RegistroLog e ControleExecoes, sendo que, para diferenciação das demais, usou-se uma coloração diferente.

Vale ressaltar que, para representar os aspectos, há o recurso do UML, pois não se trata de uma classe, especificando neste caso o estereótipo *<<aspect>>*, no diagrama.

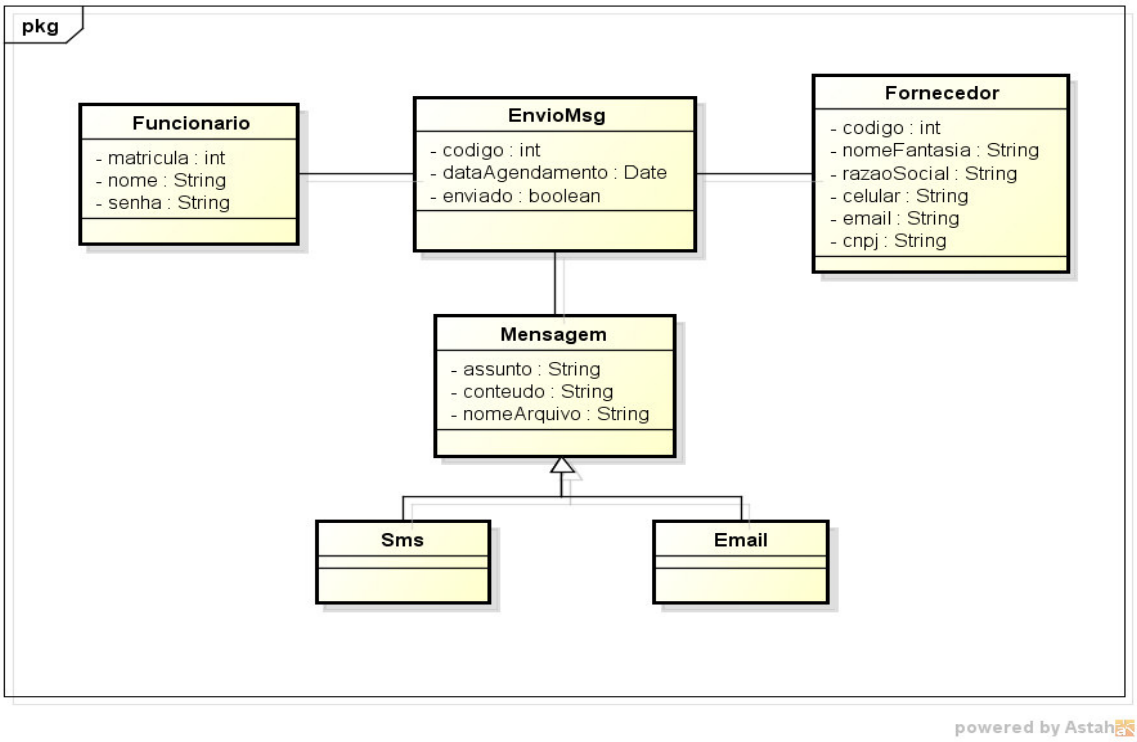

Figura 4.3 - Diagrama de Classes sem Aspectos.

