3
Uma Teoria de Aspectos

Este Capítulo descreve uma abordagem para tratar do problema da *necessidade de um framework conceitual unificador para a POA* apresentado no Capítulo 1.

Propomos uma *teoria de aspectos* – um framework conceitual para a programação orientada a aspectos que oferece uma terminologia consistente e uma semântica básica para pensar sobre um problema em termos dos conceitos e propriedades que caracterizam o estilo da POA como um paradigma emergente de desenvolvimento de software. Esses conceitos e propriedades já foram informalmente descritos pelos autores [73, 42, 45].

Nosso objetivo não é apresentar uma pesquisa sobre os conceitos existentes usados por diferentes abordagens da separação avançada de *concerns*, mas sim apresentar a definição de uma ontologia que, de acordo com nosso ponto de vista, inclui os conceitos e as propriedades essenciais para o suporte ao projeto de linguagens orientadas a aspectos e o desenvolvimento de sistemas de software orientados a aspectos. Nesse cenário informal, nossa teoria de aspectos consiste em uma descrição em linguagem natural das categorias dos principais conceitos, propriedades e regras às quais esses conceitos devem satisfazer, bem como um conjunto de modelos conceituais entidade-relacionamento [28] – seguindo a abordagem proposta pelo paradigma de Teoria-Modelo [117]. A teoria de aspectos pode ser usada para avaliar ferramentas e linguagens candidatas a serem orientadas a aspectos, além de guiar o projeto de novas linguagens orientadas a aspectos.

Nesta tese, a teoria de aspectos oferece um framework conceitual para um modelo no nível do projeto em conformidade com a UML (Capítulo 6) que especifica a semântica da linguagem de modelagem aSideML (Capítulo 5).

Organização do Capítulo

A Seção 3.1 apresenta a teoria de aspectos que, por razões históricas, chamamos de *modelo de aspectos*. A Seção 3.2 demonstra a usabilidade e a utilidade do modelo de aspectos em diversos contextos, com diferentes propósitos:

- para apresentar uma definição para linguagens orientadas a aspectos (Seção 3.2.1).
- para caracterizar algumas abordagens representativas para a separação avançada de *concerns* (Seção 3.2.2).
- para oferecer suporte ao projeto de uma linguagem orientada a aspectos (Seção 3.2.3).

A Seção 3.3 introduz os trabalhos relacionados.

3.1 O Modelo de Aspectos

Aspectos, *componentes*, *pontos de combinação*, *crosscutting* e *processos de combinação* são conceitos introduzidos por Kiczales et al. em seu trabalho seminal *Aspect-Oriented Programming* [73] e que, coletivamente constituem o coração do paradigma de orientação a aspectos. Além disso, duas propriedades, *quantificação* e *transparência* (*obliviousness*), foram propostas como propriedades necessárias para a POA [42]. Seguindo a idéia de adotar a POA como uma convergência possível para as tendências de pesquisa em SAdC [40], consideramos esses conceitos e propriedades, bem como a separação clara entre aspectos e componentes [81] – que chamamos de *dicotomia aspecto-base* – como elementos fundamentais do paradigma orientado a aspectos. Nós os adotamos como o framework conceitual principal que caracteriza tudo que é orientado a aspectos. Organizamos esses elementos em quatro modelos conceituais inter-relacionados: (i) o modelo de componentes, (ii) o modelo de pontos de combinação, (iii) o modelo de processo de combinação e (iv) o modelo *core* ou principal. Seguindo esse padrão, chamamos o modelo composto resultante de *modelo de aspectos* (Figura 3.1). Como uma primeira aproximação, definimos que uma *linguagem orientada a aspectos* é uma linguagem que oferece suporte ao modelo de aspectos.

Na Figura 3.1 existem relacionamentos com rótulo “adopts” (adota) para indicar que um *modelo de aspectos* poderá adotar ou escolher antecipadamente a natureza de seus submodelos (detalhes na Seção 3.2.3).

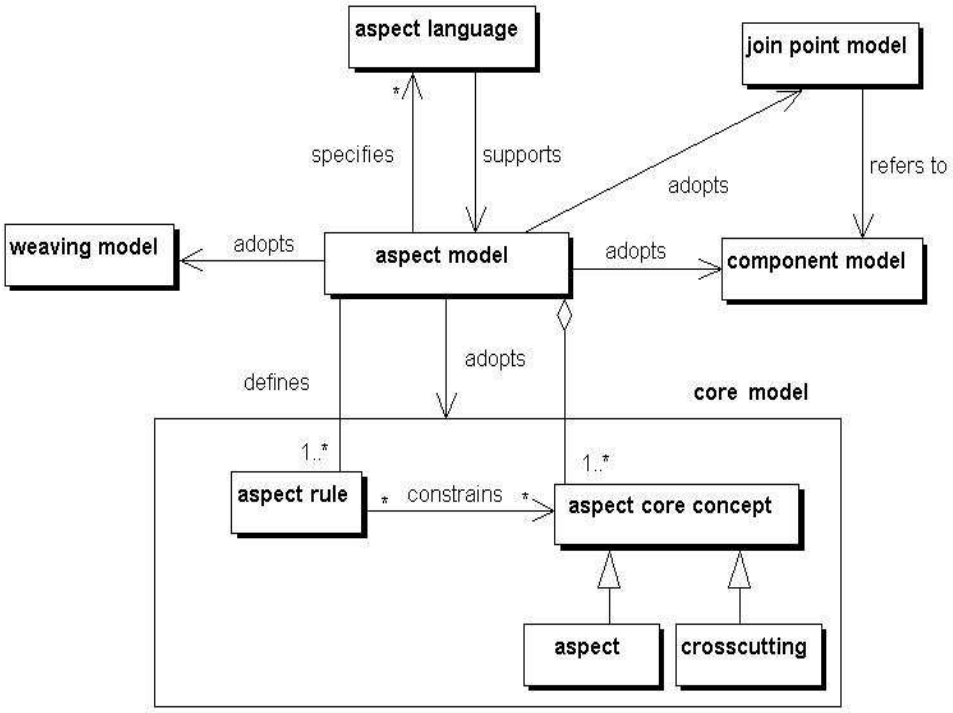

Figura 3.1: O *modelo de aspectos*.

