**UNIVERSIDADE FEDERAL DE SANTA CATARINA**
**PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO**

**Eduardo Kessler Piveta**

# AURÉLIA: UM MODELO DE SUPORTE A PROGRAMAÇÃO ORIENTADA A ASPECTOS

Dissertação submetida à Universidade Federal de Santa Catarina como parte dos requisitos para a obtenção do grau de Mestre em Ciência da Computação

Luiz Carlos Zancanella

(Orientador)

Florianópolis, junho de 2001

# AURÉLIA: UM MODELO DE SUPORTE A PROGRAMAÇÃO ORIENTADA A ASPECTOS

Eduardo Kessler Piveta

Esta Dissertação foi julgada adequada para a obtenção do título de Mestre em Ciência da Computação Área de Concentração Sistemas de Computação e aprovada em sua forma final pelo Programa de Pós-Graduação em Ciência da Computação.

______________________________

Luiz Carlos Zancanella, Dr.

______________________________

Fernando Ostuni Gauthier, Dr.

Banca Examinadora

______________________________

Luiz Carlos Zancanella, Dr.

______________________________

Ricardo Felipe Custódio, Dr.

______________________________

Carlos José Pereira de Lucena, Dr.

______________________________

Raul Sidnei Wazlawick, Dr.

______________________________

Ricardo Pereira e Silva, Dr.

**SUMÁRIO**

1 INTRODUÇÃO .... 1
1.1 ORGANIZAÇÃO DO TRABALHO .... 2
2 FUNDAMENTAÇÃO TEÓRICA .... 4
2.1 PROGRAMAÇÃO ORIENTADA A ASPECTOS .... 4
*2.1.1 Componentes .... 6*
*2.1.2 Aspectos .... 6*
*2.1.3 Linguagem de componentes .... 8*
*2.1.4 Linguagem de aspectos .... 8*
*2.1.5 Combinador de aspectos .... 9*
*2.1.6 Mecanismos de Composição .... 11*
*2.1.7 Tecnologias de implementação para programação orientada a aspectos 11*
2.2 FERRAMENTAS/LINGUAGENS DE SUPORTE A PROGRAMAÇÃO ORIENTADA A ASPECTOS .... 13
*2.2.1 QIDL .... 13*
*2.2.2 AOP /ST .... 15*
*2.2.3 AspectJ .... 15*
*2.2.4 D .... 17*
*2.2.5 IL .... 20*
*2.2.6* $D^2AL$ *.... 21*
*2.2.7 JST .... 22*
*2.2.8 AspectIX .... 24*
*2.2.9 Outras ferramentas e abordagens .... 25*
2.3 TRABALHOS RELACIONADOS À PROGRAMAÇÃO ORIENTADA A ASPECTOS .... 26
*2.3.1 Programação Orientada ao Sujeito .... 26*
*2.3.2 Demeter/Adaptative programming .... 27*
*2.3.3 Filtros de composição (Composition Filters) .... 27*
*2.3.4 Especificação orientada a aspectos .... 29*
*2.3.5 Outras abordagens .... 30*

3 MODELO DE SUPORTE A PROGRAMAÇÃO ORIENTADA A ASPECTOS ..... 31
3.1 INTRODUÇÃO ..... 31
3.2 ABSTRAÇÕES PARA A REPRESENTAÇÃO DE ASPECTOS ..... 31
3.3 MAPEAMENTO DAS ESTRUTURAS DEFINIDAS/COMBINADAS ..... 33
*3.3.1 Combinando através de herança ..... 33*
*3.3.2 Combinando através de associação ..... 36*
*3.3.3 Combinando através do padrão de projeto Decorator ..... 37*
*3.3.4 Combinando através de interceptação de mensagens ..... 40*
4 AURÉLIA: PROGRAMAÇÃO ORIENTADA A ASPECTOS EM OBJECT PASCAL ..... 43
4.1 COMPONENTES ..... 43
*4.1.1 Classe TAspect ..... 43*
*4.1.2 Classes PointCuts e PointCut ..... 45*
*4.1.3 Classe JoinPoints e JoinPoint ..... 48*
4.2 IMPLEMENTAÇÃO DO COMBINADOR ..... 49
5 APLICATIVO DE EXEMPLO ..... 53
6 CONSIDERAÇÕES FINAIS ..... 57
6.1 TRABALHOS FUTUROS ..... 58
*6.1.1 Problemas encontrados no uso do OPMOP ..... 59*
7 ANEXOS ..... 60
7.1 ARQUIVOS QUE COMPÕE A IMPLEMENTAÇÃO ..... 60
7.2 CÓDIGO FONTE DOS COMPONENTES ..... 60
*7.2.1 AOPSupport.pas ..... 60*
7.3 CÓDIGO FONTE DO COMBINADOR ..... 62
*7.3.1 AOPMetaClass.pas ..... 62*
*7.3.2 AOPExpert.pas ..... 63*
7.4 CÓDIGO UNITS AUXILIARES ..... 64
*7.4.1 AOPAspectizedClasses.pas ..... 64*
*7.4.2 AOPConst; ..... 65*
*7.4.3 AOPAddIn.pas ..... 66*

8 REFERÊNCIAS BIBLIOGRÁFICAS .................................................................... 67

**LISTA DE FIGURAS**

Figura 2.1 - Funcionamento dos filtros de combinação................................................ 28
Figura 3.1 - Diagrama de classes do modelo ............................................................... 32
Figura 3.2– Classe Pessoa sendo afetada pelos aspectos Trace e Sincronize................ 34
Figura 3.3 – Método com comportamento alterado pelo aspecto Trace........................ 34
Figura 3.4 – Possível ordem de herança ...................................................................... 36
Figura 3.5 – Possível ordem de herança ...................................................................... 36
Figura 3.6 - Classe alterada.......................................................................................... 37
Figura 3.7 – Padrão Decorator..................................................................................... 38
Figura 3.8 – Aspectos Trace e Sincronize atuando sobre a classe Pessoa..................... 39
Figura 3.9 –Metaclasse criada pelo combinador .......................................................... 40
Figura 3.10 – Metaclasse com o método interceptado.................................................. 41
Figura 4.1 – Classe Aspect definida no modelo ........................................................... 43
Figura 4.2 – Aspecto em uma aplicação....................................................................... 44
Figura 4.3 – Pontos de atuação em uma aplicação ....................................................... 45
Figura 4.4 – Classes Pointcuts e PointCut.................................................................... 46
Figura 4.5 – Pontos de combinação em uma aplicação ................................................ 48
Figura 4.6 – MOPAssocia............................................................................................. 50
Figura 4.7 – Combinador - Diagrama de Classes ......................................................... 51
Figura 4.8 - Diagrama de seqüência Combinar Projeto................................................. 52
Figura 4.9 – Diagrama de seqüência *WeaveProject* ...................................................... 52
Figura 4.10 – Combinando um projeto.......................................................................... 52
Figura 5.1 - Aplicativo Hello Horld............................................................................... 53
Figura 5.2 – Saída do aplicativo ................................................................................... 54
Figura 5.3 – Ponto de atuação Mensagens..................................................................... 54
Figura 5.4 – Saída do aplicativo – Tela 1 ..................................................................... 55
Figura 5.5 – Saída do aplicativo – Tela 2 ..................................................................... 55
Figura 5.6 – Saída do aplicativo – Tela 3 ..................................................................... 55
Figura 5.7 – Componente MOPAssocia criado pelo combinador ................................ 56

## LISTA DE EXEMPLOS

Exemplo 2.1 - Definição da IDL (BECKER & GEIHS, 1998) .................................... 14
Exemplo 2.2 – Exemplo de aspecto escrito em AspectJ................................................ 16
Exemplo 2.3 - coordenador para o problema do jantar dos filósofos. (LOPES, 1997) .. 19
Exemplo 2.4 – Portal escrito em RIDL (LOPES, 1997)............................................... 20
Exemplo 3.1 – Classe criada pelo combinador............................................................. 34
Exemplo 3.2 – Classe modificada pelo combinador...................................................... 37
Exemplo 4.1 - Classe TAspect (implementação Object Pascal).................................... 43
Exemplo 4.2 - Classe TCustomAspect : implementação Object Pascal......................... 44
Exemplo 4.3 - Classe TPointCuts.................................................................................. 47
Exemplo 4.4 - Classe TPointCut.................................................................................... 47
Exemplo 4.5 – Classe TJoinPoints ................................................................................ 48
Exemplo 4.6 – Classe TJoinPoint.................................................................................. 49
Exemplo 5.1 – Classe TPessoa ...................................................................................... 53
Exemplo 5.2 – Manipulador de eventos do clique do botão.......................................... 53
Exemplo 5.3 – Manipuladores de evento das ações do ponto de atuação ..................... 54
Exemplo 5.4 – Metaclasse criada pelo combinador ...................................................... 56

# RESUMO

Este trabalho propõe um modelo para a implementação de aplicativos utilizando o paradigma de programação orientada a aspectos, através da definição de aspectos genéricos. Este modelo pode ser aplicado a linguagens orientadas a objeto através da implementação das abstrações para a representação de aspectos e de uma ou mais estratégias de implementação do modelo.

Este modelo foi implementado através de uma abordagem reflexiva, a qual influiu decisivamente para os resultados do projeto. As facilidades providas pelo protocolo de metaobjetos utilizado, através do mecanismo de interceptação de mensagens, diminuiu o tempo para a implementação do sistema, bem como reduziu a responsabilidade final do ambiente. O modelo pode ser implementado em outros ambientes de programação, devendo ser adaptado para a geração de código na linguagem escolhida.

# ABSTRACT

This work proposes a model to implement systems using the aspect oriented programming paradigm, through the definition of generic aspects. This model can be applied to object-oriented languages by the implementation of the abstractions for aspects and one or more strategies defined in the model.

This model was implemented using a reflective approach, which decisivelly influenced to the project's results. The facilities provided by the metaobject protocol choosen, through the mechanism of messages' interception, made more easy the task of implementing the system, reducing the final responsabilities of the tool. The model can be implemented in other environments and languages. It must be adapted to generate the code in the choosen language.

# 1 Introdução

A engenharia de software e as linguagens de programação coexistem em um relacionamento de suporte mútuo. A maioria dos processos de projeto de software considera um sistema em unidades cada vez menores. As linguagens de programação, por sua vez, fornecem mecanismos que permitem a definição de abstrações de unidades do sistema e a composição destas de diversas formas possíveis para a produção do sistema como um todo (BECKER & GEIHS, 1998).

Uma linguagem de programação articula-se adequadamente com um projeto de software quando provê abstrações e mecanismos de composição que suportam claramente os tipos de unidades descritas no projeto.

Segundo IRWIN et al. (1997), os mecanismos de abstração das linguagens mais comumente utilizadas (sub-rotinas, procedimentos funções, objetos, classes, APIs) podem ser enquadrados em um modelo de procedimentos generalizados, obtidos através da decomposição funcional do sistema.

Entretanto, existem propriedades que não se enquadram em componentes da decomposição funcional, tais como: tratamento de exceções, restrições de tempo real, distribuição e controle de concorrência. Elas normalmente estão espalhadas em diversos componentes do sistema afetando a performance ou a semântica da aplicação.

Embora elas possam ser visualizadas e analisadas relativamente em separado, sua implementação utilizando linguagens orientadas a objeto ou estruturadas torna-se confusa e seu código encontra-se espalhado através do código da aplicação, dificultando a separação da funcionalidade básica do sistema dessas propriedades.

Este fenômeno de entrelaçamento de código (*code tangling*) é responsável por boa parte da complexidade encontrada nos sistemas atuais. Ele aumenta a dependência entre os componentes funcionais, desviando-os de sua finalidade principal, tornando-os menos reusáveis e mais propensos a erros.

A programação orientada a aspectos é uma abordagem que permite a separação dessas propriedades ortogonais dos componentes funcionais de uma forma natural e concisa, utilizando-se de mecanismos de abstração e de composição para a produção de código executável.

Este trabalho define um modelo de suporte a programação orientada a aspectos em linguagens orientadas a objeto, através de abstrações para a representação de aspectos e de estratégias de implementação de um combinador que torne possível mesclar componentes e aspectos de forma clara e eficiente.

Este modelo tem a intenção de prover subsídios para a definição de aspectos em linguagens orientadas a objetos, onde aspectos genéricos são implementados através de objetos derivados das classes do modelo.

O modelo descrito neste trabalho foi implementado e sua implementação, denominada de Aurélia, é descrita no capítulo 4. Tanto os componentes de suporte quanto o combinador de Aurélia foram desenvolvidos para o ambiente visual de programação Borland Delphi (DELPHI, 2001), versão 5.0.

A escolha deste ambiente deve-se principalmente a escassez de abordagens que utilizam ambientes visuais para a implementação de suporte à programação orientada a aspectos (de fato, apenas a ferramenta AspectJ (ASPECTJ, 2001) possui suporte em Java para os ambientes JBuilder (JBUILDER, 2001) e Forte (FORTE, 2001).

Deve-se ainda à inexistência de suporte a este paradigma em Object Pascal (linguagem utilizada pelo ambiente Delphi), bem como na família de produtos relacionados a este (Borland C++ Builder (BUILDER, 2001), Kylix (KYLIX, 2001)).

Este trabalho tem forte correlação com AspectJ e D (LOPES, 1997) no que tange aos requisitos e fundamentação teórica. AspectJ oferece suporte a definição de aspectos genéricos e D oferece alternativas para a implementação. A ferramenta descrita neste trabalho é influenciada em parte por JST (SENTURIER, 1999), que utiliza metaclasses descritas em OpenJava (CHIBA & TATSUBORI, 1998) para a criação de aspectos e por BÁSTAN & PRYOR (1999), que implementa uma arquitetura reflexiva em Smalltalk de suporte a programação orientada a aspectos.

## 1.1 Organização do trabalho

O trabalho está organizado em 7 capítulos, como descrito a seguir.

O capítulo 2 mostra o estado da arte em programação orientada a aspectos, passando pelos seguintes tópicos:

- Visão geral da programação orientada a aspectos;
- Ferramentas de suporte a programação orientada a aspectos e

- Abordagens relacionadas a programação orientada a aspectos.

No capítulo 3 é definido um modelo de suporte a programação orientada a aspectos, através da definição de abstrações e estratégias de combinação para a implementação;

O capítulo 4 é descreve a ferramenta Aurélia, uma implementação que utiliza o modelo e de uma de suas estratégias de implementação (interceptação de mensagens).

No capítulo 5 é descrito um exemplo de aplicativo escrito utilizando Aurélia e no capítulo 6 são apresentadas considerações finais sobre este trabalho, suas contribuições, vantagens e limitações, bem como propostas de trabalhos futuros ou relacionados a este.

O capítulo 7 contém o código fonte da implementação.

# 2 Fundamentação Teórica

## 2.1 Programação Orientada a Aspectos

Na concepção de LOPES (1997), as aplicações estão ampliando os limites das técnicas de programação atuais, de modo que certas características de um sistema afetam seu comportamento de tal forma que as técnicas atuais não conseguem capturar essas propriedades de forma satisfatória.

Como conseqüência disso, quando essas propriedades precisam ser implementadas, são inseridas no sistema de diversas maneiras, normalmente arbitrárias, que levam a distrações sobre o que um componente deve fazer, dificultando o reuso e comprometendo sua legibilidade.

Isso leva a um aumento da complexidade no que diz respeito à implementação dos componentes funcionais do sistema, os quais tornam-se mais confusos do que deviam. Esse fenômeno é chamado de entrelaçamento de código.

Logo, existem diversos benefícios em se possuir uma propriedade importante de um sistema expressada de maneira bem localizada em uma única unidade ou seção de código. A partir desta separação, pode-se entender com maior facilidade como esta age no sistema como um todo, já que não é necessário procurá-la em diferentes lugares e separá-la de outras propriedades. Ainda, é possível analisá-la, modificá-la, estendê-la e reusá-la mais facilmente (CZARNECKI & EISENECKER, 2000).

Esta separação de conceitos é um princípio fundamental na engenharia de software aplicado na análise, projeto e implementação de sistemas computacionais. No entanto, as técnicas de programação mais comumente utilizadas nem sempre se apresentam satisfatórias no que tange a essa separação.

Existem problemas de programação que as técnicas de programação orientada a objetos ou de programação estruturada não são suficientes para separar claramente todas as decisões de projeto que o programa deve implementar (BECKER & GEIHS, 1998).

Isto se deve ao fato de que as abordagens mais utilizadas concentram-se em encontrar e compor unidades funcionais da decomposição do sistema, enquanto que outras questões importantes não são bem localizadas no projeto funcional.

Como exemplo, pode-se citar propriedades que envolvem várias unidades funcionais, tais como: sincronização, restrições de tempo, concorrência, distribuição de objetos, persistência, etc. Quando duas propriedades sendo programadas devem ser compostas de maneira diferente e ainda coordenarem-se é dito que elas são ortogonais entre si.

IRWIN et al. (1997) define que uma propriedade de um sistema que deve ser implementada pode ser vista como um componente ou como um aspecto:

- A propriedade pode ser vista como um componente se puder ser encapsulada em um procedimento generalizado (objeto, método, procedimento, API). Os componentes tendem a ser unidades da decomposição funcional do sistema. Como exemplos, podem ser citados: contas bancárias, usuários ou mensagens;
- Aspectos não são normalmente unidades da decomposição funcional do sistema, mas sim propriedades que envolvem diversas unidades de um sistema, afetando a semântica dos componentes funcionais sistematicamente. Exemplos de aspectos podem ser: controle de concorrência em operações em uma mesma conta bancária, registro das transações de uma determinada conta, política de segurança de acesso aos usuários de um sistema, restrições de tempo real associadas à entrega de mensagens.

Dada a definição de componentes e aspectos, é possível colocar o objetivo da programação orientada a aspectos: oferecer suporte para o programador na tarefa de separar claramente os componentes dos aspectos, os componentes entre si e os aspectos entre si, utilizando-se de mecanismos que permitam a abstração e composição destas, produzindo o sistema desejado.

