Estudo de Caso Aplicando Programação Orientada a Aspecto

Marcel Hugo (FURB/DSC)
marcel@furb.br

Marcio Carlos Grott (Totall.com S.A.)
grott@totall.com.br

Resumo. Este artigo faz um apanhado geral da tecnologia de orientação a aspecto. É feito uma introdução do tema mostrando os detalhes teóricos da composição de um aspecto, as tecnologias que dão suporte ao desenvolvimento de software e a inserção dentro da engenharia de software. Para utilizar aspectos dentro do desenvolvimento de software existem *frameworks* que fornecem uma infra-estrutura tanto no fornecimento de uma linguagem de aspecto quanto à utilização de aspetos definidos em arquivo XML. É visto com maior grau de detalhamento a linguagem de aspecto *AspectJ* que é uma linguagem baseada na sintaxe da linguagem de programação orientada a objetos Java. Para demonstrar a utilização de aspecto é elaborado um estudo de caso consistindo de um modelo de troca de informações entre sistemas heterogêneos de envio e recebimento de notificação da compra de mercadorias.

Palavras-chave: *Aspectos*, POA, *AspectJ*, *Java*, *Framework*, *Engenharia de software*.

1 Introdução

A engenharia de software e as linguagens de programação coexistem em um relacionamento de suporte mútuo. A maioria dos processos de desenvolvimento de software considera um sistema em unidades (módulos) cada vez menores. As linguagens de programação, por sua vez, fornecem mecanismos que permitem a definição de abstrações de unidades de sistemas e a composição destas de diversas formas possíveis à produção do sistema como um todo (GRADECK; LESIECKI, 2003).

Com o advento da *Object Oriented Programming* (Programação Orientada a Objetos, POO), em meados dos anos 70 com a linguagem de programação Smaltalk, inicia-se um novo paradigma de desenvolvimento de software que está presente até os tempos atuais. A POO trouxe grandes avanços para o desenvolvimento de software, permitindo a construção de sistemas mais fáceis de serem projetados, maior reusabilidade de componentes, modularidade, implementações menos complexas e redução do custo de manutenção. Muitas aplicações não estão em apenas um único módulo de código contido em um único arquivo. Aplicações são coleções de módulos (classes) que trabalham juntas para fornecer determinadas funcionalidades para um conjunto de requisitos bem definidos (GRADECKI; LESIECKI, 2003). Porém, existem certas propriedades que se pretende programar em um sistema que não são adequadamente encapsuladas em classes, seja porque se aplicam as diversas classes de um sistema simultaneamente, seja porque não pertencem à natureza intrínseca da(s) classe(s) a(s) qual(is) se aplicam. A dificuldade de modularizar alguns tipos de funcionalidades causa um acúmulo de responsabilidade adicional que o projeto da classe não previa, ocasionando um entrelaçamento da responsabilidade inicial - intrínseca da classe - com responsabilidade extra.

Devido ao entrelaçamento de responsabilidades (*crosscutting*) de uma classe surge o paradigma de programação orientada a aspectos. A implementação dos aspectos fornece subsídio a POO para tratar de tais responsabilidades extras através da decomposição funcional da classe (SOARES; BORBA, 2002). O aspecto consiste de características estruturais e/ou comportamentais que devem ser adicionadas a diversas partes do sistema, mas que são projetadas e codificadas

separadamente. Quando o sistema é compilado os aspectos combinam as responsabilidades extras, ou ortogonais, com as responsabilidades intrínsecas de cada módulo afetado, fazendo com que todas as responsabilidades apresentem as propriedades pretendidas sem comprometer a coerência de cada módulo (LADDAD, 2003).

É importante salientar que a *Aspect-Oriented Programming* (Programação Orientada a Aspectos, POA) não propõem substituir nenhum dos paradigmas já existentes, e sim, acrescentar suporte a modularização das propriedades relativas ao *crosscutting concerns* (assuntos que entrecortam o sistema). A POA não é a única proposta para suprir tal necessidade dos paradigmas em uso atualmente. Existem, por assim dizer, uma família de soluções similares, como filtros de composição, reflexão computacional, *hyper-spaces*, *Subject-Oriented Programming* (SOP), que são coletivamente denominadas de *Advanced Separations of Concerns* (técnicas avançadas de separação de assuntos) (CHAVES, 2002).

Este artigo tem por objetivo mostrar a aplicação da programação orientada a aspectos na especificação de um software orientado a objetos em camadas, implementando em *AspectJ* um aspecto do sistema – o log da leitura de arquivos XML. O artigo organiza-se em quatro seções que apresentam a orientação a aspectos, sua implementação e o estudo de caso realizado. A última seção apresenta as conclusões alcançadas.