Nas próximas seções, descreveremos os modelos conceituais (Seções 3.1.1 a 3.1.4) e discutiremos as propriedades da POA (Seção 3.1.5). Usamos diagramas entidade-relacionamento para ilustrar cada modelo conceitual em termos dos conjuntos de entidades e relações sobre esses conjuntos. Para cada conceito principal, apresentamos as definições existentes como um ponto de partida para fornecer nossa própria definição e terminologia, seguida por uma breve discussão e alguns exemplos.

3.1.1 O Modelo de Componentes

O *modelo de componentes* representa um framework conceitual usado para pensar em um problema e decompô-lo em termos de um determinado tipo de componente. Esse framework consiste em categorias de conceitos principais (*mecanismos de composição* e *componentes*), regras que restringem elementos dessas categorias e um conjunto de princípios gerais.

Definição.

Um componente é uma unidade da decomposição funcional do sistema, uma propriedade ou *concern* que pode ser encapsulada de forma clara em um

procedimento generalizado (ou seja, objeto, método, procedimento, API) [73]. Usamos o termo *componente* para denotar uma entidade nomeável que modulariza uma *parte funcional* do sistema de software, mais ou menos independente de outras partes, e que pode ser naturalmente composta com outros componentes usando os mecanismos de composição fornecidos.

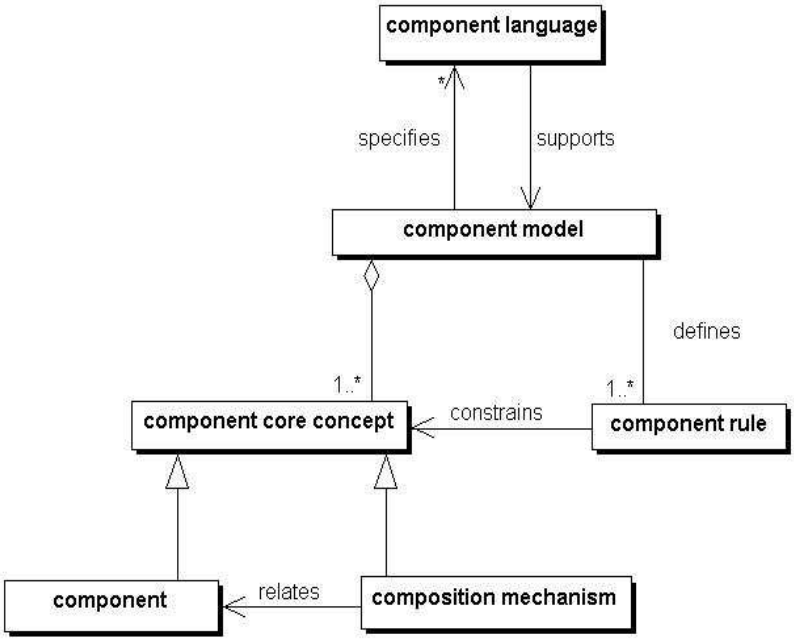

Figura 3.2: O modelo de componentes.

Discussão.

O modelo de componentes é uma parte fundamental do *modelo de aspectos*. O estado de ser um *concern* transversal é relativo a um determinado tipo de decomposição. O modelo de componentes define o tipo primário de decomposição e uma linguagem de componentes é usada para expressar de forma eficaz *concerns* funcionais. O modelo de componentes também restringe o projeto do modelo de pontos de combinação (Seção 3.1.2).

Um modelo de componentes pode ter o suporte de uma ou mais linguagens de componentes. A Figura 3.2 apresenta um modelo de dados para o modelo de componentes.

Exemplo.

Componentes podem ser concretizados por funções, procedimentos ou classes; os mecanismos de composição comuns são chamadas de função, chamadas de procedimentos e invocações de métodos. O *modelo de objetos*, ou seja, o framework conceitual para tudo aquilo orientado a objetos [16], é um modelo de componentes, em que os principais conceitos são objetos, classes e herança [137]. Uma regra típica[1] é que todos os objetos são uma instância de alguma classe. Os princípios gerais são abstração, encaspulamento, modularidade e herança [16]. Java e C++ são linguagens de programação orientadas a objetos uma vez que seu framework conceitual é o modelo de objetos, ou seja, as duas oferecem suporte ao modelo de objetos. No entanto, Java oferece suporte a herança simples enquanto C++ oferece suporte a herança múltipla, entre outras diferenças.

3.1.2 O Modelo de Pontos de Combinação

O *modelo de pontos de combinação* representa um framework conceitual usado para descrever os tipos de pontos de combinação de interesse e as restrições associadas a seu uso. Ele é altamente dependente do modelo de componentes adotado. A Figura 3.3 apresenta um modelo de dados para o modelo de pontos de combinação.