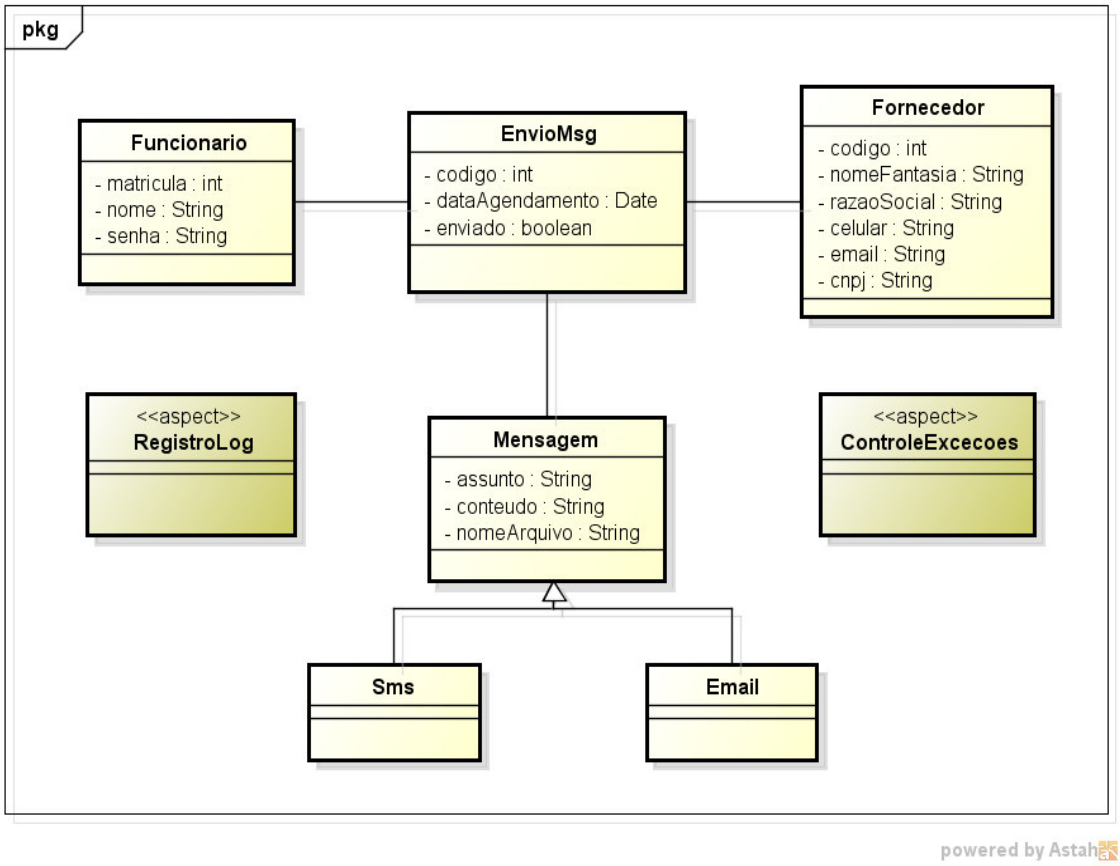

Figura 4.4 - Diagrama de Classes Com Aspectos.

4.3 Implementação

O sistema utilizado como estudo de caso foi projetado para ser implementado na linguagem Java, para armazenamento de dados foi utilizado o banco de dados Postgres [1] e o *framework* [2] de persistência de dados iBatis[3] foi utilizado para auxiliar nas operações de manipulação no banco de dados.

4.3.1 Diagrama de arquitetura

Ao se organizar a arquitetura do sistema, utilizou-se como base o conceito de separação de interesses, neste contexto o software foi dividido em dois pacotes macros que se referem primeiramente aos componentes que interagem diretamente com o usuário, denominado Visão, e os que se referem às funcionalidades do sistema. Os componentes do pacote Visão interagem com o usuário e com as funcionalidades implementadas por classes e aspectos. Na figura 4.5, é possível observar a estrutura do sistema, onde o *framework* iBatis atua como facilitador no acesso ao Banco de Dados, e as funcionalidades de agendamento e os interesses sistêmicos atuam diretamente no pacote com os componentes que interagem com o usuário.