2 Desenvolvimento de software orientado a aspectos

O advento da programação orientada a objetos tornou o desenvolvimento de sistemas mais reutilizável, flexível, com maior facilidade de manutenção e principalmente o desenvolvimento de módulos que encapsulam funcionalidades específicas do sistema. Porém o desenvolvimento orientado a objetos tem algumas limitações que são mais difíceis de serem tratadas no âmbito de objetos. Estas limitações são identificadas (OSSHER et al., 1996; OSSHER; TARR, 1999), como o entrelaçamento e espalhamento de código com diferentes propósitos, por exemplo, o entrelaçamento de código de negócio com código de apresentação, e o espalhamento de código de acesso a banco de dados em várias classes. Algumas destas limitações podem ser solucionadas com o uso de padrões de projeto (GAMA et al., 2000).

Existem algumas extensões do paradigma de orientação a objetos que tentam solucionar as suas limitações. Pode-se citar a *Aspect-Oriented Programming (*Programação Orientada a Aspectos, POA) (ELRAD; FIRMAN; BASDWER, 2001), *Subject-Oriented Programming (*Programação Orientada a Sujeito, POS) (OSSHER; TARR, 1999) e *Adaptive Programming (*Programação Adaptativa, PA). Estas técnicas visam obter uma maior modularidade de software em situações práticas onde a programação orientada a objetos e *patterns* não oferecem suporte adequado (SOARES; BORBA, 2002).

Dentro deste contexto de melhoria da qualidade e reusabilidade de código, a POA tem demonstrado ser promissora, pois procura solucionar a ineficiência em capturar algumas das principais decisões de projeto que um sistema deve implementar. Estas decisões de projeto se tornam distribuídas dentro do projeto como, por exemplo, log da aplicação, acesso a banco de dados, *cache* de objetos, tratamento de exceções, resultando num espalhamento e entrelaçamento de decisões de diferentes propósitos, aumentando a complexidade e dificultando a manutenção. Assim, a POA aumenta a modularidade separando o código que implementa funções específicas que afetam diferentes partes do sistema, chamadas preocupações ortogonais (*crosscutting concerns*). Alguns exemplos de *crosscutting concerns* é persistência, distribuição, controle de concorrência, tratamento de exceções e depuração. O aumento da modularidade implica em sistemas mais legíveis e reutilizáveis, os quais são mais facilmente projetados e mantidos.

2.1 Composição de um sistema baseado em aspectos

Um sistema que utiliza programação orientada a aspecto é composto dos seguintes componentes:

❖ *Linguagem de componentes*. Segundo Irwin et al. (1997), "a linguagem de componente deve permitir ao programador escrever programas que implementem as funcionalidades básicas do sistema, ao mesmo tempo em que não provêem nada a respeito do que deve ser implementado na linguagem de aspecto". As linguagens de componentes focalizam-se na resolução dos problemas encontrados durante a fase de análise preocupando-se em resolver os problemas de negócio. As linguagens de aspectos fornecem módulos com funcionalidades para resolver problemas de infra-estrutura (arquitetônico) da aplicação que podem ser reutilizados em diversos projetos com pouca ou nenhuma alteração.

❖ *Linguagem de aspecto*. A linguagem de 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 (IRWIN et al., 1997);

❖ *Programas de componentes*: é o arquivo-fonte escrito (*source code*) em determinada linguagem de programação, por exemplo, Java, C, C++, Delphi, etc, onde o desenvolvedor codifica as regras de negócio do sistema em classes conforme definido durante a fase de análise do projeto.

❖ *Um ou mais programas de aspectos*: é o arquivo fonte (*source code*) codificado em uma linguagem de aspecto, como *AspectJ* (abordada adiante), *AspectC*, *AspectWerkz*, *Dynaop*. Algumas linguagens são voltadas a preocupações específicas, e neste caso devem ser utilizadas múltiplas linguagens de aspectos, uma para cada preocupação. Um exemplo de uso de múltiplas linguagens de aspectos aparece no *framework* D, onde são utilizadas duas linguagens de aspectos: *Ridl* (para distribuição) e COOL (para concorrência).

❖ *Combinador de aspectos*. A tarefa do combinador de aspectos (*aspect weaver*) é combinar os programas escritos em uma linguagem de componentes (as classes onde estão codificadas as regras de negócio) com os programas escritos em linguagem de aspectos (classes de aspectos). Neste trabalho será utilizada a linguagem de programação Java e o *AspectJ* como linguagem de aspecto.

Os programas de aspectos, que podem estar implementados em múltiplas linguagens de aspecto, implementam as responsabilidades ortogonais. O código do aspecto torna explícito o comportamento que será integrado ao código dos componentes e, em que contexto tal integração ocorre: são os chamados *join points (*pontos de junção), que são elementos semânticos da linguagem de componentes com os quais os programas de aspectos se coordenam. Exemplos de pontos de junção comuns são: invocações de métodos (chamadas ou recebimentos), geração de exceções, criação de objetos, entre outros.

