UNIVERSIDADE REGIONAL DE BLUMENAU

CENTRO DE CIÊNCIAS EXATAS E NATURAIS

CURSO DE CIÊNCIAS DA COMPUTAÇÃO – BACHARELADO

PROTÓTIPO DE UM *WEAVER* PARA PROGRAMAÇÃO ORIENTADA A ASPECTOS EM DELPHI

EDMAR SOARES DE OLIVEIRA

BLUMENAU
2006

2006/1-08

EDMAR SOARES DE OLIVEIRA

PROTÓTIPO DE UM *WEAVER* PARA PROGRAMAÇÃO ORIENTADA A ASPECTOS EM DELPHI

Trabalho de Conclusão de Curso submetido à Universidade Regional de Blumenau para a obtenção dos créditos na disciplina Trabalho de Conclusão de Curso II do curso de Ciências da Computação — Bacharelado.

Prof. Marcel Hugo, M.Eng. - Orientador

BLUMENAU
2006

2006/1-08

PROTÓTIPO DE UM *WEAVER* PARA PROGRAMAÇÃO ORIENTADA A ASPECTOS EM DELPHI

Por

EDMAR SOARES DE OLIVEIRA

Trabalho aprovado para obtenção dos créditos na disciplina de Trabalho de Conclusão de Curso II, pela banca examinadora formada por:

Presidente:	Prof. Marcel Hugo, M.Eng. – Orientador, FURB
Membro:	Prof. Everaldo Artur Grahl, M.Eng. – FURB
Membro:	Prof. Jomi Fred Hübner, Dr. – FURB

Blumenau, 06 de julho de 2006

Dedico este trabalho a meus pais, José Soares e Rosa. Pessoas fundamentais na minha vida, meu alicerce, meus ídolos. A distância física que nos separa e a saudade são grandes, mas a vontade de vencer é bem maior. Essa vitória também é de vocês.

AGRADECIMENTOS

A Deus, pela força, sabedoria e inteligência concedida para chegar até aqui, e por me guiar e me acompanhar todos os dias de minha vida.

Aos meus pais, pela catequese, educação e formação que recebi. Pai, ainda lembro do dia em que achou que eu estava ficando louco por querer vim para tão longe. Mas acreditou, incentivou e confiou nos meus ideais. Obrigado pela compreensão, confiança e apoio que sempre recebi de vocês.

Aos demais familiares, em especial a minha irmã Marly. As conversas que tivemos ao longo desses anos foram importantes em algumas decisões que tomei.

Ao meu orientador, professor Marcel Hugo, pelos ensinamentos, confiança e apoio na realização deste trabalho.

À professora Joyce Martins, pela atenção e apoio dado no desenvolvimento deste trabalho e em outros momentos do curso.

À minha namorada Andréa, pela compreensão e incentivo.

Enfim, a todos que de alguma forma contribuíram para que eu chegasse até aqui. Muito obrigado!

Dedicação é a capacidade de se entregar à realização de um objetivo. A realização de um sonho depende da dedicação. Há muita gente que espera que o sonho se realize por mágica. Mas toda mágica é ilusão. E ilusão não tira ninguém do lugar onde está. Ilusão é combustível de perdedores.

Roberto Shinyashiki

RESUMO

Este trabalho apresenta uma ferramenta de suporte a programação orientada a aspectos em Delphi. É composto por uma linguagem de programação orientada a aspectos e um *weaver*. A linguagem de aspectos é especificada através de definições regulares e a notação *Backus-Naur Form* (BNF). A ferramenta é especificada utilizando a *Unified Modeling Language* (UML) como linguagem de modelagem. A ferramenta recebe como entrada um projeto Delphi, um ou mais programas de aspectos e gera um projeto Delphi mesclando as funcionalidades dos programas de entrada.

Palavras-chave: Programação orientada a aspectos. Ferramenta CASE. Linguagem de programação. Compiladores. Gerador de código.

ABSTRACT

This work presents a support tool for aspect-oriented programming in Delphi. It is composed by an aspect-oriented programming language and a weaver. The language of aspects is specified through regular definitions and the Backus-Naur Form (BNF) notation. The tool is specified using Unified Modeling Language (UML). The tool receives as input a Delphi project, one or more programs of aspects and generates a Delphi project joining the functionalities of the input programs.

Key-words: Aspect-Oriented programming. CASE Tool. Programming language. Compilers. Code generator.

LISTA DE ILUSTRAÇÕES

LISTA DE SIGLAS

POO – Programação Orientada a Objetos

POA – Programação Orientada a Aspectos

AOP – *Aspect-Oriented Programming*

DSOO – Desenvolvimento de Software Orientado a Objetos

DSOA – Desenvolvimento de Software Orientado a Aspectos

BNF – *Backus-Naur Form*

UML – *Unified Modeling Language*

IDE – *Integrated Development Environment*

SQL – *Structured Query Language*

SUMÁRIO

1 INTRODUÇÃO

Ao longo dos últimos anos, a Engenharia de Software tem disponibilizado para os desenvolvedores vários métodos, técnicas e ferramentas, para auxiliá-los a produzir com maior qualidade. Entre as características perseguidas, a busca pela reusabilidade e manutenibilidade são as mais importantes para a produtividade (RESENDE; SILVA, 2005, p. 10). Com o surgimento da Programação Orientada a Objetos (POO), houve uma mudança radical na maneira de se desenvolver software. A POO tem contribuído de forma positiva na engenharia de software, pois o modelo de objetos fornece um melhor entendimento do problema real (KICZALES et al., 1997, p. 1). Com o crescimento dos sistemas e o aumento de sua complexidade, têm-se observado alguns problemas como por exemplo, entrelaçamento e espalhamento de código, que as técnicas de POO e programação estruturada são incapazes de resolver (KICZALES et al., 1997, p. 1). Assim nasceu a Programação Orientada a Aspectos (POA), ou *Aspect-Oriented Programming* (AOP), com o intuito de apresentar técnicas capazes de cobrir essas falhas.

Para Resende e Silva (2005, p. 12), “a POA tem por objetivo separar os níveis de preocupação durante o desenvolvimento de software. [...] A proposta é poder desenvolver as partes do sistema sem se preocupar com as demais partes.” Esta técnica visa também a eliminação da redundância de código fonte no sistema (RESENDE; SILVA, 2005, p. 12). Segundo Nelson (2005, p. 18), a tecnologia de orientação a aspectos não substitui a orientação a objetos, mas complementa esta outra tecnologia. Tanto é que algumas partes de um programa orientado a aspectos são implementadas baseadas no modelo orientado a objetos. A POA oferece uma estrutura que encapsula implementações de uma responsabilidade que ficariam espalhadas no modelo orientado a objetos. Esta unidade é chamada de aspecto.

Para uma implementação básica de POA, é necessário: uma linguagem para

programação dos componentes, como por exemplo o Java; uma linguagem para programação dos aspectos, como o AspectJ; um *weaver;* um programa de componentes e um programa de aspectos (STEINMACHER, 2001, p. 3). O *weaver* (ou montador) é o combinador de aspectos. Sua função é combinar os programas escritos em linguagem de componentes, no caso, as regras de negócio implementadas com orientação a objeto, com os escritos em linguagem de aspecto. O resultado será um programa mesclando as funcionalidades implementadas nos programas de componentes e aspecto (GROTT, 2005, p. 27).

Neste trabalho é desenvolvido um protótipo de um *weaver* para geração de código na linguagem Delphi com suporte a programação orientada a aspectos. Por se tratar de uma ferramenta de engenharia de software, ela é voltada para o desenvolvedor. A ferramenta recebe como entrada um arquivo de projeto do Delphi (arquivo .dpr), e um ou mais programas na linguagem de aspecto AOPDelphi, que foi criada neste trabalho. Como saída, a ferramenta gera programas fontes na linguagem Delphi, mesclando as funcionalidades dos programas fornecidos como entrada. Este *weaver* foi desenvolvido baseado no AspectJ[1].

1.1 OBJETIVOS DO TRABALHO

O objetivo deste trabalho é desenvolver um software *weaver* (montador) para gerar código na linguagem Delphi com suporte à programação orientada a aspectos.

Os objetivos específicos do trabalho são:

a) criar uma linguagem para programação dos aspectos;

b) permitir que o *weaver* receba como entrada um arquivo de projeto do Delphi (arquivo .dpr) e um ou mais programas fontes na linguagem criada para programação dos aspectos;

[1] AspectJ é uma ferramenta de apoio a programação orientada a aspectos na linguagem Java que possui o seu *weaver* e sua linguagem de aspectos.

c) realizar análises léxica e sintática dos programas de aspectos;

d) disponibilizar um ambiente para codificação dos programas de aspecto.

1.2 ESTRUTURA DO TRABALHO

O conteúdo do trabalho está organizado em cinco capítulos. No capítulo seguinte é feita uma apresentação da tecnologia da programação orientada a aspectos, com seus conceitos básicos, vantagens, elementos e construções que formam um programa orientado a aspectos. No final do capítulo são apresentados alguns trabalhos correlatos. No capítulo três é feito um embasamento teórico sobre compiladores e construção de linguagens de programação. O capítulo quatro trata da especificação e implementação da ferramenta desenvolvida. Para finalizar, no capítulo cinco são apresentadas as conclusões alcançadas no desenvolvimento do trabalho, além de sugestões para extensões do mesmo.

2 PROGRAMAÇÃO ORIENTADA A ASPECTOS

Neste capítulo é apresentada a tecnologia de programação orientada a aspectos. Primeiramente é visto o princípio da separação de interesses, que é a teoria que norteia tal tecnologia. Em seguida são apresentados os conceitos básicos, algumas vantagens e os elementos que compõem a POA. São apresentadas também algumas construções que formam uma linguagem de POA.

2.1 SEPARAÇÃO DE INTERESSES

Todo sistema de software lida com diferentes interesses, sejam eles de natureza funcional ou não funcional (NELSON, 2005, p. 11). Separação de interesses, ou *Separation of Concerns* é a teoria estabelecida pela engenharia de software que prega que as preocupações envolvidas no desenvolvimento de um software devem ser focadas e trabalhadas separadamente, concentrando em uma de cada vez, com o intuito de obter o domínio da complexidade do desenvolvimento do sistema como um todo. Um interesse (*concern)* é uma fração do problema que se deseja tratar como um componente à parte, podendo ser um requisito, uma funcionalidade ou uma propriedade do sistema (KULESZA; SANT'ANNA; LUCENA, 2005, p. 266). Dijkstra (1976 apud KULESZA; SANT'ANNA; LUCENA, 2005, p.266), primeira pessoa a utilizar o termo *separation of concern*, afirma que "a principal característica do pensamento inteligente é ser capaz de estudar em profundidade um aspecto de determinado problema isoladamente, para o bem de sua própria consistência, mas sabendo o tempo todo que outros aspectos estão esperando sua vez, pois a mente humana é tão limitada que não consegue lidar com todos eles simultaneamente sem ficar confusa."

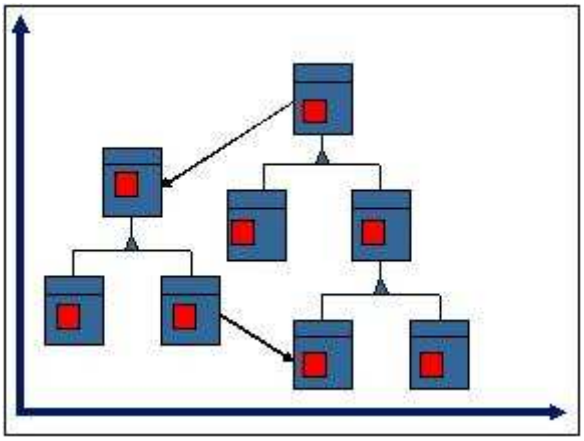
Resende e Silva (2005, p. 9) definem o princípio da separação de interesses como "a

teoria que investiga como separar as diversas preocupações pertinente a um sistema, propiciando que cada uma das preocupações seja tratada separadamente a fim de reduzir a complexidade do desenvolvimento, evolução e integração de software.” Dividindo os interesses e responsabilidade do sistema em partes, o desenvolvedor estará concentrado em resolver um problema menor. Dessa forma, a probabilidade da ocorrência de erros em cada módulo é reduzida e conseqüentemente, o mesmo acontecerá com o sistema como um todo (RESENDE; SILVA, 2005, p. 9).

2.2 CONCEITOS BÁSICOS

Os desenvolvedores de software utilizam diferentes abstrações provindas de linguagens, métodos e ferramentas para modularizar os interesses de um software. No Desenvolvimento de Software Orientado a Objetos (DSOO), essas abstrações são as classes, objetos, métodos e atributos (KULESZA; SANT’ANNA; LUCENA, 2005, p. 266). A orientação a objetos melhorou as possibilidades de separação de interesses, porém, nos casos de sistemas mais complexos, onde há interesses que atravessam os componentes responsáveis pela modularização de outros interesses, suas abstrações não são eficazes (KULESZA; SANT’ANNA; LUCENA, 2005, p. 266). Esses interesses que atravessam as funcionalidades do sistema são chamados de interesses transversais (*crosscutting concerns*), ou interesses ortogonais, pois sua implementação atravessa a estrutura de outros módulos do sistema, dificultando sua modularização (RESENDE; SILVA, 2005, p. 8). Dessa forma, aparecem os problemas de entrelaçamento (*code tangling*) e espalhamento de código (*spreading code*), reduzindo a reusabilidade e aumentando a dificuldade de evolução do software. Tratamento de exceções, distribuição e persistência são exemplos de interesses transversais (ELRAD apud KULESZA; SANT’ANNA; LUCENA, 2005, p. 266). Na figura 1 supõe-se um modelo de

classes onde há a implementação de um interesse transversal, como um controle de *log* por exemplo. Os pontos vermelhos em cada classe se referem à implementação desse interesse. Observa-se que eles estão espalhados, poluindo as classes com implementações que não são comuns do negócio.



Fonte: adaptado de Barros (2004, p. 9).

Figura 1 – Exemplo de interesse transversal em um modelo de classes

A POA surgiu através de pesquisas realizadas em laboratórios de empresas e universidades, com o objetivo de solucionar essas falhas da POO. Essas limitações contribuem para que o desenvolvimento de software fique mais complexo e impedem que os processos e produtos sejam mais gerenciáveis e flexíveis (RESENDE; SILVA, 2005, p. 1). Kiczales et al. (1997, p.1) propuseram a POA como uma tecnologia de programação para melhorar a separação de interesses no desenvolvimento de software e contribuir com o aumento da reusabilidade e manutenibilidade.

Para Nelson (2005, p. 13), o objetivo do desenvolvimento orientado a aspecto é encapsular interesses transversais em módulos fisicamente separados do restante do código. “Pensando em termos abstratos, a orientação a aspectos introduz uma terceira dimensão de decomposição. Além de decompor o sistema em objetos (dados) e métodos (funções), decompõe-se cada objeto e função de acordo com o interesse sendo servido e agrupa-se cada interesse em um módulo distinto” (NELSON, 2005, p. 13).

Filman (2005 apud KULESZA; SANT'ANNA; LUCENA, 2005, p. 267) afirma que o Desenvolvimento de Software orientado a Aspecto (DSOA) vem se consolidando como um novo paradigma de desenvolvimento e seu objetivo é disponibilizar métodos, técnicas e ferramentas para apoiar a implementação, projeto e especificação de requisitos para endereçar os interesses transversais. Este paradigma envolve duas etapas de trabalho: a primeira é a decomposição do sistema em partes não espalhadas e não entrelaçadas. A outra consiste em juntar essas partes novamente de tal forma que se obtenha o sistema desejado. Na POA não há chamada explícita de métodos entre as partes. Ao invés disso, é especificada em uma unidade separada, como uma parte deve reagir a eventos que ocorrem em outra parte. Isso reduz o acoplamento entre as partes, pois elas não se acessam diretamente (NELSON, 2005, p. 17).