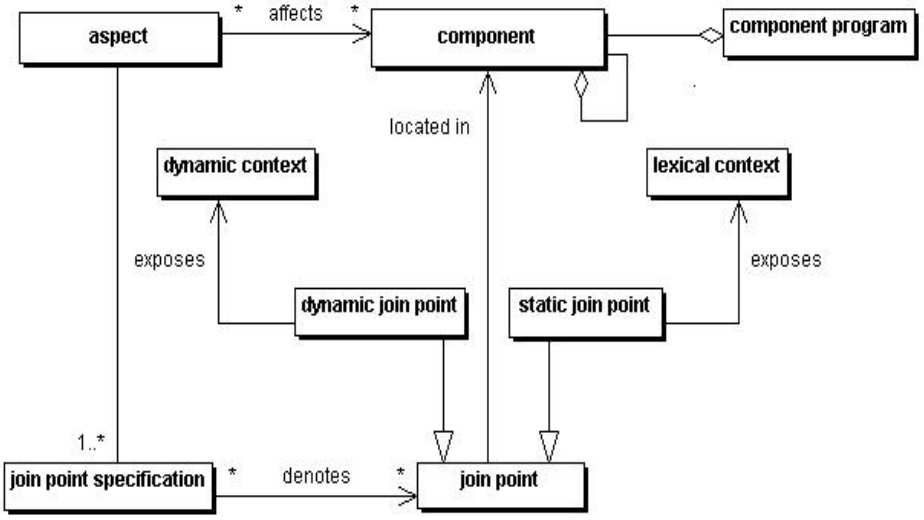

Figura 3.3: O modelo de pontos de combinação.

[1] para modelos baseados em classe

Definição.

Pontos de combinação são elementos da semântica da linguagem de componentes com os quais os aspectos interagem [73]. Usamos o termo *ponto de combinação* a fim de denotar um elemento relacionado à estrutura estática ou dinâmica de um programa de componentes, referenciado e possivelmente afetado por um aspecto. Um *ponto de combinação estático* (*static join point*) é um local na estrutura estática de um componente, enquanto que um *ponto de combinação dinâmico* (*dynamic join point*) é um local no comportamento dinâmico de um programa de componentes. Um ponto de combinação dinâmico possui uma *sombra estática* (*static shadow*) correspondente na parte estrutural do componente. Uma sombra estática é uma região no componente que representa um ponto de combinação dinâmico relacionado a tal componente.

Discussão.

O conceito de *ponto de combinação* refere-se a algum local relacionado a um componente, sob a perspectiva do aspecto. Os componentes "detêm" os locais reais, enquanto os aspectos os descrevem como pontos de combinação de interesse e os usam para afetar componentes com sua funcionalidade *crosscutting*. Os pontos de combinação podem expor outras informações relacionadas ao contexto em que aparecem – chamamos de *contexto de crosscutting*. A natureza das informações de contexto disponíveis depende do tipo de ponto de combinação: pode ser *léxica* ou *estática* (disponível a partir do texto do programa) ou *dinâmica* (disponível a partir da execução do programa). O combinador de aspectos combina aspectos e componentes nos pontos de combinação especificados pelo aspecto (Seção 3.1.4).

Há muitos locais na estrutura de componentes ou comportamento de componentes que podem ser usados como pontos de combinação, mas, na prática, somente um subconjunto é considerado útil [107]. As linguagens orientadas a aspectos devem definir seu conjunto de pontos de combinação levando em consideração sua linguagem de componentes correspondente. O subconjunto selecionado influenciará o conjunto de quantificação naquelas linguagens orientadas a aspectos (Seção 3.1.5).

Exemplo.

Se considerarmos o modelo de objetos um modelo de componentes, alguns possíveis pontos de combinação estáticos de interesse são classes, interfaces, métodos e atributos; alguns possíveis pontos de combinação dinâmicos são chamadas e execuções de métodos, instanciações, execuções de construtor, referências a campos e execuções do tratador. Para uma linguagem orientada a aspectos que adota outro modelo de componentes, o modelo de pontos de combinação será certamente diferente.

3.1.3 O Modelo Core

O objetivo da POA é oferecer suporte ao programador na separação entre *componentes* e *aspectos*, fornecendo mecanismos que possibilitam *abstraí-los* e *compô-los* a fim de produzir o sistema completo [73].

O modelo *core* representa um framework conceitual usado para descrever *aspectos* como um mecanismo de abstração e *crosscutting*, como um mecanismo de composição.

3.1.3.1 Aspectos

Definição.

Aspectos são definidos como *propriedades de sistemas que afetam componentes* e, mais especificamente, como *propriedades que afetam o desempenho ou a semântica de componentes de forma sistêmica* [73]. Também usamos o termo *aspecto* para denotar uma entidade nomeável de primeira classe que oferece uma representação modular para um *concern* transversal.

Discussão.

A modularização de um *concern* transversal envolve a provisão de um mecanismo de abstração – o aspecto – que localiza:

– a especificação de um conjunto particular de pontos de combinação, e

– extensões ou melhorias (*enhancements*)[2] que devem ser combinadas em pontos de combinação específicos.

Para estabelecer uma terminologia independente de linguagem, usamos o termo *característica transversal* (*crosscutting feature*) a fim de denotar qualquer melhoria comportamental ou estrutural especificada dentro do aspecto para afetar um ou mais componentes nos pontos de combinação especificados, e o termo *interface transversal* (*crosscutting interface*) para denotar o conjunto de pontos de combinação especificados dentro do aspecto (Figura 3.4).