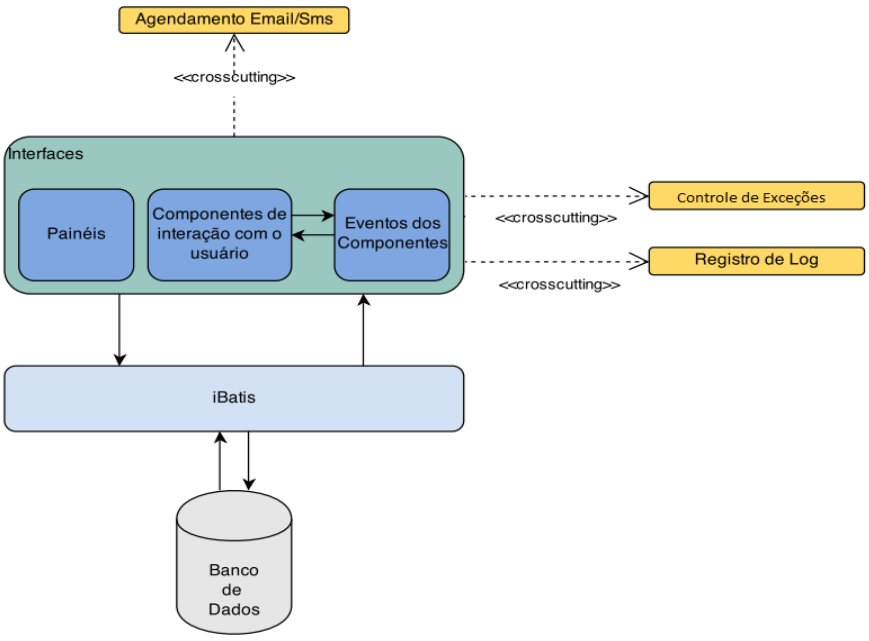

Figura 4.5 - Diagrama de Arquitetura Sistema Despachante.

[1] Sistema gerenciador de banco de dados relacional, desenvolvido como código aberto.
[2] Abstração que une códigos comuns entre vários projetos de software provendo uma funcionalidade genérica.
[3] Framework de mapeamento objeto-relacional, usando descritor XML, utilizado para programas em Java, .NET e Ruby.

4.3.2 Configuração do projeto

Tendo em vista que o projeto inicialmente implementado somente com Orientação a Objetos já estava quase que completamente implementado, segue as configurações que foram necessárias para esta nova proposta, com a modularização de interesses transversais.

- Adicionar as dependências relativas ao AspectJ, necessárias para habilita-lo. Parte do arquivo pom.xml, na figura 4.6 mostra estas dependências. Neste arquivo de configuração. ficam adicionadas todas as dependências e repositórios que o sistema irá precisar.

```
55        <dependency>
56            <groupId>org.springframework</groupId>
57            <artifactId>spring-aspects</artifactId>
58            <version>3.0.5.RELEASE</version>
59        </dependency>
60        <dependency>
61            <groupId>org.aspectj</groupId>
62            <artifactId>aspectjrt</artifactId>
63            <version>1.6.11</version>
64        </dependency>
65
66        <dependency>
67            <groupId>org.aspectj</groupId>
68            <artifactId>aspectjweaver</artifactId>
69            <version>1.6.11</version>
70        </dependency>
```

Figura 4.6 - Dependências AspectJ.

- No arquivo de configuração, adicionar a *tag* “<aop:aspectj-autoproxy />“ , que define o bean interceptor padrão. A figura 4.7 detalha o código adicionado.

```
WebDespachante-Context-Beans.xml
1  <beans xmlns="http://www.springframework.org/schema/beans"
2      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
3      xmlns:tx="http://www.springframework.org/schema/tx"
4      xmlns:sec="http://www.springframework.org/schema/security"
5      xmlns:context="http://www.springframework.org/schema/context"
6      xmlns:aop="http://www.springframework.org/schema/aop"
7
8      xsi:schemaLocation="http://www.springframework.org/schema/beans
9      http://www.springframework.org/schema/beans/spring-beans-3.0.xsd
10     http://www.springframework.org/schema/context
11     http://www.springframework.org/schema/context/spring-context-3.0.xsd
12     http://www.springframework.org/schema/tx
13     http://www.springframework.org/schema/tx/spring-tx-3.0.xsd
14     http://www.springframework.org/schema/security
15     http://www.springframework.org/schema/security/spring-security-3.0.xsd
16     http://www.springframework.org/schema/aop
17     http://www.springframework.org/schema/aop/spring-aop-3.0.xsd">
18
19
20
21     <aop:aspectj-autoproxy />
22
23     <bean id="LogAspect" class="web.despachante.aspecto.RegistroLog"/>
24     <bean id="ExcecaoAspect" class="web.despachante.aspecto.ControleExcecao"/>
25
```

Figura 4.7 - Arquivo de configuração da aplicação.