A programação orientada a aspectos estende outras técnicas de programação (orientada a objetos, estruturada, etc.) que oferecem suporte apenas para separar componentes entre si abstraindo e compondo-os para a produção do sistema desejado. (IRWIN et. al., 1997)

Uma implementação baseada no paradigma de programação orientada a aspectos é composta normalmente de:

- Uma linguagem de componentes para a programação de componentes;

- Uma ou mais linguagens de aspectos para a programação de aspectos;
- Um combinador de aspectos (*aspect weaver*) para a combinação das linguagens;
- Um programa escrito na linguagem de componentes e
- Um ou mais programas escritos na linguagem de aspectos.

### 2.1.1 Componentes

AKSIT & TEKINERDOGAN (1998b) consideram que os componentes no contexto da programação orientada a aspectos são abstrações providas pela linguagem que permitem a implementação da funcionalidade do sistema. Logo procedimentos, funções, classes, objetos, procedimentos são denominados de componentes em programação orientada a aspectos.

Eles originam-se da decomposição funcional de um sistema, independentes de a linguagem ser orientada a objetos ou estruturada. A programação orientada a aspectos não é restrita ou delimitada pela orientação a objetos, ela estende o modelo.

### 2.1.2 Aspectos

Propriedades de um sistema envolvendo diversos componentes funcionais não podem ser expressas utilizando as notações e linguagens atuais de uma maneira bem localizada (tais como: sincronização, interação entre componentes, distribuição, persistência). Estas propriedades são expressas através de fragmentos de código espalhados por diversos componentes do sistema (CZARNECKI & EISENECKER, 2000).

A seguir tem-se a descrição de algumas propriedades que normalmente são vistas como aspectos em relação à funcionalidade básica da aplicação:

#### 2.1.2.1 Sincronização de Objetos Concorrentes

Um programa em que dados globais sejam compartilhados necessita sincronizar suas atividades através de mecanismos que possibilitem sua execução. Um exemplo de política de sincronização seria que todos os objetos podem realizar operações de leitura em um determinado objeto se ele não estiver sendo modificado, mas somente um pode realizar uma operação de escrita por vez (CAHILL & DEMPSEY, 1997).

#### 2.1.2.2 Distribuição

O projeto e a implementação de sistemas distribuídos requer abordar diversos problemas que não existem em sistemas "não-distribuídos". No projeto, é necessário mudar a perspectiva de como os componentes devem ser especificados, envolvendo o entendimento de diferentes componentes e como eles relacionam-se.

Na implementação, as linguagens de programação atualmente utilizadas deixam a desejar em prover suporte para programar em termos dessas diferentes perspectivas e de propriedades que são ortogonais uma às outras (LOPES, 1997).

#### 2.1.2.3 Tratamento de exceções

Algumas formas de tratamento de exceções são inerentes à definição de um tipo de objeto (por exemplo, tentar retirar um elemento de uma pilha vazia). Entretanto, políticas de tratamento de exceções mais genéricas podem ser vistas e implementadas como aspectos. Como exemplo, pode-se citar o uso de pré-condições e pós-condições que devem ser satisfeitas no momento em que uma operação é realizada. Estas condições podem ser implementadas como aspectos (OSSHER & TARR, 1998).

#### 2.1.2.4 Coordenação de múltiplos objetos

A implementação da coordenação de múltiplos objetos apresenta diversos problemas quanto ao reuso e quanto à extensão. Esta implementação é mais difícil que a primeira vista, devido à necessidade de sincronizar a integração entre os objetos ativos em busca de um objetivo global (HARRISON & OSSHER, 1993).

Estes problemas ocorrem por causa da falta de mecanismos de alto nível no modelo convencional de objetos que permitam abstrair o comportamento coordenado e devido ao algoritmo de coordenação estar espalhado, isto é; distribuído entre conjuntos de objetos dificultando seu entendimento.

Os algoritmos usados para a coordenação de múltiplos objetos são normalmente ligados às implementações, não podendo ser reusados para objetos diferentes. Pode-se dizer ainda que a coordenação é expressa através de mecanismos de sincronização específicos da linguagem de programação, tornando difícil o reuso das classes destes objetos (HARRISON & OSSHER, 1993).

Um exemplo de coordenação pode ser a execução de um vídeo, onde a exibição de imagens, a reprodução de música e a locução são responsabilidade de objetos distintos que necessitam coordenarem-se.

#### 2.1.2.5 Persistência

A persistência é um elemento importante em sistemas computacionais que aparece espalhado através do código do sistema. Uma alternativa seria a implementação desta característica como um aspecto, independente de como e quais são os componentes devem ser tornados persistentes. A partir desta implementação, pode ser feita a composição entre o aspecto e os objetos.

A recuperação de dados é feita no momento em que o objeto é acessado e a atualização da base de dados ocorre no momento da criação ou modificação do estado do objeto (OSSHER & TARR, 1998).

#### 2.1.2.6 Outras propriedades

Existem outras características que podem ser vistas como aspectos: serialização, atomicidade, replicação, segurança, visualização, *logging*, *tracing*, tolerância à falhas, obtenção de métricas, dentre outras (OSSHER & TARR, 1998).

### 2.1.3 Linguagem de componentes

A linguagem de componentes deve permitir ao programador escrever programas que implementem as funcionalidades básicas do sistema, ao mesmo tempo em que não prevêem nada a respeito do que deve ser implementado na linguagem de aspectos (IRWIN et al., 1997).

Embora a programação orientada a aspectos não esteja limitada à orientação a objetos, as linguagens de componentes mais comumente utilizadas são linguagens orientadas a objeto, como Java ou Smalltalk.

### 2.1.4 Linguagem de aspectos

A linguagem de aspectos deve suportar a implementação das propriedades desejadas de forma clara e concisa, fornecendo construções necessárias para que o programador crie estruturas que descrevam o comportamento dos aspectos e definam em que situações eles ocorrem (IRWIN et al.,1997).

BÖLLERT (1999) afirma que existem alguns requisitos que devem ser observados na especificação de uma linguagem de aspectos:

- Sua sintaxe deve ser fortemente relacionada com a da linguagem de componentes, de maneira a facilitar o aprendizado e obter maior aceitação por parte dos desenvolvedores;
- A linguagem deve ser projetada para especificar o aspecto de maneira concisa e compacta. Normalmente as linguagens de aspecto são de mais alto nível de abstração que as linguagens de programação de uso geral e
- Sua gramática deve possuir elementos que permitam ao combinador compor os programas escritos usando as linguagens de aspectos e componentes.

### 2.1.5 Combinador de aspectos

O processo de combinação realizado pelo combinador de aspectos é o produto do cruzamento de um ou mais aspectos com os componentes descritos na linguagem de componentes, e pode ser visto como:

- M x A → R, onde M é um modelo de programação, A é um aspecto, x é o operador de combinação, → é o processo de transformação e R é o resultado (AKSIT & TEKINERDOGAN, 1998a).

A função do combinador de aspectos é processar a linguagem de aspectos e a de componentes, compondo essas linguagens corretamente a fim de produzir a operação geral desejada. Para que isso ocorra, é essencial o conceito de pontos de combinação (*join-points*), que são os elementos da semântica da linguagem de componentes que os programas escritos na linguagem de aspectos coordenam-se. Pontos de combinação podem ser chamadas a métodos, construtores, destrutores, acesso a atributos, chamadas a funções, etc. (IRWIN et. al., 1997)

A representação dos pontos de combinação pode ser gerada em tempo de execução usando um ambiente de execução reflexivo para a programação dos componentes. Neste caso, a linguagem de aspectos é implementada através de metaobjetos, chamados a cada invocação de um método, que utiliza as informações dos pontos de combinação e dos programas de aspectos para combinar os argumentos (KICZALES et al., 1997).

O projeto de um sistema orientado a aspectos requer o entendimento sobre o que deve ser descrito na linguagem de componentes e de aspectos, bem como as características que são compartilhadas entre elas.

Apesar de estas linguagens terem diferentes mecanismos de abstração e composição, elas devem utilizar alguns termos comuns, que possibilitarão ao combinador mesclar os diferentes tipos de programas (IRWIN et al., 1997).

O combinador percorre os programas de aspectos e coleta informações a respeito dos pontos de combinação referenciados pelos programas. A seguir, busca localizar os pontos de coordenação entre as linguagens, combinando o código de maneira a implementar o que é especificado na linguagem de aspectos e componentes. Este processo que é feito manualmente quando usadas técnicas orientadas a objeto ou estruturadas, é automatizado com a sua utilização (BÖLLERT, 1998a).

Um exemplo de uma implementação básica de um combinador é o de um pré-processador que varre o código de componentes à procura dos pontos de combinação e insere as sentenças declaradas no programa de aspectos.

O processo de combinação entre componentes e aspectos pode ser dividido em dois tipos: estático ou dinâmico.

#### 2.1.5.1 Combinação estática

Em um processo de combinação estático, o resultado pode ser comparado em eficiência e performance com o programa escrito sem o uso de técnicas de programação orientada a aspectos. O uso de uma combinação estática previne que um nível adicional de abstração cause um impacto negativo na performance do sistema (BÖLLERT, 1998a).

Uma desvantagem da combinação estática é que a identificação de sentenças relativas a de aspectos em tempo de execução é difícil. Como conseqüência, a adaptação ou modificação de aspectos dinamicamente torna-se trabalhosa.

#### 2.1.5.2 Combinação dinâmica

Para que a combinação possa ser dinâmica é indispensável que os aspectos existam tanto em tempo de compilação quanto em tempo de execução. Através de uma interface reflexiva, o combinador de aspectos tem a possibilidade de adicionar, adaptar e remover aspectos em tempo de execução.

### 2.1.6 Mecanismos de Composição

Herança, parametrização e chamadas de funções são exemplos de mecanismos de composição. Os mecanismos de composição de aspectos, segundo CZARNECKI & EISENECKER (2000) devem permitir (idealmente) os seguintes requisitos:

- **Fraco acoplamento:** Os aspectos devem possuir o mínimo possível de ligações com os componentes; Um fraco acoplamento entre aspectos e componentes é desejável.
- **Adição não invasiva de aspectos ao código existente:** é a capacidade de adaptar um componente ou um aspecto sem modificá-lo manualmente. Deve ser possível expressar mudanças como uma operação aditiva. Mecanismos de composição são utilizados para adicionar mudanças no código existente. O código não é fisicamente modificado, mas a expressão da composição indica que o código original possui uma semântica diferente. Um exemplo é o mecanismo de herança, onde a herança é não invasiva com respeito à sua superclasse.

### 2.1.7 Tecnologias de implementação para programação orientada a aspectos

Prover suporte para aspectos envolve duas etapas: implementar abstrações para expressar aspectos e implementar um combinador para compor o código de aspectos com o de componentes.

#### 2.1.7.1 Abstrações para a representação de aspectos

Para a implementação de abstrações que permitam expressar aspectos de maneira concisa e simples, de acordo com CZARNECKI & EISENECKER (2000), existem duas abordagens principais:

- A primeira delas consiste em codificar o suporte a aspectos como uma biblioteca convencional. Ao utilizar esta abordagem é necessário ainda definir mecanismos de composição que descrevam o tipo de ortogonalidade que o aspecto necessita.

- A segunda é projetar uma linguagem específica para o aspecto, implementada através de um pré-processador, compilador ou interpretador.

Segundo CZARNECKI & EISENECKER (2000), as vantagens de utilizar linguagens especializadas são:

- Representação declarativa, explícita, referente ao aspecto tratado, com termos utilizados no domínio do aspecto;
- As construções de uma linguagem especializada capturam a intenção do desenvolvedor de maneira mais apropriada. Não é necessária a utilização de técnicas específicas para o reconhecimento das abstrações do domínio e
- Permite otimizações e verificação de erros relativas ao domínio da linguagem.

Linguagens especializadas nem sempre são a melhor opção, devido à dificuldade de aceitação por parte do usuário. Nos casos que os aspectos não são específicos de um domínio, as vantagens não são tão aparentes quanto nos casos em que os aspectos tratados são específicos de um domínio como no framework D (LOPES, 1997).

#### 2.1.7.2 Implementação de um Combinador de Aspectos

A combinação de aspectos e componentes envolve transformação de código. Como o código de aspectos é definido separadamente do código de componentes é necessário que estes códigos sejam combinados de maneira a produzir o comportamento especificado.

Esta transformação pode ser realizada através de duas tecnologias: transformação do código-fonte e reflexão dinâmica.

As transformações no código-fonte podem ser implementadas utilizando-se um compilador ou pré-processador. Compiladores e pré-processador normalmente provêm uma interface para a inclusão e edição de nós gerados na árvore de parse. Normalmente o código combinado é rearranjado em tempo de desenvolvimento, estaticamente, de maneira eficiente.

A reflexão em tempo de execução envolve a existência explícita de mecanismos que permitam interferir nas computações que ocorrem nos objetos do nível base.

Utilizando-se destas representações meta é possível modificar o comportamento das entidades do sistema em tempo de execução. (CZARNECKI & EISENECKER, 2000)

## 2.2 Ferramentas/Linguagens de suporte a programação orientada a aspectos

A seguir tem-se a descrição de diversas ferramentas de suporte a programação orientada a aspectos disponíveis na literatura.

### 2.2.1 QIDL

A qualidade de serviço (QoS) aborda o tratamento de propriedades não-funcionais em sistemas distribuídos, sendo originada de problemas inerentes à distribuição que reduzem a transparência do sistema distribuído (tais como: flutuação dinâmica de banda, transmissão de erros, falhas, etc).

Normalmente a qualidade de serviço refere-se a atributos de performance que não são explicitamente visíveis nos serviços definidos na interface entre as partes do sistema. É necessária a existência de mecanismos para a especificação e a gerência da qualidade de serviço de forma ortogonal à associação dos serviços das interfaces.

BECKER & GEIHS (1998) ponderam que a qualidade de serviço pode ser entendida como um aspecto na visão da programação orientada a aspectos, devendo, portanto ser especificada em uma linguagem apropriada, o que pode ser feito juntamente com a especificação do aplicativo e integrada a implementação através de um framework de integração.

Um serviço oferece uma interface funcional para seus clientes, para que garantias de qualidade de serviço sejam asseguradas aos clientes quando o serviço for executado (tais como: restrições de tempo-real, qualidade de som). O gerenciamento destas características requer manipulações tanto no lado cliente quanto do lado servidor. Ambos formam um contrato formado pela qualidade e pela funcionalidade do serviço (BECKER & GEIHS, 1998).

Como as restrições de qualidade de serviço são elementos integrantes de um sistema distribuído, afetando tanto cliente, quanto *middleware*, quanto servidor, elas não podem ser encapsuladas de forma satisfatória através das abstrações providas pelas linguagens orientadas a objetos.

As operações visíveis aos clientes de um serviço onde restrições de qualidade de serviço tenham de ser observadas podem ser encapsuladas na interface de QoS. O estado de uma qualidade de serviço é descrito por um atributo. O aspecto qualidade de serviço é modelado através da descrição de sua interface sem levar em conta sua implementação.

Ao invés de usar um combinador de aspectos os elementos de QoS são gerados em um framework através do compilador IDL. Os aspectos são definidos em IDL e a implementação é feita na própria linguagem de componentes.

QIDL é uma extensão de CORBA que funciona através de definições de qualidade de serviço, que são projetadas para suportar uma grande variedade de restrições diferentes. O compilador QIDL pode auxiliar o usuário a implementar serviços que possuam QoS disponíveis.

#### 2.2.1.1 Definição de qualidade de serviço em QIDL

Uma definição de qualidade de serviço em QIDL permite a especificação de um atributo e de uma interface de QoS, através de uma construção sintática provida por QIDL, conforme o Exemplo 2.1.

```
"qos" QoSname [ ":" baseQoS { "," baseQoS } ]
"{"
{ simple-type-spec QoSParam ";" }
interface
"{"
{ operation-decl ";" }
"}" ";"
"}" ";"
```

**Exemplo 2.1 - Definição da IDL (BECKER & GEIHS, 1998)**

A palavra reservada **qos** introduz uma nova definição de QoS, que contém duas partes: a primeira define os parâmetros que um QoS pode ter (estes parâmetros devem ser de qualquer tipo definido na IDL ou um tipo especificado pelo usuário) e a segunda parte refere-se às operações disponíveis na interface de QoS.

As interfaces definidas na IDL são convertidas para uma linguagem de implementação através de um compilador IDL, mantendo consistentes tanto as

definições de QoS quanto às estruturas e interfaces descritas. Atualmente, o mapeamento é feito apenas de QIDL para C++.

### 2.2.2 AOP /ST

A ferramenta AOP/ST adiciona extensões de suporte a programação orientada a aspectos para Visualworks / Smalltalk. O AOP/ST é composto de um combinador para Smalltalk e duas linguagens de aspectos: uma para a sincronização de processos e outra para o acompanhamento do fluxo de execução de um programa (BÖLLERT, 1998b).

O combinador de AOP/ST é aberto para a integração com outras linguagens de aspectos, bem como possui um conjunto de componentes que possibilitam ao desenvolvedor criar suas próprias linguagens de aspectos.

#### 2.2.2.1 Linguagem de sincronização de processos

Em D (LOPES, 1997), foi apresentada uma linguagem para a sincronização de processos chamada COOL, bem como uma primeira implementação em Java. COOL foi utilizada também como subsídio para o desenvolvimento de AspectJ, e adaptada para tornar possível sua implementação em Smalltalk. Suas idéias foram utilizadas como base da linguagem de sincronização de AOP/ST. (BÖLLERT, 1998b)

#### 2.2.2.2 Linguagem de *tracing*

A linguagem de *tracing* é baseada no conceito de módulos de *tracing* (*tracers*). Um *tracer* especifica quais as mensagens que chegam em um determinado objeto devem ser observadas. Um *tracer* é composto de três partes:

1. Definição;
2. Lista de mensagens a “rastrear” e
3. Código personalizado para as mensagens a serem observadas.

Os métodos referenciados em um *tracer* são chamados de *traced methods.*

### 2.2.3 AspectJ

AspectJ (ASPECTJ, 2001) é uma extensão de Java para suporte a programação orientada a aspectos de maneira genérica.