O combinador de aspectos identifica nos componentes pontos de junção onde os aspectos se aplicam, produzindo o código final da aplicação, que programam tanto as propriedades definidas pelos componentes quanto aquelas definidas pelos aspectos. Combinadores de aspectos podem atuar em tempo de compilação ou execução. Implementações de combinadores em tempo de execução têm a possibilidade interessante de permitir a adição/exclusão de aspectos com a aplicação em pleno funcionamento. Uma visão geral de todo este processo é apresentada na figura 1.

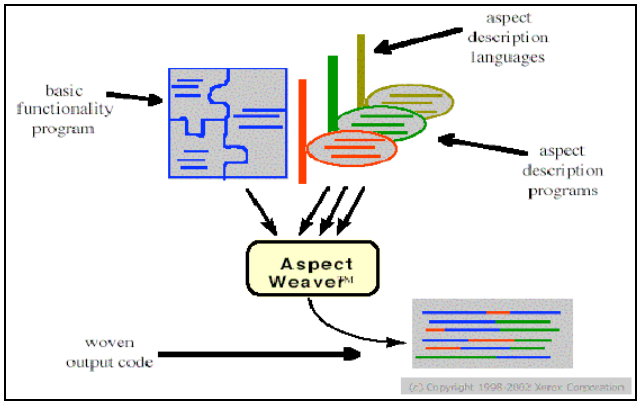

Figura 1: Composição de sistemas utilizando programação orientada a aspectos

Com este novo paradigma de desenvolvimento, POA, os analistas devem estar mais atentos em suas especificações para separar as preocupações ortogonais e os programadores devem ter em mente que as responsabilidades ortogonais devem ser deixadas para os aspectos. Assim, os programadores podem se concentrar nas reais necessidades da aplicação e o que não for inerente à resolução do problema ficará codificado em programas de aspectos. Pode-se observar que começará a existir equipes dedicadas somente com o desenvolvimento de aspectos que poderão estar presentes em diversos projetos. Enquanto, outras equipes somente estarão preocupadas com o desenvolvimento dos requisitos funcionais do sistema, utilizando os aspectos criados para compor os requisitos não funcionais do projeto.

2.2 Linguagens, propósito e características

As linguagens de aspecto de propósito específico, como o próprio nome informa, são linguagens que tratam de comportamentos específicos dentro de um sistema, como por exemplo, sincronismo, tratamento de exceções, não fornecendo suporte a qualquer outro tipo de componente. As linguagens de propósito específico possuem um nível de abstração mais alto que a linguagem de componentes para a qual foi projetada. Para garantir a utilização dos aspectos conforme foram projetadas, essas linguagens geralmente impõem alguma restrição no uso da linguagem de componente. Isto é, a linguagem de aspecto poderá impedir a utilização de determinadas palavras que são reservadas da linguagem de componentes para evitar que a linguagem de componente implemente preocupações, que seriam aspectos (STAINMACHER, 2003).

As linguagens de aspecto de propósito geral permitem a implementação de qualquer tipo de aspecto. Não é possível programar restrições junto à linguagem de componentes, pois, geralmente ambas possuem o mesmo conjunto de instruções. Como exemplo pode-se citar a linguagem *AspectJ*, que possui a linguagem Java como linguagem de componente.

A linguagem de propósito geral é mais familiar e de fácil adoção para o desenvolvimento, uma vez que poderá compartilhar o mesmo ambiente de desenvolvimento utilizado pela linguagem de componente. Como características têm:

❖ Capacidade de implementar todos os aspectos do sistema sem a necessidade de troca de ferramenta para implementar determinados aspectos não suportados pelo ambiente;

❖ Não é necessário o aprendizado de uma nova linguagem a cada aspecto que se implementa;

❖ Quando se implementa um único aspecto para uma determinada aplicação uma linguagem de aspecto específica terá um melhor desempenho e maior facilidade de desenvolvimento do aspecto. Em se tratando de uma aplicação de múltiplos aspectos a utilização de diferentes linguagens de aspecto torna-se inadequada por impor restrições à linguagem de componente a cada linguagem de aspectos que se necessite, podendo desencadear erros nos aspectos já implementados e dificultando a legibilidade da aplicação como um todo e, principalmente a manutenção da aplicação.

3 AspectJ

Aspects, os elementos básicos desta abordagem, podem alterar a estrutura estática ou dinâmica de um programa. A estrutura estática é alterada adicionando, por meio das declarações *inter-types*, membros (atributos, métodos ou construtores) a uma classe, modificando assim a hierarquia do sistema. Já a alteração em uma estrutura dinâmica de um programa ocorre em tempo de execução por meio de *join points*, os quais são selecionados por *pointcuts*, e através da adição de comportamentos (*advices*) antes ou depois dos *join points* (KISELEV, 2002).