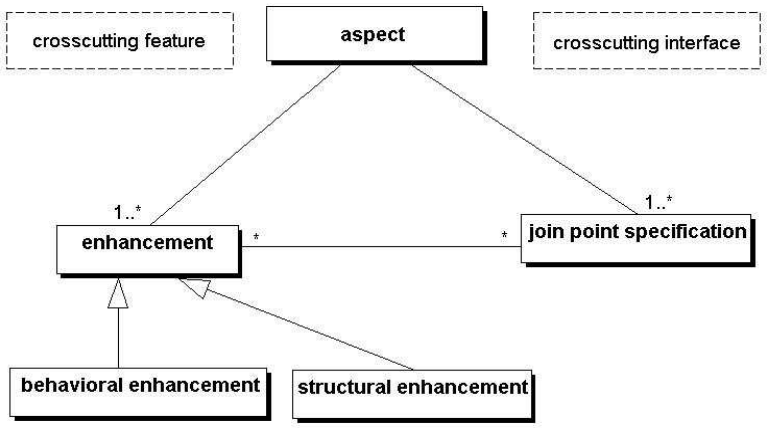

Figura 3.4: Aspecto.

Interfaces Transversais.

As *interfaces transversais* incorporam *especificações de pontos de combinação*, ou seja, elas descrevem os tipos de pontos de combinação de interesse para o aspecto, restritos pelo modelo de pontos de combinação adotado.

Características Tranversais.

As *características transversais* são atributos e operações que descrevem melhorias na estrutura e no comportamento de componentes. Essas melhorias podem *adicionar* uma nova estrutura e comportamento para um

[2]O termo *melhoria* ou *enhancement* é usado para denotar qualquer modificação feita nos componentes, não necessariamente de forma monotônica: os aspectos também podem excluir comportamentos.

ou mais componentes, *refinar* ou até mesmo *redefinir* o comportamento existente.

Sempre que necessário, usamos o termo *característica transversal estrutural* para denotar melhorias estruturais, e o termo *característica transversal comportamental* para denotar melhorias comportamentais. Mais adiante fazemos a distinção entre *características transversais estáticas* e *características transversais dinâmicas*, ou seja, as características transversais que usam pontos de combinação estáticos e dinâmicos, respectivamente.

	Crosscutting Estático	Crosscutting Dinâmico
Estrutural	atributos	
Comportamental	operações	operações

Tabela 3.1: Características transversais.

Exemplo.

O aspecto `Logging` não é apenas o código que registra algum dado específico. O *concern* transversal – e portanto, o aspecto – é que *um determinado conjunto de pontos deve registrar alguns dados específicos.* A linguagem AspectJ [75] oferece suporte a esta visão: os aspectos são definidos como elementos de implementação de primeira classe que incorporam *declarações de pointcuts* para expressar especificações de pontos de combinação, e *declarações de advice/introduction* para expressar características transversais.

3.1.3.2 Crosscutting

Definição.

Crosscutting é definido como um fenômeno observado sempre que duas propriedades programadas devem compor de forma diferente e mesmo assim serem coordenadas [73]. Usamos o termo *crosscutting* para denotar o mecanismo de composição usado para compor aspectos e componentes nos pontos de composição designados. Os aspectos podem afetar um ou mais componentes, possivelmente afetando sua estrutura e comportamento. Além disso, ampliamos o uso do termo para denotar também um *relacionamento* genérico de um aspecto para um ou mais componentes.

Discussão.

A partir da primeira definição citada anteriormente [73], temos que os aspectos podem afetar outros aspectos. Contudo, nessa teoria core, ficamos restritos ao *crosscutting* como um mecanismo de composição entre aspectos e componentes. Ademais, fazemos a distinção das questões a seguir ao considerar *crosscutting* um mecanismo de composição.

Direção de Crosscutting.

A direção de crosscutting é sempre de aspectos para componentes. O termo *herança reversa* (*reverse inheritance*) tem sido usado para denotar o mecanismo de composição que relaciona aspectos e classes, uma vez que a direção da composição é o oposto da herança convencional. Apesar de o modelo de objetos ser normalmente a opção para o modelo de componentes, preferimos usar o termo *crosscutting* em vez de herança reversa para preservar a independência do *modelo de aspectos* em relação a seus submodelos.

Dimensões de Crosscutting.

Crosscutting pode ser aplicado *homogeneamente*, fornecendo o mesmo conjunto de melhorias feitas em um ou mais componentes, ou *heterogeneamente*, quando subconjuntos de melhorias são aplicados ao mesmo tempo a diferentes tipos de componentes. Chamamos o primeiro de *crosscutting vertical* e, o segundo, *crosscutting horizontal*.

Cardinalidade de Crosscutting.

Os aspectos podem afetar um ou mais componentes simultaneamente. Os componentes podem ser afetados por um ou mais aspectos de forma simultânea.

Natureza de Crosscutting.

Chamamos de *crosscutting estrutural estático*, ou simplesmente, *crosscutting estático*, o tipo de *crosscutting* que usa pontos de combinação estáticos e afeta a estrutura estática dos componentes. Chamamos de *crosscutting comportamental dinâmico*, ou simplesmente, *crosscutting dinâmico*,

o tipo de *crosscutting* que usa pontos de combinação dinâmicos e, portanto, afeta o comportamento dinâmico dos componentes.

Crosscutting sensível ao contexto.