Sua implementação é aberta (*Open Source*) desde a versão 0.7b4, possuindo suporte a ambientes integrados de desenvolvimento (Emacs, Jbuilder 3.5, Forte 4J). O

projeto da linguagem é dirigido através da resposta dos usuários em relação ao ambiente. AspectJ está atualmente na versão 0.8b4.

É a linguagem de suporte à programação orientada a aspectos mais conhecida, tendo sua origem baseada nas idéias do framework D (LOPES, 1997).

O combinador de aspectos é implementado como um pré-processador que recebe como entrada o(s) programa(s) de componentes e o(s) programa(s) de aspectos e gera arquivos Java que podem ser compilados normalmente.

Java é a linguagem de componentes utilizada pelo ambiente AspectJ. A linguagem de aspectos é genérica, possibilitando ao desenvolvedor especificar instruções nos seguintes pontos de combinação:

- Execução de métodos;
- Recebimento de chamadas a construtores;
- Execução de construtores;
- Acesso a campos e
- Execução de manipuladores de exceção.

A seguir, no Exemplo 2.2, tem-se um aspecto denominado *AutoReset* que atua em todos os objetos da classe *Point*. Sempre que ocorrer uma chamada ao método *set* ou *setX*, ocorre um teste relativo ao número de chamadas a esses métodos. Quando o número chega a 100, o contador é zerado e o objeto da classe *Point* é colocado na posição (0,0).

```
aspect AutoReset of eachobject(instanceof(Point)){
        int count=0;
        pointcut setters(Point p): instanceof(p) & receptions(void setY(..) | void setX(...) );
        after(Point p): setters(p){
            if (++count>=100){
                count = 0;
                p.setx(0);
                p.sety(0);
            }
        }
}
```

**Exemplo 2.2 – Exemplo de aspecto escrito em AspectJ**

### 2.2.4 D

As linguagens de programação orientadas a objeto possuem capacidades poderosas para expressar a funcionalidade básica dos componentes de um sistema, entretanto essas linguagens possuem poucas capacidades que permitam ao desenvolvedor capturar o comportamento dos componentes quando eles estão distribuídos através de uma rede ou quando a modularidade de seu comportamento seqüencial é quebrada pela execução concorrente dos serviços disponíveis no sistema.

A distribuição de objetos e a programação concorrente normalmente são ortogonais à funcionalidade básica dos componentes, afetando-os de maneira sistemática. Quando são programadas utilizando linguagens orientadas a objeto, tendem a produzir um código entrelaçado, onde a funcionalidade dos componentes mistura-se com as sentenças relativas à sincronização e a distribuição de objetos, tornando os componentes menos reusáveis e mais difíceis de entender e manter. (LOPES, 1997)

D é um framework que fornece abstrações para a implementação de esquemas de sincronização e transferência remota de dados em sistemas distribuídos. Para isso, fornece duas linguagens para a descrição destes aspectos: COOL e RIDL. Elas podem ser integradas a linguagens orientadas a objeto com poucas ou nenhuma alteração na linguagem escolhida.

Os aspectos descritos em D afetam o comportamento distribuído e concorrente dos componentes, possibilitando ao desenvolvedor preocupar-se primeiramente com a funcionalidade dos componentes e a seguir, de uma maneira explícita, descrever como os aspectos afetam os componentes de um modo bem localizado e relativamente separado do restante da aplicação.

A partir de D foi implementado um protótipo usando Java como linguagem de componentes. Esta implementação foi chamada de DJ.

#### 2.2.4.1 COOL

Esta linguagem provê meios para especificar mecanismos de exclusão mútua em *threads* concorrentes, sincronização, notificação e comandos de guarda de maneira separada do código referente às classes do sistema. Um programa escrito em COOL consiste em uma coleção de módulos de coordenação.

Os coordenadores são associados a classes. Cada coordenador pode estar associado a uma ou mais classes, tendo a responsabilidade da sincronização de *threads* através da execução dos métodos das classes. Eles são associados aos exemplares das classes no momento em que os exemplares são criados.

Como os coordenadores não são classes, eles não podem ser instanciados. São mecanismos que servem para um propósito específico.

LOPES (1997) define que o protocolo entre um objeto e seu(s) coordenador(es) funciona através da execução de 8 passos:

1. Dentro de uma *thread* T, ocorre uma invocação ao método m do objeto X;
2. A mensagem é interceptada e enviada ao coordenador do objeto;
3. O coordenador verifica restrições de exclusão e pré-condições para o método m. Se alguma das restrições não é atendida, T é suspensa. No momento em que todas as restrições são atendidas, o coordenador executa as sentenças definidas no bloco on_entry para o método m.
4. A requisição é enviada para X;
5. O método m é executado pela *thread* T;
6. Quando a invocação retornar, o resultado é apresentado ao coordenador;
7. O coordenador executa as sentenças definidas no bloco on_exit para o método m e
8. A invocação do método retorna.

Um exemplo de um coordenador pode ser visualizado no Exemplo 2.3. O exemplo contém um coordenador para o problema do jantar dos filósofos.

```
per_class coordinator Philosopher {
condition OKToEat[] = {true, true, true, true, true};
boolean eating[] = {false, false, false, false, false};
eat: requires OKToEat[mynumber];
on_entry {
OKToEat[(mynumber+1) % max] = false;
OKToEat[(mynumber-1) % max] = false;
eating[mynumber] = true;
}
on_exit {
if (eating[(mynumber+2) % max] == false)
    OKToEat[(mynumber+1) % max] = true;
```

```
if (eating[(mynumber-2) % max] == false)
   OKToEat[(mynumber-1) % max] = true;
eating[mynumber] = false;
}
}
```

**Exemplo 2.3 - coordenador para o problema do jantar dos filósofos. (LOPES, 1997)**

### 2.2.4.2 RIDL

A linguagem RIDL disponibiliza mecanismos para a manipulação de dados entre diferentes espaços de execução separadamente dos componentes funcionais.

O módulo principal de um programa em RIDL é chamado de portal. Para cada classe pode haver no máximo um portal. As declarações especificadas em um portal identificam classes que podem ser referenciadas em chamadas remotas.

Os exemplares destas classes são chamados de objetos remotos, e suas operações que podem ser invocadas remotamente são especificadas através do portal e são denominadas de operações remotas.

Um portal é automaticamente associado a um exemplar de uma classe quando a referência a este objeto for exportada para fora do espaço de execução em que o objeto foi criado.

Conforme LOPES (1997), o protocolo entre um objeto e seu portal funciona através da execução de 8 passos:

1. De algum espaço de endereçamento remoto ocorre uma invocação ao método m do objeto X;
2. A requisição é encaminhada para o portal do objeto;
3. O método é processado de acordo com sua declaração no portal, onde os parâmetros são passados do espaço remoto para o espaço local de acordo com a especificação do portal;
4. A requisição é enviada para o objeto;
5. O método m é executado;
6. Quando a invocação retorna, o retorno é apresentado ao portal do objeto;
7. O valor de retorno é processado de acordo com sua declaração de operação remota e

8. A invocação do método retorna e seu valor de retorno é enviado para o espaço remoto.

Os portais relacionam-se com as classes através da definição de um novo portal. Cada portal é nomeado com o nome da classe à qual ele refere-se. No Exemplo 2.4, pode-se visualizar dois portais, um para a classe *BookLocator* e um para a classe *ProjectManager*.

```
portal BookLocator {
void register(Book b, Location l);
void unregister(Book b);
Location locate(Book b);
default:
Book : copy { Book only all.String, all.int; };
}
portal ProjectManager {
boolean newBook(Book b, Price p) {
        Book : copy { Book only owner, all.String, all.int; };
}
}
```

**Exemplo 2.4 – Portal escrito em RIDL (LOPES, 1997)**

### 2.2.5 IL

IL é uma linguagem de aspectos para descrever as interações entre objetos através de suas interfaces. Utilizando-se IL, independente da linguagem de componentes utilizada, as classes e seus comportamentos são escritos na linguagem de componentes sem precisar se preocupar com interações (BERGER et al., 1998).

Em IL, elas são especificadas fora do objeto, e a descrição das interações dos objetos é feita através de regras de interação, que definem (para uma mensagem qualquer) um conjunto de mensagem parcialmente ordenado, que correspondem ao fluxo de execução.

Estas regras de interação são descritas mediante o uso de um gerente de interação. O gerente permite especificar o estado e os comportamentos contidos na

interação projetada. Ao ser especializado, um gerente adiciona novas regras ou complementa aquelas já existentes.

Em IL, os gerentes podem ser declarados estaticamente entre dois objetos ou dinamicamente, podendo estabelecer ou destruir interações entre objetos em tempo de execução.

Para descrever as regras e os gerentes de interação um ambiente de programação é proporcionado juntamente com a linguagem. Este ambiente, desenvolvido em Smalltalk possui três geradores: um para Smalltalk, um para CORBA e um para C++ e pode ser visto como um editor de interações.

A linguagem IL foi "experimentada" em algumas linguagens de componentes como: Smalltalk, OpenC++, CORBA-C++, CORBA-OpenC++. Estão sendo realizados estudos para a integração de IL em linguagens de componentes que integram gerenciamento de redes, como: Java RMI e Corba-Smalltalk.

Dependendo do tipo da linguagem de componentes, o combinador de aspectos deve trabalhar com diferentes paradigmas, portanto para cada linguagem de componentes existe um combinador associado. Entretanto, pode-se visualizar que os combinadores têm características comuns, gerando classes para as regras de interação e para os gerentes como descrito na especificação IL.

Após a geração das classes básicas, o combinador adiciona o código necessário para associar os gerentes aos objetos e adiciona um conjunto de unidades que controlam as interações. Este conjunto pode ser visto como uma biblioteca de tempo de execução.

No código resultante, a cada chamada de mensagens, é determinada se a mesma é passível de interações, se sim, um novo comportamento é chamado para adequar-se às interações descritas na linguagem de aspectos.

### 2.2.6 D$^2$AL

Algumas linguagens e arquiteturas como Emerald (BLACK et al., 1988) e CORBA (OMG, 1998) fornecem mecanismos para o desenvolvedor controlar propriedades (replicação de objetos, migração, dentre outras) através da adição de sentenças ou operações nas linguagens de programação utilizadas para a descrição da funcionalidade básica do sistema.

Essa abordagem leva a problemas devido à mistura de código relativo a funcionalidade básica com código relativo a distribuição. Por exemplo, pode-se citar o fato que ao especializar uma classe herda-se tanto as características funcionais da mesma quanto às características relativas à distribuição, limitando o reuso destas estruturas.

Segundo BECKER (1998), $D^2AL$ (*Design-Based Distribution Aspect Language*) foi desenvolvida para a especificação de distribuição através de uma linguagem de aspectos, permitindo ao desenvolvedor separar o que diz respeito à distribuição do que faz parte da funcionalidade básica do sistema.

A diferença fundamental entre $D^2AL$ e outras linguagens de aspectos é que as construções das linguagens referem-se ao projeto e não a implementação da funcionalidade do sistema.

O uso do projeto do sistema como base da linguagem aumenta a expressividade da linguagem. Porém, existem alguns problemas relacionados a como o combinador deve ser implementado, de modo a mapear as construções de projeto para a implementação. (BECKER, 1998)

O combinador de aspectos da proposta de $D^2AL$ funciona como um pré-processador que transforma o código Java transparente de distribuição e a especificação $D^2AL$ em código Java.

BECKER (1998) enfatiza que para que esta operação seja bem sucedida, é indispensável que o ambiente em tempo de execução seja capaz de acompanhar os eventos que são relevantes para a distribuição dos objetos, como por exemplo, uma mudança de estado.

Em cada local no código fonte onde um destes eventos possa ocorrer, o combinador inclui as sentenças necessárias para manter a transparência de localização, por exemplo, conforme especificado na linguagem de aspectos.

### 2.2.7 JST

A linguagem JST introduz uma linguagem de aspectos para Java que permite ao desenvolvedor expressar sincronização de objetos separadamente do código de componentes, sendo baseada em uma linguagem de transição de estados. A idéia básica

é definir em uma classe de sincronização todos os estados que um objeto da classe pode estar e quais os métodos que podem ser executados neste estado. (SENTURIER, 1999)

Os benefícios de utilizar JST são:

- Separar o código de sincronização da funcionalidade básica do sistema;
- Prover uma linguagem onde os estados e transições são elementos sintáticos do sistema e
- Prover uma abordagem semelhante a diagramas de estados para escrever classes de sincronização.

Em JST, as classes de sincronização são chamadas de comportamentos, que implementam uma interface Java e podem estender outros comportamentos. Cada comportamento define alguns estados, onde o estado inicial é o de criação. Um estado pode ser um estado básico ou particionado. No caso de ser particionado, devem ser descritos seus sub-estados, se classificados como básico, a descrição de transições é necessária.

Uma transição é composta de uma guarda (descrita por uma condição booleana e o nome do método a esperar) e de instruções.

Durante a execução de um aplicativo, um ou mais estados de um comportamento JST podem estar ativos. São definidos na linguagem seis tipos de estados para controlar a maneira como ocorre a entrada e a saída de cada estado. O tipo de cada um deles é dividido em dois elementos: um para entrar no estado (*null ou join*) e outro para sair do estado (*sequential, parallel ou server*).

O mecanismo de herança em JST funciona de maneira similar a herança de classes, onde um comportamento filho herda todos os estados e todas as transições do comportamento pai, e quando herdados, estados e transições podem ser redefinidos.

Em JST, os comportamentos são compostos de maneira transparente através do combinador da linguagem. Ele é implementado em C++ e funciona da seguinte maneira: dado um comportamento em JST é gerada uma meta-classe em OpenJava (CHIBA & TATSUBORI, 1998) que é conectada ao código da classe base.

O uso de uma linguagem reflexiva para a implementação do combinador possui vantagens, na medida em que diversas características necessitariam ser implementadas caso não fosse utilizada uma linguagem desse tipo.

### 2.2.8 AspectIX

Sistemas distribuídos baseados no conceito de *middleware*, como CORBA e DCE (OPEN GROUP, 2001) possuem as características básicas para a implementação de aplicações distribuídas. No entanto, as propriedades não-funcionais são normalmente descritas através de mecanismos de adição ou então não são descritas.

Para aspectos que envolvem conceitos de programação distribuída, o combinador deve gerar código que interage com o *middleware* para que as propriedades do sistema correspondam à sua especificação, entretanto isto o torna muito dependente do *middleware* e da linguagem de programação utilizada.

Segundo BECKER et al. (1998), AspectIX é uma arquitetura que possibilita a implementação e o controle de propriedades não funcionais de objetos distribuídos, proporcionando uma interface aberta e genérica para a especificação destas propriedades. A arquitetura permite que a implementação de objetos distribuídos seja modificada em tempo de execução para permitir aos objetos refletir as mudanças na configuração dos aspectos.

De um ponto de vista externo, uma implementação AspectIX é parecida com uma descrição CORBA. Entretanto a arquitetura AspectIX adota um modelo de objetos fragmentados semelhante a encontrado em INRIA (GOURHANT et al., 1994) e Globe (HOMBURG et al, 1997).

Em AspectIX, um objeto distribuído é composto de vários “fragmentos” que interagem entre si. Um fragmento pode ser um *stub* do lado cliente, um objeto servidor (o qual contém a funcionalidade básica do objeto), etc. Estes fragmentos podem se comunicar de modo a realizar as tarefas da aplicação.

Os aspectos podem ser ativados ou desativados, bem como pode ter seus parâmetros modificados em tempo de execução. Ao serem modificadas as configurações de um aspecto, o fragmento é substituído por uma implementação mais atual.

O *middleware* AspectIX é aberto para a inclusão de novas implementações ou novos aspectos. Ele estende CORBA provendo uma interface genérica e aberta para controlar e implementar aspectos não funcionais de objetos distribuídos. As extensões de AspectIX estão sendo definidas para CORBA e apenas um mapeamento para Java está disponível. (BECKER et al., 1998)

### 2.2.9 Outras ferramentas e abordagens

Existem ainda diversas outras abordagens e ferramentas que utilizam os conceitos da programação orientada a aspectos ou são abordagens relacionadas.

Dentre essas ferramentas podemos citar:

- **HYPERJ**: implementa os conceitos de *Hyperspaces* (OSSHER & TARR, 1999);
- **Syncronization rings model**: abordagem para a sincronização de objetos concorrentes (HOLMES, et al., 1997);
- **Piccola**: abordagem baseada em Cálculo Pi (ACHERMANN & NIERSTRASZ);
- **DCOOL**: versão em Smalltalk de COOL (CZARNECKI, 2000);
- **Distributed OZ**: trata de sistemas distribuídos e mobilidade (BRAND et al., 1997);
- **QuO**: provê um framework para a integração de restrições de QoS genéricas em CORBA (BAKKEN et al., 1997);
- **Composer Tool Suite**: ferramentas para suporte a especificação orientada a aspectos (BLAIR L. & BLAIR G, 1999);
- **Moderator Framework:** abordagem utilizando um framework para suporte à programação orientada a aspectos para tratar de sistemas concorrentes (BADER et al., 1999);
- **Aspect language for robust programming**: linguagem de aspectos para tratamento de exceções, controle de invariantes, etc (FRADET & SÜDHOLT, 1999);
- **Fragment System in BETA**: explica como utilizar uma ferramenta existente no ambiente de programação BETA para suportar os conceitos de programação orientada a aspectos (KNUDSEN, 1999).
- **Arquitetura reflexiva em Smalltalk**: arquitetura reflexiva para suporte a programação orientada a aspectos em Smalltalk. (BASTÁN & PRYOR, 1999). Esta arquitetura reflexiva utiliza-se do framework Luthier (CAMPO & PRICE, 1998).

## 2.3 Trabalhos Relacionados à Programação Orientada a Aspectos

Existem diversas abordagens que se concentram em encapsular propriedades que são ortogonais às unidades funcionais do sistema. Estas abordagens, na visão de CZARNECKI & EISENECKER (2000), estendem o modelo de programação orientado a objeto, fornecendo mecanismos para a descrição de propriedades que são ortogonais a funcionalidade básica do sistema.