3.1 Crosscutting elements

AspectJ utiliza extensões da linguagem de programação Java para construir novas regras para preocupações dinâmicas e estáticas. As extensões de *AspectJ* são projetadas para que programadores sintam-se a vontade enquanto estão programando os aspectos. O *AspectJ* utiliza construções próprias para os blocos de aspecto: a implementação dos aspectos é construída em blocos que formam os módulos que expressam os interesses ortogonais.

Os elementos básicos em *AspectJ* são: *join points, pointcuts, advices*, declarações *inter-types* e *aspects*. Os tópicos seguintes conceituam cada um desses elementos de *AspectJ*.

3.2 Join points

O conceito de *Join Point* é fundamental para o entendimento de *AspectJ*. O *Join Point* é qualquer ponto de execução identificado dentro de um sistema. O *AspectJ* pode operar sobre os seguintes tipos de *join points*: a) chamada de métodos, b) execução de métodos, c) chamada de construtores, d) execução de inicialização, e) execução de construtores, f) execução de inicialização estática, g) pré-inicialização de objetos, h) inicialização de objetos, i) referência a campos, j) tratamento de exceções (GRADECKI; LESIECKI, 2003).

3.3 Pointcuts

Em *AspectJ*, um aspecto normalmente define *pointcuts*, que são formados pela composição de pontos de combinação, através de combinadores lógicos. Para definir um *pointcut* utiliza-se construtores de *AspectJ* chamados designadores de *pointcuts* (*pointcuts designators*).

Um *pointcut designator* identifica o *pointcut* pelo nome ou por uma expressão. Os termos *pointcut* e *pointcut designator* são usados freqüentemente como sinônimos. Pode ser declarado um *pointcut* dentro de um aspecto, classe ou interface. Da mesma forma que atributos e métodos de classes, pode ser especificado um qualificador de acesso aos *pointcuts* (*public, private, protected* ou *default*) para restringir o acesso.

Em *AspectJ*, um *pointcut* pode ter um nome ou ser anônimo. *Pointcuts* anônimos, como classes anônimas, são definidas no lugar onde serão utilizadas, tais como parte de uma *advice*, ou

no momento da definição de outro *pointcut*. *Pointcuts* nomeados são elementos que podem ser referenciados de múltiplos lugares, aumentando a reusabilidade.

3.4 Advice

Advice é o código para ser executado em um *join point* que está sendo referenciado pelo *pointcut*. *Advice* pode ser executado antes, durante ou depois de um *join point*. O *advice* pode modificar a execução do código no *join point*, pode substituir ou passar por ele. Usando o *advice* pode-se "logar" as mensagens antes de executar o código de determinados *join points* que estão espalhados em diferentes módulos. O corpo de um *advice* é muito semelhante ao de qualquer método, encapsulando a lógica a ser executada quando um *join point* é alcançado (GRADECKI; LESIECKI, 2003).

3.5 Introduction

Introduction é um *crosscutting* estático que introduz alterações nas classes, interfaces e aspectos do sistema. Alterações estáticas em módulos não tem efeito direto no comportamento. Por exemplo, pode ser adicionado um método ou um atributo na classe.

3.6 Compile-time declaration

O *compile-time declaration* é uma declaração de *crosscutting* estático que permite adicionar alerta e erro quando se detecta certos padrões de utilização de classes. Por exemplo, pode-se declarar que é um erro fazer qualquer chamada a classes de *Abstract Window Toolkit* (AWT) em um *Enterprise Java Beans* (EJB).

3.7 Aspect

O *aspect* é a unidade central do *AspectJ* da mesma forma que a classe é a unidade central de Java. Nele estão implementadas as regras de construção de *crosscutting* dinâmico e estático. *Pointcut*, *advice*, *introduction* e *declarations* são combinados em um *aspect*. Como qualquer classe Java, os *aspects* podem conter atributos, métodos e classes internas.

Antes de começar a implementar o comportamento do *crosscutting*, precisa ser feita uma análise para identificar os *join points* e quais argumentos de comportamentos se desejam modificar, para que em seguida seja definido o novo comportamento. Para iniciar a implementação se escreve um *aspect* que serve como um modelo que contém uma implementação geral. Então, dentro do *aspect*, escrevem-se os *pointcuts* para capturar os *join points* desejados. Finalmente, se escreve os *advice* para cada *pointcut* e codifica-se a ação que se deseja executar quando o *join point* for chamado.

3.8 A lógica de compilação

O *AspectJ* oferece um conjunto de ferramentas para auxiliar na criação de aspectos, desde um compilador de aspectos, visualizadores de aspectos, *plugins* para integração com as mais populares IDE´s de desenvolvimento. Neste trabalho é utilizada a IDE Eclipse versão 3.0 com o *plugin* ADJT, desenvolvido pelo mesmo grupo de desenvolvedores da IDE, disponíveis no site da IDE (www.eclipse.org).