4.3.3 Log

Arquivos de log são muito importantes durante o processo de desenvolvimento de software, pois facilitam quando há a necessidade de encontrar problemas, conflitos e identificar falhas de funcionamento. Funcionam como registro das operações ocorridas durante a execução do sistema.

Diante da necessidade de implementação do registro de Log do sistema, caso fosse implementado da maneira convencional, seria necessário adicionar em cada método onde deseja-se efetuar algum registro de operação, as linhas de código referentes ao log.

Modularizando este interesse, tem-se a classe ResgistroLog como é detalhado na figura 4.8.

RegistroLog.java

```
1  package web.despachante.aspecto;
2
3  import java.util.Calendar;
10
11 @Aspect
12 public class RegistroLog {
13
14     private final Logger logger;
15
16     public RegistroLog() {
17         logger = Logger.getLogger(RegistroLog.class);
18     }
19
20     @Before("execution(* web.despachante.dao.*.*(..))")
21     public void logBefore(JoinPoint joinPoint) {
22
23         logger.info("chamada logBefore()");
24         logger.info("execução : " + joinPoint.getSignature().getName() + " em " + Calendar.getInstance());
25         logger.info("******");
26
27     }
28
29     @AfterReturning(pointcut = "execution(* web.despachante.dao.*.*(..))", returning = "result")
30     public void logAfterReturning(JoinPoint joinPoint, Object result) {
31
32         logger.info("chamada logAfterReturning() ");
33         logger.info("execução : " + joinPoint.getSignature().getName());
34         logger.info("retorno : " + result);
35         logger.info("******");
36
37     }
38
39 }
40
```

Figura 4.8 - Implementação da classe RegistroLog.

O método *logBefore* será acionado antes de todo método contido no pacote web.despachante.dao, ou seja, esta nomenclatura “dao” é comumente utilizada para pacotes que conterão as classes que fazem algum tipo se manipulação com o banco de dados. Desta forma, um único método, contido na classe que condiz com sua utilidade, será responsável por executar este interesse.

Da mesma forma, o método *logAfterReturning*, será acionado para todos os métodos do pacote “dao”, porém, após a execução destes métodos, registrando também o retorno do método executado.

4.3.4 Controle de exceções

O mecanismo de tratamento de exceções é de grande importância para sistemas baseados na Programação Orientada a Objetos, pois com ele é possível tratar erros e falhas que podem eventualmente ocorrer durante a execução programa. As formas de tratar exceções são infinitas, ficando a cargo do programador definir qual a mais apropriada para cada caso.

Dentre as formas mais comuns de tratamento de exceções estão: desfazer operações que foram realizadas até aquele ponto do programa, apresentar uma mensagem de aviso ao usuário, registrar o ocorrido num arquivo de log ou até mesmo todas elas ao mesmo tempo.

O tratamento de exceções, por padrão, é um interesse transversal, pois é de interesse de todo o sistema. Geralmente ele é implementado com códigos espalhados sobre todos os módulos existentes, porém é possível tratá-lo separadamente do restante dos outros códigos do sistema. Diante disso, se faz necessário utilizar a POA para centralizar todas as regras e estruturas relacionadas ao tratamento de exceções. A figura 4.9 demostra a modularização desse interesse, na classe ControleExcecao.