### 2.3.1 Programação Orientada ao Sujeito

Conforme HARRISON & OSSHER (1993), a programação orientada ao sujeito (*Subject-Oriented Programming*) tem o enfoque baseado em dois pontos principais:

1. Facilitar a identificação e descrição de conceitos ortogonais (*crosscuting concerns*) em sistemas de software;
2. Facilitar a identificação e integração dos pontos de combinação de um sistema. Os pontos de combinação são as localizações em um sistema as quais são afetadas por mais de um conceito (conceitos ortogonais) ao mesmo tempo. O processo de integração descreve como um conceito ortogonal afeta o código em um ou mais pontos de combinação.

A programação orientada ao sujeito foi proposta como uma extensão ao modelo orientado a objetos para tentar solucionar o problema da manipulação de diferentes perspectivas para objetos a serem modelados. (CZARNECKI & EISENECKER, 2000)

É possível visualizar como exemplo um objeto representando um livro. Para o setor de marketing, o livro contém atributos como: assunto (classificação), resumo, autor. Para o setor de produção, os atributos relevantes são, tipo de papel, formato, etc. Outro uso ocorre quando é necessário integrar sistemas desenvolvidos que operam de maneira independente.

O objetivo é ser capaz de adicionar extensões até antes não visualizadas de maneira não invasiva. Nesta abordagem, cada perspectiva é chamada de sujeito. Um sujeito é uma coleção de classes ou fragmentos relacionados por herança ou outros relacionamentos contidos no sujeito. Um sujeito é um modelo completo ou parcial de objetos, que são compostos através de regras de composição.

IRWIN et al. (1997) observam que enquanto aspectos tendem a ser propriedades que afetam a performance ou a semântica dos componentes, sujeitos tendem a ser

características adicionais inseridas em outros sujeitos. A programação orientada a sujeitos é complementar e compatível com a programação orientada a aspectos.

### 2.3.2 Demeter/Adaptative programming

Conforme CZARNECKI & EISENECKER (2000), a idéia original da abordagem Demeter/Adaptative programming é promover uma melhor separação entre o comportamento e a estrutura dos objetos segundo o paradigma de orientação a objetos. A abordagem é motivada pela observação que programas orientados a objeto normalmente contém diversos métodos que não fazem nada, ou fazem muito pouca coisa, e chamam outros métodos passando informações de uma parte do diagrama para outra parte do diagrama.

Esta abordagem tenta minimizar o problema, tentando evitar longas seqüências de acesso a métodos, como por exemplo: *object1.part1.part2.part3();*, o que liga muitas estruturas à um comportamento ou a um objeto, onde na verdade nem todas os objetos referenciados são necessários para a resolução do problema.

Como exemplo, para computar o total de salários pagos por uma empresa, seria necessário somar o salário de todos os funcionários. Podem existir diversas classes que são diretamente ligadas aos funcionários e à empresa, mas que não fazem parte da computação. Ainda assim são necessárias porque são referenciadas pelo funcionário, por exemplo. (Ex.: Empresa→ Divisão → Departamento → Funcionário → Salário)

A abordagem Demeter/Adaptative programming tenta resolver este problema através de uma especificação parcial somente das classes envolvidas no problema juntamente com uma descrição comportamental. Estas especificações parciais são chamadas de estratégias transversais. Um exemplo para o cálculo do salário seria: *from Empresa to Salario*.

### 2.3.3 Filtros de composição (*Composition Filters*)

Filtros de composição é uma técnica de programação orientada a aspectos onde diferentes aspectos são expressos em filtros, de maneira declarativa, juntamente com especificações para a transformação de mensagens. Esta abordagem permite ao desenvolvedor expressar aspectos de uma forma geral através do uso de filtros e possibilita sua composição sem a necessidade da construção de geradores específicos.

Os filtros de composição podem ser anexados a objetos de diferentes linguagens orientadas a objeto. (AKSIT & TEKINERDOGAN, 1998b)

A expressividade das linguagens orientadas a objeto pode ser ampliada através de extensões modulares independentes que afetam a semântica das mensagens sem a necessidade da modificação do modelo. O uso de extensões ortogonais ao modelo pode aprimorar a clareza dos sistemas desenvolvidos sob este paradigma sem prejuízos.

Nesta abordagem as mensagens que chegam a um objeto são avaliadas e, se necessário, manipuladas pelos filtros que atuam sobre aquele objeto. Estes filtros podem ser anexados a linguagens orientadas a objeto, sem a necessidade de modificá-las.

Os filtros são definidos em uma determinada ordem, as mensagens que chegam ao objeto são interceptadas, passam pelos filtros e após são descartadas, ativadas pelo objeto ou delegadas para outro objeto. Neste processo, cada filtro pode aceitar ou rejeitar uma mensagem dependendo da semântica associada com o tipo do filtro. Uma visão geral do funcionamento dos filtros de composição pode ser vista na Figura 2.1, no qual mensagens que chegam a um objeto passam através dos filtros se as condições definidas forem satisfeitas ou são rejeitadas caso contrário.

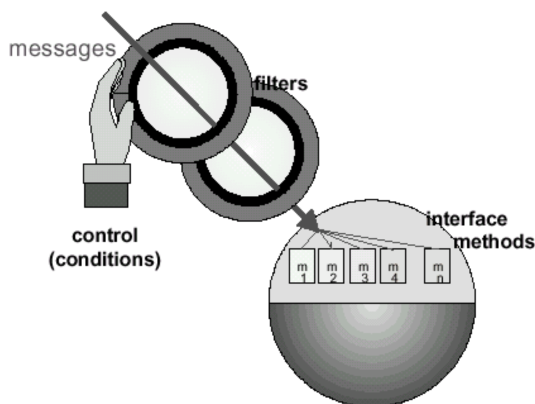


**Figura 2.1 - Funcionamento dos filtros de combinação.**

IRWIN et al. (1997) afirma que a linguagem de componentes utilizada nesta abordagem é uma linguagem orientada a objetos e a linguagem de aspectos é o mecanismo de filtros de composição. A maioria do processo de combinação ocorre em tempo de execução. Nesta abordagem os pontos de combinação são o envio e recebimento de mensagens por parte de um objeto.

### 2.3.4 Especificação orientada a aspectos

Existem muitas técnicas diferentes de especificação formal, as quais exibem uma grande diversidade no que diz respeito ao estilo da linguagem e ao domínio da aplicação. Elas podem ser agrupadas em diversas categorias como álgebra de processos, autômatos, redes de petri, etc. (BLAIR L. & BLAIR G., 1999)

Extensões para estas linguagens estão constantemente sendo desenvolvidas, normalmente abordando uma falha de muitas técnicas de especificação: enquanto elas são muito eficientes para tratar do comportamento funcional do sistema, poucas têm a habilidade de tratar de aspectos não funcionais do sistema.

Por exemplo, muitas extensões abordam a especificação de sistemas de tempo real, comportamentos híbridos ou probabilísticos, mas o comportamento funcional (qualitativo) e o comportamento quantitativo tornam-se entrelaçados na especificação.

Para problemas pequenos, isto pode não ser um problema significativo, mas para problemas mais complexos estes entrelaçamentos de diferentes aspectos tornam a especificação difícil de desenvolver, entender e manter.

Existem, nesta área, estudos onde os aspectos são divididos em três classes: funcionais, de interação e não-funcionais (restrições de tempo, tolerância à falhas, concorrência, etc.) (BIDAN & ISSARNYE, 1996).

Existe a distinção entre o comportamento funcional (qualitativo) e o comportamento quantitativo do sistema. O quantitativo é visto como um aspecto na especificação enquanto que o quantitativo engloba restrições e políticas de gerenciamento. Para cada aspecto pode ser utilizada a linguagem de especificação que mais se adapte ao modelo.

Em BLAIR L. & BLAIR G. (1998), é utilizada a ferramenta LOTOS para uma especificação abstrata do comportamento funcional e lógica temporal e autômatos para a especificação de aspectos quantitativos. Outros trabalhos, como (JACKSON & ZAVE 1993 e JACKSON & ZAVE 1996) consideram o uso da linguagem Z juntamente com autômatos, redes de petri e gramáticas formais.

No caso de uma especificação orientada a aspectos é necessária a presença de um elemento que faça o equivalente ao papel do combinador de aspectos, mesclando o comportamento do programa de componentes e de aspectos.

No que tange a especificações formais uma característica imprescindível é a semântica: ela deve ser clara, concisa e não ambígua.

Para uma técnica de especificação orientada a aspectos o projetista deve ainda se preocupar com a semântica das composições entre os aspectos do sistema. Em BLAIR et al. (1996) o resultado da composição das especificações é um modelo global baseado em autômatos.

A validação do modelo pode ser verificada utilizando-se uma das técnicas a seguir:

- Prova formal demonstrando que os requisitos de usuário satisfazem a especificação como um todo; (BLAIR et al., 1996)
- Análise de performance no modelo global gerado usando uma ferramenta de simulação;

Existem diversas vantagens no uso de uma especificação orientada a aspectos, entretanto existem também desvantagens que devem ser analisadas. A mais significativa destas é a complexidade envolvida no processo de composição e validação da especificação, bem como a possibilidade de inconsistências ou contradições à medida que aumentam as ferramentas de especificação e os aspectos envolvidos. Mecanismos que verificam inconsistências devem ser incluídos no processo de validação da especificação. (BLAIR L. & BLAIR G., 1998)

### 2.3.5 Outras abordagens

Existem outras abordagens semelhantes ou relacionadas com a programação orientada a aspectos, dentre elas podemos citar o conceito de Hiper-espaços (OSSHER & TARR, 1999), Monads (DE MEUTER, 1997) e Generative Programming (CZARNECKI & EISENECKER, 2000).

# 3 Modelo de suporte a programação orientada a aspectos

## 3.1 Introdução

O modelo visa definir abstrações e mecanismos de composição que ofereçam suporte a programação orientada a aspectos de maneira genérica, ou seja, os aspectos tratados não são aspectos específicos de um domínio.

Ele baseia-se principalmente nas idéias de AspectJ e D. A primeira ferramenta foi tomada como parâmetro devido a ser um modelo de suporte genérico para a programação de aspectos. A segunda, devido a teoria relativa à implementação tanto do combinador quanto das classes resultantes.

A proposta é oferecer suporte através de componentes visuais para a implementam de aspectos, bem como oferecer alternativas para a implementação do modelo proposto e do combinador.

## 3.2 Abstrações para a representação de aspectos

O modelo consiste em um conjunto de classes inter-relacionadas que fornecem mecanismos para representar aspectos em uma aplicação. Objetos destas classes são referenciados e utilizados da mesma forma que outros objetos no sistema, com a particularidade que não são feitas chamadas para métodos ou propriedades destes pelos componentes da aplicação.

Um aspecto pode estar presente em diversas classes, podendo atuar em diversos eventos do sistema. Ao ser inserido na aplicação, é necessária sua associação com as classes com os quais ele interage.

Cada aspecto possui diversos pontos de atuação (*pointcuts*) que determinam um conjunto de situações (pontos de combinação) e um conjunto de ações que devem ser tomadas no decorrer de cada situação.

Os pontos de combinação são definidos através do nome da classe e do nome do método a serem afetados pelo aspecto. Na Figura 3.1, pode-se visualizar o diagrama de classes do modelo.

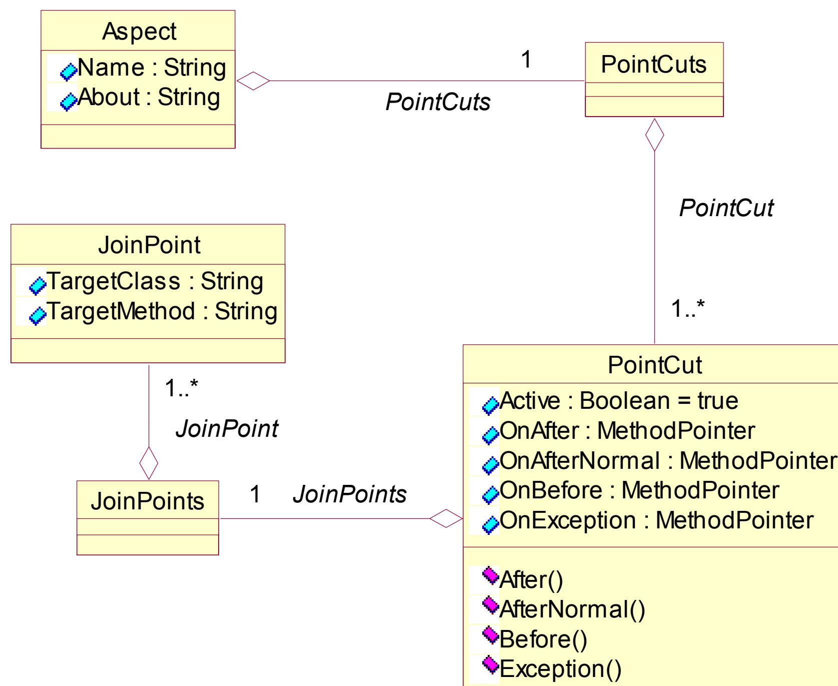


**Figura 3.1 - Diagrama de classes do modelo**

No modelo, cada aspecto é definido como um exemplar de uma classe chamada *Aspect* ou de uma de suas subclasses. Nesse contexto, é possível incluir um exemplar da classe *Aspect* para cada propriedade ortogonal à funcionalidade básica do sistema.

Como cada aspecto pode conter diferentes ações que ocorrem de maneira relacionada, cada aspecto é dividido em pontos de atuação. Cada ponto de atuação corresponde a uma ação que pode ocorrer diversas vezes em componentes diferentes, mas que pode ser generalizada a ponto de se enquadrar em diversas classes.

Como exemplo pode ser visualizado um aspecto segurança, no qual pontos de atuação diferentes poderiam ser: controle de acesso, *logging*, restrições de tempo, etc.

Um ponto de atuação é um exemplar da classe *PointCut*, onde *JoinPoints* é uma propriedade que contém uma lista de classes e métodos que são afetados pelo ponto de atuação.

Essa lista é um exemplar da classe *JoinPoints* que contém diversos pontos de combinação, os quais são os pontos onde a linguagem de aspectos e a linguagem de componentes coordenam-se.

No modelo é possível definir pontos de combinação nos seguintes eventos:

- Chamadas a métodos;

- Chamadas a construtores e
- Chamadas a destrutores.

Em cada um desses pontos de atuação é possível inserir sentenças relativas a implementação dos aspectos desejados. Essas sentenças são inseridas nas seguintes ocasiões:

- Antes dos pontos de combinação;
- Após os pontos de combinação, em um retorno normal do método;
- Após os pontos de combinação, independente do sucesso do retorno do método e
- Na ocorrência de exceções.

De acordo com essas características, um aspecto pode ter diversos pontos de atuação. Cada ponto de atuação pode conter diversos pontos de combinação, que são exemplares da classe *JoinPoint* contendo o atributos *TargetClass* e *TargetMethod*, que são respectivamente o nome da classe afetada e o nome do método afetado pelo aspecto.

Em um ponto de atuação são definidas as instruções relativas a cada evento em que o aspecto é ativado.

## 3.3 Mapeamento das estruturas definidas/combinadas

O combinador funciona através da inclusão de novas classes, novos métodos ou modificação de classes ou métodos existentes.

### 3.3.1 Combinando através de herança

O mecanismo de herança pode ser utilizado para implementar o comportamento descrito nos aspectos. Se um aspecto afeta uma classe em um ou mais pontos, deve ser gerada uma subclasse que efetue o comportamento modificado.

O código-fonte das classes afetadas não é modificado e o código dos aspectos é inserido nas subclasses criadas. No Figura 3.2, pode ser visualizada a classe Pessoa sendo afetada pelos aspectos *Trace* e *Sincronize*. Os métodos afetados são sobrescritos pelo método combinado, conforme o Exemplo 3.1. O método combinado encapsula o método herdado, executando as sentenças relativas aos aspectos, como pode ser visualizado na Figura 3.3.

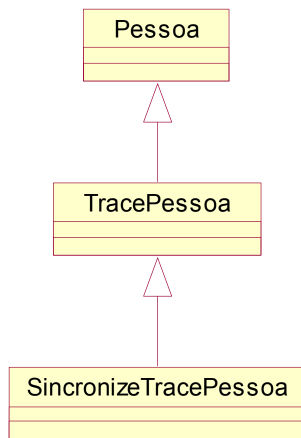


**Figura 3.2– Classe Pessoa sendo afetada pelos aspectos Trace e Sincronize**

```
⇒ TTracePessoa = class(TPessoa)
⇒   GetIdade:integer; override;
⇒   SetIdade:(x:integer) ; override;
⇒ End;
```

**Exemplo 3.1 – Classe criada pelo combinador**

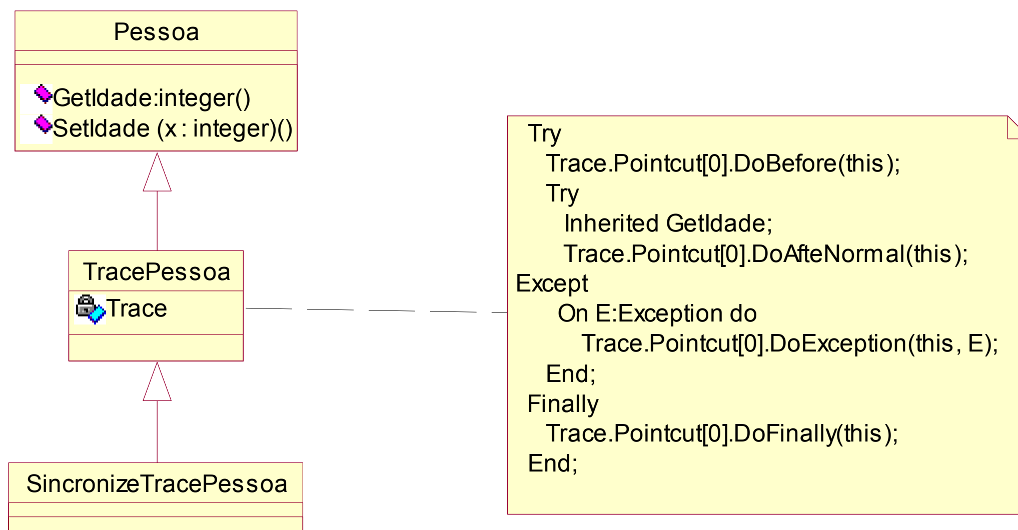


**Figura 3.3 – Método com comportamento alterado pelo aspecto Trace**

Uma abordagem semelhante a esta é utilizada pelo combinador da ferramenta AOP/ST. Para que os métodos combinados funcionem, é necessário que os objetos da classe *Pessoa* sejam substituídos por objetos da classe modificada, no caso, *TTracePessoa*. Se for necessário que o aspecto atinja também exemplares das

subclasses de *Pessoa*, é necessário que as subclasses referenciadas sejam também modificadas pelo combinador.