3.9 O compilador

O compilador é a unidade central da implementação da linguagem *AspectJ*. Ele é responsável pela combinação das classes Java com as unidades de aspectos produzindo os arquivos para serem utilizados em produção. O compilador de aspectos aceita arquivos de classes Java, unidades de

aspectos ou uma mistura de ambos. Os arquivos resultantes da compilação contêm puro *bytecodes* Java e por isto pode ser executado em qualquer máquina virtual Java.

4 Análise e desenvolvimento de um protótipo para importação de dados

Segundo Wazlawick (2004), o processo de desenvolvimento de software se divide em quatro grandes fases: análise, projeto, implementações e teste. Neste artigo será desenvolvido um ciclo de desenvolvimento que corresponde à execução das quatro etapas. Na fase de análise será feita a investigação do problema para descobrir o problema a ser resolvido.

O problema a ser resolvido corresponde a um problema do mundo real encontrado durante a análise de um B2B (*bussines to bussines*) solicitado por um cliente da empresa Totall.com S.A., que precisava que as compras de mercadorias para reposição do estoque fossem geradas e enviadas pelo sistema da Totall.com S.A. aos devidos fornecedores.

Para que pudesse dar a solução ao cliente, fez primeiramente um levantamento de requisitos avaliando as reais necessidades do cliente. Após, feito o levantamento de requisitos procedeu-se a definição dos casos de uso conforme define a especificação da UML utilizando para isto a ferramenta CASE Jude que podem ser visto na Figura 2.

Os casos de uso encontrados foram modelados para que fosse possível ter uma melhor visualização da interação do usuário com o protótipo, conforme apresentados na Figura 2.

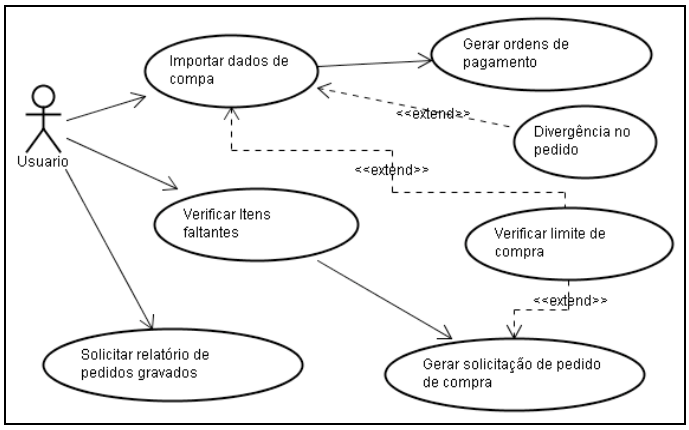

Figura 2: Modelo UML dos casos de uso

Para melhor organizar e visualizar os casos de uso se criou uma tabela que contém as informações sobre os casos e uma referência aos requisitos que compõem o caso. Na **Tabela 1** é apresentada à descrição dos casos de uso que compõem o protótipo.

Tabela 1: Listagem dos casos de uso

Nome	Atores	Descrição
Importar dados de compra	Usuário	O usuário, ao receber o arquivo referente à compra de mercadorias para o seu estabelecimento, inicia a importação dos dados da compra. Os arquivos já têm um formato predefinido entre o fornecedor e o lojista, sendo definido em formato XML, que após processado dará origem às ordens de pagamento a serem feitas ao fornecedor pelo departamento financeiro da loja. O sistema deverá fazer à baixa do pedido de compra gerado para o fornecedor informando que o pedido está concluído.
Verificar Itens faltantes	Usuário	O usuário uma vez por dia verificará quais os produtos que necessitará comprar. Caso existam produtos, o sistema deverá gerar os pedidos de compra.
Solicitar relatório	Usuário	O usuário ao final da importação poderá solicitar um relatório com o(s) pedido(s) que foram

de pedidos gravados		recebidos (importados) do fornecedor. A geração do relatório poderá ter alguns filtros que são: data, fornecedor, número do pedido.
Gerar ordens de pagamento	Usuário	Após feita a importação das compras deverá ser feita à geração da ordem de pagamento ao fornecedor. A ordem de pagamento gerada deverá notificar, por e-mail, o departamento financeiro da loja que em determinada data deverá ser feito o pagamento da compra.
Gerar solicitação de pedido de compra ao fornecedor	Usuário	As gerações dos pedidos de compras são feitos pelo usuário que depois de conferido o relatório de necessidades de mercadorias, deverá encaminhar aos fornecedores dos produtos o pedido de compra. Somente existirá um fornecedor por pedido de compra.