Os aspectos podem melhorar os componentes em situações especiais, por exemplo, quando esses componentes estão relacionados a ou interagem com determinado(s) componente(s). Chamamos de *crosscutting sensível ao contexto* o tipo de *crosscutting* que usa as informações de contexto como um critério para a composição.

Exemplo.

Uma característica importante da programação orientada a objetos é a *herança* [130]. Entre outros usos, a herança é considerada um mecanismo para a composição estrutural que permite novas definições de classe que se basearão em classes existentes. Uma nova classe herda as propriedades existentes de seus parentes e pode adicionar novas propriedades, refinar ou redefinir as propriedades herdadas.

Uma característica importante da programação orientada a aspectos é *crosscutting*, um mecanismo de composição que permite aos aspectos serem combinados com os componentes existentes em pontos de combinação bem definidos. Um aspecto afeta um ou mais componentes e pode adicionar novas propriedades, refinar ou redefinir propriedades existentes.

3.1.4 O Modelo de Processo de Combinação

O *modelo de processo de combinação* representa um framework conceitual usado para descrever os tipos de mecanismos de combinação (*weaving*).

Definição.

Processo de combinação é o processo de compor aspectos e componentes relacionados através de *crosscutting* em pontos de combinação específicos. O termo *combinador de aspectos* (*aspect weaver*) designa a ferramenta que compõe aspectos e componentes [73].

Discussão.

O modelo de processo de combinação é parte de nosso framework conceitual uma vez que pode ser útil para interpretar certas propriedades e/ou características de linguagem. Por exemplo, é normalmente útil saber se alguma linguagem de componentes precisa ser interpretada ou se também pode ser compilada. Nos próximos parágrafos, identificamos e apresentamos algumas facetas importantes do processo de combinação.

Entradas do processo de combinação.

Um combinador de aspectos pode trabalhar com código-fonte, bytecode ou código objeto.

Saídas do processo de combinação.

Um combinador de aspectos pode gerar código-fonte, bytecode ou código-objeto.

Estratégia de processo de combinação.

Um combinador de aspectos pode fornecer modificação local (*in-place modification*) ou migração de cliente (*client migration*) [105]. A *modificação local* é destrutiva, ou seja, o código componente original não está mais disponível depois do processo de combinação. A *migração de cliente* significa que o componente original e as versões combinadas estão disponíveis. Em vez de alterar permanentemente o componente original, são criados novos clientes que se referem a um novo componente combinado.

Tempo de Combinação.

O processo de combinação pode ser *estático* (tempo de carregamento ou compilação) ou *dinâmico* (tempo de execução). O *processo de combinação estático* (*static weaving*) é uma tecnologia de combinação na qual o programa de componente e o programa de aspecto são mesclados em uma nova versão dos códigos fonte, antes ou durante a compilação. O *processo de combinação em tempo de carregamento* (*load-time weaving*) é um tipo especial de tecnologia de combinação que não requer o código-fonte. O programa de aspecto pode ser combinado ao programa componente no formato binário

usando instrumentação de código. O *processo de combinação dinâmico* (*dynamic weaving*) é uma tecnologia de combinação que permite que os aspectos sejam combinados e separados durante a execução.

Exemplo.

Muitas ferramentas de combinação para POA baseiam-se no processo de combinação estático, ou sobre o código fonte ou diretamente no código-objeto. O compilador AspectJ (`ajc`) é um combinador de aspectos. Em suas primeiras versões, ele trabalhava realizando um pré-processamento de um conjunto de arquivos-fonte .java que continham descrições de aspectos e gerando arquivos-fonte .java ou arquivos em java .class. Agora, AspectJ também oferece suporte ao processo de combinação de bytecode (*bytecode weaving*), ao permitir como entrada arquivos em java .class.

AspectJ realiza modificação local, que altera permanentemente os componentes originais, enquanto Hyper/J realiza migração de cliente.

3.1.5 Propriedades

A programação orientada a aspectos incorpora alguns princípios bem conhecidos e novas propriedades. Os dois princípios mais importantes da POA são a *separação de concerns* e a *modularidade*. A separação de *concerns* é alcançada por meio da separação clara entre aspectos e componentes. A modularidade é alcançada por meio de um novo tipo de unidade modular para *concerns* transversais , o aspecto. Discutimos esses dois princípios a seguir em *dicotomia aspecto-base*. Além disso, duas propriedades, *quantificação* e *transparência*, foram propostas como as principais características da POA e foram usadas para definir se uma linguagem é orientada a aspectos ou não [45].

3.1.5.1 Dicotomia aspecto-base

A *dicotomia aspecto-base* significa a adoção de uma distinção clara entre componentes (ou *componentes base*) e aspectos. Quando a dicotomia é válida, levando em consideração os princípios de separação de *concerns* e de modularidade, temos que:

– os sistemas são decompostos em componentes e sistemas;

– os aspectos modularizam *concerns* transversais ;
– os componentes modularizam *concerns* funcionais; e
– os aspectos devem ser explicitamente representados e de modo separado dos componentes e de outros aspectos.

Em nosso framework conceitual, a dicotomia aspecto-base é considerada uma característica essencial da POA.

3.1.5.2 Transparência

A transparência (*obliviousness*) é o ato ou o efeito de deixar transparente, no sentido de poder ser esquecido ou passar despercebido. No contexto da POA, a transparência é a idéia de que os componentes não precisam ser especificamente preparados para receber as melhorias proporcionadas pelos aspectos [42]. Quando há a propriedade de transparência, temos que:

– os componentes não têm ciência de que os aspectos irão afetá-los; e
– os programadores não precisam de esforço adicional enquanto implementam a funcionalidade dos componentes para fazer a POA funcionar e comprovar seus benefícios [42].