#### 3.3.1.1 Vantagens

A combinação é realizada de maneira não invasiva em relação à classe afetada, ou seja, é preservada a classe original, criando uma subclasse para cada classe afetada por um aspecto introduzido no sistema. Entretanto, isto causa a modificação do aplicativo que se utiliza desta classe.

#### 3.3.1.2 Problemas relacionados

Um dos problemas relacionados com esta abordagem são anomalias de herança em que a classe é modificada, mas suas subclasses continuam a utilizar a classe original (MATSUOKA & YONEZAWA, 1993).

Dependendo do número de classes afetadas, bem como do numero de aspectos que agem sobre uma mesma classe, a hierarquia de classes pode tornar-se extensa. As classes criadas não são utilizadas diretamente pelo usuário, mas podem consumir tempo extra devido as chamadas na tabela de métodos.

Se mais de um aspecto ou mais de um ponto de combinação de um mesmo aspecto agir sobre uma classe, é necessária a combinação por etapas, na qual é gerada uma subclasse para o primeiro aspecto, e uma subclasse da classe gerada para o segundo e assim por diante. O problema é a definição da ordem de combinação que deve ser realizada. Possíveis ordens de herança podem ser visualizadas na Figura 3.4 e na Figura 3.5.

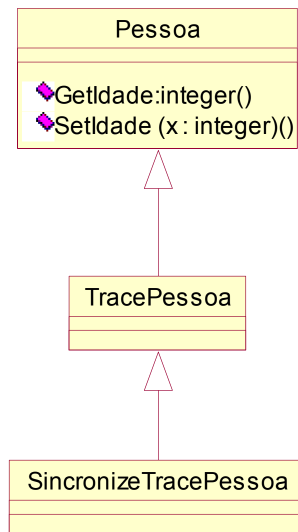


**Figura 3.4 – Possível ordem de herança**

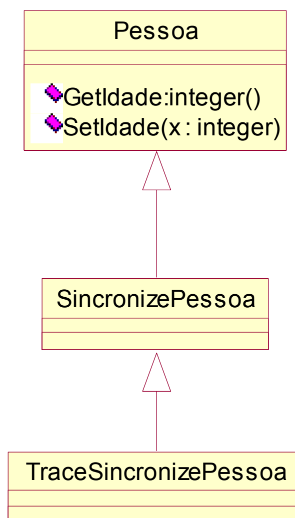


**Figura 3.5 – Possível ordem de herança**

Outra complicação é originada do fato que nem todas as linguagens possibilitam sobrescrever um método diretamente. Ex.: Em Object Pascal os comportamentos que podem ser sobrescritos pelas subclasses devem conter a palavra reservada *virtual* após a declaração dos métodos.

### 3.3.2 Combinando através de associação

A associação é um mecanismo de composição que pode ser utilizado para a implementação dos comportamentos descritos nos aspectos. Para cada aspecto que age

sobre uma classe é criada uma propriedade nomeada pelo aspecto, exemplar da classe *Aspect*, conforme o Exemplo 3.2. Os comportamentos afetados são modificados através da chamada aos métodos referentes ao aspecto (Figura 3.6).

```
TPessoa = class
⇒ Aspecto: TAspect;
    GetIdade:integer
    SetIdade:(x:integer)
End;
```

**Exemplo 3.2 – Classe modificada pelo combinador**

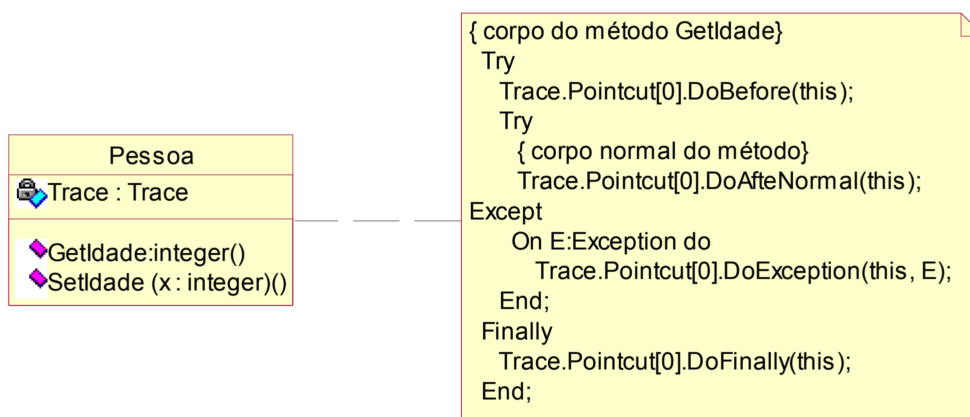


**Figura 3.6 - Classe alterada**

#### 3.3.2.1 Vantagens

Uma das vantagens desta abordagem é que não são criadas novas classes, e nem os objetos destas precisam ser modificados. Apenas as classes afetadas são modificadas.

#### 3.3.2.2 Problemas relacionados

Uma desvantagem desta abordagem é exatamente a modificação da classe afetada, dificultando a recuperação da classe original, bem como dificultando a identificação no código resultante das sentenças relativas a aspectos.

### 3.3.3 Combinando através do padrão de projeto Decorator

Outra alternativa de implementação é a utilização do padrão de projeto Decorator (GAMMA et al., 2000). Nesta abordagem, é criada uma subclasse da classe afetada, contendo uma referência a um objeto da superclasse da classe criada, como pode ser visualizado na Figura 3.7. A partir desta subclasse, chamada no padrão de

*Decorator*, é possível a criação de decoradores concretos, que implementam a funcionalidade descrita nos aspectos. O uso deste padrão é uma alternativa flexível ao uso de subclasses para a extensão da funcionalidade.

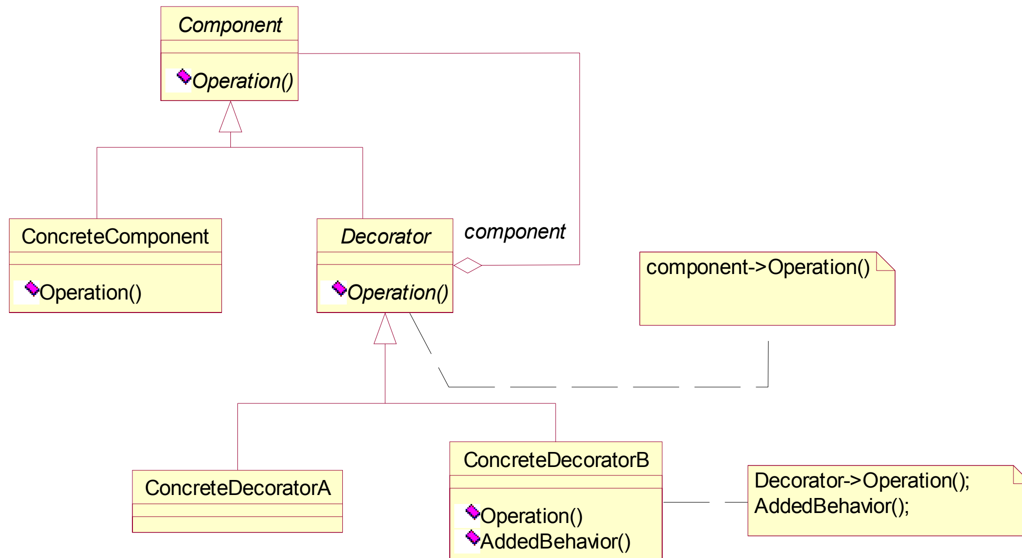


**Figura 3.7 – Padrão Decorator**

Como exemplo podemos visualizar a classe *Pessoa* sendo afetada por dois aspectos *Trace* e *Sincronize.* É criado um Decorator chamado de PessoaDecorator que repassa as chamadas para o objeto que está sendo decorado. Logo após são criadas mais duas classes que implementam a funcionalidade descrita nos aspectos, conforme a Figura 3.8.

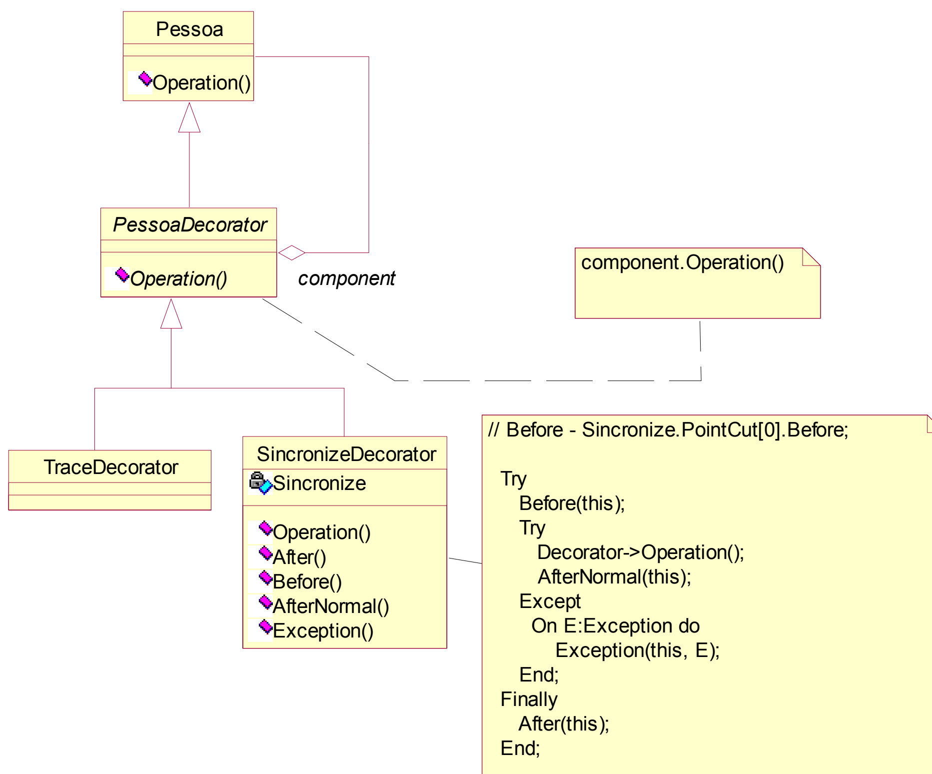


**Figura 3.8 – Aspectos Trace e Sincronize atuando sobre a classe Pessoa**

Os objetos da classe pessoa devem ser substituídos por objetos decorados com os decoradores *TraceDecorator* e *SincronizeDecorator*. Ao implementar esta estratégia é necessário prover mecanismos de definição de ordem em que os aspectos atuam sobre os comportamentos, da mesma forma que na estratégia que utiliza herança.

#### 3.3.3.1 Vantagens

As vantagens no uso desta abordagem estão principalmente na flexibilidade proporcionada pelo uso do padrão, onde a ordem dos decoradores pode ser alterada em tempo de execução. Destaca-se também que as mudanças são feitas de maneira não invasiva ao código existente.

#### 3.3.3.2 Problemas relacionados

Uma das dificuldades desta abordagem é que os decoradores são diferentes dos componentes no que tange a sua identidade.

O uso de decoradores para classes que são afetadas por mais de um aspecto ou aplicativos em que aspectos devam ter a possibilidade de serem removidos dinamicamente é mais vantajosa em relação à utilização de herança, entretanto para classes afetadas por somente um aspecto não têm proveito da flexibilidade proporcionada pelo padrão.

### 3.3.4 Combinando através de interceptação de mensagens

Nesta abordagem, para cada classe referenciada por um aspecto, é criada pelo combinador uma meta-classe que possui informações sobre a classe afetada, oferecendo mecanismos de interceptação de mensagens, o que permite a inclusão de sentenças relativas a cada aspecto em pontos chave da aplicação.

Como exemplo pode ser visualizada uma classe pessoa que é afetada por um aspecto (Figura 3.9). Para cada classe afetada é gerada uma metaclasse que simula a semântica especificada.

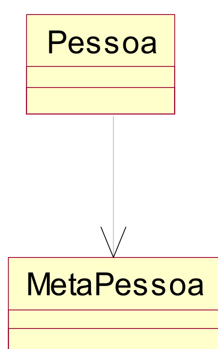


**Figura 3.9 –Metaclasse criada pelo combinador**

```
MetaPessoa metaclass of Pessoa

Method MetaPessoa.InterceptMessages{
  Try
    Trace.Pointcut[0].DoBefore(this);
    Try
      // Executa o método da classe base
      ExecuteBaseMethod (ParseTree);
      Trace.Pointcut[0].DoAfteNormal(this);     Except
     On E:Exception do
        Trace.Pointcut[0].DoException(this, E);
    End;
  Finally
    Trace.Pointcut[0].DoFinally(this);
  End;
}
```

**Figura 3.10 – Metaclasse com o método interceptado**

LUNAU (1998), descreve uma proposta de implementar aspectos como metaobjetos através do mecanismo de interceptação de mensagens. A arquitetura descrita foi originalmente desenvolvida para permitir reflexão computacional em aplicações de controle de processos e permite a troca de metaobjetos em tempo de execução.

BÁSTAN & PRYOR (1999) definem uma arquitetura reflexiva de suporte a programação orientada a aspectos em Smalltalk através da representação de aspectos utilizando-se de metaclasses que descrevem o comportamento desejado. É possível incluir instruções antes ou depois do código original. O combinador, nesta estratégia, é representado pela classe *AspectManager*, responsável pela interceptação de mensagens.

#### 3.3.4.1 Vantagens

Uma das principais vantagens desta abordagem é a adição não invasiva do código relativo aos aspectos nas classes da aplicação. Outra vantagem é a facilidade de implementação desta abordagem, que utiliza os mecanismos de interceptação de mensagens, e fornece conhecimentos acerca do nível base. Parte das responsabilidades do sistema são transferidas para o protocolo de metaobjetos.

#### 3.3.4.2 Problemas relacionados

Uma dificuldade é a escassez de implementações que utilizam os conceitos de reflexão computacional em linguagens orientadas a objeto. Outra questão a ser vislumbrada é que a adição de um novo nível de abstração pode afetar a performance do sistema.

# 4 Aurélia: Programação orientada a aspectos em Object Pascal

Aurélia é uma implementação do modelo descrito no capítulo anterior, implementado no ambiente de programação visual Borland Delphi 5.0, através da abordagem de interceptações de mensagens.

Esta abordagem foi implementada através do uso do protocolo de metaobjetos *OP*MOP (LEITE, 2001), que disponibiliza componentes que promovem facilidades na confecção de aplicativos com características reflexivas.

## 4.1 Componentes

As classes foram implementadas e a convenção de nomes segue a abordagem Object Pascal, onde os nomes de classes começam com a letra “T”.

### 4.1.1 Classe TAspect

O suporte a programação orientada a aspectos em Aurélia consiste em um conjunto de componentes não-visuais, modificados em tempo de desenvolvimento através do ambiente Delphi. Para representar um aspecto em uma aplicação é criado um exemplar da classe *TAspect* (Figura 4.1) ou de sub-classes dela.