Encerrada a fase de levantamento de requisitos e definição dos casos de uso pode-se desenvolver o modelo da camada de domínio do protótipo. Este modelo visa produzir uma solução para o problema identificado pela análise, sem nenhuma referência às classes de aspecto bem com a sua interação dentro do protótipo (**Figura 3**).

Após definidas as classes de negócios contemplando toda a estrutura necessária para resolver os requisitos levantados no cliente e definidos nos diagramas de use-case verificou-se que havia classes que não faziam parte do problema a ser resolvido e que eram de extrema importância. Desta forma iniciou-se o levantamento destas classes e ao final do levantamento das classes faltantes concluiu-se eram classes que constituíam os aspectos da aplicação. Estes aspectos são demonstrados nos diagramas de classes com o estereótipo "Aspect" e na cor amarela.

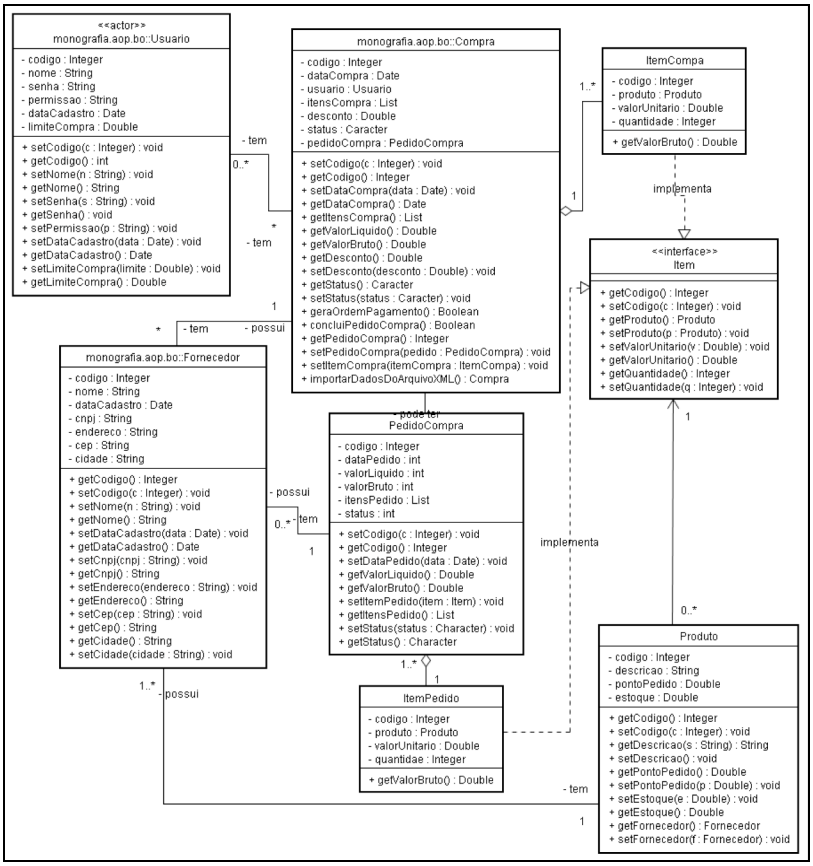

Figura 3: Diagrama de classe do protótipo

Na **Figura 4**, se tem o diagrama de classe que faz a importação dos dados oriundos do fornecedor referente à compra feita. Neste diagrama é colocado um log das operações de importação dos pedidos. Para que não seja necessário configurar um log para cada classe deste pacote, utiliza-se um aspecto que faz um log dos métodos das classes desejadas. O aspecto responsável pelo log é nomeado de AspectLog.

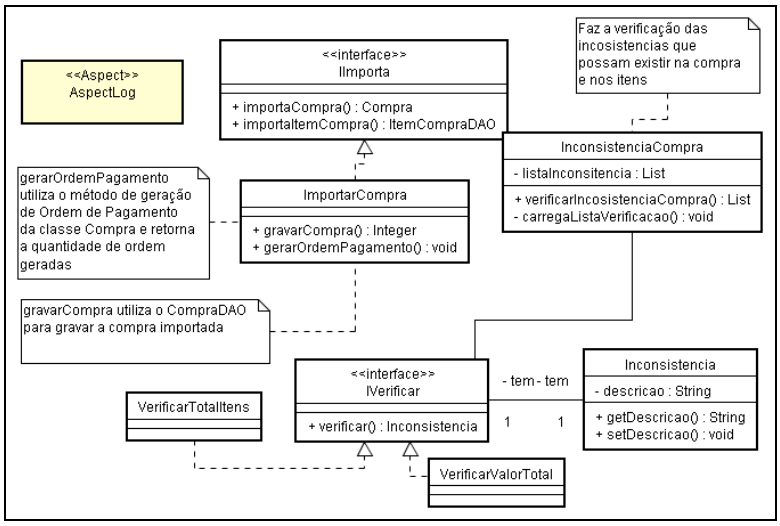

Figura 4: Diagrama de classe para importação dos pedidos

Na **Figura 5** tem-se o diagrama de classe utilizado para fazer o log das operações do protótipo definido em uma classe de aspecto. A classe de aspecto chamada de LogAspecto é utilizada para fazer gravação e leitura de arquivo(s) de log da classe GravarLog, responsável por fazer a persistência dos dados recebidos em um arquivo.