Resende e Silva (2005, p. 9) enfatizam que "o objetivo final da POA é reduzir a complexidade do desenvolvimento de software, permitindo que as diversas preocupações possam ser componentizadas, independentemente de sua natureza, funcional ou não funcional."

2.2.1 Vantagens da POA

Nelson (2005, p. 17) lista algumas vantagens da POA:

a) menos responsabilidades em cada parte: como os interesses transversais são separados em um módulo específico, as classes de negócio não ficam "poluídas" com código que lida com interesses periféricos;
b) melhor modularização: o nível de acoplamento é reduzido pelo fato dos módulos não se chamarem diretamente;
c) evolução facilitada: novos aspectos podem ser acrescentados ao software sem a necessidade de alterar o código existente;

d) maior possibilidade de reutilização: como o código não estará entrelaçado, a possibilidade dele ser reutilizado em outros sistemas é maior;

e) implementação de requisitos não funcionais: para Resende e Silva (2005, p. 13) “a POA tem facilitado principalmente a implementação de requisitos não funcionais que demandam soluções que não podem ser contidas em apenas um módulo, mas se espalham ao longo de todo o sistema.”

2.3 ELEMENTOS QUE COMPÕEM A POA

Uma aplicação que utiliza POA é composta pelos seguintes componentes:

a) linguagem de componentes: segundo Irwin (1997 apud GROTT, 2005, p. 26), “a linguagem de componentes 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”. Pode-se citar Java, C, Delphi como exemplos de linguagem de componentes;

b) linguagem de aspecto: a linguagem de aspecto deve permitir que as propriedades desejadas sejam implementadas de forma clara e concisa e fornecer ao programador construções necessárias para criar estruturas que realizam o comportamento dos aspectos, definindo o que e quando serão executadas (IRWIN, 1997 apud GROTT, 2005, p. 26);

c) programas de componentes: é o arquivo fonte escrito em uma determinada linguagem de programação, como por exemplo, Delphi, C, Java, onde o desenvolvedor implementa as regras de negócio do sistema utilizando técnicas de orientação a objeto (GROTT, 2005, p. 27);

d) um ou mais programas de aspecto: programas que serão codificados utilizando a

linguagem de aspecto (KICZALES et al., 1997, p. 24);

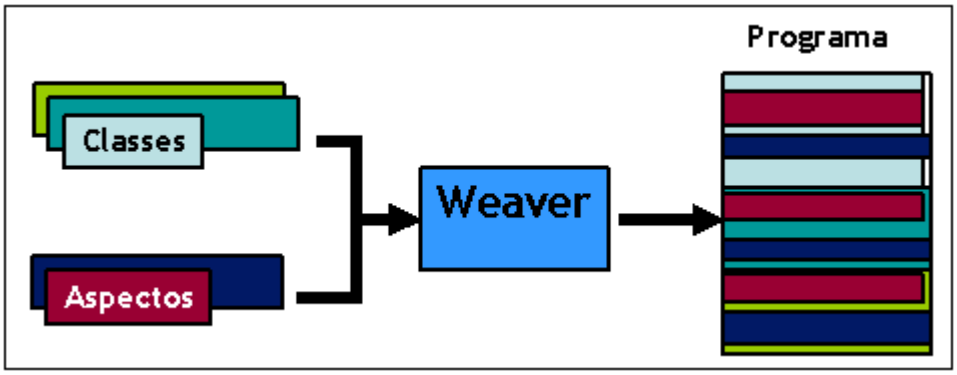
e) *weaver*: é o termo utilizado para referenciar aos compiladores de linguagens orientadas a aspectos" (RESENDE; SILVA, 2005, p. 27).

2.3.1 *Weaver*

Também conhecido como combinador de aspectos, ou montador, a função deste componente é combinar os programas escritos em linguagem de componentes, no caso, as regras de negócio implementadas com orientação a objeto, com os escritos em linguagem de aspectos, gerando um programa que mescle as funcionalidades definidas nesses programas de entrada. Este processo é chamado de *weaving* (GROTT, 2005, p. 27). Traduzindo para o português, *weaver* quer dizer tecelão. Para Nelson (2005, p.17), a ferramenta é chamada dessa forma porque "ela 'tece' os vários fragmentos de programas em um programa único."

O *weaver* tem a capacidade de interceptar as chamadas de métodos em tempo de compilação ou execução, conforme seu propósito, podendo desviar o fluxo de execução e até mesmo abortar a execução do método (SILVEIRA et al., 2004, p. 6).

Na Figura 2 é apresentado o modelo de funcionamento de um *weaver*.



Fonte: Barros (2004, p. 13).

Figura 2 - Modelo de funcionamento de um *weaver*

2.4 POA DINÂMICA E POA ESTÁTICA

A POA pode ser estática ou dinâmica. Isso quer dizer que os pontos do sistema que serão interceptados e os respectivos códigos que serão introduzidos podem ser definidos em tempo de compilação ou execução (RESENDE; SILVA, 2005, p. 27).

No caso da POA estática, a junção dos programas de aspecto e componentes é feita em tempo de compilação. Para criar ou alterar um aspecto é necessário implementar essas definições na linguagem de aspecto, para depois compilar. Uma vantagem neste processo é a segurança, pois podem ser evitados erros em tempo de execução. Como exemplo de *weaver* estático tem-se o AspectJ (STEINMACHER, 2003, p. 7).

Na POA dinâmica, a combinação dos programas de aspectos e componentes é feita em tempo de execução. Piveta (2001, p. 10) afirma que "através de uma interface reflexiva, o combinador de aspectos tem a possibilidade de adicionar, adaptar e remover aspectos em tempo de execução." Steinmacher (2003, p. 8) cita o AOP/ST[2] e o AspectS como combinadores de aspectos dinâmico.

2.5 ASPECTO

A POA oferece recursos para apoiar na separação dos interesses transversais, a fim de melhorar a modularização desses interesses. Isso é feito através de abstrações que permitem a separação e composição desses interesses para produzir o sistema desejado. Essa nova abstração é chamada de aspecto (KULESZA; SANT'ANNA; LUCENA, 2005, p. 266).

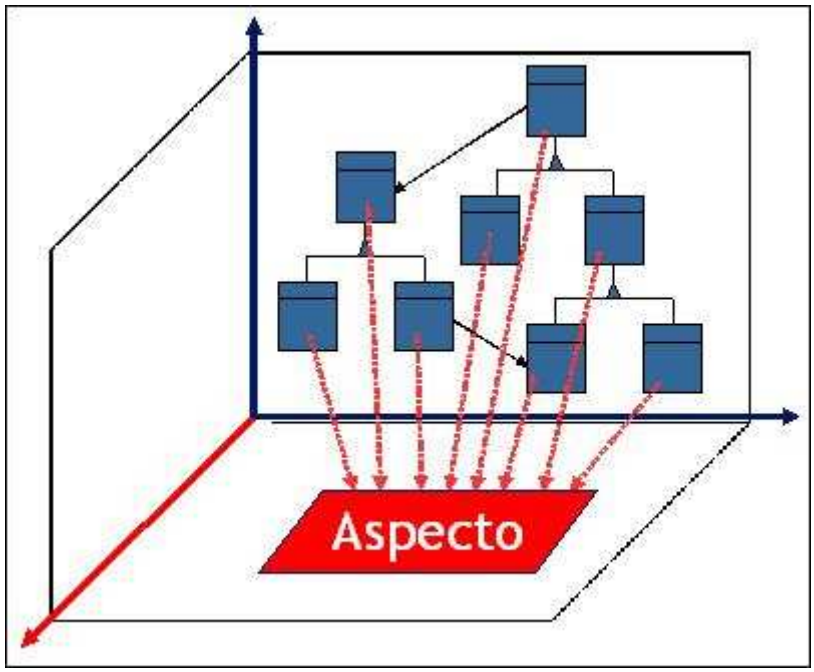
O aspecto é a principal unidade de modularização na orientação a aspectos. Ela está para a orientação a aspectos assim como a classe está para a orientação a objetos. Da mesma

[2] AOP/ST e AspectS são ferramentas de suporte a POA na linguagem SmallTalk (STEINMACHER, 2003, p.11).

forma que as classes encapsulam métodos e atributos, os aspectos encapsulam os *pointcuts* e *advices*. Dependendo da linguagem de aspecto, eles podem conter variáveis, métodos e restrições de acesso (NELSON, 2005, p. 22).

Para Resende e Silva (2005, p. 42) um aspecto deve ser visto como uma unidade que contém uma estrutura de *pointcuts*, que faz referência a determinados pontos definidos no sistema onde serão executadas instruções que não fazem parte do fluxo normal do programa.

Kulesza, Sant'anna e Lucena (2005, p. 267) definem o aspecto como "a unidade de modularidade para interesses transversais. Eles encapsulam uma funcionalidade que atravessa várias classes do sistema." Baseado no exemplo da figura 1, a figura 3 mostra como fica encapsulada a implementação de um interesse transversal utilizando técnicas de orientação a aspectos. A implementação que antes estaria espalhada, agora está encapsulada dentro de uma única unidade: o aspecto.



Fonte: adaptado de Barros (2004, p. 9).

Figura 3 – Representação de um interesse transversal utilizando orientação a aspecto

2.6 *JOINPOINT*

O *joinpoint* ou ponto de junção é qualquer ponto identificável dentro da execução de um programa, como por exemplo: chamadas de métodos e construtores, tratamentos de exceções, acesso a atributos das classes (GROTT, 2005, p. 39). São nesses pontos que o código implementando nos *advices*, definidos dentro dos aspectos, serão introduzidos.

No Quadro 1 tem-se um exemplo de um método, na linguagem Delphi, que poderá ter um *joinpoint* associado.

```
procedure TSintatico.parse(scanner : TLexico; semanticAnalyser : TSemantico);
begin
    self.scanner := scanner;
    self.semanticAnalyser := semanticAnalyser;

    stack.clear;
    stack.add(Pointer(DOLLAR));
    stack.add(Pointer(START_SYMBOL));

    if (previousToken <> nil) and (previousToken <> currentToken) then
        previousToken.destroy;
    previousToken := nil;

    if currentToken <> nil then
        currentToken.destroy;
    currentToken := scanner.nextToken;

    while not step do
        ;
end;
```

Quadro 1 – Exemplo de método que poderá estar associado a um *joinpoint.*

Resende e Silva (2005, p. 26) enfatizam que “a filosofia da POA doutrina que é possível interromper o fluxo natural de um programa em seus *joinpoints*, executar uma outra rotina e continuar a execução do programa como se nada tivesse ocorrido.”

2.7 *POINTCUT*

Os *pointcuts*, também conhecidos como conjuntos de junção, são utilizados para

especificar os pontos no programa de componentes que serão interceptados e inseridos um comportamento diferente (RESENDE; SILVA, 2005, p. 45). Ou seja, eles selecionam um conjunto de pontos de junção (*joinpoints*). Nelson (2005, p. 22) cita que existem regras que permitem flexibilidade na declaração de *pointcuts*, não sendo necessário especificar um *pointcut* para cada *joinpoint*, o que tornaria a POA praticamente sem sentido.

Resende e Silva (2005, p. 45) comparam os *pointcuts* com uma variável que armazena uma lista de assinaturas de métodos que serão interceptados e terão o fluxo normal do programa alterado. Na execução do programa, ao ser invocado um desses métodos, o aspecto entrará em ação executando as instruções definidas para o respectivo *pointcut.*

2.8 *ADVICE*

Os *advices* são construções semelhantes aos métodos na orientação a objetos e são associados aos *pointcuts*. Eles contêm o código que será inserido no ponto de junção quando o *pointcut* correspondente for atingido. Podem ser executados antes, depois ou substituir o método interceptado (KULESZA; SANT'ANNA; LUCENA, 2005, p. 268).

O *weaver* seleciona os *joinpoints* baseado nos *pointcuts* definidos no aspecto. Quando um método associado ao *joinpoint* for invocado, o aspecto entra em ação executando o código contido no *advice*, antes ou depois do método invocado, conforme definido, podendo também desviar para uma outra rotina como se nada tivesse acontecido (GROTT, 2005, p. 28).

Segundo Resende e Silva (2005, p. 86) a POA permite que nos *advices* seja capturado o objeto que está invocando o método interceptado. Dessa forma ele pode ser utilizado no escopo do *advice*, como se tivesse sido criado localmente e é uma referência do objeto que chamou o método.

2.9 TRABALHOS CORRELATOS

AspectJ (KULESZA; SANT'ANNA; LUCENA, 2005, p. 267-268) é uma extensão da linguagem Java para suporte a POA. Entre as ferramentas com suporte a POA ela é a mais conhecida e utilizada. Permite a definição de pontos de junção, que são os locais onde serão executados os códigos de aspecto, como invocação de métodos e tratamentos de exceções. O aspecto encapsula os *pointcuts*, que é um conjunto de pontos de junção e os *advices*, que são o código de aspecto que deverá ser executado nos pontos de junção. Em AspectJ há três tipos de *advices*: *before*, onde o código contido no *advice* é executado antes da execução das instruções do método interceptado; *after*, onde o código do *advice* é executado após o código do método interceptado; e *around*, que toma o total controle de execução do método interceptado, podendo não permitir a sua execução ou executar um outro método no lugar. O *weaver* recebe como entrada programas de componentes e programas de aspectos e gera arquivos Java que podem ser compilados normalmente.

No Quadro 2 é mostrado um exemplo de um aspecto definido na linguagem AspectJ.

```
public aspect MeuAspecto {
    // declaração de um pointcut
    pointcut PointCut_Um(): call (int Contas.subtrair(int, int));

    // declaração de um advice para o PointCut_Um()
    before(): PointCut_Um(){
        System.out.println("Capturado:" + thisJoinPoint);
    }
}
```

Quadro 2 – Exemplo de aspecto implementado em AspectJ

Na Quadro 2 é implementado um aspecto com o nome de "MeuAspecto" e nele é definido um *pointcut* (PointCut_Um) que seleciona como *joinpoints* as chamadas do método subtrair da classe Contas, que passam dois argumentos do tipo int. O aspecto possui um *advice* do tipo *before* associado a este *pointcut.*

AspectC (STEINMACHER, 2002, p. 15) é uma extensão da linguagem C com suporte

a POA. Os *advices*, que são os códigos de aspecto, são introduzidos nos limites das chamadas de funções do programa de componentes utilizado como entrada, podendo ser executados antes, durante ou depois dessas funções. Assim como a linguagem C, o *weaver* do AspectC é estático. Steinmacher (2002, p. 15) afirma que, “o AspectC é ideal para o desenvolvimento de sistemas onde a eficiência é primordial, como na implementação de sistemas operacionais.”

Grott (2005) faz um apanhado geral da tecnologia de orientação a aspectos e apresenta detalhes teóricos da composição de um aspecto, as tecnologias que dão suporte ao desenvolvimento de software e a inserção dentro da engenharia de software. É visto com mais detalhes a linguagem de aspecto AspectJ. É elaborado um estudo de caso consistindo de modelo de troca de informações entre sistemas heterogêneos de envio e recebimento de notificação de compra de mercadorias.