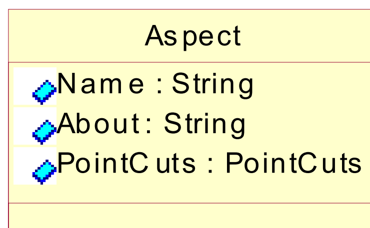


**Figura 4.1 – Classe Aspect definida no modelo**

```
TAspect = class (TCustomAspect)
Published
  property PointCuts;
  property About;
end;
```

**Exemplo 4.1 - Classe TAspect (implementação Object Pascal)**

A classe *TAspect* (Exemplo 4.1) representa aspectos em uma aplicação, e é derivada da classe *TCustomAspect* (Exemplo 4.2) que implementa as funcionalidades de uma aspecto mas deixa a cargo da classe *TAspect* a publicação das propriedades mais relevantes de um aspecto .

```
        TCustomAspect = class (TComponent)
        private
        ...
        public
           constructor Create(AOwner:TComponent); override;
           destructor Destroy; override;
        protected
           property PointCuts:TPointCuts ...;
           property About:string ...;
end;
```

**Exemplo 4.2 - Classe TCustomAspect : implementação Object Pascal**

Para definir um aspecto em uma aplicação Delphi, o usuário deve inserir um componente da classe *TAspect* na aplicação. Para uma melhor separação, é conveniente a colocação dos aspectos de um sistema em um *DataModule*, o qual é um Container de componentes não-visuais. Após a inserção do componente *TAspect* é necessária a criação de pontos de atuação, através da propriedade *PointCuts,* conforme visualizado na Figura 4.2**.**

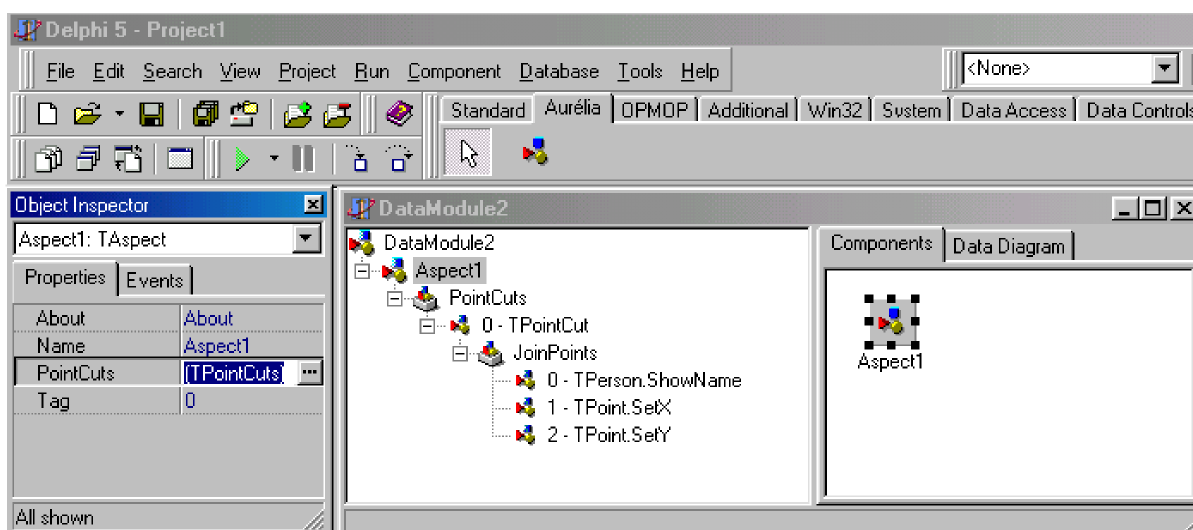


**Figura 4.2 – Aspecto em uma aplicação**

**Propriedades:**

**About**: Informações acerca do projeto;

**Name**: Identificador único para cada objeto do sistema;

**PointCuts**: Lista de pontos de atuação;

### 4.1.2 Classes PointCuts e PointCut

Para cada aspecto podem ser criados diversos pontos de atuação, definidos na classe *TPointcuts* (Exemplo 4.3), que funcionam como um repositório para os pontos de combinação em que um aspecto atua (Figura 4.3). Nos pontos de atuação são definidas as operações que os aspectos devem executar nos pontos de combinação.

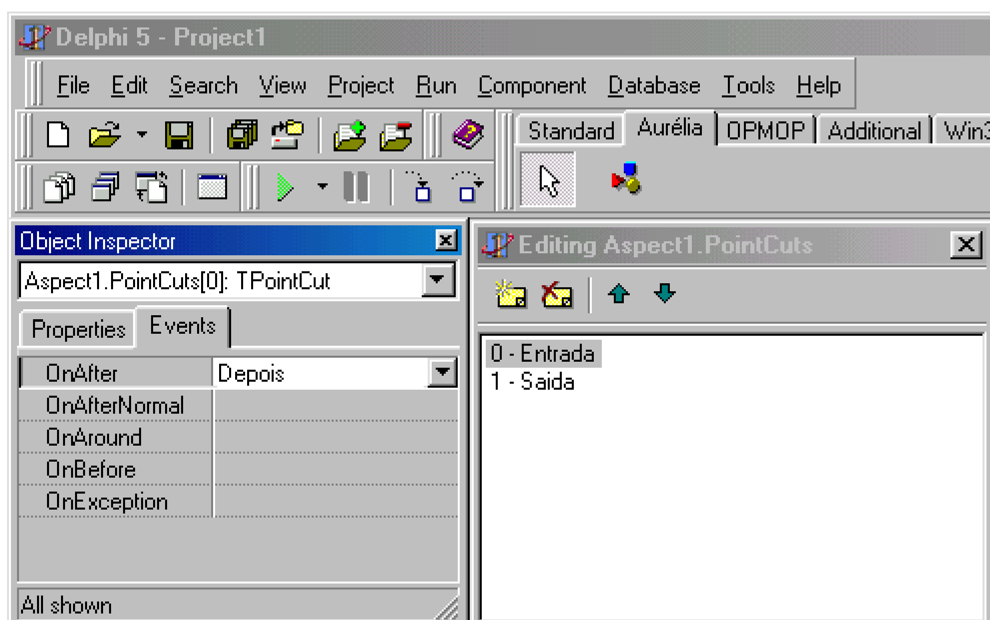


**Figura 4.3 – Pontos de atuação em uma aplicação**

Cada ponto de atuação (*TPointcut* - Exemplo 4.4) contém as seguintes características:

**Propriedades:**

**Active**: Determina se as operações definidas neste ponto devem ser executadas

**JoinPoints**: define pontos de junção entre os aspectos e os componentes;

**Eventos:**

**OnAfter**: Ocorre após a execução dos métodos definidos na propriedade *JoinPoints*, independente de ocorrer exceções no momento da execução do comportamento definido na linguagem de componentes;

**OnAfterNormal**: Ocorre após a execução dos métodos definidos na propriedade *JoinPoints*, se o método tiver um retorno normal;

**OnBefore**: Ocorre antes da execução dos métodos definidos na propriedade *JoinPoints*, se o método tiver um retorno normal;

**OnException**: Ocorre no momento em que uma exceção é levantada no decorrer da execução dos métodos definidos na propriedade *JoinPoints*

**Métodos:**

**After**: Chama o manipulador do evento definido na propriedade *OnAfter*;

**AfterNormal**: Chama o manipulador do evento definido na propriedade *OnAfterNormal*;

**Before**: Chama o manipulador do evento definido na propriedade *OnBefore*;

**Exception**: Chama o manipulador do evento definido na propriedade *OnException*;

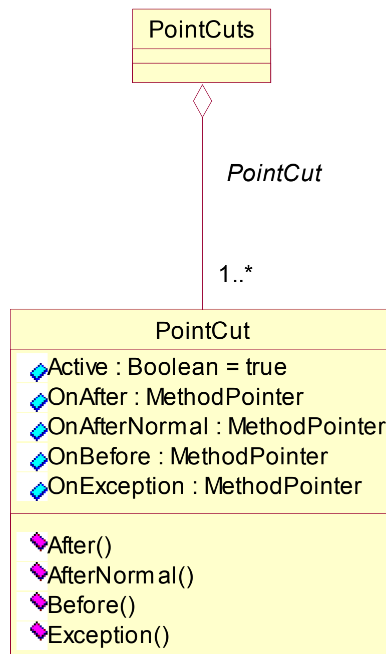


**Figura 4.4 – Classes Pointcuts e PointCut**

```
        TPointCuts = class(TCollection)
        private
          FAspect:TCustomAspect;
        ...
        protected
        ...
        public
          constructor Create(Aspect:TCustomAspect);
          function Add: TPointCut;
          property Items[Index: Integer]: TPointCut ...; default;
end;
```

**Exemplo 4.3 - Classe TPointCuts**

```
        TPointCut = class(TCollectionItem)
        private
            function GetName: String;
        protected
           function GetDisplayName:string;override;
        public
           procedure Before (ThisJoinPoint: TObject); dynamic;
           procedure AfterNormal (ThisJoinPoint: TObject); dynamic;
           procedure Exception (ThisJoinPoint: TObject); dynamic;
           procedure After (ThisJoinPoint: TObject); dynamic;
           procedure Around (ThisJoinPoint: TObject); dynamic;
           procedure Assign (Source:TPersistent); override;
           constructor Create (Collection:TCollection); override;
           destructor Destroy;override;
        published
           property Active:Boolean ...;
           property Name:String ...;
           property JoinPoints:TJoinPoints ...;
           property OnBefore:TPointCutEvent ...;
           property OnAfterNormal:TPointCutEvent ...;
           property OnException:TPointCutEvent ...;
           property OnAfter:TPointCutEvent ...;
           property OnAround:TPointCutEvent ...;
        end;
```

**Exemplo 4.4 - Classe TPointCut**

### 4.1.3 Classe JoinPoints e JoinPoint

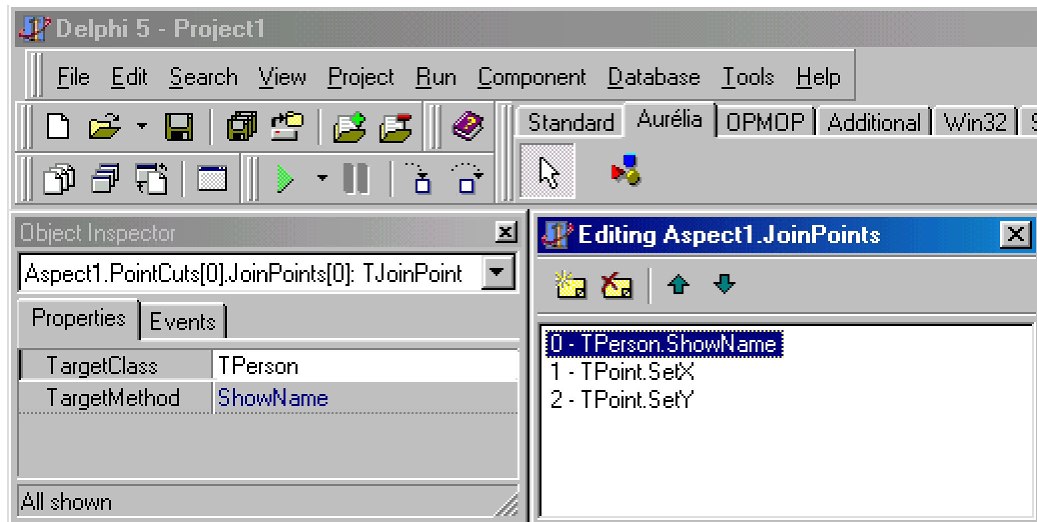


**Figura 4.5 – Pontos de combinação em uma aplicação**

Os pontos de combinação (*TJoinPoints* - Exemplo 4.5) são os pontos onde a linguagem de aspectos coordena-se com a linguagem de componentes. Em Aurélia é possível definir pontos de combinação nos seguintes eventos:

- Chamadas a métodos;
- Chamadas a construtores e
- Chamadas a destrutores.

```
TJoinPoints = class(TCollection)
private
  FPointCut:TPointCut;
  ...
protected
  ...
public
  constructor Create(PointCut:TPointCut);
  function Add: TJoinPoint;
  property Items[Index: Integer]: TjoinPoint ...; default;
end;
```

**Exemplo 4.5 – Classe TJoinPoints**

```
TJoinPoint = class(TCollectionItem)
protected
 ...
```

```
public
 ...
published
  property TargetClass:String ...;
  property TargetMethod:String ...;
  end;
```

**Exemplo 4.6 – Classe TJoinPoint**

Cada ponto de combinação (*TJoinPoint* - Exemplo 4.6) é definido através das seguintes propriedades:

**TargetClass**: Classe afetada pelo aspecto;

**TargetMethod**: Método afetado pelo aspecto;

A partir destas classes é definido um componente que permite a inclusão de aspectos em uma aplicação. O próximo capítulo descreve como os objetos definidos a partir do componente *TAspect* são combinados com o programa de componentes.

## 4.2 Implementação do Combinador

A função do combinador é mesclar as especificações de aspectos e componentes, gerando um programa equivalente, utilizando-se somente a linguagem de componentes.

O combinador de Aurélia é implementado em Object Pascal, utilizando–se da API Open Tools (OTA). Ele consiste em um componente que acessa implementações de um conjunto de interfaces que permitem a manipulação de grande parte do ambiente Borland Delphi.

Através da OTA é possível criar e manipular menus, formulários, componentes, debugger, mensagens, editor do ambiente, gerenciar projetos, etc. A OTA disponível no Delphi 5.0 é a mesma utilizada pelo C++ Builder, e segundo a Borland, o Kylix 1.0 será compatível com projetos escritos em Delphi 6.0.

Para que a tarefa tivesse êxito, foi utilizado um protocolo de metaobjetos (MOP) denominado *OP*MOP. Este MOP permite o controle da execução de um método através da definição de metaclasses, que interagem com o nível base, interceptando mensagens e realizando operações sobre o nível base.

O *OP*MOP é composto de 14 componentes, entretanto para o escopo deste trabalho, apenas 4 deles foram considerados e 1 deles efetivamente utilizado, o componente MOPAssocia.

Este componente, visualizado na Figura 4.6, é encarregado de prover o suporte à realização do processo de interceptação de mensagens, onde o nível base é controlado por um metanível através de um metaobjeto que intercepta as mensagens enviadas ao nivel base. O componente além de associar a classe do nível base à sua metaclasse promove a reificação do nível base (LEITE, 2001).

**Figura 4.6 – MOPAssocia**

Quando acionado, o combinador (Figura 4.7) executa as seguintes operações:

1. Obtém informações acerca do projeto a ser combinado, gerando uma lista de unidades e uma lista de aspectos que atuam no sistema;
2. Cria um diretório temporário e monta um projeto temporário que será combinado;
3. Transforma as informações referentes a cada aspecto do sistema em uma lista de classes e de que maneira essas classes são afetadas;
4. Para cada classe afetada por aspectos é gerada uma metaclasse que reproduz a semântica descrita nos aspectos;
5. São criados componentes MOPAssocia para cada classe, associando as classes do nível base com as metaclasses que representam o comportamento alterado;
6. A cláusula *uses* de cada unidade é afetada para refletir a configuração atual do sistema;

As metaclasses criadas são associadas com as classes do nível base através do componente TMOPAssocia, que faz parte do OPMOP e segundo LEITE (2001), é encarregado de prover os mecanismos de interceptação de mensagens.

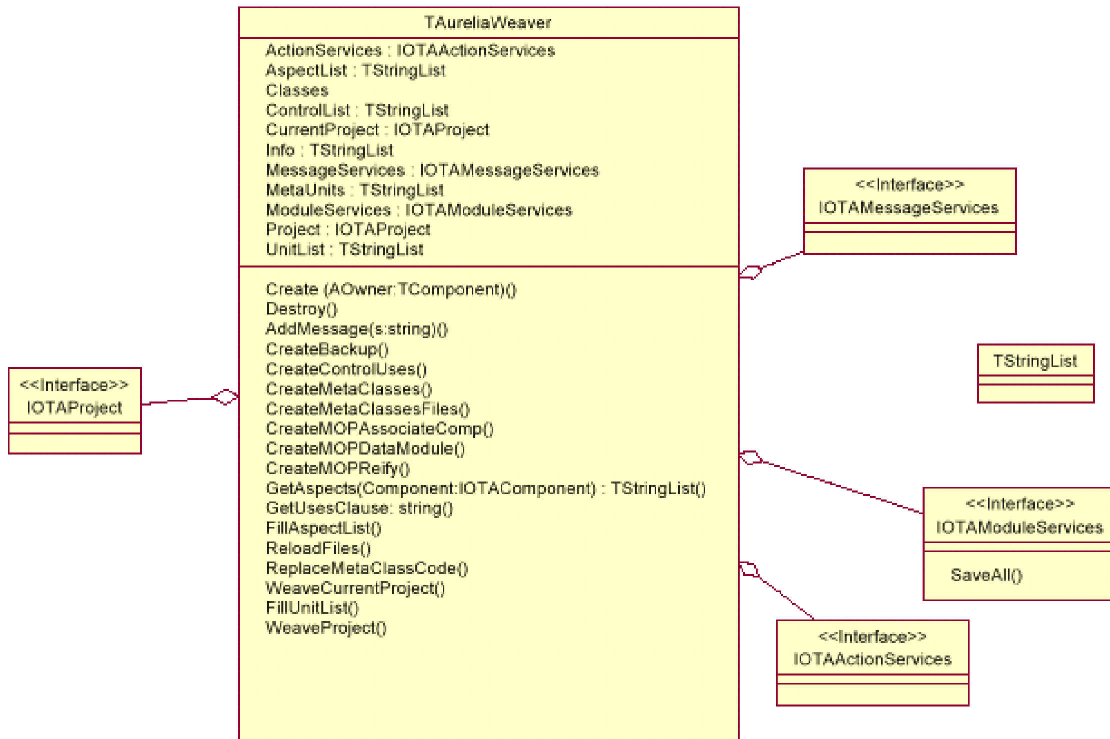


**Figura 4.7 – Combinador - Diagrama de Classes**

O funcionamento do combinador pode ser visualizado na Figura 4.8 e Figura 4.9, através dos diagramas de seqüência .

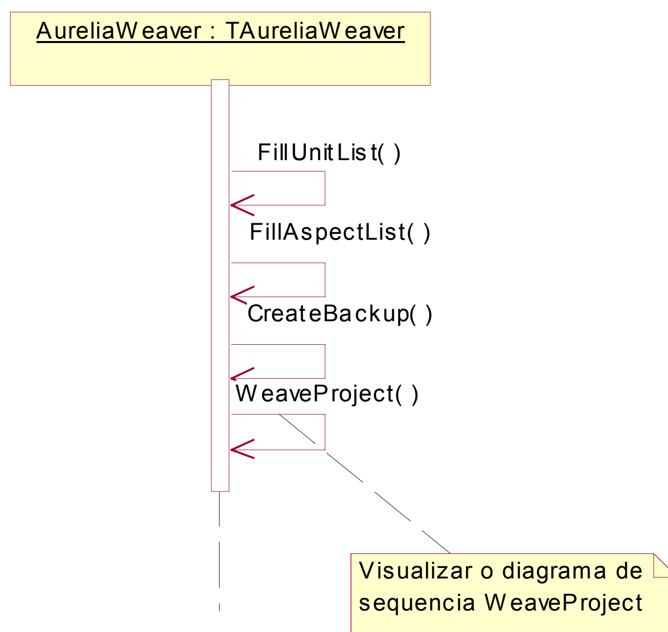

**Figura 4.8 - Diagrama de seqüência Combinar Projeto**

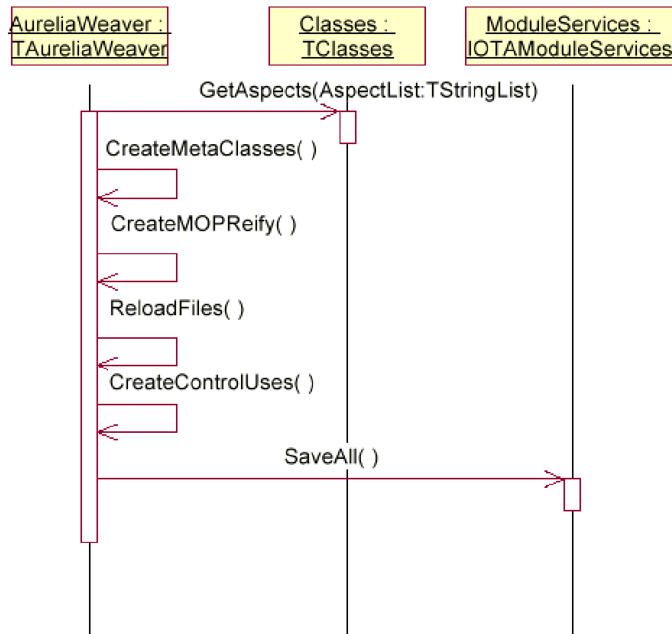


**Figura 4.9 – Diagrama de seqüência *WeaveProject***

Para combinar um projeto, o usuário deve adicionar um componente da classe *TAurelia* no aplicativo e modificar a propriedade *Weaved* para *true*, conforme a Figura 4.10. O programa será combinado e poderá ser compilado normalmente.