ControleExcecao.java

```
1  package web.despachante.aspecto;
2
3  import javax.faces.application.FacesMessage;
10
11 @Aspect
12 public class ControleExcecao {
13
14     private final Logger logger;
15
16     public ControleExcecao() {
17         logger = Logger.getLogger(RegistroLog.class);
18     }
19
20     @AfterThrowing(pointcut = "execution(* web.despachante.*.*.*(..))", throwing = "error")
21     public void afterThrowing(JoinPoint joinPoint, Throwable error) {
22
23         FacesContext.getCurrentInstance().addMessage(null,
24                 new FacesMessage(FacesMessage.SEVERITY_ERROR, "Algum erro ocorreu, consulte o administrador do sistema", "Erro"));
25         logger.info("erro");
26         logger.info("execução : " + joinPoint.getSignature().getName() + " erro " + error.getMessage());
27
28     }
29
30 }
31
```

Figura 4.9 - Implementação da classe ControleExcecao.

O método *afterThrowing* executará após qualquer método contido no pacote web.despachante e que apresente algum erro de execução.

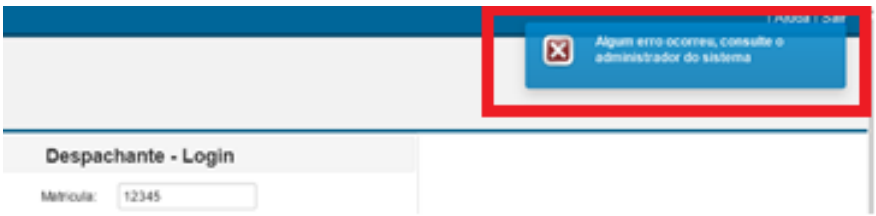

Figura 4.10 – Erro apresentado utilizando aspecto ControleExcecao

A figura 4.10 mostra o erro apresentado no momento do login no sistema. O detalhamento do erro foi apresentado no console do próprio eclipse, como mostra a figura 4.11.

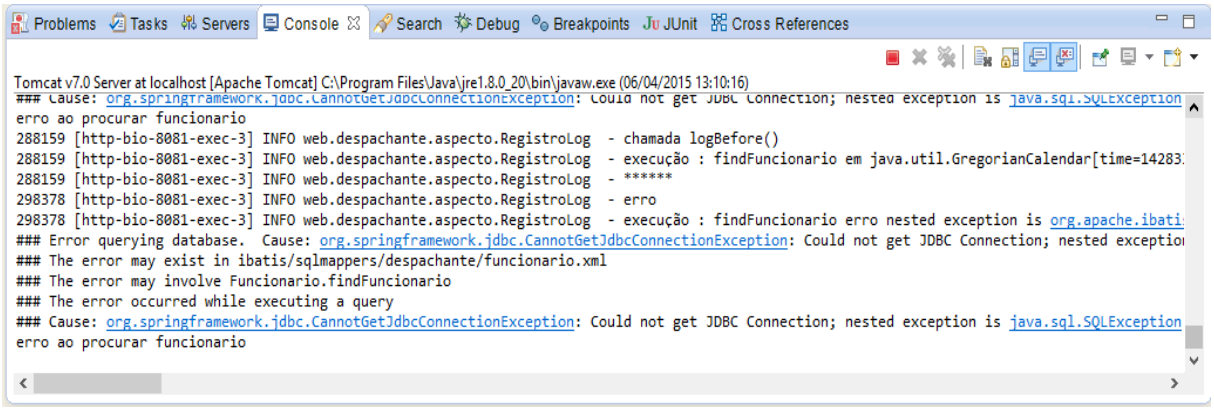

Figura 4.11 – Erro capturado pelo aspecto ControleExcecao.

Estas mudanças tem grande impacto quando se fala de reutilização de código, pois apenas esta implementação abrange vários métodos sem a necessidade de alterar o código destes. Tem-se, portanto, interesses transversais, que normalmente são implementados de forma espalhada por todo o sistema, agora modularizados na forma de aspectos, o que evidencia um ganho de reutilização e manutenção, bem como no próprio entendimento do código como um todo.