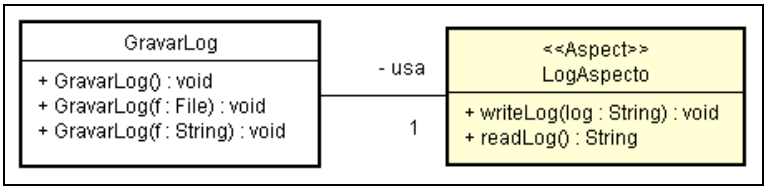

Figura 5: Diagrama de classe do aspecto que registra o log do sistema

4.1 Codificação

Com a análise do estudo de caso encerrada e elaborado os diagramas de classes pode ser feita à codificação. A codificação utiliza a linguagem de programação Java da Sun Microsystem e o *framework* de aspectagem *AspectJ* versão 1.2 desenvolvido por um consórcio de empresas e abrigado no projeto principal da IDE Eclipse.

Este protótipo está divido em duas camadas denominadas de camada de acesso a uma fonte de dados, que pode ser um banco de dados relacional, orientado a objeto, arquivos simples ou arquivo XML, e a camada de regras de negócio onde se tem os objetos que tratam da regra de negócio da aplicação. A camada de interface não será desenvolvida, pois não é o foco principal do

trabalho. Existe ainda uma camada de objeto paralela a estas duas que compreende aos objetos que fazem o transporte de dados entre as camadas.

No **Quadro 1**, apresenta-se um aspecto de log de todas as operações que serão realizadas com as classes que pertencem ao pacote `monografia.`

```
package monografia.aop.aop;
import java.text.*;
import java.util.Date;
public aspect LogAspecto {
    private    static    final    DateFormat    horaFmt    =    new
SimpleDateFormat("HH:mm:ss''S");
    pointcut recepcao() :
        execution(* monografia..*.*(..))
        && !within(Log);
    before() : recepcao() {
        System.out.println(getTimestamp() + " Iniciando "
                + thisJoinPoint.getSignature() +":" );
        Object[] arg = thisJoinPoint.getArgs();
        for(int i=0;i<arg.length;i++){
            System.out.print("arg:" + arg[i].toString() + "");
        }
    }
    after() : recepcao() {
        System.out.println(getTimestamp() + " Concluindo "
                + thisJoinPoint.getSignature()  );
    }
    public String getTimestamp() {
        return "[" + horaFmt.format(new Date()) + "]";
    }
}
```

Quadro 1: Aspecto responsável por fazer o log da aplicação

No **Quadro 2**, apresenta-se o código necessário para fazer a execução da classe de `Compra,` encontrada no diagrama de classes da **Figura 3,** que utiliza o aspecto de log para gravar as suas operações.

```
package monografia.aop.test;
import java.util.Date;
import monografia.aop.vo.Compra;
import monografia.aop.vo.Fornecedor;
import monografia.aop.vo.ItemCompra;
import monografia.aop.vo.Produto;
public class CompraTest {
    public static void main(String[] args) {
        Compra compra = new Compra();
        compra.setCodigo(Integer.valueOf(1));
        compra.setDataCompra(new Date());
        compra.setDesconto(Double.valueOf(10));
        compra.setFornecedor(new Fornecedor("02882917945"));
        compra.setStatus(Character.valueOf('C'));
        //   método   que   realiza   a   importação   dos   dados   do
arquivo XML
        compra.importarDadosDoArquivoXML();

        System.out.println("done");
    }}
```

Quadro 2: Trecho de código utilizado para demonstrar a utilização do aspecto de log.