Piveta (2001) propõe um modelo de suporte a programação orientada a aspectos visando definir abstrações e mecanismos de composição que ofereçam suporte de maneira genérica. A implementação do modelo proposto através da abordagem de interceptação de mensagens foi batizada de Aurélia, e provê orientação a aspectos para linguagem Object Pascal. A abordagem foi implementada através do uso do protocolo de metaobjetos OPMOP (LEITE, 2001 apud PIVETA, 2001), desenvolvido na Universidade Federal de Santa Catarina, que disponibilizam componentes que promovem facilidades na confecção de aplicativos com características reflexivas.

3 COMPILADORES

Um compilador é um programa que recebe como entrada um programa fonte de uma linguagem de programação e gera um programa equivalente em uma outra linguagem. Este programa gerado pode ser uma outra linguagem de programação ou uma linguagem de máquina específica (AHO; SETHI; ULLMAN, 1995, p. 1).

Vargas (2005, p. 22) afirma que a compilação se divide em dois processos, denominados análise e síntese. No processo de análise, o código fonte é dividido, sendo verificadas as propriedades de suas partes. Este processo é realizado nas fases de análises léxica, sintática e semântica do compilador. O processo de síntese é responsável por gerar o programa destino. As fases de geração de código intermediário, otimização de código e geração de código objeto fazem parte deste processo.

3.1 ANÁLISE LÉXICA

Price e Toscani (2001, p. 7) afirmam que o principal objetivo da fase léxica é identificar seqüências de caracteres que constituem unidades léxicas, também chamadas de *tokens*. O analisador léxico ler cada caractere do programa fonte verificando se os mesmos fazem parte do alfabeto da linguagem, a fim de identificar os *tokens*. Os comentários e os brancos desnecessários são ignorados pelo analisador. *Tokens* são as unidades básicas de um programa fonte, como identificadores, constantes numéricas, palavras reservadas, operadores, entre outros (PRICE; TOSCANI, 2001, p. 22).

Vargas (2005, p. 22) ratifica que se durante a análise léxica for identificado algum *token* que não atenda a especificação da linguagem, deve ser informado a ocorrência de um erro léxico no programa fonte.

3.2 ANÁLISE SINTÁTICA

A função da análise sintática é fazer a verificação da estrutura gramatical do programa, se ela contempla as regras gramaticais da linguagem (PRICE; TOSCANI, 2001, p. 9). Um dos pontos mais importantes durante a fase de análise sintática é a detecção de erros, pois boa parte dos erros encontrados durante o processo de análise são detectados nesta fase. Para Vargas (2005, p. 24) “um erro sintático ocorre quando a seqüência de *tokens* não está na ordem definida na gramática da linguagem.”

3.3 ANÁLISE SEMÂNTICA

“A fase de análise semântica verifica os erros semânticos no programa fonte e captura as informações de tipo para a fase subseqüente de geração de código” (AHO; SETHI; ULLMAN, 1995, p. 4). Para Price e Toscani (2001, p. 9), a função do analisador semântico é verificar se o significado nas estruturas sintáticas do programa, como comandos, identificadores e expressões, são compatíveis, como por exemplo, a atribuição de um valor decimal para uma variável definida como inteiro.

3.4 LINGUAGENS DE PROGRAMAÇÃO

Marshall (1986, p. 14) afirma que, assim como uma linguagem natural, a linguagem de programação tem a finalidade de propiciar a comunicação. Ela é o meio de comunicação entre o desenvolvedor que estará solucionando um determinado problema e o computador que estará o auxiliando (PRICE; TOSCANI, 2001, p. 1). As linguagens de programação são projetadas com regras gramaticais comparativamente simples, de tal forma que sejam evitados

os problemas com ambigüidade (MARSHALL, 1986, p. 16).

Price e Toscani (2001, p. 1) classificam como linguagem de alto nível aquelas que cujas construções estão mais próximas da linguagem natural ou do domínio da aplicação. Elas são as mais utilizadas hoje em dia.

3.5 BNF

A definição formal da sintaxe de uma linguagem de programação é chamada de gramática, que consiste em um conjunto de regras que especificam a seqüência de caracteres quem formam programas válidos para a linguagem. Uma notação bastante utilizada para descrever gramáticas de linguagem de programação é a *Backus-Naur Form* (BNF) (VAREJÃO, 2004, p. 13).

Na década de 50, através de pesquisas distintas e não relacionadas, John Backus e Noam Chomsky inventaram a mesma notação para definir sintaxe de linguagens de programação. Na década de 60 essa notação foi ligeiramente modificada por Peter Naur para especificação da linguagem ALGOL 60. Essa nova versão da notação foi batizada de BNF, também conhecida como Forma de Backus-Naur (SEBESTA, 2000, p. 115).

A BNF utiliza abstrações para definir as estruturas sintáticas. As definições são chamadas de regras ou produções e são compostas pelos símbolos não terminais e símbolos terminais. Os símbolos não terminais são representados por um nome entre os sinais de “<” e “>”. Os símbolos terminais são representados pelos *tokens* (SEBESTA, 2000, p. 116).

Na Quadro 3 tem-se um exemplo com produções BNF para especificar uma linguagem de programação.

```
<programa>   ::=  begin <lista_stmt> end

<lista_stmt> ::=  <stmt>
                | <stmt> ; <lista_stmt>

<stmt>       ::=  <var> := <expressao>

<var>        ::=  A | B | C

<expressao>  ::=  <var> + <var>
                | <var> - <var>
                | <var>
```

Fonte: Sebesta (2000, p. 117).
Quadro 3 – Exemplo de uma BNF de linguagem de programação

Como pode ser visto no Quadro 3, ao especificar uma produção, sua identificação fica no lado esquerdo. Em seguida aparece o símbolo “::=” que quer dizer, “é definida por”, e no lado direito aparece a definição da produção, formada por símbolos terminais e não terminais. O símbolo “|” que aparece na definição de algumas regras significa “ou”.

3.6 EXPRESSÕES REGULARES

Expressão regular “é uma composição de símbolos, caracteres com funções especiais, que, agrupados entre si e com caracteres literais, formam uma seqüência, uma expressão” (JARGAS, 2001). Essa expressão é interpretada como uma regra, e considerando uma seqüência de caracteres como entrada de dados, a expressão obterá sucesso se parte ou total da entrada de dados contemple as regras da expressão (JARGAS, 2001).

Menezes (1998, p. 50) define expressão regular como “um formalismo denotacional, também considerado gerador, pois pode-se inferir como construir as palavras de uma linguagem. É definida a partir de conjuntos básicos e operações de concatenação e união. São consideradas adequadas para comunicação homem x homem e principalmente homem x máquina.”

Os caracteres com funções especiais são chamados de metacaracteres. Cada símbolo tem sua função específica, que pode mudar dependendo do contexto no qual está inserido, e

pode-se agregá-los uns com os outros, combinando suas funções e fazendo construções mais complexas (JARGAS, 2001). No Quadro 4 são apresentados alguns metacaracteres com sua respectiva função.

METACARACTER	FUNÇÃO
.	Reconhece um caracter qualquer
*	Concatenação sucessiva zero ou mais vezes
+	Concatenação sucessiva uma ou mais vezes
?	Opcional
[]	Delimita uma lista de caracteres permitidos
()	Determina um grupo de caracteres

Fonte: Jargas (2001).

Quadro 4 – Exemplo de metacaracteres

4 AOPDELPHI

Neste capítulo é feita a apresentação da ferramenta desenvolvida, batizada de AOPDelphi. Inicialmente são apresentados os requisitos principais e em seguida a sua especificação e a operacionalidade da implementação.

4.1 REQUISITOS PRINCIPAIS

O AOPDelphi é uma ferramenta de engenharia de software voltada para o desenvolvedor. Deve prover suporte a programação orientada a aspectos para a linguagem Delphi. Os programas de aspectos devem ser escritos em uma linguagem própria.

Deve possuir um *weaver* para fazer a junção dos programas de componentes e de aspectos. O *weaver* deve receber como entrada um ou mais programas de aspectos e um arquivo de projeto do Delphi (arquivo .dpr). O arquivo .dpr deve ser de um projeto Delphi que não possua erros de compilação e seus programas devem estar implementados utilizando orientação a objetos. A ferramenta deve realizar análises léxica e sintática dos programas de aspectos fornecidos como entrada. Como saída, a ferramenta deve gerar programas na linguagem Delphi que mesclem as funcionalidades dos programas de componente e de aspectos fornecidos na entrada.

A ferramenta deve possuir um ambiente para implementação dos programas de aspectos.

4.2 ESPECIFICAÇÃO

Nos tópicos seguintes é apresentada a especificação da ferramenta. Primeiramente é

exibida a especificação da linguagem criada para implementação dos aspectos. Em seguida é apresentada a especificação do *weaver* através dos diagramas de casos de uso, atividades e de classes.

4.2.1 Especificação da linguagem de aspecto

A linguagem de aspecto criada é uma linguagem não *case sensitive*. Apesar de ter sido inspirada na linguagem de aspectos do AspectJ, sua sintaxe e seus comandos são semelhantes aos da linguagem Delphi, pois ela é uma extensão dessa última linguagem. Para especificação da linguagem foi utilizada a ferramenta GALS (GESSER, 2003). As definições regulares de auxílio para definição de *tokens*, os *tokens* e a gramática da linguagem foram definidas nesta ferramenta.

Na definição da gramática foi utilizada a notação BNF. O quadro 5 apresenta a gramática da linguagem AOPDelphi e algumas definições regulares por ela utilizada.

Definições regulares utilizadas na BNF

```
IDENTIFICADOR : (<letra> | "_") (<letra> | <digito> | "_")*
CURINGA_IDENTIFICADOR: ("*" (<letra> | <digito> | "_")+)
                          | ((<letra> |"_") (<letra> | <digito> | "_")* "*"
                            (<letra> | <digito> | "_")* (<letra> |"_" | <digito>))
                          | ((<letra> |"_") (<letra> | <digito> | "_")* "*")
CODIGO : .
```

BNF

```
<AOPDelphi> ::= <define_aspecto> <declara_variaveis>
                <implementation>
                End.
<define_aspecto> ::= IDENTIFICADOR = Aspect
<letra> ::= [a-zA-Z]
<digito> ::= [0-9]
<declara_variaveis> ::= Var <lista_variaveis>
                        | ε
<lista_variaveis> ::= <lista_identificador> : <tipos_primitivos>;
                      <lista_variaveis_linha>
<tipos_primitivos>::= string
                     | integer
                     | real
                     | double
                     | extended
                     | currency
                     | boolean
                     | variant
                     | word
                     | byte
                     | char
<lista_identificador> ::= IDENTIFICADOR <lista_identificador_linha>
<lista_identificador_linha> ::= , IDENTIFICADOR <lista_identificador_linha>
                                | ε
<lista_variaveis_linha> ::= <lista_variaveis>  | ε
<implementation> ::= Implementation <declara_pointcuts> <declara_advices>
                   | ε
<declara_pointcuts>::= IDENTIFICADOR : Pointcut = ( <joinpoint> ) ;
                       <lista_pointcut>
<joinpoint>::= <excecao> | <metodo>
<excecao>::= Exception <indentificador_excecao>
<metodo>::= <tipo_metodo> <id_classe_metodo> . <id_classe_metodo>
            ( <parametros> )
<tipo_metodo>::= Procedure | Function | Constructor
                 | Destructor | <curinga>
<id_classe_metodo>::= IDENTIFICADOR
                     | <curinga>
                     | CURINGA_IDENTIFICADOR
<parametros>::= <tipos_primitivos> <parametros_linha>
               | <curinga>
<parametros_linha>::= , <tipos_primitivos> <parametros_linha>
                      | ε
<lista_pointcut>::= <declara_pointcuts>
                    | ε
<declara_advices>::= Advice IDENTIFICADOR : <tipo_advice> ;
                     <codigo_advice> ;
                     <lista_advice>
<tipo_advice>::= Before
                 | After
<lista_advice>::= <declara_advices>
                  | ε
<indentificador_excecao>::= IDENTIFICADOR | <curinga>
<curinga>  ::= *
<codigo_advice> ::= Begin (<codigo>)* EndAdvice
<codigo> ::= CODIGO
```

Quadro 5 - BNF da linguagem AOPDelphi

4.2.2 Especificação do protótipo

A ferramenta foi especificada utilizando a *Unified Modeling Language* (UML) como linguagem de modelagem. A seguir são apresentados os diagramas de casos de uso, atividades e de classes, que foram implementados na ferramenta Enterprise Architect.

4.2.2.1 Diagrama de casos de uso

A ferramenta possui três casos de uso associados a um único ator: o desenvolvedor. Os casos de uso são: Implementar programas de componentes; Implementar programas de aspectos; e Compilar projeto.

O caso de uso Implementar programas de componentes consiste no desenvolvimento de um projeto escrito na linguagem Delphi, com técnicas de orientação a objeto. Esse projeto será fornecido como entrada no *weaver* e seus programas serão afetados pelos programas de aspectos.

O caso de uso Implementar programas de aspectos é baseado na implementação dos programas de aspectos na linguagem criada para esta finalidade. Para isso, a ferramenta conta com um ambiente para programação dos aspectos, com comandos básicos de manipulação de arquivos, como abrir, salvar ou criar um arquivo novo. O comando compilar realiza análises léxica e sintática do programa de aspectos, indicando a ocorrência de erros com sua respectiva posição no programa fonte.

O caso de uso Compilar projeto é a principal funcionalidade da ferramenta. O desenvolvedor especifica os parâmetros de um projeto, ou seja, seleciona um projeto Delphi (arquivo .dpr) que terá seus programas afetados pelos aspectos e indica esses programas de aspectos, que irão interagir com o projeto Delphi. Quando o desenvolvedor executa o

comando *weaving* da ferramenta, que é o processo de compilação de um projeto orientado a aspectos, o *weaver* inicialmente compila o projeto Delphi para certificar de que este não possua erros de compilação. Em seguida são compilados os programas de aspectos. Não havendo erros de compilação nos programas de componentes e de aspectos, o *weaver* varre os aspectos do projeto e baseado em seus *pointcuts*, localiza os *joinpoints* correspondentes nos programas de componentes, e faz a mesclagem necessária do código *advice* no método interceptado. Para finalizar o processo, o projeto Delphi alterado é compilado para certificar de que não houve erro durante a junção dos fontes. Antes de iniciar as alterações, os fontes do projeto Delphi são duplicados e todas as modificações são feitas no projeto cópia, a fim de preservar íntegros os fontes originais, para que o desenvolvedor possa mantê-los normalmente.

Na figura 4 é apresentado o diagrama de casos de uso da ferramenta e nas figuras 5, 6 e 7 são apresentados com detalhes os casos de uso implementar programas de componentes, implementar programas de aspectos e compilar projeto, respectivamente.