Em nosso framework conceitual, adotamos a transparência como uma propriedade essencial da POA, que será usada como uma medida informal que indica a utilidade de sistemas orientados a aspectos.

Discussão.

A transparência é considerada a característica que torna a POA especial [45]. O uso de aspectos para modularizar *concerns* transversais com suporte da propriedade de transparência melhora a separação dos *concerns* no desenvolvimento de software e simplifica a análise, o projeto e a implementação de componentes.

Nos primórdios da POA acreditava-se que, quanto mais transparentes, melhores os sistemas orientados a aspectos [42]; no entanto, a transparência total é um objetivo difícil de alcançar. Nesse contexto, *intimidade* (*intimacy*) é definida como o esforço adicional necessário para preparar os componentes para os aspectos [40].

3.1.5.3 Quantificação

Quantificação é definida como a capacidade de escrever declarações unitárias e separadas que melhoram muitos lugares não-locais no sistema de programação. A quantificação proporciona a capacidade de declarar coisas como: "*em programas P, sempre que ocorrer a condição C, faça a ação A*" [45].

Quando há a propriedade de quantificação, temos que:

– os aspectos podem afetar um número arbitrário de componentes simultaneamente.

Em nosso framework conceitual, adotamos a quantificação como uma propriedade essencial da POA.

Discussão.

A capacidade de quantificar em sistemas da POA está relacionada ao suporte ao raciocínio declarativo sobre diversos tipos de elementos que pertencem a um programa de componente, por exemplo, campos, conjuntos de campos, métodos, chamadas de método etc. Esse suporte requer algum tipo de linguagem de quantificação que permita a expressão das declarações quantificadas. Essas sentenças podem conter uma ou mais *variáveis de quantificação* que serão ligadas a valores em algum universo de discurso, e uma *sentença aberta* que pode ser avaliada ou aplicada nesse contexto.

Em nosso framework, definimos como o universo de discurso os elementos que pertencem à estrutura de um programa (elementos estáticos) ou à execução de um programa (elementos dinâmicos); portanto, as variáveis de quantificação podem ser associadas aos elementos estáticos (*quantificação estática*) ou elementos dinâmicos (*quantificação dinâmica*). As sentenças abertas correspondem a características transversais comportamentais (Seção 3.1.3.1).

Na quantificação estática, as variáveis de quantificação estão associadas a elementos como classes, campos, métodos etc. Na quantificação dinâmica, as variáveis de quantificação estão associadas a elementos como objetos, criação de objetos, parâmetros reais, conjuntos de campos, chamadas de métodos etc.

Exemplo.

Na linguagem AspectJ, uma declaração de *advice* especifica a *ação A* que deverá ser realizada sempre que alguma *condição C* surgir durante a execução do *programa P*. A condição é especificada por um predicado chamado de *designador de pointcut*. Por exemplo, para expressar uma declaração como "*para cada chamada a um método público, alguns dados particulares deverão ser registrados*" em AspectJ, o programador pode definir o seguinte trecho de código dentro de um aspecto:

```
pointcut publicCall():     // pointcut designator
    call(public *.*(..));  // condition C
after(): publicCall() {    // advice
   log particular data ... // action A
}
```


3.2 Utilização do Modelo de Aspectos

O principal benefício de ter um framework conceitual como o *modelo de aspectos* é fornecer suporte para a avaliação das abordagens existentes, bem como para o desenvolvimento de novos métodos e ferramentas com base em uma terminologia unificada, conceitos e propriedades definidas no framework. A Tabela 3.2 resume os conceitos e as propriedades que constituem o *modelo de aspectos*.

Modelo de Componentes	componente, mecanismo de composição, regra de componente, linguagem de componente
Modelo de pontos de combinação	ponto de combinação, ponto de combinação estático, ponto de combinação dinâmico, contexto estático, contexto dinâmico, sombra estática
Modelo do processo de combinação	combinador de aspectos, processo de combinação, processo de combinação estático, processo de combinação dinâmico, estratégia do processo de combinação
Modelo principal	*aspecto*, interface transversal, característica transversal, característica transversal estrutural, característica transversal comportamental, *crosscutting*, *crosscutting* estático, *crosscutting* dinâmico, *crosscutting* horizontal, *crosscutting* vertical
Propriedades	dicotomia aspecto-base, quantificação, transparência

Tabela 3.2: Conceitos e propriedades definidos no *modelo de aspectos*.

Nesta Seção, o *modelo de aspectos* é usado para fornecer uma definição para *linguagens orientadas a aspectos* (Seção 3.2.1) e para caracterizar quatro abordagens representativas de SAdC (Seção 3.2.2). No Capítulo 5, ilustramos como o *modelo de aspectos* pode ser usado para oferecer suporte ao projeto da linguagem de modelagem `aSide`.

3.2.1 Linguagens Orientadas a Aspectos

Definição.

Uma *linguagem orientada a aspectos* (LOA) é uma linguagem que oferece suporte ao modelo de aspectos e, portanto, satisfaz os requisitos a seguir:

- adoção de um modelo de componentes;
- adoção de um modelo de pontos de combinação;
- adoção de um modelo de processo de combinação;
- adoção de alguma abstração nomeável a fim de modularizar *concerns* transversais;
  - provisão de algum meio para especificar pontos de combinação (interfaces transversais);
  - provisão de algum meio para especificar melhorias a serem combinadas em pontos de combinação (características transversais);
- suporte para a dicotomia aspecto-base;
- suporte para algum tipo de quantificação na estrutura ou no comportamento dos componentes; e
- suporte a algum nível de transparência de componentes.