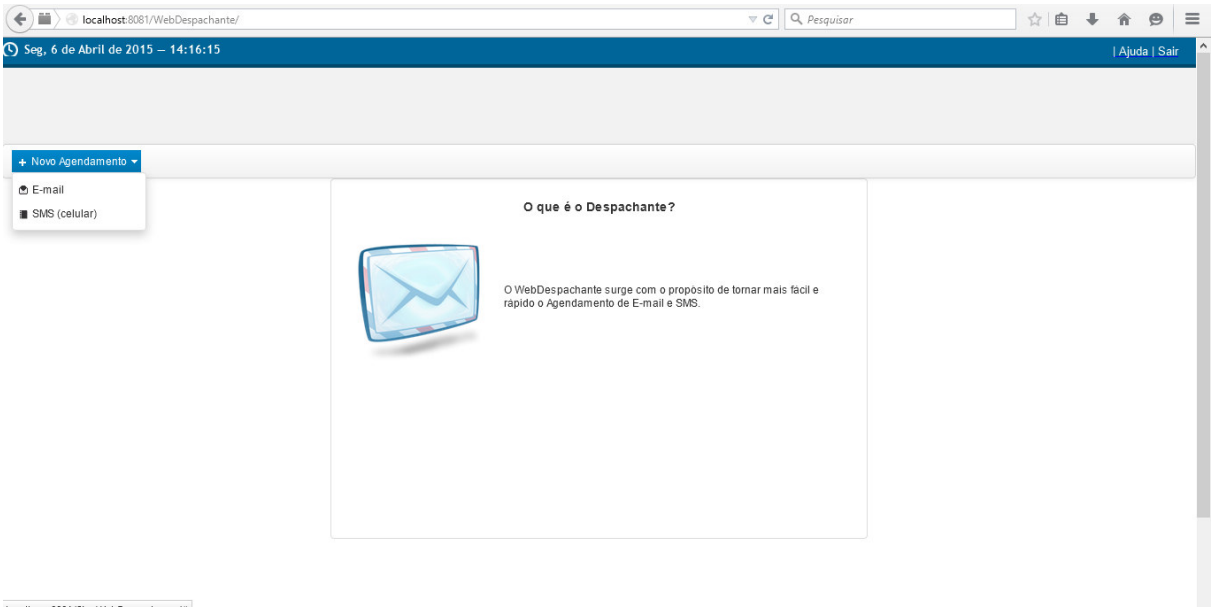

Figura 4.10 – Tela inicia sistema Despachante.

Vale ressaltar, porém, que não há mudanças na interface da aplicação, já que o objetivo do uso de aspectos é modularizar com mais eficiência, tendo impacto então somente no código-fonte da aplicação, como mostra a figura 4.12 com a tela inicial do sistema, que permaneceu a mesma antes e depois da utilização de aspectos.

5 CONCLUSÃO

A implementação da POA se mostra muito atrativa ao ser utilizada em complemento ao paradigma orientado a objetos. Foi possível verificar sua eficácia em modularizar interesses transversais, garantindo assim maior flexibilidade para a evolução dos sistemas.

Sobre a linguagem AspectJ, além da compatibilidade com a JVM, por ser uma extensão orientada a aspectos da linguagem JAVA, a linguagem se mostrou muito poderosa, podendo alcançar várias partes do sistema, possibilitando a implementação de funcionalidades de forma separada dos outros códigos do sistema, deixando o código muito mais legível. Além disso, é muito fácil fazer alguma modificação nessas classes, como inserir e remover aspectos.

Sobre o processo de tornar um sistema no paradigma orientado a objetos, para um sistema também com orientação a aspectos, foi observado que o esforço para isto pode ser considerado bem leve, pois com apenas algumas configurações e adição de dependências, os aspectos podem ser implementados. É necessário somente que o programador tenha um bom conhecimento sobre o código do sistema.