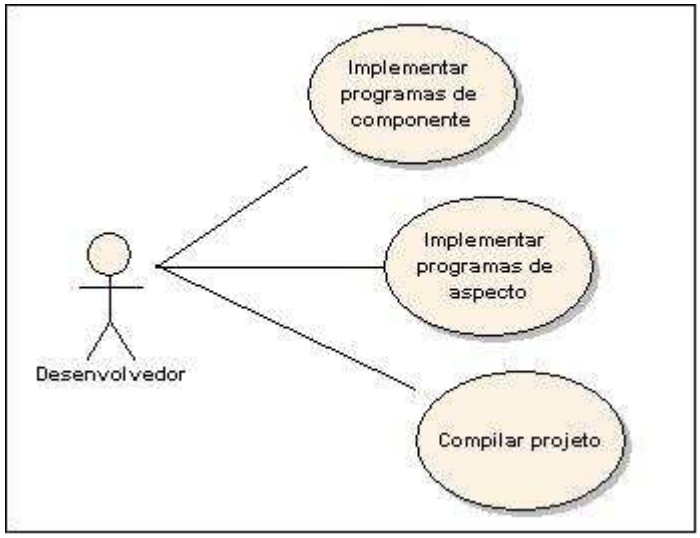


Figura 4 – Diagrama de casos de uso da ferramenta AOPDelphi

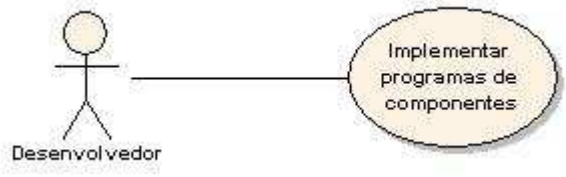
	
Descrição	O desenvolvedor deve implementar os programas de componente escritos na linguagem Delphi, com técnicas de orientação a objeto. Este programa será fornecido como entrada no *weaver*.
Ator	Desenvolvedor
Requisitos Atendidos	- O *weaver* deve receber um arquivo de projeto do Delphi como programa de componente.
Pós-condições	- Programa de componente implementado poderá ser fornecido como entrada na ferramenta.

Figura 5 – Caso de uso Implementar programas de componentes

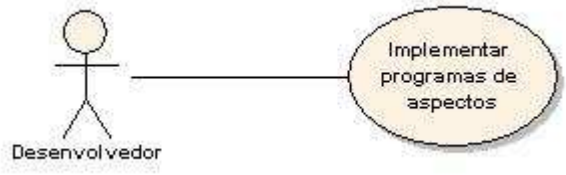
	
Descrição	O desenvolvedor deve implementar um ou mais programas de aspectos escritos na linguagem própria da ferramenta.
Ator	Desenvolvedor
Requisitos Atendidos	- Os programas de aspectos devem ser escritos em uma linguagem própria. - O *weaver* deve receber como entrada um ou mais programas de aspectos. - A ferramenta deve possuir um ambiente para implementação dos programas de aspectos.
Pré-condições	- Deve existir a pasta Base no diretório corrente do aplicativo AOPDelphi, com os arquivo .xml que armazenam os projetos.
Pós-condições	- Programas de aspectos implementados poderão ser fornecidos como entrada na ferramenta.

Figura 6 – Caso de uso Implementar programas de aspectos

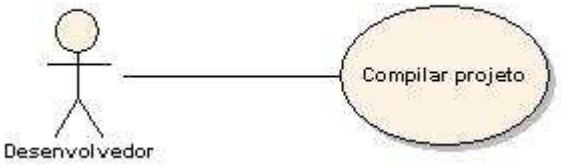
	
Descrição	O desenvolvedor deve informar ao *weaver* o projeto Delphi (arquivo .dpr) que será afetado pelos programas de aspectos. - Deve informar também um ou mais programas de aspectos escritos na linguagem da ferramenta. - O desenvolvedor informa à ferramenta as listas de *units* que não serão analisadas pelo compilador durante o processo de *weaving*. - A ferramenta realiza a análise léxica e sintática dos programas de aspectos fornecidos como entrada. - Em seguida a ferramenta realiza o *weaving* baseado no projeto Delphi e nos programas de aspectos fornecidos como entrada, mesclando suas funcionalidades. - Para finalizar, o projeto Delphi resultante do *weaving* é compilado. Se durante a compilação for encontrado algum erro, este deve ser informado se provém de código Delphi inserido dentro do aspecto (*advice*) ou se é resultado da junção de código durante o *weaving*.
Ator	Desenvolvedor
Requisitos Atendidos	- Deve prover suporte a programação orientada a aspectos para a linguagem Delphi. - A ferramenta deve realizar análises léxica e sintática dos programas de aspectos fornecidos como entrada. - A ferramenta deve gerar programas na linguagem Delphi que mesclem as funcionalidades dos programas de componente e de aspectos fornecidos na entrada.
Pré-condições	- Deve existir a pasta Base no diretório corrente do aplicativo AOPDelphi, com os arquivo .xml que armazenam os projetos. - Deve existir a pasta Saída no diretório corrente do aplicativo AOPDelphi, que é a pasta onde será gerado o projeto após o *weaving*.
Pós-condições	É gerado um projeto Delphi mesclando as funcionalidades dos programas de componentes e aspectos fornecidos na entrada.

Figura 7 – Caso de uso Compilar projeto

4.2.2.2 Diagrama de atividades

No diagrama de atividades estão especificadas as funcionalidades da ferramenta de forma geral. Ele está dividido em duas raias, representadas pelo desenvolvedor e a ferramenta, com suas respectivas atividades. Na figura 8 é apresentado o diagrama de atividades da

ferramenta AOPDelphi.

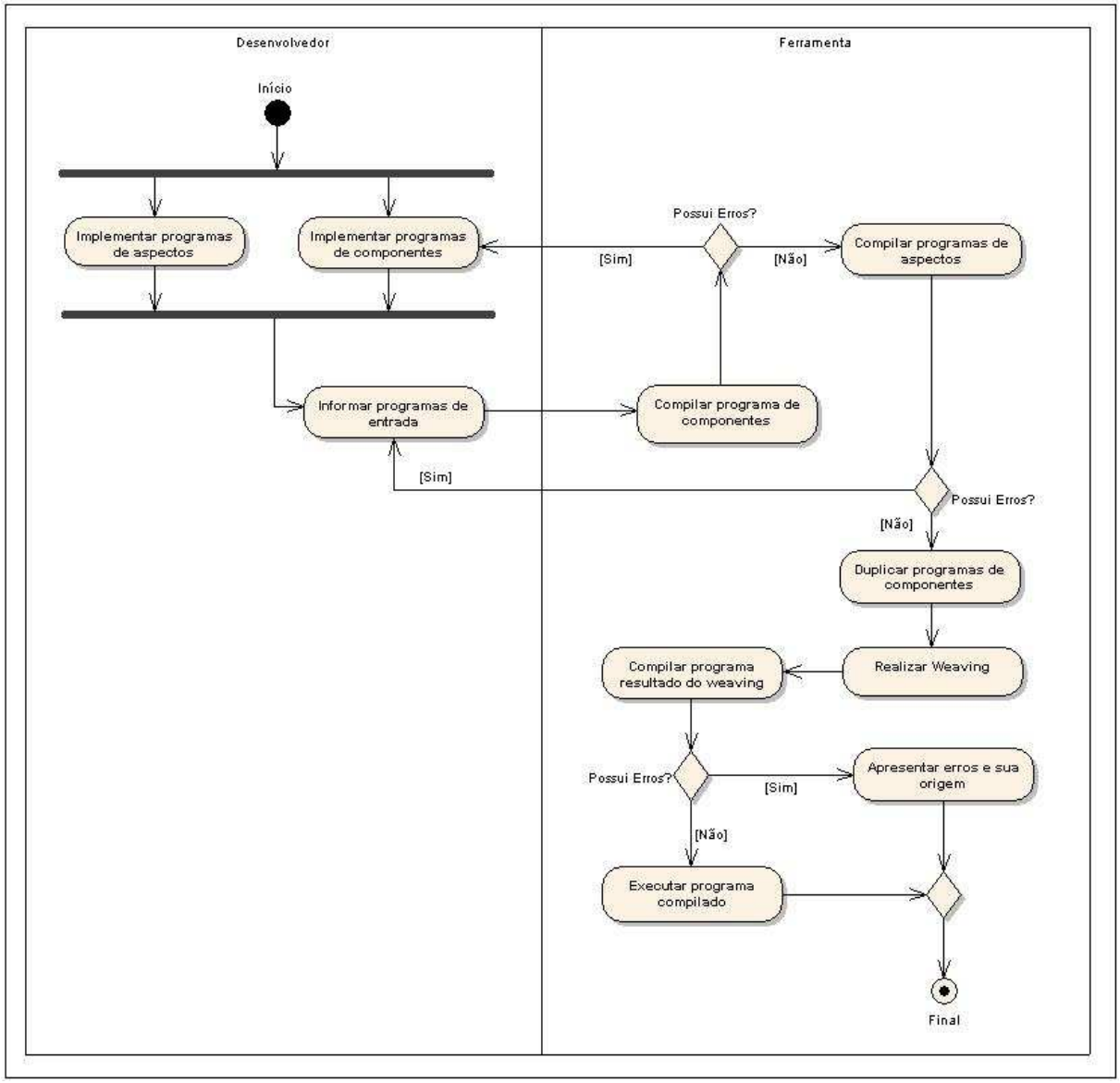


Figura 8 – Diagrama de atividades da ferramenta AOPDelphi

A descrição detalhada das atividades do diagrama é apresentada no quadro 6.

Atividade	Raia	Descrição
Implementar programas de componentes	Desenvolvedor	O desenvolvedor implementa programas de componentes escritos na linguagem Delphi, utilizando técnicas de orientação a objeto. Este programa será fornecido como entrada no *weaver*.
Implementar programas de aspectos	Desenvolvedor	O desenvolvedor implementa programa(s) de aspectos escrito(s) na linguagem específica da ferramenta, para ser fornecido como entrada no *weaver*.
Informar programas de entrada	Desenvolvedor	O desenvolvedor informa ao *weaver* um ou mais programas de aspectos na linguagem AOPDelphi (arquivo .dao) e um arquivo de projeto do Delphi (arquivo .dpr). Pode informar também listas de units que não devem ser afetadas pelos programas de aspectos.
Compilar programas de componentes	Ferramenta	Inicialmente, a ferramenta compila o programa de componentes (projeto Delphi) para certificar de que este não possui erros, pois deve ser fornecido para o *weaver* um projeto que esteja compilando.
Compilar programas de aspectos	Ferramenta	A ferramenta compila os programas de aspectos e organiza em objetos as informações necessárias para em seguida fazer o *weaving*. Para cada fonte de aspecto (arquivo .dao) é instanciado um objeto de TAspecto. O código Delphi escrito entre "Begin" e "EndAdvice" no fonte de aspecto não é analisado pelo compilador.
Duplicar programas de componentes	Ferramenta	A ferramenta faz uma cópia de todos os fontes do projeto Delphi. As alterações que serão feitas pelo *weaver* serão realizadas no programa cópia.
Realizar *weaving*	Ferramenta	Baseado nos *pointcuts* extraídos em cada aspecto, a ferramenta gera expressões regulares para varrer os fontes do projeto Delphi, localizar os *joinpoints* que devem ser interceptados e implementar a alteração escrita no respectivo *advice*.
Compilar programa resultado do *weaving*	Ferramenta	A ferramenta compila o projeto Delphi após o *weaving* para certificar de que o mesmo não possui erros.
Apresentar erros e sua origem	Ferramenta	Havendo erros de compilação no projeto Delphi após o *weaving*, a ferramenta apresenta os tais erros e sua origem. Ou seja, é informado se o erro em questão provém do código escrito no *advice* ou é em função da junção feita no *weaving*.
Executar programa compilado	Ferramenta	Não havendo erros de compilação no projeto Delphi após o *weaving*, a ferramenta executa o programa gerado (arquivo .exe).

Quadro 6 – Descrição das atividades

4.2.2.2.1 Atividade *Weaving*

Dentre as atividades apresentadas no diagrama ilustrado na figura 8, se destaca a atividade *Weaving*, principal atividade da ferramenta. Nela é representado o processo desde a seleção das *units* que serão analisadas pelo *weaver* até a junção do código contido no *advice*, quando este é inserido no método interceptado por um *pointcut*. O diagrama de atividades do processo *weaving* é representado na figura 9.

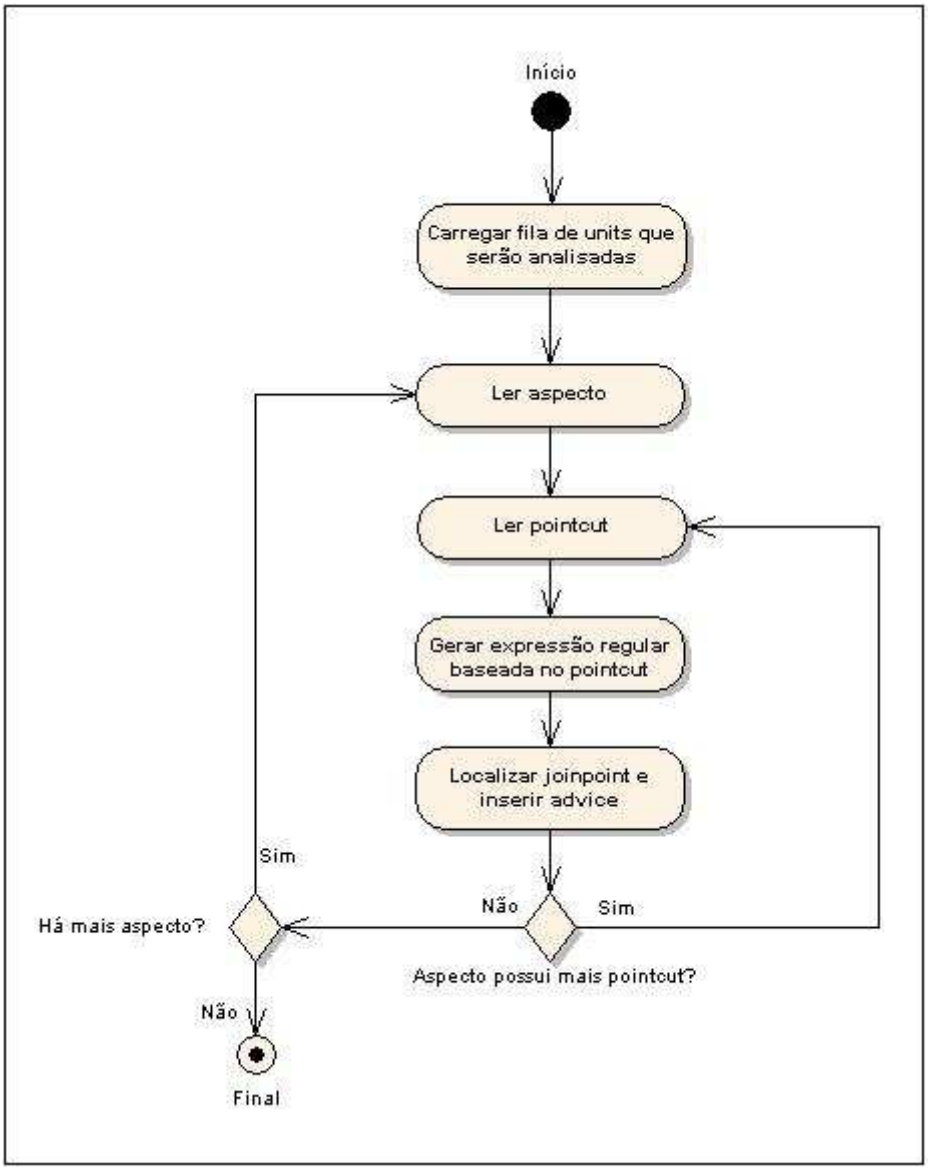


Figura 9 – Diagrama de atividades do processo de *weaving*

Neste processo, inicialmente a ferramenta extrai do arquivo de projeto Delphi, que será

afetado pelos aspectos, as *units* que estão diretamente associadas ao projeto e adiciona-as a uma fila. Essas *units* também são lidas, e delas são extraídas as *units* as quais elas se referem, e as mesmas são adicionadas à fila. Este processo se repete para todas as *units* referenciadas no projeto. Em seguida, o *weaver* varre a lista de aspectos do projeto e para cada aspecto, varre a lista de *pointcuts*, gerando expressões regulares baseadas em suas informações. Essas expressões regulares são parâmetros de busca nas *units* adicionadas à fila para serem analisadas pelo *weaver*. Ao detectar nos fontes analisados algum método que contemple a expressão regular em questão, o código contido nos *advices* associados ao *pointcut* é inserido neste método, antes ou depois do código já implementado, conforme definição do *advice*.