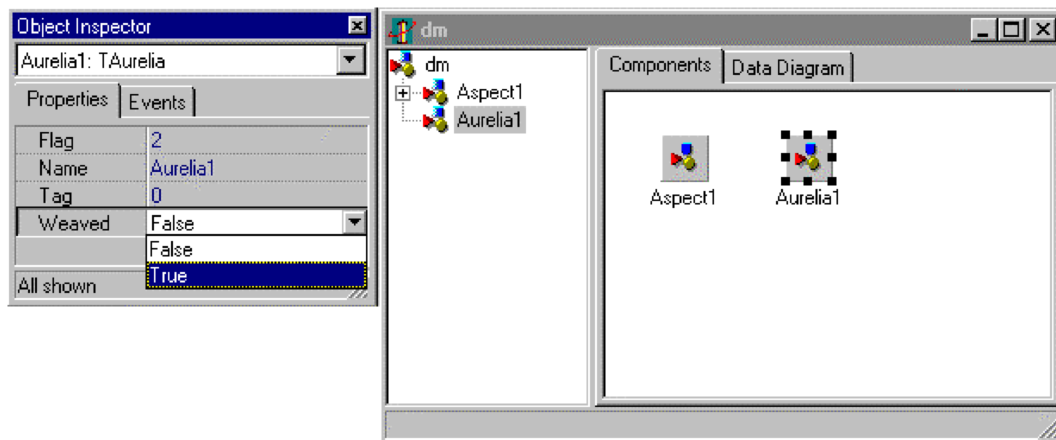


**Figura 4.10 – Combinando um projeto**

# 5 Aplicativo de exemplo

O exemplo consiste em um aplicativo denominado de *HelloWorld* (Figura 5.1). Ele consiste em um formulário contendo um botão que faz uma chamada ao método *ShowName* da classe *TPessoa* (Exemplo 5.1).

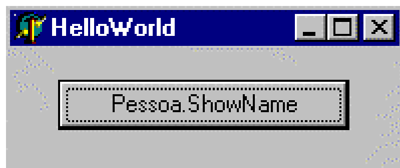


**Figura 5.1 - Aplicativo Hello Horld**

```
TPessoa = class
        Procedure ShowName;
End;

Procedure TPessoa.ShowName;
Begin
        Showmessage('Eduardo Kessler Piveta');
End;
```

**Exemplo 5.1 – Classe TPessoa**

No clique do botão são executadas as instruções contidas no Exemplo 5.2.

```
procedure TForm1.Button1Click(Sender: TObject);
var
  Pessoa:TPessoa;
Begin
  Pessoa:=TPessoa.Create;
  Pessoa.ShowName;
  Pessoa.Free;
end;
```

**Exemplo 5.2 – Manipulador de eventos do clique do botão**

A execução deste aplicativo tem como resultado a saída da Figura 5.2

**Figura 5.2 – Saída do aplicativo**

Definindo um aspecto chamado HelloWorld, possuindo um ponto de atuação chamado Mensagens (Figura 5.3) que atua no recebimento de mensagens *Showmessage* em objetos da classe *TPessoa*.

**Figura 5.3 – Ponto de atuação Mensagens**

No ponto de atuação *Mensagens*, são definidas ações que são executadas antes do ponto de combinação e depois dele ().

```
procedure Tdm.Before(ThisJoinPoint: TObject);
begin
  showmessage(‘Antes da execução');
end;

procedure Tdm.AfterNormal(ThisJoinPoint: TObject);
begin
  showmessage(‘Depois da execução’);
end;
```

**Exemplo 5.3 – Manipuladores de evento das ações do ponto de atuação**

Ao ser combinado e compilado, o aplicativo obtém a seguinte saída na execução do clique do botão ().

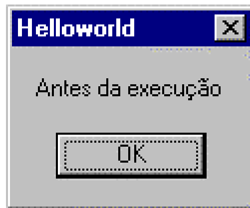


**Figura 5.4 – Saída do aplicativo – Tela 1**

**Figura 5.5 – Saída do aplicativo – Tela 2**

**Figura 5.6 – Saída do aplicativo – Tela 3**

Isto ocorre porque foi gerada uma metaclasse chamada de *TTPessoaMetaClass*, descrita no Exemplo 5.4.

```
TTPessoaMetaClass = class(TMOP)
public
 function controleExecucao (BaseObjeto:TObject;metodo:TMOP_Metodo;args:
TMOP_Args):Boolean; override;
end;

function TTPessoaMetaClass.controleExecucao (BaseObjeto:TObject;metodo:
TMOP_Metodo; args: TMOP_Args) : Boolean;
begin
   Result := False;
   metodo.nome := Trim(UpperCase(metodo.nome));
   if (metodo.nome = 'SHOWNAME') then
   begin
   try
      U_dm.dm.Aspect1.PointCuts[0].Before(self);
```

```
            try
              Result := oNBControle.executa(metodo, args);
              U_dm.dm.Aspect1.PointCuts[0].AfterNormal(self);
            except
              On E:Exception do
                  U_dm.dm.Aspect1.PointCuts[0].Exception(self, E);
           end;
           finally
              U_dm.dm.Aspect1.PointCuts[0].After(self);
           end
        end
     else
        Result := oNBControle.executa(metodo, args);
     end;
```

**Exemplo 5.4 – Metaclasse criada pelo combinador**

Então, a metaclasse é associada com a classe base através da criação de um componente MOPAssocia, visualizado na figura X.

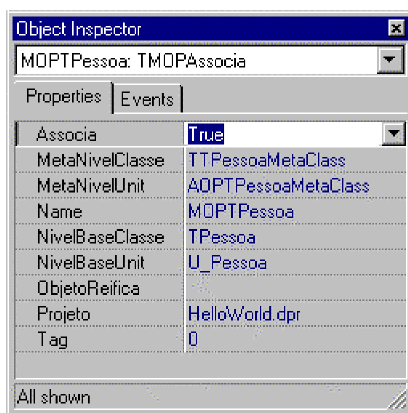


**Figura 5.7 – Componente MOPAssocia criado pelo combinador**

O aplicativo volta ao estado inicial ao ser modificada a propriedade *Weaved* do componente *Aurelia1* para *false*.

# 6 Considerações finais

Este trabalho apresenta uma proposta de suporte a programação orientada a aspectos através de componentes, bem como uma implementação deste modelo para Object Pascal.

O uso de uma estrutura reflexiva como base para a implementação de Aurélia influiu decisivamente para os resultados do projeto. A facilidade provida pelo MOP, principalmente através do mecanismo de interceptação de mensagens, diminuiu o tempo total para a implementação do sistema, bem como reduziu a responsabilidade final do ambiente.

Sem o uso do *OP*MOP ou outro MOP, a implementação seria muito mais trabalhosa, devido ao fato que seria necessária a implementação de parte das características providas pelo MOP, como a interceptação de mensagens e conhecimentos da classe base.

Este modelo pode ser implementado em outros ambientes visuais de programação, devendo ser adaptado para a geração de código na linguagem “alvo”. Além disso é de grande valia a existência de mecanismos de reflexão estrutural ou comportamental para facilitar a implementação das estratégias definidas no modelo.

Tanto o MOP utilizado (o *OP*MOP) quanto a ferramenta desenvolvida (Aurélia) necessitam de melhoramentos. Dentre os melhoramentos visíveis ao *OP*MOP existem diversas vantagens em se usar a OTA ao invés da estratégia utilizada pelo protocolo. Isso implica no fato que os arquivos são acessados e modificados dentro do próprio ambiente, bem como é possível obter informações como: projeto atual, unidades, componentes, etc.

Outra limitação do *OP*MOP, a persistência dos componentes TMOPExecuta, pode ser resolvida criando esses componentes através da OTA em tempo de desenvolvimento.

No caso de Aurélia as principais restrições dizem respeito a falta de informações mais detalhadas a respeito do programa e das classes do nível base, como as funcionalidades providas por um MOP estrutural, o que permitiria a inclusão, alteração e exclusão de propriedades, métodos e classes, facilitando a implementação das outras estratégias propostas no modelo.

Outra restrição, herdada do *OP*MOP é que apenas classes diretamente descendentes de TObject podem ser alvo de aspectos.

Apesar destas restrições existem diversos benefícios do uso da ferramenta Aurélia:

- Redução do entrelaçamento de código nos aplicativos que possuem propriedades ortogonais entre si;
- Facilidade de especificação dos aspectos em tempo de desenvolvimento;
- Fácil combinação/recuperação do programa original e
- Aspectos podem ser ativados ou desativados em tempo de execução através dos pontos de atuação;

Além disso, este trabalho tem como contribuições principais:

- Definição de diferentes estratégias de implementação que podem ser utilizadas para implementação de suporte a programação orientada a aspectos;
- Implementação de uma ferramenta que implementa estes conceitos em um ambiente visual de programação e
- Descrição do estado da arte em programação orientada a aspectos.

Aurélia oferece uma combinação em tempo de desenvolvimento através de um ambiente reflexivo e promove a separação de interesses de maneira clara, possibilitando ao usuário visualizar o resultado do processo antes da compilação.

## 6.1 Trabalhos Futuros

Existem diversas propostas de continuação deste trabalho, dentre elas podemos destacar:

- Implementação do modelo em outras linguagens de programação e utilizando outros protocolos de metaobjetos;
- Definição e implementação de um MOP estrutural para Object Pascal;
- Implementação das diferentes estratégias de implementação;
- Criação de sintaxe para a definição de aspectos;
- Utilização de Aurélia em cursos de graduação/pós-graduação em disciplinas que falem sobre programação orientada a aspectos;
- Eliminar as restrições atuais, tanto do *OP*MOP quanto de Aurélia;

### 6.1.1 Problemas encontrados no uso do OPMOP

Existem diversos problemas encontrados no uso do OPMOP, que devem ser reparados se for disponibilizado para utilização em escala maior. Sugere-se a definição de aplicativos a serem desenvolvidos usando o *OP*MOP, por pessoas diferentes e seus comentários utilizados como base para correções.

Como usuário de uma versão “alfa”, utilizada por um número reduzido de pessoas, estabeleço alguns problemas reportados que devem ser corrigidos para a distribuição de uma versão “beta”.

Problemas encontrados:

1 - Se o nome da *Unit* for igual ao nome da classe base, o nome da *unit* é substituído pelo nome da classe alterada. Ex.: “unit Pessoa” é transformada para “unit {Pessoa}PessoaAlterada”;

2 - Ao ser criada a *unit* <unitBase>Controle, devem ser observadas as bibliotecas que já são utilizadas pela classe base, incluíndo-as no arquivo de controle;

3 – Os métodos privados de uma classe não podem ser sobrescritos pela classe “alterada” no arquivo de controle. Isto gera um erro de compilação. As possíveis soluções para isto são a modificação do método para *protected*, ou ignorar este método na criação da classe alterada.

4 – As diretivas utilizadas por procedimentos e funções (override, virtual, etc) não devem ser incluídas na parte implementation de uma unit, devendo ser somente definida na parte interface;

5 – Métodos abstratos não devem ser implementados, ou melhor, interceptados pela metaclasse, devido ao fato que eles não possuem implementação;

6 – Apenas classes descendentes diretamente de TObject podem ter seus métodos interceptados devido ao modo como os objetos de controle são criados;

# 7 Anexos

## 7.1 Arquivos que compõe a implementação

A implementação é composta dos seguintes arquivos:

- AOPAbout.pas: Informações sobre o aplicativo;
- AOPAspectizedClasses.pas: Unit de suporte para preenchimento;
- AOPConst.pas: Constantes utilizadas pelos componentes;
- AOPDataModule.pas: Arquivo contendo o Data Module que é gerado no momento da combinação;
- AOPExpert.pas: Contém as classes que implementam o combinador;
- AOPMetaClass.pas: Contém informações para a criação das metaclasses;
- AOPSupport.pas: Contém as classes que formam os componentes visuais e
- AOPAddIn: Componente que faz o papel do combinador.

## 7.2 Código Fonte dos Componentes

A seguir têm-se parte do código fonte dos componentes da implementação de Aurélia.

### 7.2.1 AOPSupport.pas

```
interface
uses    Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs, ExtCtrls,
ComCtrls, dsgnintf, typinfo, OPMOP;

type

TPointCutExceptionEvent = procedure (ThisJoinPoint: TObject; E:Exception) of object;
TPointCutEvent = procedure (ThisJoinPoint: TObject) of object;
TAspect = class;
TCustomAspect = class;
TPointCut = class;

TJoinPoint = class(TCollectionItem)
private
   …
protected
  function GetDisplayName:String; override;
public
  procedure Assign(Source:TPersistent); override;
published
```

```
  property TargetClass:String read FTargetClass write FTargetClass;
  property TargetMethod:String read FTargetMethod write FTargetMethod;
end;

TJoinPoints = class(TCollection)
private
  FPointCut:TPointCut;
  function GetItem(Index: Integer): TJoinPoint;
  procedure SetItem(Index: Integer; const Value: TJoinPoint);
protected
  function GetOwner: TPersistent; override;
public
  constructor Create(PointCut:TPointCut);
  function Add: TJoinPoint;
  property Items[Index: Integer]: TJoinPoint read GetItem write SetItem; default;
end;

TPointCut = class(TCollectionItem)
private
   …
   FName: String;
   FJoinPoints: TJoinPoints;
   FActive: Boolean;
   procedure SetJoinPoints(const Value: TJoinPoints);
   procedure SetName(const Value: String);
   function GetName: String;
   procedure SetActive(const Value: Boolean);
protected
   function GetDisplayName:string;override;
public
   procedure Before(ThisJoinPoint: TObject); dynamic;
   procedure AfterNormal(ThisJoinPoint: TObject); dynamic;
   procedure Exception(ThisJoinPoint: TObject; E:Exception); dynamic;
   procedure After(ThisJoinPoint: TObject); dynamic;
   procedure Around(ThisJoinPoint: TObject); dynamic;
   procedure Assign(Source:TPersistent); override;
   constructor Create(Collection:TCollection); override;
   destructor Destroy;override;
published
   property Active:Boolean read FActive write SetActive;
   property Name:String read GetName write SetName;
   property JoinPoints:TJoinPoints read FJoinPoints write SetJoinPoints;
   property OnBefore:TPointCutEvent read FOnBefore write FOnBefore;
   property OnAfterNormal:TPointCutEvent read FOnAfterNormal write FOnAfterNormal;
   property OnException:TPointCutExceptionEvent read FOnException write FOnException;
   property OnAfter:TPointCutEvent read FOnAfter write FOnAfter;
   property OnAround:TPointCutEvent read FOnAround write FOnAround;
end;

TPointCuts = class(TCollection)
private
  FAspect:TCustomAspect;
  function GetItem(Index: Integer): TPointCut;
  procedure SetItem(Index: Integer; const Value: TPointCut);
protected
  function GetOwner: TPersistent; override;
public
  constructor Create(Aspect:TCustomAspect);
```

```
  function Add: TPointCut;
  property Items[Index: Integer]: TPointCut read GetItem write SetItem; default;
end;

TCustomAspect = class(TComponent)
private
   FPointCuts: TPointCuts;
   FAbout: string;
   procedure SetPointCuts(const Value: TPointCuts);
public
   constructor Create(AOwner:TComponent); override;
   destructor Destroy; override;
protected
   property PointCuts:TPointCuts read FPointCuts write SetPointCuts;
   property About:string read FAbout write FAbout;
end;

TAspect = class(TCustomAspect)
published
  property PointCuts;
  property About;
end;
```

## 7.3 Código Fonte do Combinador

### 7.3.1 AOPMetaClass.pas

```
unit AOP{<TargetClass>}MetaClass;

interface

Uses
   Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs, StdCtrls, uMOP;

Type
  T{<TargetClass>}MetaClass = class(TMOP)
  public
    function controleExecucao (metodo: TMOP_Metodo; args: TMOP_Args):Boolean; Override;
  end;

implementation

uses {<BaseUnits>};

function T{<TargetClass>}MetaClass.controleExecucao (metodo: TMOP_Metodo; args: TMOP_Args) : Boolean;
begin
   Result := False;
   metodo.nome := Trim(UpperCase(metodo.nome));
   {<AspectizedMethods>}
   Result := oNBControle.executa(metodo, args);
end;

end.
```

## 7.3.2 AOPExpert.pas

```
interface

uses sharemem, dialogs, sysutils, windows, virtintf, exptintf, toolintf, toolsAPI, classes, contnrs,
AOPAspectizedClasses, AOPConst;

type

EWeaverException = class(Exception);

TAureliaWeaver = class(TComponent)
private
   …
   function GetInfo:TStringList;
   procedure FillUnitList;
   procedure FillAspectList;
   function  SearchWord(var Editor:IOTAEditBuffer;s:string;var row,
   column:integer):Boolean;
   procedure ReplaceTargetClass(var Editor:IOTAEditBuffer;ModuleIndex:integer);
   procedure ReplaceAspectizedMethods(var Editor:IOTAEditBuffer;
   ModuleIndex: integer);
   procedure ReplaceBaseUnits(var Editor:IOTAEditBuffer);
   procedure WeaveProject;
public
   constructor Create(AOWner:TComponent);override;
   destructor Destroy; override;
   procedure AddMessage(const s:string);
   procedure CreateBackup;
   procedure CreateControlUses;
   procedure CreateMetaClasses;
   procedure CreateMetaClassesFiles;
   function CreateMOPAssociateComp(var Form:IOTAFormEditor;
     var offset:integer;Name:String):IOTAComponent;
   function CreateMOPDataModule:String;
   procedure CreateMOPReify;
   function GetAspects(Component:IOTAComponent):TStringList;
   function GetCurrentProject:IOTAProject;
   function GetControlList:string;
   function GetUsesClause:string;
   procedure ReloadFiles;
   procedure RestoreProject;
   procedure ReplaceMetaClassCode(var Editor:IOTAEditBuffer;
   ModuleIndex:integer);
   procedure WeaveCurrentProject;
published
   property ActionServices:IOTAActionServices read FActionServices  write FActionServices;
   property AspectList:TStringList read FAspectList write FAspectList;
   property Classes:TClasses read FClasses write FClasses;
   property ControlList:TStringList read FControlList write FControlList;
   property CurrentProject:IOTAProject read GetCurrentProject;
   property Info:TStringList read GetInfo;
   property MessageServices:IOTAMessageServices read FMessageServices
   write FMessageServices;
   property MetaUnits:TStringList read FMetaUnits write FMetaUnits;
   property ModuleServices:IOTAModuleServices read FModuleServices
   write FModuleServices;
```

```
  property Project:IOTAProject read FProject write FProject;
  property UnitList:TStringList read FUnitList write FUnitList;
end;
```

## 7.4 Código Units Auxiliares

A seguir têm-se o código fonte das unidades auxiliares da implementação de Aurélia. Algumas declarações triviais foram omitidas.