Se nos abstrairmos do modelo de componentes (já que o modelo de pontos de combinação depende dele) e do modelo de processo de combinação, a definição anterior pode ser resumida em uma equação para "orientação a aspectos", como se segue:

$$\begin{aligned} \text{orientação a aspectos} \;=\; & \text{aspectos} + \textit{crosscutting} + \\ & \text{modelo de pontos de combinação} + \\ & \text{transparência} + \text{quantificação} + \\ & \text{dicotomia aspecto-base} \end{aligned} \tag{3-1}$$

Discussão.

As linguagens orientadas a aspectos podem ser *linguagens específicas a domínio* ou *linguagens de propósito geral*. Nos dois casos, elas devem fornecer uma interpretação para os elementos do *modelo de aspectos*.

Exemplo.

AspectJ [75] e AspectC [33] são linguagens de programação orientadas a aspectos de propósito geral, uma vez que seu framework conceitual é o *modelo de aspectos*. No entanto, AspectJ adota o modelo de objetos como seu modelo de componentes (Java como a linguagem de componentes) enquanto AspectC adota o modelo procedimental (C como a linguagem de componentes).

3.2.2 Avaliação de Características de Linguagens

Nesta Seção, usamos o *modelo de aspectos* para analisar quatro abordagens de SAdC apresentadas no Capítulo 2 – AspectJ, Hyper/J, Filtros de Composição e Demeter/DJ – a fim de avaliar se são orientadas a aspectos ou não. A Tabela 3.3 apresenta um resumo das interpretações resultantes.

3.2.2.1 AspectJ

AspectJ [75] é uma extensão orientada a aspectos a Java que oferece suporte à programação orientada a aspectos de propósito geral. AspectJ foi desenvolvida pela equipe da Xerox PARC que propôs as idéias paradigmáticas da POA e boa parte de sua terminologia [73]. Em 2002, AspectJ foi transferida da PARC para um projeto de código aberto, Eclipse [11].

Terminologia.

Programação orientada a aspectos, aspecto, ponto de combinação, *weaving*, *pointcut*, *advice,* declaração *inter-type*, *crosscutting*.

	AspectJ	Hyper/J	Composition Filters	DJ
Modelo de componentes	modelo de objetos	modelo de objetos	modelo de objetos	modelo de objetos
Componente Linguagem de componentes	objeto Java	objeto Java	objeto orientada a objetos	objeto Java
Modelo de pontos de combinação				
Dinâmicos Estáticos	pontos na execução classes	... classes, atributos, métodos	envio/ recepção de mensagens ...	nós em grafo de chamada nós em grafo de classe
Modelo Core				
Aspecto interface transversal característica transversal	aspecto pointcuts introduction, advice	hyperslice? hypermodule atributos métodos	filtro expressões de filtro wrappers	método adaptativo estratégias transversais visitante adaptativo
Modelo de combinação	estático	estático	estático e dinâmico	estático
Dicotomia aspecto-base	sim	não	sim	sim
Quantificação	sim	sim	sim	sim
Transparência	sim	sim	sim	sim

Tabela 3.3: Interpretações usando o *Modelo de Aspectos*.

3.2.2.2 DJ

A biblioteca DJ (Demeter/Java) [84] é um pacote Java que oferece suporte à Programação Adaptativa (PA) [82], uma das primeiras abordagens de SAdC. A PA oferece suporte à separação de *concerns* comportamentais (métodos, estratégias) a partir de *concerns* estruturais (grafos de classes), no contexto de software orientado a objetos.

Terminologia.

Programação adaptativa, grafo de classes, método adaptativo, estratégia transversal, visitante adaptativo.

Discussão.

A PA é considerada um caso especial da POA, no qual os componentes são expressáveis em termos de grafos, e aspectos (*métodos adaptativos*) referem-se aos grafos e os afetam usando *estratégias transversais* e *visitantes adaptativos*. De acordo com nosso framework conceitual, a biblioteca DJ oferece suporte à POA.

3.2.2.3 Filtros de Composição

A abordagem dos Filtros de Composição (FCs) [3, 15] é uma extensão ortogonal e modular ao modelo de objetos a fim de lidar com os problemas da modelagem orientada a objetos e para aumentar a adaptabilidade e a reusabilidade em sistemas orientados a objetos. Na abordagem de FC, os aspectos são expressos como *filtros*. Um *filtro* é definido como uma função que manipula as mensagens recebidas e enviadas pelos objetos [15]. Os filtros são independentes da linguagem.

Terminologia.

Filtros de Composição, filtro de entrada, filtro de saída, tipo de filtro, interface de filtro, elemento de filtro, classe interna, condição, seletor, sobreposição, filtro de *dispatch*.

Discussão.

Os FCs são considerados um caso especial da POA, no qual os filtros envolvem (*wrap*) os componentes base a fim de fornecer o comportamento *crosscutting*. De acordo com nosso framework conceitual, a abordagem de FC oferece suporte à POA.

3.2.2.4 Hyper/J

Hyper/J [64] é uma ferramenta desenvolvida no Centro de Pesquisa T.J. Watson da IBM para oferecer suporte à Separação Multidimensional de Concerns (SMDdC) [132], uma evolução do trabalho precursor de programação orientada a sujeitos [59]. Hyper/J permite a modularização e

a composição de *concerns* sem a exigência de extensões especiais de linguagens. Hyper/J usa hyperslices – conjuntos de pacotes de Java, classes, métodos, campos etc. – para modularizar qualquer tipo de *concern*.