4.2.2.3 Diagrama de classes

Como foi utilizada a ferramenta GALS (GESSER, 2003) na especificação da linguagem de aspecto, as classes que se referem ao compilador dos programas de aspectos foram geradas por essa ferramenta. Essas classes estão representadas na figura 10.

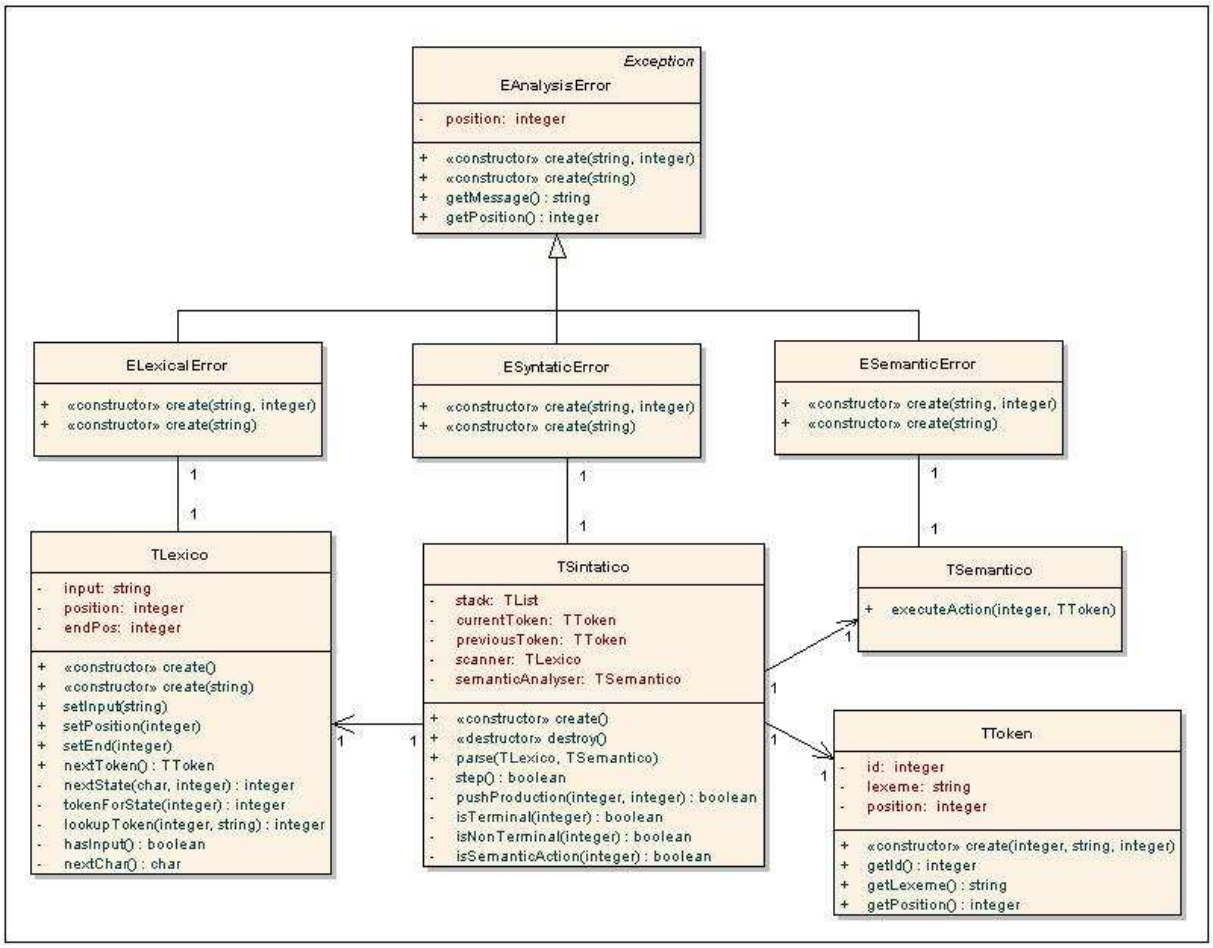


Figura 10 - Classes geradas pelo GALS (GESSER, 2003)

Das classes geradas pelo GALS (GESSER, 2003), a única que sofreu modificações foi a classe TSemantico. Como o objetivo do GALS é gerar analisadores léxico e sintático, a classe TSemantico foi gerada apenas com a assinatura do método executeAction (action : integer; const token : TToken).

Na figura 11 estão representadas as classes criadas para desenvolvimento da ferramenta. A classe TSemantico gerada pelo GALS aparece novamente, desta vez com as alterações feitas para atender as necessidades do protótipo. A fim de extrair as informações necessárias dos programas de aspectos para posteriormente realizar o *weaving*, e para garantir a semântica dos programas de aspectos, a classe TSemantico foi alterada, sendo implementado o método executeAction e adicionados outros métodos e atributos.

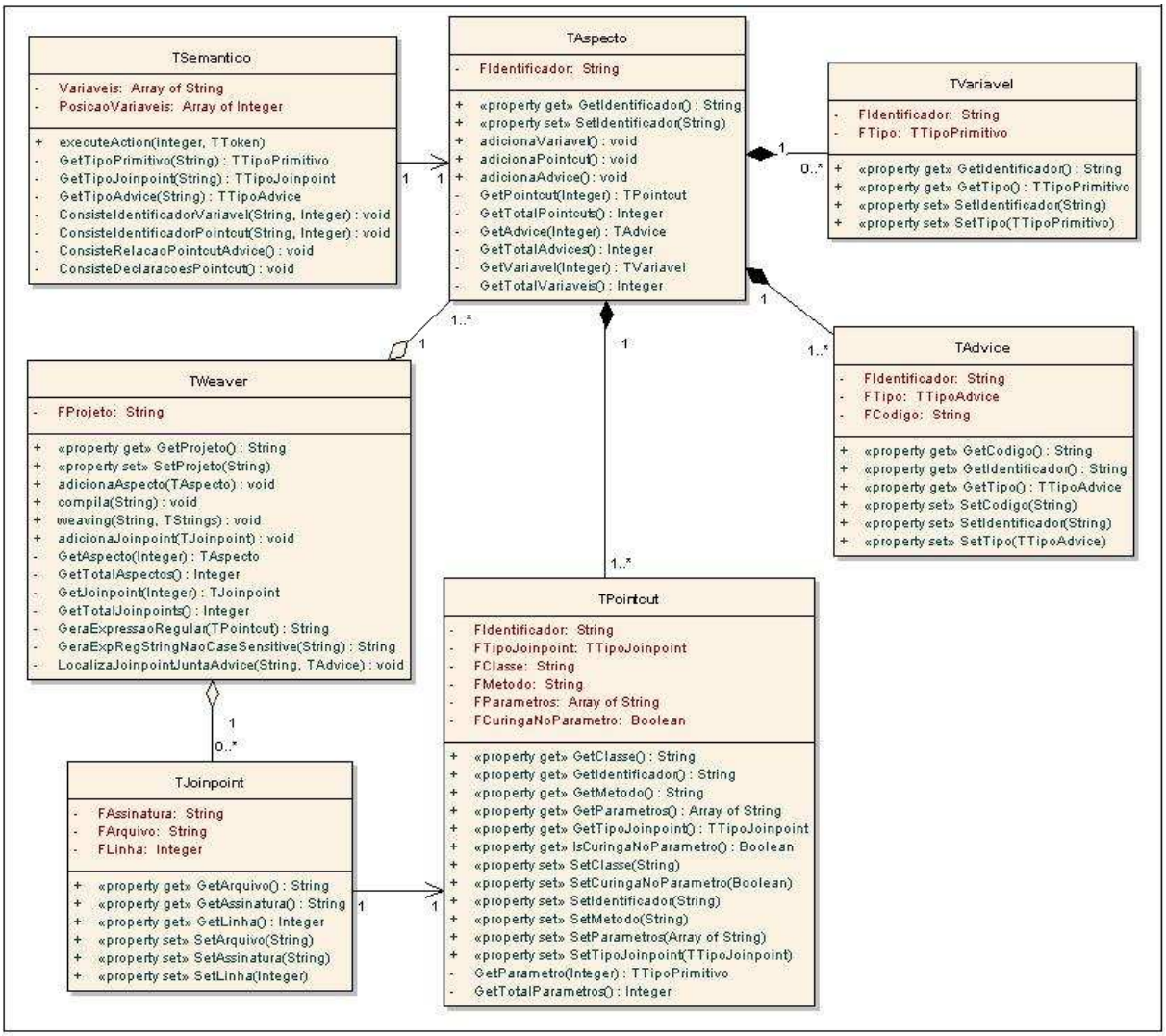


Figura 11 – Diagrama de classes da ferramenta AOPDelphi

Na compilação de um projeto AOPDelphi, para cada programa de aspecto é instanciado um objeto de TAspecto. O mesmo acontece com a classe TSemantico. Quando o analisador sintático faz o *parsing*, é invocado o método executeAction da classe TSemantico para cada ação semântica definida na gramática da linguagem.

Durante o *parsing*, o compilador identifica no programa fonte as variáveis, os *pointcuts* e os *advices* do aspecto. Para cada um desses são instanciados objetos de TVariavel, TPointcut e TAdvice, respectivamente, armazenando as devidas informações. A classe TAspecto possui uma composição para cada uma dessas três classes. A composição de TPointcut armazena os *pointcuts* detectados no programa de aspecto. No momento do *weaving* eles serão utilizados para saber quais os métodos dos programas de componentes

devem ser interceptados. A composição de TAdvice armazena os *advices* declarados no aspecto, sendo que para cada *pointcut* deve haver pelo menos um *advice*. O atributo FCodigo de TAdvice armazena o código que será inserido quando o *pointcut* correspondente for acionado.

A classe TWeaver é a principal classe da ferramenta. Um objeto de TWeaver é instanciado no momento do *weaving* de um projeto AOPDelphi. No atributo FProjeto é armazenado o caminho do projeto Delphi (arquivo .dpr) afetado pelos programas de aspectos que compõem o projeto AOPDelphi. A classe agrega os objetos de TAspectos que são instanciados na compilação dos programas de aspectos. Agrega também objetos de TJoinpoint, instanciados no momento que é localizado um método que será interceptado pelo aspecto.

No método *weaving* de TWeaver acontece a principal funcionalidade da ferramenta. Tendo os programas de aspectos compilados, o *weaver* percorre a lista de aspectos do projeto e para cada *pointcut* dos aspectos é chamado o método GeraExpressaoRegular, que baseado nas informações do *pointcut* gera uma expressão regular para localizar o método que deve ser interceptado. A busca e a junção do fonte é feita pelo método LocalizaJoinpointJuntaAdvice, que recebe como argumentos a expressão regular do método a ser localizado e o *advice* que se referem ao *pointcut*.

4.3 IMPLEMENTAÇÃO

Nesta seção são abordados os aspectos sobre a implementação do protótipo, as ferramentas utilizadas para sua construção e o seu funcionamento.

4.3.1 Ferramentas utilizadas

A ferramenta foi implementada utilizando a linguagem de programação Delphi no ambiente Borland Delphi 7. No quadro 7 é apresentado um trecho de código fonte da ferramenta. O código se refere a *interface* da classe TPointcut.

```
TAspecto = class
private
  FIdentificador: String;
  FLista_Pointcuts: TList;
  FLista_Advices: TList;
  FLista_Variaveis: TList;

  Function GetPointcut(I: Integer): TPointcut;
  Function GetTotalPointcuts :Integer;
  Function GetAdvice(I: Integer): TAdvice;
  Function GetTotalAdvices: Integer;
  Function GetVariavel(I: Integer): TVariavel;
  Function GetTotalVariaveis: Integer;
public
  function adicionaVariavel(Nome: String): TVariavel;
  function adicionaPointcut(Nome : String): TPointcut;
  function adicionaAdvice(Nome: String): TAdvice;
  constructor Create;
  destructor Destroy; override;
  property Identificador : String read FIdentificador write FIdentificador;
  property Pointcut[index : Integer] : TPointcut read GetPointcut;
  property TotalPointcuts : Integer read GetTotalPointcuts;
  property Advice[index: Integer]: TAdvice read GetAdvice;
  property TotalAdvices : Integer read GetTotalAdvices;
  property Variavel[index: Integer]: TVariavel read GetVariavel;
  property TotalVariaveis : Integer read GetTotalVariaveis;
end;
```

Quadro 7 – *Interface* da classe TAspecto

Para trabalhar com as expressões regulares, foi utilizada a biblioteca RegExp Studio (SOROKIN), que é uma biblioteca *freeware* para trabalhar com expressões regulares no Delphi.

Na implementação do compilador dos programas de aspectos foi utilizada também a ferramenta GALS (GESSER, 2003). O GALS é um gerador de analisadores léxico e sintático. Nele foram geradas as classes para a implementação dos analisadores léxico e sintático, e a *interface* da classe do analisador semântico. O GALS tem a opção de gerar essas classes nas

linguagens C, Java ou Delphi. Isso é feito com base nas definições regulares, palavras reservadas, símbolos especiais, gramática e outras informações que são fornecidas como entrada na ferramenta. No quadro 8 é apresentada a *interface* da classe TLexico, gerada pelo GALS.

```
TLexico = class
public
    constructor create; overload;
    constructor create(input : string); overload;

    procedure setInput(input : string);
    procedure setPosition(pos : integer);
    procedure setEnd(endPos : integer);
    function nextToken : TToken; //raises ELexicalError

private
    input : string;
    position : integer;
    endPos : integer;

    function nextState(c : char; state : integer) : integer;
    function tokenForState(state : integer) : integer;
    function lookupToken(base : integer; key : string) : integer;

    function hasInput : boolean;
    function nextChar : char;
end;
```

Quadro 8 – *Interface* da classe TLexico

4.3.2 Operacionalidade da implementação

Ao ser executado o aplicativo AOPDelphi.exe, inicialmente a ferramenta verifica a necessidade de criar alguns diretórios. A estrutura de diretórios do AOPDelphi está representada na figura 12.

Figura 12 – Estrutura de diretórios do AOPDelphi

A pasta Projetos é criada para armazenar os projetos AOPDelphi salvos. A ferramenta trabalha com o conceito de projeto. Um projeto AOPDelphi é um conjunto de informações organizadas em um arquivo que especifica o projeto Delphi que será afetado pelos aspectos, os programas de aspectos que irão interagir com o projeto Delphi e algumas restrições de *units* que não devem ser aspectadas[3]. Este arquivo possui a extensão .apr e é organizado como um arquivo .ini. Ao ser acionado o comando para salvar um projeto, a ferramenta sugere que seja salvo no diretório projetos criado no mesmo local do arquivo executável.

A pasta Base armazena os arquivos XML utilizados pela aplicação. São utilizados três arquivos XML (Aspectos.xml, ListaUnits.xml, Projetos.xml) para armazenar as informações de projetos AOPDelphi. Esta pasta não é criada pela ferramenta, porém é fundamental a sua existência no diretório da aplicação com os três arquivos .xml. Do contrário, ao ser executada a ferramenta, será emitida uma mensagem informando a ausência dos arquivos .xml e em seguida a aplicação será abortada.