Para demonstrar a execução do aspecto no **Quadro 3**, é capturado um trecho do resultado da execução do aspecto de log `LogAspecto`.

```
[21:04:10'484] Iniciando void
monografia.aop.test.TesteCompra.main(String[]):
arg:[Ljava.lang.String;@8813f2[21:04:10'609] Iniciando void
monografia.aop.vo.Compra.setCodigo(Integer):
arg:1[21:04:10'609] Concluindo void
monografia.aop.vo.Compra.setCodigo(Integer)
[21:04:10'609] Iniciando void
monografia.aop.vo.Compra.setDataCompra(Date):
arg:Sun May 08 21:04:10 BRT 2005[21:04:10'625] Concluindo void
monografia.aop.vo.Compra.setDataCompra(Date)
[21:04:10'625] Iniciando void
monografia.aop.vo.Compra.setDesconto(Double):
arg:10.0[21:04:10'625] Concluindo void
monografia.aop.vo.Compra.setDesconto(Double)
[21:04:10'625] Iniciando void
monografia.aop.vo.Fornecedor.setCnpj(String):
arg:02882917945[21:04:10'625] Concluindo void
monografia.aop.vo.Fornecedor.setCnpj(String)
[21:04:10'625] Iniciando void
monografia.aop.vo.Compra.setFornecedor(Fornecedor):
arg:monografia.aop.vo.Fornecedor@de6f34[21:04:10'625] Concluindo void
monografia.aop.vo.Compra.setFornecedor(Fornecedor)
[21:04:10'625] Iniciando void
monografia.aop.vo.Compra.setStatus(Character):
arg:C[21:04:10'625] Concluindo void
monografia.aop.vo.Compra.setStatus(Character)
done
[21:04:10'625] Concluindo void
monografia.aop.test.TesteCompra.main(String[])
```

Quadro 3: Listagem da execução do aspecto de log.

5 Conclusão

A programação orientada a aspecto prova ser uma nova abordagem de desenvolvimento de software caracterizada pela capacidade de abstração de módulos que não fazem parte da análise do problema solicitado pelo cliente. Outra grande melhoria que traz é a capacidade de reutilização de grande parte dos módulos desenvolvidos, sendo a reutilização algo que se almeja cada vez mais no desenvolvimento orientado a aspecto.

Durante o desenvolvimento do protótipo pôde-se notar a facilidade com que foi implementado um aspecto de log utilizado por todas as demais classes sem a necessidade de fazer a programação manual em cada classe afetada por este aspecto. Por outro lado, houve certa dificuldade no desenvolvimento de aspectos no sentido de realizar a análise do problema sem se preocupar com os aspectos ortogonais da aplicação, ou seja, ao longo da análise sempre temos em mente todo o arcabouço de recursos necessários para resolução do problema, que nem sempre faz parte do domínio do problema e sim de sua implementação. Isto faz com que não consigamos facilmente abstrair todos estes recursos e pensar somente na resolução do problema proposto.

Foi possível verificar que a programação orientada a aspecto traz de forma transparente e com resultados imediatos a reutilização de unidades de códigos. O aspecto de log desenvolvido para fazer o log de uma importação de dados de compras do fornecedor poderá ser utilizado em todas as classes de módulo como também, em vários outros sistemas sem a necessidade de recodificação de um aspecto de log.

Referências

CHAVES, Rafael Alves. **Aspectos e Midlware**. 2002. 50f. Trabalho Individual submetido à Universidade Federal de Santa Catarina como parte dos requisitos para obtenção do grau de Mestre em Ciências da Computação, Florianópolis.

ELRAD, T., FILMAN, R. E., and BADER, A. (2001). **Aspect–oriented programming. Communications of the ACM**, 44 (10):29–32.

GAMMA, Erich, et al. **Padrões de Projeto: soluções reutilizáveis de software orientado a objetos**. Trad. Luiz A. Meirelles Salgado. Porto Alegre: Bookman, 2000.

GRADECK, Joe, LESIECKI, Nicolas. **Mastering AspectJ: aspect-oriented programming in Java**. Indianopolis, Indiana. Wiley, 2003, p. 453.

IRWIN, John, et al. **Aspect – Oriented Programming**. Proceeding of ECOOP'97, Finland: Springer - Verlag, 1997.

KISELEV, I. **Aspect – Oriented Programming with AspectJ**. Ed. Sams Publishing, 2002.

LADDAD, Ramnivas. **AspectJ in Action: Practical Aspect-Oriented Programming**. Ed. Mannig, Canada, [2003].

OSSHER, H., et al (1996). **Specifying suject–oriented composition**. TAPOS, 2 (3):179–202. Special Issue on Subjectivity in OO Systems.

OSSHER, H. and TARR, P. (1999). **Using subject–oriented programming to overcome common problems in object–oriented software development/evolution**. In International Conference on Software Engineering, ICSE'99, pages 698–688. ACM, 1999.

SOARES, S. and BORBA, P. (2002). **Progressive implementation with aspect–oriented programming**. In Verlag, S., editor, The 12th Workshop for PhD Students in Object-Oriented Systems, Malaga, Spain. To appear.

STAINMACHER, Igor Fábio. **Estudo de Princípios para Modelagem Orientada a Aspectos**. 2003. 70 f. Trabalho de conclusão apresentado ao Curso de Ciências da Computação, do Centro de Tecnologia da Universidade Estadual de Maringá, Maringá.

WAZLAZWICK, Raul Sidnei. **Análise de Projeto de Sistemas de Informação Orientados a Objetos**. Rio de Janeiro: Elsevier, 2004.