Uma dificuldade passada ao trabalhar no estudo de caso, foi diante do pouco material disponível abordando essa "transformação" de um sistema baseado somente em POO, para um sistema também com POA. Trata-se de configurações básicas como foi mostrado no estudo de caso, porém, quase não existem materiais abordando esta questão da configuração do projeto.

Diante do que foi proposto no trabalho, a documentação de um sistema com aspectos e a criação de aspectos no sistema já implementado, pode-se afirmar que o resultado obtido, bem como a facilidade de desenvolver essas tarefas possibilitam a conclusão de que seu uso favorece o desenvolvimento de software e sua manutenção. Com base no que foi desenvolvido, existem alguns pontos no que se refere à sugestões para trabalhos futuros, como pode ser citado:

- Formalização da implementação de outros interesses sistêmicos, como a própria camada de persistência, autenticação etc;
- Pesquisa sobre a integração do Paradigma de Programação Orientada a Aspecto no desenvolvimento da Tecnologia Adaptativa;
- Pesquisa sobre métodos de testes automatizados voltados para aspectos.

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

REFERÊNCIAS

SOMMERVILLE, Ian. **Engenharia de Software**. 8ª d. São Paulo: Pearson Addison-Wesley,2007.

LADDAD, Rammivas. **AspectJ in action**. Greenwich: Manning Publications Co, 2003.
GOETTEN Junior, WINCK Diogo, 2006. AspectJ – **Programação Orientada a Aspectos com Java**. São Paulo – SP. Novatec Editora, 2006.

GOETTEN, Junior; WINCK Diogo e MACHADO Caio, 2004. **Programação Orientada a Aspectos – Abordando Java e aspectJ**. Workcomp Sul – SBC.

CHAGAS José Daniel Ettinger, OLIVEIRA; Marcus Vinicius de Gois. **Programação Orientada a Objetos versus Programação Orientada a Aspectos: um estudo de caso comparativo através do desenvolvimento de um banco de questões**. UFS. São Cristóvão, 2009.

RESENDE, Antônio Maria Pereira; SILVA, Claudiney Calixto. **Programação Orientada a Aspectos em Java**. Rio de Janeiro : Brasport, 2005.

RICARTE, I. (2001) **Programação Orientada a Objetos: Uma Abordagem com Java**. UNICAMP. Disponível em: http://www.dca.fee.unicamp.br/cursos/PooJava/sobre.html.

ROYCE, W. Software Project Management - A Unified Framework.Addison- Wesley, 1998.
SANTOS, Rafael. **Introdução à Programação Orientada a Objetos Usando Java**. Campus, São Paulo, 2003.

SUN MICROSYSTEMS. (1997) **Java RMI Tutorial**. Revision 1.3, JDK 1.1 FCS, 1997. Disponível em: http://www.metz.supelec.fr/~vialle/course/SI/java-rmi/doc- JavaRMI.pdf

ELRAD, T.; FILMAN, R.E.; BADER, A. **Aspect–oriented Programming**. Communications of the ACM. Vol. 44, n. 10, 2001.

GRADECKI, J.; LESIEC, N. **Mastering AspectJ: aspect-oriented programming in Java**. John Wiley & Sons, 2003.

INFOWESTER, **Linguagem Java**. Disponível em: http://www.infowester.com/lingjava.php. Acesso em: 05/10/2014.

SOARES, Sérgio; BORBA, Paulo. **AspectJ - programação orientada a aspectos em java**. In: SIMPÓSIO BRASILEIRO DE LINGUAGENS DE PROGRAMAÇÃO, 6., 2002, Rio de Janeiro. Anais... Rio de Janeiro: [s.n.], 2002.

CHAVEZ, Christina von Flach Garcia; LUCENA, Carlos José Pereira de. **Um enfoque baseado em modelos para o design orientado a aspectos**. 2004. 298 f. Tese de Doutorado (Pós-Graduação em Informática) - Departamento de Informática, Pontifícia Universidade Católica do Rio de Janeiro.