A pasta Entrada guarda os arquivos de aspectos implementados na ferramenta. Ao executar o comando para salvar um programa de aspecto, a aplicação sugere que este seja salvo no diretório entrada. Os arquivos de programas de aspectos em AOPDelphi possuem extensão .dao.

A pasta Lista de *units* é utilizada para armazenar as listas de *units* que não serão afetadas pelos programas de aspectos. A ferramenta dispõe de mecanismos para criar tais

[3] Trata-se de um neologismo. É o mesmo que ser afetado pelo aspecto.

listas. Ao salvar uma lista, o diretório lista de *units* é sugerido como local para salvar o arquivo. Os arquivos de lista de *units* são arquivos do tipo texto com extensão .lun.

A pasta Saída é utilizada para armazenar os fontes dos projetos AOPDelphi já compilados.

Com exceção da pasta base, as demais pastas citadas são criadas no diretório da aplicação ao iniciar a ferramenta, caso alguma delas não exista. Quando é executada a aplicação ela é apresentada conforme mostra a figura 13.

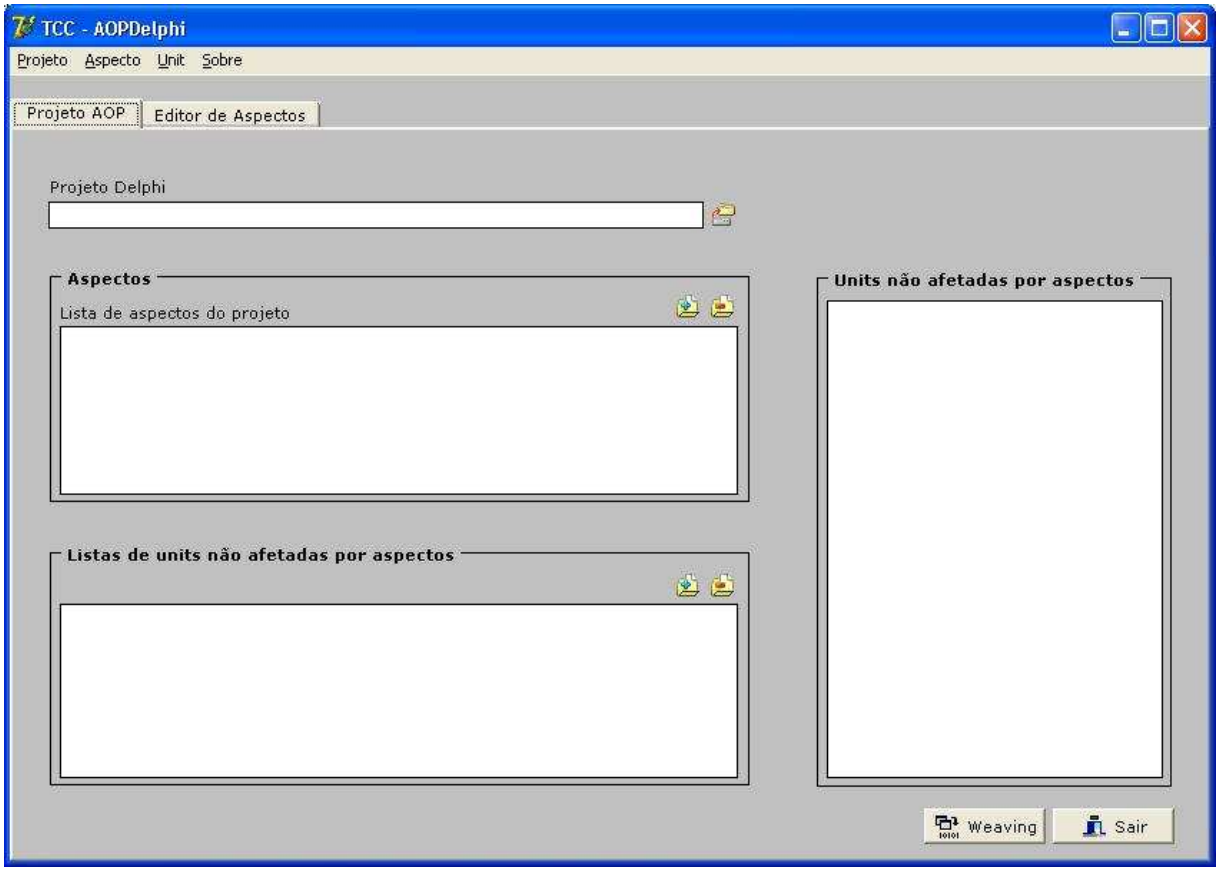


Figura 13 – Apresentação da ferramenta AOPDelphi

A tela do AOPDelphi é formada por duas guias. Projeto AOP e Editor de Aspectos. A guia Editor de Aspectos é o ambiente para programação dos aspectos. O ambiente possui comandos básicos de manipulação de arquivos como abrir, salvar, solicitar um arquivo novo, além da função compilar. Possui uma área com o editor para implementar os programas de aspectos e abaixo do editor, uma área destinada às mensagens de compilação (erros, avisos). A tela com a guia Editor de Aspectos ativa, com um programa de aspecto aberto, e as opções

de comando para programas de aspectos estão representadas nas figuras 14 e 15, respectivamente.

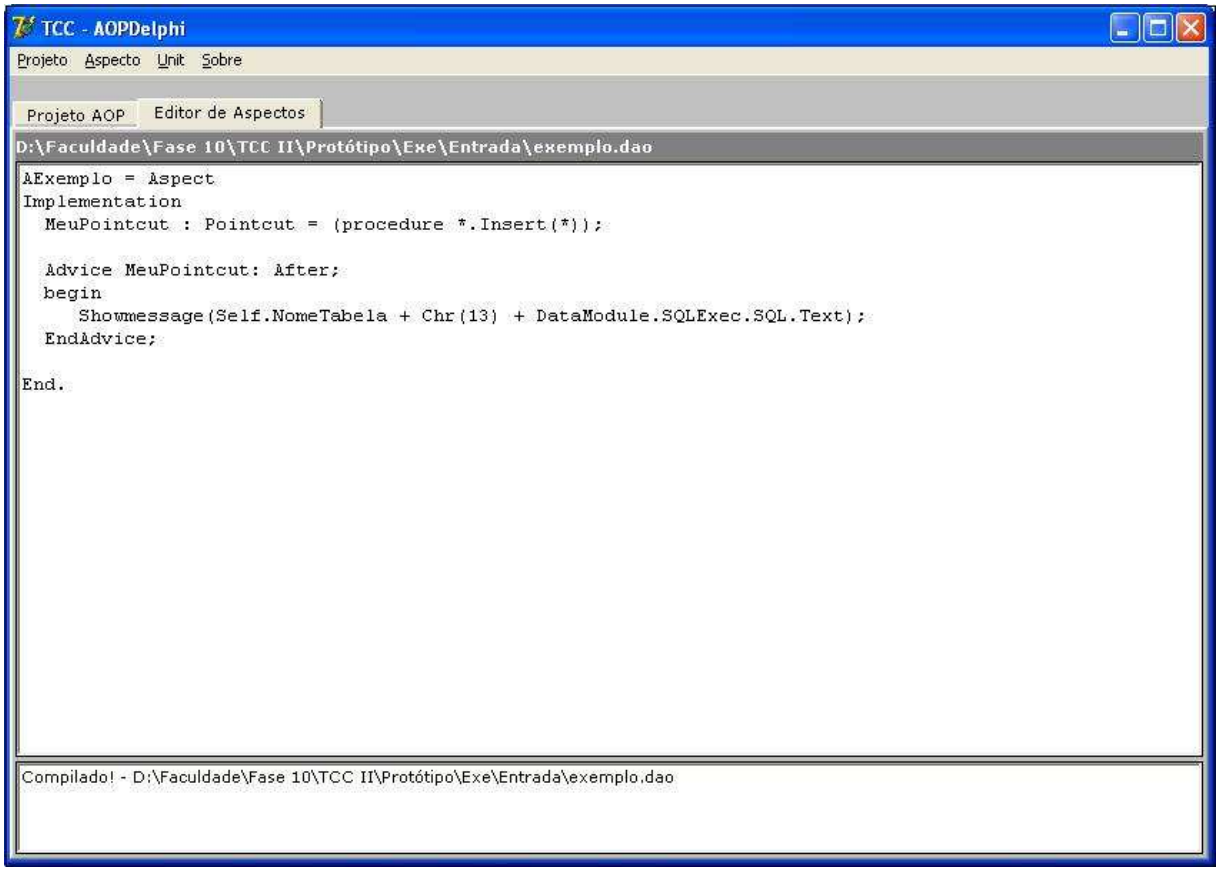


Figura 14 – Ambiente para programação dos aspectos

Figura 15 – Opções de comando para um programa de aspecto

A figura 14 mostra o ambiente com um programa de aspectos aberto. No programa foi implementado um aspecto de nome AExemplo que possui o *pointcut* MeuPointcut. Este *pointcut* se refere a todos os métodos de inserção (*Insert*), seja qual for a classe e independente da quantidade de parâmetros que ele esteja passando. O *pointcut* possui um *advice* do tipo *after* associado, que conforme foi implementado, deve ser executado depois da

execução do método que será interceptado. Ou seja, neste exemplo, qualquer classe do sistema que invocar o método *Insert*, após a execução da última instrução do método, será executada a linha Showmessage(Self.NomeTabela + Chr(13) + DataModule.SQLExec.SQL.Text). O termo *Self* dentro do código *advice* estará sempre fazendo referência ao objeto que está invocando o método interceptado. O manual da linguagem de aspecto AOPDelphi está disponível no apêndice A deste trabalho.

No editor de aspectos, ao executar o comando compilar, o programa fonte é analisado léxica e sintaticamente. Sendo encontrado algum erro, este é informado ao usuário na parte inferior da tela, através da mensagem e a posição do erro (linha, coluna) no programa fonte. O cursor também é posicionado no local do erro no programa fonte. No caso dos *advices*, o código escrito entre `begin` e `EndAdvice` não é analisado pelo compilador. Para isso acontecer seria necessário trabalhar com a gramática completa da linguagem Delphi.

Na guia Projeto AOP são definidos os parâmetros para compilação de um projeto orientado a aspectos. A figura 16 apresenta as opções de comando para um projeto AOPDelphi.

Figura 16 – Opções de comando para projetos AOPDelphi

No campo Projeto Delphi é informado o arquivo de projeto Delphi (arquivo .dpr) que será envolvido no projeto orientado a aspectos. Todas as *units* que fazem parte desse projeto serão aspectadas, com exceção daquelas que estiverem incluídas nas listas de *units* que não devem ser afetadas por aspectos. A ferramenta oferece mecanismos para criar essas listas e sua tela é apresentada na figura 17.

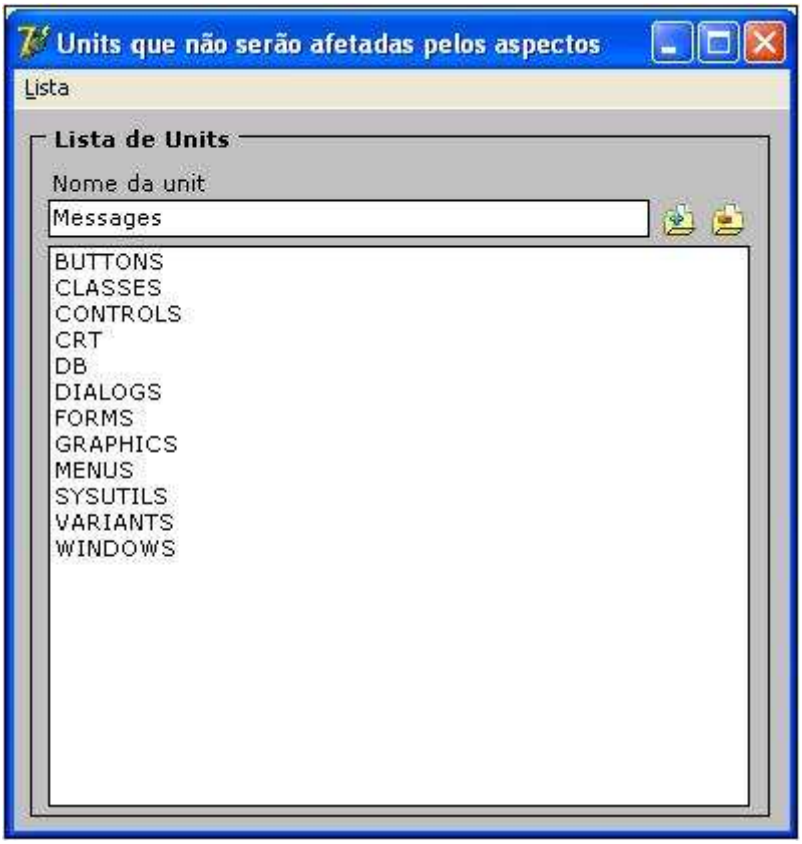


Figura 17 – Tela para criar listas de units que não serão afetadas por aspectos

Na tela apresentada na figura 17 é possível criar novas listas e abrir listas já existentes para serem editadas. Ao salvar uma lista é criado um arquivo do tipo texto com extensão .lun. As *units* aparecem na lista sempre em caixa alta e em ordem alfabética.

Após informar o projeto Delphi, o desenvolvedor deve indicar à ferramenta quais são os programas de aspectos que irão interagir com o projeto Delphi informado. A ferramenta dispõe de funcionalidades para adicionar e remover programas de aspectos (arquivo .dao) na lista de aspectos do projeto. Ao ser selecionada a opção de adicionar um programa à lista de aspectos do projeto, a ferramenta abre uma caixa de diálogo com o diretório **Entrada** ativo, para selecionar um arquivo .dao. Obviamente é possível selecionar os programas que estejam em diretórios diferentes. Através de um clique duplo em um item da lista, a guia Editor de aspectos é ativada e o programa que se refere o item é aberto no editor. A figura 18 mostra a lista de aspectos de um projeto AOPDelphi.

Figura 18 – Lista de aspectos de um projeto AOPDelphi

No campo seguinte o desenvolvedor pode indicar à ferramenta as listas de *units* que não serão afetadas pelos aspectos do projeto. Ao selecionar a opção para adicionar uma lista, será aberta uma caixa de diálogo com o diretório Lista de *units* ativo. A cada lista adicionada, suas *units* são inseridas no campo *Units* não afetadas por aspectos. Durante o processo de *weaving*, cada *unit* referenciada em algum fonte lido será analisada, a menos de que seu nome conste nesta lista de restrição.

Tendo informado esses parâmetros, o projeto AOPDelphi está pronto para ser compilado. Isto pode ser feito através do comando Projeto/*Weaving* na barra de menu, pelo botão *Weaving* na guia Projeto AOP, ou simplesmente pela tecla de atalho F9.

Ao executar o comando *Weaving*, o projeto Delphi informado no projeto será compilado para certificar de que não possua erros. Em seguida são compilados os programas de aspectos. É criado um diretório com o mesmo nome do arquivo .dpr do projeto dentro da pasta Saída e para ele são copiados os fontes do projeto Delphi. As alterações que tiverem que ser feitas durante o processo de *weaving* serão feitas nos fontes que foram copiados para o diretório de saída. Após ocorrer a junção dos programas de aspectos e do programa de componentes, os fontes gerados serão compilados. Ocorrendo sucesso, a aplicação será executada. Havendo erros de compilação no projeto Delphi após o *weaving*, a ferramenta apresenta os tais erros e sua origem. Ou seja, é informado se o erro em questão provém do código escrito no *advice* ou é em função da junção feita no *weaving*.

Considerando o aspecto apresentado na figura 11, o quadro 9 apresenta o resultado da

compilação de um projeto AOPDelphi onde foi afetado o método de uma classe.