Terminologia.

Separação multidimensional de concerns, *hyperspace*, *hyperslice*, *hypermodule*, mapa de *concerns*, unidades correspondentes, *merge*, sobreposição (*override*).

Discussão.

Hyper/J permite a composição de vários modelos de objetos separados; ele não requer uma base distinta e não oferece suporte à dicotomia aspecto-base. Além disso, hyperslices não conseguem modularizar *concerns* transversais , isto é, eles não localizam as interfaces transversais e características transversais . Portanto, de acordo com nosso framework conceitual, Hyper/J não oferece suporte à POA.

3.2.3 Suporte ao Projeto de Linguagens

O *modelo de aspectos* agrega os conceitos e as propriedades essenciais que dão suporte ao projeto de linguagens orientadas a aspectos. Esses conceitos essenciais podem ser vistos como candidatos a abstrações de nível mais alto para linguagens orientadas a aspectos. Como se trata de um framework conceitual genérico, o modelo de aspectos precisa ser instanciado para ser usado. Isso significa que o projetista de uma nova linguagem deve, inicialmente:

- adotar um modelo de componentes;
- adotar um modelo de pontos de combinação adequado;
- adotar um modelo de processo de combinação;
- oferecer representações adequadas para os elementos dos modelos adotados;
- fornecer uma semântica clara para *crosscutting*;
- fornecer algum meio de oferecer suporte à quantificação.

A lista anterior de tarefas de projeto é apenas uma tentativa e deverá ser refinada à medida em que o *modelo de aspectos* for utilizado para diferentes modelos de componentes de vários paradigmas. Neste trabalho, ela será usada para orientar o projeto de `aSideML`, uma linguagem de modelagem orientada a aspectos (Capítulo 5).

3.3 Trabalhos Relacionados

Há alguns trabalhos que pesquisam frameworks unificadores para DSOA. Este trabalho bem como seus trabalhos relacionados partem de abordagens orientadas a aspectos bem estabelecidas no nível de implementação, e procuram abstrair conceitos e propriedades comuns a essas abordagens.

Na Universidade de Lancaster, Mehner e Rashid buscam interfaces padrão para a inspeção em tempo de execução de programas orientados a aspectos [96]. Mehner e Rashid argumentam que essa interface padrão deve estar centrada em uma base ou núcleo comum para POA. Em [97], eles apresentaram o estado atual de `GEMA`, seu modelo genérico para a POA.

Nagy, Aksit e Bergmans defendem que os mecanismos de composição de aspectos são uma característica importante das LOAs; em [103] eles apresentam os Grafos de Composição (GC), um modelo genérico que permite a descrição uniforme e a comparação de diferentes mecanismos de composição orientados a aspectos.

Em [95], Masuhara e Kiczales oferecem um framework para modelar a semântica básica de cinco tecnologias de desenvolvimento de software orientado a aspectos e seus mecanismos. Eles tentam caracterizar quais propriedades de cada mecanismo permitem a modularidade *crosscutting*, em oposição à modularidade estruturada em blocos ou hierárquica. Uma propriedade crítica de seu framework é que ele modela os pontos de combinação como pontos existentes no resultado do processo de combinação em vez de em um dos programas de entrada. Ademais, o trabalho de Masuhara e Kiczales destaca-se por outras duas razões. Primeiro, ele trata componentes e aspectos uniformemente, assumindo que não há linguagem de componentes primária nem dicotomia aspecto-base, o que torna o seu framework adequado para caracterizar Hyper/J como uma ferramenta orientada a aspectos. Segundo, ele permite a descrição de aspectos que afetam outros aspectos naturalmente.

Ultimamente, as propriedades de transparência e dicotomia aspecto-base têm recebido menor atenção na caracterização de abordagens de

DSOA. As vantagens de uma dicotomia aspecto-base (por exemplo, simplificação conceitual) podem ser superadas por um tratamento uniforme dado aos módulos, como é proposto em abordagens de separação multidimensional de concerns. Nesse contexto, a teoria de aspectos descrita aqui pode ser revista no futuro para relaxar algumas restrições expressas em sua equação (dicotomia, por exemplo), e se tornar mais abrangente.

3.4 Considerações Finais

Neste Capítulo, buscamos inspiração em Wegner [137], que nos primórdios do paradigma de orientação a objetos tentou esclarecer as dimensões de projeto de linguagens orientada a objetos. Da mesma forma, a comunidade de POA precisa ser capaz de excluir as linguagens não-POA da definição de POA e estabelecer um vocabulário para falar sobre as diferenças das diversas linguagens de POA.

O modelo de aspectos proposto aqui se concentra no desenvolvimento de um framework conceitual a fim de explicar as abstrações para o DSOA, bem como a relação entre a abstração de aspectos e outros tipos de abstrações que objetivam fornecer suporte à descrição de *concerns transversais*. Será fornecida uma descrição detalhada da semântica de *crosscutting*, bem como de outros mecanismos no modelo, assim que for apresentada uma instanciação para a teoria, no Capítulo 6.

Apesar de as abordagens avaliadas aqui dependerem do modelo de objetos, decidimos abstrair sobre o tipo de componente afetado pelos aspectos. Isso permite que outros modelos sejam adotados no futuro, como por exemplo, o modelo de agentes [120].