### 7.4.1 AOPAspectizedClasses.pas

```
interface

uses   Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs,
  ExtCtrls, ComCtrls, dsgnintf, typinfo, OPMOP;

type

  TClasses = class;

  TAspectizedClass = class(TCollectionItem)
  private
         …
  public
    constructor Create(Collection: TCollection); override;
    destructor Destroy;
  published
    property ClassName:String read FClassName write FClassName;
    property Methods:TStringList read FMethods write FMethods;
    property Paths:TStringList read FPaths write FPaths;
  end;

  TAspectizedClasses = class(TCollection)
  private
    FClasses:TClasses;
    function GetItem(Index: Integer): TAspectizedClass;
    procedure SetItem(Index: Integer; const Value: TAspectizedClass);
  protected
    function GetOwner: TPersistent; override;
  public
    constructor Create(Classes:TClasses);
    function Add: TAspectizedClass;
    property Items[Index: Integer]: TAspectizedClass read GetItem write SetItem; default;
  end;

  TClasses = class(TComponent)
  private
    FClasses: TAspectizedClasses;
    procedure SetClasses(const Value: TAspectizedClasses);
  public
    constructor Create(AOwner:TComponent); override;
    destructor Destroy; override;
    function IndexOf(aClassName:String):Integer;
    procedure GetAspects(AspectList:TStringList);
```

```
 published
   property Classes:TAspectizedClasses read FClasses write SetClasses;
 end;
```

## 7.4.2 AOPConst;

```
interface

const
InfoText:string =
   '{*****************************************************}'+#13+
   '{                                    }'+#13+
   '{      MetaClasses Generated by                }'+#13+
   '{      Aspect Oriented Programming Unit            }'+#13+
   '{      Universidade Federal de Santa Catarina - UFSC     }'+#13+
   '{      Florianópolis - Brasil                 }'+#13+
   '{                                    }'+#13+
   '{    Authors:                            }'+#13+
   '{      Eduardo Kessler Piveta                  }'+#13+
   '{      kessler@inf.ufsc.br                  }'+#13+
   '{      http://www.inf.ufsc.br/~kessler             }'+#13+
   '{                                    }'+#13+
   '{      Luiz Carlos Zancanella                 }'+#13+
   '{      zancanel@inf.ufsc.br                  }'+#13+
   '{      http://www.inf.ufsc.br/~zancanel             }'+#13+
   '{                                    }'+#13+
   '{*****************************************************}'+#13;

MethodBody:string =
     '{<AspectizedClass>} '+#13+
     'if (metodo.nome = {<TargetMethod>}) then '+#13+
     'begin '+#13+
     '  try '+#13+
     '    {<Aspect>}.Before(self); '+#13+
     '    try '+#13+
     '      Result := oNBControle.executa(metodo, args); '+#13+
     '      {<Aspect>}.AfterNormal(self); '+#13+
     '    except '+#13+
     '      On E:Exception do '+#13+
     '      {<Aspect>}.Exception(self, E); '+#13+
     '    end; '+#13+
     '  finally '+#13+
     '    {<Aspect>}.After(self); '+#13+
     '  end '+#13+
     'end '+#13+
     'else '+#13;

MetaClassBody:string =
'unit AOP{<TargetClass>}MetaClass;'+#13+
''+#13+
'interface'+#13+
''+#13+
'Uses'+#13+
''+#13+
'Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs,'+
```

```
'StdCtrls, uMOP;'+#13+
''+#13+
'Type'+#13+
'    T{<TargetClass>}MetaClass = class(TMOP)'+#13+
'     public'+#13+
'                function controleExecucao (BaseObjeto:TObject;metodo: TMOP_Metodo; args: TMOP_Args) : Boolean; Override;'+#13+
'     end;'+#13+
''+#13+
'implementation'+#13+
''+#13+
'uses {<BaseUnits>};'+#13+
''+#13+
'function T{<TargetClass>}MetaClass.controleExecucao (BaseObjeto:TObject;metodo: TMOP_Metodo; args: TMOP_Args) : Boolean;'+#13+
'begin'+#13+
'    Result := False;'+#13+
'    metodo.nome := Trim(UpperCase(metodo.nome));'+#13+
'    {<AspectizedMethods>}'+#13+
'    Result := oNBControle.executa(metodo, args);'+#13+
'end;'+#13+
''+#13+
'end.';

implementation
end.
```

### 7.4.3 AOPAddIn.pas

```
interface

uses sharemem, dialogs, sysutils, windows, virtintf, exptintf, toolintf, toolsAPI, classes, contnrs, Menus, Extctrls;

type

TAurelia = class(TComponent)
private
  FWeaved: Boolean;
  FFlag: integer;
  procedure SetWeaved(const Value: Boolean);
  procedure SetFlag(const Value: integer);
public
published
  property Flag:integer read FFlag write SetFlag;
  property Weaved:Boolean read FWeaved write SetWeaved;
end;

TAureliaMenu = class (TObject)
  procedure WeaveItem(Sender:TObject);
end;
```

# 8 Referências Bibliográficas

ACHERMANN, Franz, NIERSTRASZ, Oscar. **Separation of Concerns through unification of concepts.** Proceedings of the Workshop on Aspects & Dimensions of Concerns at ECOOP'2000, Springer-Verlag, 1999.

AKSIT, Mehmet, TEKINERDOGAN, Bedir. **Formalizing adaptability aspects**. Proceedings of the Aspect-Oriented Programming Workshop at ICSE'98. Kyoto - Japão, 1998a.

AKSIT, Mehmet, TEKINERDOGAN, Bedir. **Solving the modeling problems of object oriented languages by composing multiple aspectos using composition filters**. Proceedings of the Aspect-Oriented Programming Workshop at ECOOP'98, 1998b.

ASPECTJ, **AspectJ Home Page.** http://www.aspectj.org

BADER, Atef, CONSTANTINIDES, Constantinos A., ELRAF, Tzilla. **An Aspect-Oriented Design Framework for Concurrent Systems.** Proceedings of the Aspect-Oriented Programming Workshop at ECOOP'99, Finlândia: Springer-Verlag, 1999.

BAKKEN, D. E., SCHANITZ, R. E., ZINKY, J. A. **Architetural Support for Quality of Service for CORBA Objects**. Theory and Practice of Object Systems, Estados Unidos, 1997.

BASTÁN, Natalio, PRYOR, Jane,. **A reflective architecture for the support of Aspect-Oriented Programming in Smalltalk.** Proceedings of the Aspect-Oriented Programming Workshop at ECOOP'99, Finlândia: Springer-Verlag, 1999.

BECKER, Christian, GEIHS, Kurt. **Quality of Service - Aspects of Distributed Programs**. Proceedings of the Aspect-Oriented Programming Workshop at ICSE'98. Kyoto (Japão), 1998.

BECKER, Ulrich, GEIER, Martin, HAUCK, Franz J., MEIER, Erich, RASTOFER, Uwe, STECKERMEIER, Martin. **AspectIX: A Middleware for Aspect-Oriented Programming.** Proceedings of the Aspect-Oriented Programming Workshop at ECOOP'98, Finlândia: Springer-Verlag, 1998.

BECKER, Ulrich. **D$^2$AL: A design-based Aspect language for distribution control.** Proceedings of the Aspect-Oriented Programming Workshop at ECOOP'98, Finlândia: Springer-Verlag, 1998.

BERGER, L., DERY, A.M, FORNARINO, M. **Interaction Between objets: an Aspect of object-oriented languages.** Proceedings of the Aspect-Oriented Programming Workshop at ICSE'98. Kyoto (Japão), 1998.

BIDAN C , ISSARNY, V. **Aster: A Framework for Sound Customization of Distributed Runtime Systems**. Proceedings of 16º IEEE International Conference on Distributed Computing Systems, p 586-593, www.irisa.fr/solidor/work/aster.html ,1996.

BLACK, A., HUTCHINSON, N. JUL, E., LEVY, H. **Fine-Grained Mobility in the Emerald System**. ACM Transactions on Computer Systems, vol 6., No. 1, 1988.

BLAIR, G. S., CHETWIND, A., LAKAS, A.,. **Specification and Verification of Real-Time properties using LOTOS and SQTL.** Proceedings of the 8º International Workshop on Software Specification and Design. P 75-84, PaderBorn, Alemanha, 1996.

BLAIR, Lynne; BLAIR S. Gordon. **A Tool Suíte to Support Aspect-Oriented Specification.** Proceedings of the Aspect-Oriented Programming Workshop at ECOOP'99, Finlândia: Springer-Verlag, 1999.

BLAIR, Lynne; BLAIR S. Gordon. **The Impact of Aspect-Oriented Programming on Formal Methods.** Proceedings of the Aspect-Oriented Programming Workshop at ECOOP'98, Finlândia: Springer-Verlag, 1998.

BÖLLERT, Kai. **Aspect-Oriented Programming Case Study: System management applicaction**. Fachhochschule Flensburg, 1998a.

BÖLLERT, Kai. **Implementing an Aspect Weaver in Smalltalk**. in the proceeding of STJA '98 - Smalltalk und Java in Industrie und Ausbildung, Erfurt (Alemanha), 1998b.

BÖLLERT, Kai. **On Weaving Aspects.** In the Proceedings of the Aspect-Oriented Programming Workshop at ECOOP'99, 1999.

BRAND, Per, HARIDI, Seif, MEHL, Michael, SCHEIDHAUER, Ralf, SMOLKA, Gert, VAN ROY, Peter. **Using mobility to make transparent distribution parcial**.

Proceedings of the Aspect-Oriented Programming Workshop at ECOOP'97, Finlândia: Springer-Verlag LNCS, 1997.

BUILDER. **Borland C++ Builder Home Page.** www.borland.com/bcppbuilder, 2001.

CAHILL, Vinn, DEMPSEY, John, y. **Aspects of System Support for Distributed Computing.** Proceedings of the Aspect-Oriented Programming Workshop at ECOOP'97, Finlândia: Springer-Verlag LNCS, 1997.

CAMPO, Marcelo, PRICE, Tom. **Luthier: A framework for building Framework-Visualization Tools.** Object-Oriented Application Frameworks. Mohamed Fayad, Ralph Johnson (eds.), John Wiley & Sons, Estados Unidos, 1998.

CHIBA, S., TATSUBORI, M., 1998. **Open Java 1.0 API and Specification.** http:://www.softlab.is.tsukuba.ac.jp/~mich/openjava, 1998.

CZARNECKI Krzysztof, EISENECKER, Ulrich. **Generative Programming: Methods, Tools, and Applications**. Addison-Wesley, 2000.

CZARNECKI, Krzysztof. **Dynamic COOL, a prototype implementation of a dynamic version of COOL in Smalltalk.** http://nero.prakinf.tu-ilmenau.de/~czarn/aop, 2000

DE MEUTER, Wolfgang. **Monads as a theorical foundation for AOP.** Proceedings of the Aspect-Oriented Programming Workshop at ECOOP'97, Finlândia: Springer-Verlag LNCS, 1997.

DELPHI. **Borland Delphi Home Page**. www.inprise.com/delphi, 2001.

FORTE. **Forte for Java.** http://www.sun.com/forte/ffj/, 2001

FRADET, Pascal, SÜDHOLT, Mario. **An Aspect language for robust programming**. Proceedings of the Aspect-Oriented Programming Workshop at ECOOP'99, Finlândia: Springer-Verlag, 1999.

GILBERT, John R., IRWIN, John, KICKZALES, Gregor, LAMPING, John, LOINGTIER, Jean-Marc, LOPES, Cristina Videira, MAEDA, Chris, MENDHEKAR, Anurag, SHPEISMAN, Tatiana. **Aspect Oriented Programming of Sparse Matrix Code**. Proceedings of International Scientific Computing in Object-Oriented Parallel Environments (ISCOPE), Springer Verlag: Estados Unidos, 1997.

GOURHANT, Y., LE NARZUL, J., MAKAPANGOU, M., SHAPIRO, M. **Fragmented Object for Distributed Abstractions**. IEEE Computer Society Press, Readings in Distributed Computing Systems, 1994.

HARRISON, William, OSSHER, Harold. **Subject-Oriented Programming (a critique of pure objects)**. Proceedings of the Conference on Object-Oriented Programming: Systems, Languages and Applications. 411-428, Washington, ACM, 1993.

HOLMES, David, NOBLE, James, POTTER, John, **Aspects of Synchronisation.** Proceedings of the Aspect-Oriented Programming Workshop at ECOOP'97, Finlândia: Springer-Verlag LNCS, 1997.

HOMBURG, P. TANEMBAUM, A.S., VAN STEEN, M. **The architetural design of Globe: a wide-area distributed system.** Technical Report IR-422, Vrije Universiteit, Amsterdam, 1997.

IRWIN, John, KICKZALES, Gregor, LAMPING, John, MENDHEKAR, Anurag, MAEDA, Chris, LOPES, Cristina Videira, LOINGTIER, Jean-Marc. **Aspect-Oriented Programming.** proceeding of ECOOP'97, Finland: Springer-Verlag, 1997.

JACKSON, M. A., ZAVE P. **Conjunction as Composition.** ACM Transactions on Software Engineering and Metodology II, p. 379-411, ACM Press, 1993.

JACKSON, M. A., ZAVE P. W**here do Operationms come from? A multiparadigm specification technique.** IEEE Transactions on Software Engineering XXII, p. 508-528, IEEE, 1996.

JBUILDER. **JBuilder Home Page.** www.borland.com/jbuilder, 2001.

JST. **JST Weaver Home Page**. www-src.lip6.fr/homepages/Lionel.Seiturier/JST/., 2001

KICKZALES, Gregor, LAMPING, John, MENDHEKAR, Anurag. **RG: A Case-Study for Aspect-Oriented Programming.** Technical Report SPL97-009 p97100044 Xerox Palo Alto Research Center, 1997.

KICZALES, Gregor, LEE, Arthur, LOPES, Cristina Videira, MURPHY, Gail. **Workshop on Aspect-Oriented Programming**. Proceedings of the Aspect-Oriented Programming Workshop at ICSE'98. Kyoto (Japão), 1998.

KNUDSEN, Jorgen Linskov. **Aspect-Oriented Programming in BETA using the Fragment System.** Proceedings of the Aspect-Oriented Programming Workshop at ECOOP'99, Finlândia: Springer-Verlag, 1999.

KYLIX. **Borland Kylix Home Page**. www.borland.com/kylix, 2001.

LEITE, Alexandre V. **Suporte à reflexão computacional em ambientes de desenvolvimento visual de software**. Dissertação de Mestrado, CPGCC - UFSC: Florianópolis, 2001.

LOPES, Cristina Isabel Videira. **D: a language framework for distributed programming.** Ph. D. Thesis, Northeast University, 1997.

LUNAU, Charlotte Pii. **Is Composition of Metaobjects = Aspect Oriented Programming.** Proceedings of the Aspect-Oriented Programming Workshop at ECOOP'98, Finlândia: Springer-Verlag, 1998.

MATSUOKA S., YONEZAWA A.. **Analysis of Inheritance Anomaly in Object-Oriented Concurrent Programming Languages.** In: G.Agha, A.Yonezawa, and P.Wegner [Ed.], Research Directions in Concurrent Object-Oriented Programming, The MIT Press, 1993

OMG. **Object Management Group: The Common Object Request Broker Architeture, Version 2.2**, 1998.

OPEN GROUP. **Open Group Home Page – DCE Portal**, http://www.opengroup.org/dce, 2001.

OSSHER, Harold, TARR, Peri. **Multi-dimentional separation of concerns in Hyperspace.** Proceedings of the Aspect-Oriented Programming Workshop at ECOOP'99, Finlândia: Springer-Verlag, 1999.

OSSHER, Harold, TARR, Peri. **Operation-Level Composition: A Case in (Join) Point.** Proceedings of the Aspect-Oriented Programming Workshop at ICSE'98. Kyoto (Japão), 1998.

SENTURIER, Lionel. **JST: An Object Syncronization Aspect for Java.** Proceedings of the Aspect-Oriented Programming Workshop at ECOOP'99, Finlândia: Springer-Verlag, 1999.