Antes do *weaving*

```
procedure TGrupoParticipante.Insert;
begin
  With DataModule.SQLExec do
       begin
         Close;
         Sql.Clear;
         Sql.Add('Insert Into GRUPO');
         Sql.Add('(CD_GRUPO, DS_GRUPO, CD_ESTEREOTIPO)');
         Sql.Add('Values (:CD_GRUPO, :DS_GRUPO, :CD_ESTEREOTIPO)');
         ParamByName('CD_GRUPO').AsInteger := FCodigo;
         ParamByName('DS_GRUPO').AsString := FDescricao;
         ParamByName('CD_ESTEREOTIPO').AsInteger := FEstereotipo;
         ExecQuery;
       end;
end;
```

Após o *weaving*

```
procedure TGrupoParticipante.Insert;
begin
  With DataModule.SQLExec do
       begin
         Close;
         Sql.Clear;
         Sql.Add('Insert Into GRUPO');
         Sql.Add('(CD_GRUPO, DS_GRUPO, CD_ESTEREOTIPO)');
         Sql.Add('Values (:CD_GRUPO, :DS_GRUPO, :CD_ESTEREOTIPO)');
         ParamByName('CD_GRUPO').AsInteger := FCodigo;
         ParamByName('DS_GRUPO').AsString := FDescricao;
         ParamByName('CD_ESTEREOTIPO').AsInteger := FEstereotipo;
         ExecQuery;
       end;
// Código inserido por AOPDelphi - Inicio
  Showmessage(Self.NomeTabela + Chr(13) + DataModule.SQLExec.SQL.Text);
// Código inserido por AOPDelhi - Fim
end;
```

Quadro 9 – Método afetado por um aspecto

Como é apresentado no quadro 9, durante o processo de *weaving* foi identificado o método `Insert` na classe TGrupoParticipante. O método foi interceptado e o código implementado no *advice* correspondente ao *pointcut* foi inserido.

Para finalizar, a figura 19 apresenta a tela Sobre o AOPDelphi, que pode ser exibida acionando o comando Sobre/Sobre o AOPDelphi na barra de menu.

Figura 19 – Tela Sobre o AOPDelphi

4.3.3 Estudo de caso

O objetivo nesse estudo de caso é implementar um controle de *log* e autenticação no paradigma da programação orientada a aspectos através do AOPDelphi. Os aspectos de *log* e autenticação estão representados nos quadros 10 e 11, respectivamente.

```
ALog = Aspect
Implementation
  logInsert : Pointcut = (* *.SQLInsert(*));
  logUpdate : Pointcut = (* *.SQLUpdate(*));
  logDelete : Pointcut = (* *.SQLDelete(*));

  Advice logInsert: After;
  begin
    With DMTaskMan.qryLog do
    begin
      ParamByName('LO_IDUSUARIO').AsInteger := giCodUsuario;
      ParamByName('LO_TABELA').AsString := Self.NomeTabela;
      ParamByName('LO_OPERACAO').AsInteger := 1; //1=Insert, 2=Update,
3=Delete
      ParamByName('LO_DATAHORA').AsDateTime := Now;
      ParamByName('LO_SQL').AsString := DMTaskMan.SQLExec.SQL.Text;
      ExecSQL;
    end;
  EndAdvice;

  Advice logUpdate: After;
  begin
    With DMTaskMan.qryLog do
    Begin
      ParamByName('LO_IDUSUARIO').AsInteger := giCodUsuario;
      ParamByName('LO_TABELA').AsString := Self.NomeTabela;
      ParamByName('LO_OPERACAO').AsInteger := 2; //1=Insert, 2=Update,
3=Delete
      ParamByName('LO_DATAHORA').AsDateTime := Now;
      ParamByName('LO_SQL').AsString := DMTaskMan.SQLExec.SQL.Text;
      ExecSQL;
    end;
  EndAdvice;

  Advice logDelete: After;
  begin
    With DMTaskMan.qryLog do
    begin
      ParamByName('LO_IDUSUARIO').AsInteger := giCodUsuario;
      ParamByName('LO_TABELA').AsString := Self.NomeTabela;
      ParamByName('LO_OPERACAO').AsInteger := 3; //1=Insert, 2=Update,
3=Delete
      ParamByName('LO_DATAHORA').AsDateTime := Now;
      ParamByName('LO_SQL').AsString := DMTaskMan.SQLExec.SQL.Text;
      ExecSQL;
    end;
  EndAdvice;

End.
```

Quadro 10 – Aspecto de log

```
AAutenticacao = Aspect
Implementation
  AutenticaInsert : Pointcut = (* *.SQLInsert(*));
  AutenticaUpdate : Pointcut = (* *.SQLUpdate(*));
  AutenticaDelete : Pointcut = (* *.SQLDelete(*));

  Advice AutenticaInsert: Before;
  begin
    With DMTaskMan.qryAutentica do
    begin
      Close;
      Sql.Clear;
      Sql.Add('Select count(*) from DIREITO_USUARIO');
      Sql.Add('Where IDUSUARIO = :IDUSUARIO and OPERACAO = :OPERACAO');
      Sql.Add(' and TABELA = :TABELA');
      ParamByName('IDUSUARIO').AsInteger := giCodUsuario;
      ParamByName('OPERACAO').AsInteger := 1;  //1=Insert, 2=Update, 3=Delete
      ParamByName('TABELA').AsInteger := Self.NomeTabela;
      ExecSQL;
      if  Fields[0].AsInteger = 0 then
          Raise Exception.Create('Você não tem permissão para essa atividade. ' +
Chr(13) + 'Entre em contato com o seu superior.');
    end;
  EndAdvice;

  Advice AutenticaUpdate: Before;
  begin
    With DMTaskMan.qryAutentica do
    begin
      Close;
      Sql.Clear;
      Sql.Add('Select count(*) from DIREITO_USUARIO');
      Sql.Add('Where IDUSUARIO = :IDUSUARIO and OPERACAO = :OPERACAO');
      Sql.Add(' and TABELA = :TABELA');
      ParamByName('IDUSUARIO').AsInteger := giCodUsuario;
      ParamByName('OPERACAO').AsInteger := 2;  //1=Insert, 2=Update, 3=Delete
      ParamByName('TABELA').AsInteger := Self.NomeTabela;
      ExecSQL;
      if  Fields[0].AsInteger = 0 then
          Raise Exception.Create('Você não tem permissão para essa atividade. ' +
Chr(13) + 'Entre em contato com o seu superior.');
    end;
  EndAdvice;

  Advice AutenticaDelete: Before;
  begin
    With DMTaskMan.qryAutentica do
    begin
      Close;
      Sql.Clear;
      Sql.Add('Select count(*) from DIREITO_USUARIO');
      Sql.Add('Where IDUSUARIO = :IDUSUARIO and OPERACAO = :OPERACAO');
      Sql.Add(' and TABELA = :TABELA');
      ParamByName('IDUSUARIO').AsInteger := giCodUsuario;
      ParamByName('OPERACAO').AsInteger := 3;  //1=Insert, 2=Update, 3=Delete
      ParamByName('TABELA').AsInteger := Self.NomeTabela;
      ExecSQL;
      if  Fields[0].AsInteger = 0 then
          Raise Exception.Create('Você não tem permissão para essa
atividade. ' + Chr(13) + 'Entre em contato com o seu superior.');
    end;
  EndAdvice;

End.
```

Quadro 11 – Aspecto de autenticação

O projeto Delphi envolvido no estudo de caso é um sistema com alguns programas de cadastros. Somente usuários autorizados podem ter acesso ao sistema e as suas funcionalidades. Para isso, ao entrar no sistema o usuário deve informar o seu *login* e senha. Sendo validado o seu acesso, o código do usuário é armazenado na variável global `giCodUsuario`. Ainda para contemplar o acesso restrito às funcionalidades autorizadas, o sistema possui a tabela DIREITO_USUARIO no banco de dados onde são armazenadas as operações (*insert, update, delete*) que cada usuário tem direito de executar nas tabelas. Para cada operação que for executada, devem ser registrados na tabela de *log* o usuário responsável, a tabela envolvida, a operação, data e hora e instrução SQL. E a cada operação nas tabelas, deve ser consistido se o usuário ativo tem direito. Três classes no sistema (`TTarefas`, `TClientes` e `TEncaminhaTarefas`) possuem os métodos `SQLInsert`, `SQLUpdate` e `SQLDelete` que realizam inserção, atualização e exclusão, respectivamente.

O aspecto de *log* possui um *pointcut* para cada operação e um *advice* do tipo `after` associado a cada *pointcut*. O aspecto de autenticação também possui um *pointcut* para cada operação e um *advice* associado, porém do tipo `before`.

Os quadros 12, 13 e 14 apresentam os resultados do estudo de caso elaborado.

Antes do *weaver*

```
Function TEncaminhaTarefa.SQLInsert:Integer;
begin
  With DMTaskMan.SQLQuery do
    begin
      Close;
      Sql.Clear;
      Sql.Add('Insert into ENCAMINHAMENTO Values(:EN_ID_ENCAM, :EN_CLIENTE,');
      Sql.Add(' :EN_USUARIO, :EN_DATA_CRIACAO, :EN_DATA_CONCLUSAO, :EN_STATUS)');
      ParamByName('EN_ID_ENCAM').AsInteger := flGetProximoCodigo;
      Result:= ParamByName('EN_ID_ENCAM').AsInteger;
      ParamByName('EN_CLIENTE').AsInteger := uaClientes[cbCliente.ItemIndex];
      ParamByName('EN_USUARIO').AsInteger := uaUsuarios[cbUsuario.ItemIndex];
      ParamByName('EN_DATA_CRIACAO').AsDate := edtDataCriacao.Date;
      ParamByName('EN_DATA_CONCLUSAO').Value := Null;
      ParamByName('EN_STATUS').AsString := 'P'; // Pendente
      ExecSQL;
    end;
End;
```

Após o *weaver*

```
Function TEncaminhaTarefa.SQLInsert:Integer;
begin
  // Código inserido por AOPDelphi - Início
    With DMTaskMan.qryAutentica do
    begin
       Close;
       Sql.Clear;
      Sql.Add('Select count(*) from DIREITO_USUARIO');
      Sql.Add('Where IDUSUARIO = :IDUSUARIO and OPERACAO = :OPERACAO');
      Sql.Add(' and TABELA = :TABELA');
      ParamByName('IDUSUARIO').AsInteger := giCodUsuario;
      ParamByName('OPERACAO').AsInteger := 1;  //1=Insert, 2=Update, 3=Delete
      ParamByName('TABELA').AsInteger := Self.NomeTabela;
      ExecSQL;
      if  Fields[0].AsInteger = 0 then
          Raise Exception.Create('Você não tem permissão para essa atividade. ' +
Chr(13) +
                                 'Entre em contato com o seu superior.');
    end;
  // Código inserido por AOPDelphi - Fim

  With DMTaskMan.SQLQuery do
    begin
      Close;
      Sql.Clear;
      Sql.Add('Insert into ENCAMINHAMENTO Values(:EN_ID_ENCAM, :EN_CLIENTE,');
      Sql.Add(' :EN_USUARIO, :EN_DATA_CRIACAO, :EN_DATA_CONCLUSAO, :EN_STATUS)');
      ParamByName('EN_ID_ENCAM').AsInteger := flGetProximoCodigo;
      Result:= ParamByName('EN_ID_ENCAM').AsInteger;
      ParamByName('EN_CLIENTE').AsInteger := uaClientes[cbCliente.ItemIndex];
      ParamByName('EN_USUARIO').AsInteger := uaUsuarios[cbUsuario.ItemIndex];
      ParamByName('EN_DATA_CRIACAO').AsDate := edtDataCriacao.Date;
      ParamByName('EN_DATA_CONCLUSAO').Value := Null;
      ParamByName('EN_STATUS').AsString := 'P'; // Pendente
      ExecSQL;
    end;

    // Código inserido por AOPDelphi - Início
    With DMTaskMan.qryLog do
    begin
      ParamByName('LO_IDUSUARIO').AsInteger := giCodUsuario;
      ParamByName('LO_TABELA').AsString := Self.NomeTabela;
      ParamByName('LO_OPERACAO').AsInteger := 1; //1=Insert, 2=Update, 3=Delete
      ParamByName('LO_DATAHORA').AsDateTime := Now;
      ParamByName('LO_SQL').AsString := DMTaskMan.SQLExec.SQL.Text;
      ExecSQL;
    end;
    // Código inserido por AOPDelphi - Fim
end;
```

Quadro 12 – Método SQLInsert da classe TEncaminhaTarefa, antes e depois da ação dos aspectos de *log* e autenticação.

Antes do *weaver*

```
        procedure TTarefas.SQLUpdate;
        begin
          With DMTaskMan.SQLQuery do
            begin
              Close;
              Sql.Clear;
              Sql.Add('Update TAREFA set TA_DESCRICAO = :TA_DESCRICAO');
              Sql.Add('Where TA_CODIGO = :TA_CODIGO');
              ParamByName('TA_CODIGO').AsInteger := StrtoInt(Trim(edtCodigo.Text));
              ParamByName('TA_DESCRICAO').AsString := edtDescricao.Text;
              ExecSQL;
            end;
        end;
```

Após o *weaver*

```
        procedure TTarefa.SQLUpdate;
        begin
          // Código inserido por AOPDelphi - Início
            With DMTaskMan.qryAutentica do
            begin
              Close;
              Sql.Clear;
              Sql.Add('Select count(*) from DIREITO_USUARIO');
              Sql.Add('Where IDUSUARIO = :IDUSUARIO and OPERACAO = :OPERACAO');
              Sql.Add(' and TABELA = :TABELA');
              ParamByName('IDUSUARIO').AsInteger := giCodUsuario;
              ParamByName('OPERACAO').AsInteger := 2;  //1=Insert, 2=Update, 3=Delete
              ParamByName('TABELA').AsInteger := Self.NomeTabela;
              ExecSQL;
              if  Fields[0].AsInteger = 0 then
                  Raise Exception.Create('Você não tem permissão para essa atividade. ' +
Chr(13) +
                                         'Entre em contato com o seu superior.');
            end;
          // Código inserido por AOPDelphi - Fim

          With DMTaskMan.SQLQuery do
            begin
              Close;
              Sql.Clear;
              Sql.Add('Update TAREFA set TA_DESCRICAO = :TA_DESCRICAO');
              Sql.Add('Where TA_CODIGO = :TA_CODIGO');
              ParamByName('TA_CODIGO').AsInteger := StrtoInt(Trim(edtCodigo.Text));
              ParamByName('TA_DESCRICAO').AsString := edtDescricao.Text;
              ExecSQL;
            end;

            // Código inserido por AOPDelphi - Início
            With DMTaskMan.qryLog do
            begin
              ParamByName('LO_IDUSUARIO').AsInteger := giCodUsuario;
              ParamByName('LO_TABELA').AsString := Self.NomeTabela;
              ParamByName('LO_OPERACAO').AsInteger := 2; //1=Insert, 2=Update, 3=Delete
              ParamByName('LO_DATAHORA').AsDateTime := Now;
              ParamByName('LO_SQL').AsString := DMTaskMan.SQLExec.SQL.Text;
              ExecSQL;
            end;
            // Código inserido por AOPDelphi - Fim
        end;
```

Quadro 13 – Método SQLUpdate da classe TTarefa, antes e depois da ação dos aspectos de *log* e autenticação.

Antes do *weaver*

```
        procedure TClientes.SQLDelete(vId: Integer);
        begin
          With DMTaskMan.SqlQuery do
             begin
               Close;
               Sql.Clear;
               Sql.Add('Delete from CLIENTE Where CL_CODIGO = :CL_CODIGO');
               ParamByName('CL_CODIGO').AsInteger := vID;
               ExecSql;
             end;
        end;
```

Após o *weaver*

```
        procedure TClientes.SQLDelete(vId: Integer);
        begin
          // Código inserido por AOPDelphi - Início
            With DMTaskMan.qryAutentica do
            begin
              Close;
              Sql.Clear;
              Sql.Add('Select count(*) from DIREITO_USUARIO');
              Sql.Add('Where IDUSUARIO = :IDUSUARIO and OPERACAO = :OPERACAO');
              Sql.Add(' and TABELA = :TABELA');
              ParamByName('IDUSUARIO').AsInteger := giCodUsuario;
              ParamByName('OPERACAO').AsInteger := 3;  //1=Insert, 2=Update, 3=Delete
              ParamByName('TABELA').AsInteger := Self.NomeTabela;
              ExecSQL;
              if  Fields[0].AsInteger = 0 then
                  Raise Exception.Create('Você não tem permissão para essa atividade. ' +
Chr(13) +
                                         'Entre em contato com o seu superior.');
            end;
          // Código inserido por AOPDelphi - Fim

            With DMTaskMan.SqlQuery do
             begin
               Close;
               Sql.Clear;
               Sql.Add('Delete from CLIENTE Where CL_CODIGO = :CL_CODIGO');
               ParamByName('CL_CODIGO').AsInteger := vID;
               ExecSql;
             end;

            // Código inserido por AOPDelphi - Início
            With DMTaskMan.qryLog do
            begin
              ParamByName('LO_IDUSUARIO').AsInteger := giCodUsuario;
              ParamByName('LO_TABELA').AsString := Self.NomeTabela;
              ParamByName('LO_OPERACAO').AsInteger := 3; //1=Insert, 2=Update, 3=Delete
              ParamByName('LO_DATAHORA').AsDateTime := Now;
              ParamByName('LO_SQL').AsString := DMTaskMan.SQLExec.SQL.Text;
              ExecSQL;
            end;
            // Código inserido por AOPDelphi - Fim
        end;
```

Quadro 14 – Método SQLDelete da classe TClientes, antes e depois da ação dos aspectos de *log* e autenticação.

4.4 RESULTADOS E DISCUSSÃO

Considerando a proposta da ferramenta, ela apresentou bons resultados, no sentido de prover suporte a programação orientada a aspectos na linguagem Delphi. Porém, a linguagem de aspecto é bem limitada. A falta do recurso de criar variáveis, procedimentos e funções locais dentro dos *advices* dificulta um pouco o trabalho do desenvolvedor. Apesar da eliminação da redundância de código ser um dos objetivos da POA, devido às limitações da linguagem AOPDelphi, é comum haver redundância dentro do código escrito nos *advices*. Um exemplo disso é a implementação dos aspectos de autenticação e *log* no estudo de caso da seção 4.3.3. Na implementação dos *advices* de ambos aspectos, o código escrito para sua implementação é quase idêntico, alterando apenas o valor que é passado para o parâmetro Operação. Isso porque a linguagem não permite a implementação de procedimentos e funções locais dentro dos *advices*. Com procedimentos e funções locais esse comportamento seria diferente, sendo necessário a implementação de apenas uma dessas rotinas e o valor da operação poderia ser passado por parâmetro.

Considerando o estudo de caso elaborado na seção 4.3.3, observa-se a facilidade e a confiabilidade que a ferramenta pode proporcionar no desenvolvimento. Para a implementação do *log* do sistema através da ferramenta, foi necessário a implementação de um aspecto com três *pointcuts* e três *advices*. O mesmo aconteceu na implementação do aspecto de autenticação. Além de manter íntegros os programas que têm seu interesse específico, não gerando entrelaçamento de código, a implementação dos controles de *log* e autenticação estão prontos até mesmo para as novas classes que vierem a serem adicionadas no sistema.

Os controles de *log* e autenticação poderiam também ser implementados sem fazer uso da tecnologia de orientação a aspectos. Supõe-se por exemplo a implementação de uma classe

para esse fim. Porém dessa forma, cada operação (método) controlada pelo *log* e autenticação deveria ser alterada para contemplar a implementação. Isso gera mais trabalho, redundância e entrelaçamento de código, além de aumentar a possibilidade da ocorrência de erros, já que mais código será implementado. Além do mais, neste caso, se novas classes forem adicionadas ao sistema, deverão surgir mais alterações que diz respeito ao controle de *log* e autenticação.

5 CONCLUSÕES

Este trabalho propicia um conhecimento sobre a tecnologia da programação orientada a aspectos. Considerando a proposta e as vantagens desse paradigma de desenvolvimento, os softwares desenvolvidos com essas técnicas tendem a serem mais flexíveis, melhorando a manutenibilidade e reusabilidade, pois a implementação dos interesses transversais fica encapsulada em módulos fisicamente separados do restante do código. Dessa forma, cada componente terá apenas código específico da implementação de seu negócio, permitindo uma melhor evolução do software.

Sendo uma tecnologia relativamente nova, e pelo fato das linguagens de programação existentes não proverem suporte nativo a POA, é importante o surgimento de novas ferramentas de apoio nessa área.

Os objetivos do trabalho foram atingidos. Foi implementada uma ferramenta composta por uma linguagem de aspectos e um *weaver*, que provêem suporte à programação orientada a aspectos para linguagem Delphi.

A ferramenta GALS foi muito importante na especificação e implementação da linguagem de aspectos, sendo responsável pela implementação dos analisadores léxico e sintático. A biblioteca RegExp Studio, utilizada para trabalhar com expressões regulares, foi fundamental no desenvolvimento da ferramenta na parte que envolve a localização e junção dos fontes.

Uma limitação da linguagem de aspecto é a não possibilidade de criar variáveis, procedimentos e funções locais dentro dos *advices*, o que poderia facilitar bastante algumas implementações.

5.1 EXTENSÕES

Tendo a sua própria *interface*, a ferramenta AOPDelphi trabalha de forma separada do ambiente Delphi. Visando uma maior integração com este ambiente de desenvolvimento e uma melhor usabilidade, sugere-se como trabalho futuro integrar as funcionalidades do AOPDelphi como opções de menu na IDE do Delphi. Os programas de aspectos poderão ser escritos no próprio ambiente do Delphi. Dessa forma, todo o suporte a programação orientada a aspectos estaria disponível em apenas uma ferramenta.

Outra sugestão seria uma melhoria na linguagem de aspecto no sentido de permitir a declaração de variáveis e implementação de procedimentos e funções locais nos *advices*.

Para finalizar, sugere-se também a implementação de uma versão que disponibilize as funcionalidades da ferramenta através de linhas de comando, visando facilidade e maior usabilidade.

REFERÊNCIAS BIBLIOGRÁFICAS

AHO, Alfred V.; SETHI, Ravi; ULLMAN Jeffrey D. **Compiladores:** princípios, técnicas e ferramentas. Tradução Daniel de Ariosto Pinto. Rio de Janeiro: LTC, 1995.

BARROS, Alexandra. **Aspect oriented programming.** [Recife], 2004. Disponível em: <http://www.cin.ufpe.br/~abab/ppt/aop.ppt>. Acesso em: 21 abr. 2006.

GESSER, Carlos E. **GALS:** gerador de analisadores léxicos e sintáticos. 2003. 150 f. Trabalho de Conclusão de Curso (Bacharelado em Ciências da Computação) – Centro Tecnológico, Universidade Federal de Santa Catarina, Florianópolis.

GROTT, Márcio C. **Estudo de caso aplicando programação orientada a aspectos.** 2005. 65 f. Monografia (Especialização em Tecnologia da Informação na Gestão Integrada de Negócios) – Universidade Regional de Blumenau, Blumenau.

JARGAS, Aurélio M. **Expressões regulares:** guia de consulta rápida. São Paulo: Novatec, 2001. Disponível em <http://guia-er.sourceforget.net/>. Acesso em: 29 maio 2006.

KICZALES, Gregor et al. **Aspect-oriented programming.** proceeding of ECOOP'97, Finland: Springer - Verlag, 1997.

KULESZA, Uirá; SANT'ANNA, Cláudio; LUCENA, Carlos J. P. **Técnicas de projeto orientado a aspectos.** Uberlândia, 2005. Disponível em: <http://www.sbbd-sbes2005.ufu.br/mini_cursos.aspx>. Acesso em: 01 maio 2006.

MARSHALL, Garry. **Linguagens de programação para micros.** Rio de Janeiro: Campus, 1986.

MENEZES, Paulo B. **Linguagens formais e autômatos.** 2. ed. Porto Alegre: Sagra Luzzatto, 1998.

NELSON, Torsten. **Apostila do curso de programação orientada a aspectos com AspectJ.** Belo Horizonte, 2005. Disponível em: <http://www.aspectos.org/courses/aulasaop/curso_poa.pdf>. Acesso em: 26 abr. 2006.

PIVETA, Eduardo K. **Aurélia:** um modelo de suporte a programação orientada a aspectos. 2001. 80 f. Dissertação (Mestrado em Ciência da Computação) – Curso de Pós-graduação em Ciência da Computação, Universidade Federal de Santa Catarina, Florianópolis.

PRICE, Ana M. A.; TOSCANI, Simão S. **Implementação de linguagens de programação:** compiladores. 2. ed. Porto Alegre: Sagra Luzzatto, 2001.

RESENDE, Antônio M. P.; SILVA, Claudiney C. **Programação orientada a aspectos em Java.** Rio de Janeiro: Brasoft, 2005.

SEBESTA, Robert W. **Conceitos de linguagens de programação.** Tradução José Carlos Barbosa dos Santos. Porto Alegre: Bookman, 2000.

SILVEIRA, Fábio F. et al. **Uma abordagem sobre atualização dinâmica em componentes de sistemas orientados a objetos.** São Paulo, 2004. Disponível em: <www.comp.ita.br/~ffs/articles/CACIC_ffs_tonio_cunha_lisboa_2004.pdf>. Acesso em: 01 maio 2006.

SOROKIN, Andrey V. **RegExp studio.** Saint Petersburg, Rússia, 2004. Disponível em: <http://regexpstudio.com/TRegExpr/TRegExpr.html>. Acesso em: 01 mar. 2006.

STEINMACHER, Igor F. **Estudo de caso aplicando programação orientada a aspectos.** 2002. 72 f. Trabalho de Conclusão de Curso (Bacharelado em Ciências da Computação) – Universidade Estadual de Maringá, Maringá.

VAREJÃO, Flávio. **Linguagens de programação:** Java, C e C++ e outras. Rio de Janeiro: Campus, 2004.

VARGAS, Karly S. **Ferramenta para apoio ao ensino de introdução à programação.** 2005. 95 f. Trabalho de Conclusão de Curso (Bacharelado em Ciências da Computação) - Centro de Ciências Exatas e Naturais, Universidade Regional de Blumenau, Blumenau.

APÊNDICE A – Manual da linguagem AOPDelphi

Como o AOPDelphi é uma ferramenta de suporte para a linguagem Delphi, a sintaxe de sua linguagem de aspectos é semelhante a da linguagem Delphi. Um programa AOPDelphi é composto por seu identificador, declarações de variáveis, declarações de *pointcuts* e declarações de *advices*, como mostra o quadro 10.

```
ALog = Aspect
Var
  FId : Integer;
  Str1, Str2 : String;
Implementation
  logInsert : Pointcut = (procedure Arquivo.Insert(*));

  Advice logInsert: Before;
  begin
     ShowMessage('Olá Mundo!');
  EndAdvice;

End.
```

Quadro 10 – Exemplo de programa AOPDelphi

A declaração de variáveis é opcional e as regras são as mesmas de um programa Delphi. A restrição é que é aceito apenas declaração de variáveis de tipos primitivos do Delphi.

Após a palavra reservada `Implementation` inicia-se a declaração de *pointcuts*. O quadro 11 apresenta a sintaxe para declaração de *pointcuts.*

```
<Identificador> : Pointcut = (<tipo_joinpoint> <joinpoint>);
```

Quadro 11 – Sintaxe para declaração de *pointcuts*

<Identificador> é o nome do *pointcut*.

<tipo_joinpoint> representa o tipo de método que será interceptado. Os possíveis valores para <tipo_joinpoint> estão representados no quadro 12.

Valor	Ação
Constructor	Intercepta construtores
Destructor	Intercepta Destrutores
Procedure	Intercepta procedures
Function	Intercepta funções
Exception	Intercepta exceções
*	Funciona como curinga. Seleciona qualquer *joinpoint*, exceto Excption.

Quadro 12 – Possíveis valores para <tipo_joinpoint>

<joinpoint> especifica a classe, método e parâmetros ou a exceção que será interceptada. Se <tipo_joinpoint> for especificado como Exception, <joinpoint> deve ser um identificador de exceção ou *. Sendo informado *, o *pointcut* entrará em ação para qualquer exceção que for executada. Se for especificado um identificador, o *pointcut* terá efeito somente para aquela exceção específica.

No caso dos demais tipos de *joinpoint* há uma maior flexibilidade para especificar <joinpoint>. A especificação completa é composta pelo nome da classe, um ponto (.), nome do método, e se for necessário, os tipos dos parâmetros entre parênteses separados por vírgula. Por exemplo: Arquivo.Insert(Integer, Integer, String). Porém, para flexibilizar é possível fazer referência a várias classes e métodos através de curinga na declaração. Alguns exemplos são mostrados no quadro 13.

Valor	Ação
Arquivo.Insert(Integer)	Intercepta a execução do método Insert da classe Arquivo que tenha um parâmetro do tipo integer.
Arquivo.Insert(*)	Intercepta a execução do método Insert da classe Arquivo independente da quantidade de parâmetros.
*.Insert(*)	Intercepta a execução do método Insert de qualquer classe, independente da quantidade de parâmetros.
Arquivo.* (*)	Intercepta a execução de qualquer método da classe Arquivo.
TCustom*.* (*)	Intercepta a execução de qualquer método de classes cujo nome começam com **TCustom**
TCustom*.Set* (*)	Intercepta a execução de qualquer método cujo nome comece com **Set**, das classes que começam com **TCustom.**

Quadro 13 – Exemplo de valores para <joinpoint>

Ao fazer referência às classes e métodos, o curinga pode aparecer tanto no início, como no meio ou final do identificador. Porém, a linguagem AOPDelphi não permite que seja especificado como *pointcut* a seguinte declaração: `logInsert : Pointcut = (procedure *.*(*));` Dessa forma estaria selecionando todos as classes de todos os métodos, ou seja, todos os métodos do sistema seriam afetados. É obrigatório que a classe ou o método tenha algo especificado.

Após a declaração de *pointcuts* são declarados os *advices.* Todo *pointcut* deve ter um ou dois *advices*. O quadro 14 mostra a sintaxe para definição de um *advice*.

```
Advice <identificador>: <tipo_advice>;
 begin
    <codigo_advice>
 EndAdvice;
```

Quadro 14 – Sintaxe de um *advice*

<identificador> define o identificador do *advice*, e deve ser o mesmo identificador do *pointcut* o qual está associado.

<tipo_advice> pode assumir dois valores: before e after. Os *advices* do tipo before são executados antes da execução do método interceptado, e os do tipo after, depois.

<codigo_advice> define o código na linguagem Delphi que será inserido no método interceptado. Este código não é analisado pelo compilador do programa de aspectos. Em <codigo_advice>, ao ser utilizado o termo self, este fará referência ao objeto que invocou o método interceptado.