4
Modelagem Orientada a Aspectos

Este Capítulo discute questões relacionadas à Modelagem Orientada a Aspectos (MOA) com o objetivo de definir os requisitos para ferramentas e linguagens de modelagem orientadas a aspectos.

A modelagem de software contemporânea adota uma perspectiva *orientada a objetos*, na qual os principais blocos básicos dos sistemas de software são *objetos* e *classes*. Essa perspectiva demonstrou sua utilidade em termos da compreensibilidade, extensibilidade e reusabilidade em vários domínios. Depois de uma variedade de linguagens de modelagem orientadas a objetos criadas nos anos 90, foi introduzida a UML (Unified Modeling Language) [17, 116, 134] como uma linguagem de modelagem de propósito geral padrão, usável em um rico espectro de domínios de aplicações. A UML oferece visões inter-relacionadas, complementares e separadas a fim de descrever um sistema orientado a objetos, desde seus requisitos até sua implementação. Algumas dessas visões podem ainda ser separadas em modelos comportamentais e estruturais. Portanto, UML usa o princípio de separação de *concerns* em vários níveis diferentes, mas a realização desse princípio é ligeiramente diferente daquela promovida pela POA.

A POA embasa-se na idéia de que para qualquer perspectiva e blocos básicos usados, *concerns* transversais podem estar espalhados e entrelaçados em vários elementos. Assim, propõe o uso de aspectos para modularizar esses elementos transversais a fim de melhorar a compreensão, além de facilitar a reutilização e a evolução.

Sistemas de software orientados a aspectos também precisam ser visualizados e comunicados entre os desenvolvedores. Ademais, suas propriedades precisam ser especificadas de modo a permitir a análise de modelos, e a evitar o entrelaçamento e espalhamento nas especificações. A *Modelagem Orientada a Aspectos* (MOA) [142, 145] é uma parte crítica do DSOA que se concentra em notações e técnicas para a especificação, representação, visualização e comunicação de soluções orientadas a aspectos através do percurso que leva da Engenharia de Requisitos Orientada a Aspectos (EROA)

até POA.

Nesse contexto, algumas questões importantes que serão tratadas neste Capítulo são:

- Como a MOA influencia a prática atual da modelagem de software?
- Quais são as principais questões da MOA?
- Quais são os requisitos das linguagens de modelagem orientadas a aspectos?
- Como podemos usar UML para oferecer suporte à MOA?

4.1 Avanços na Modelagem de Software

No Capítulo 2, a evolução da Computação durante os últimos trinta anos foi descrita em termos dos avanços da separação de *concerns* no desenvolvimento de programas de software complexos. Essa evolução também pode ser caracterizada por uma *mudança de paradigma de algoritmos para interação* [138]:

> A mudança da tecnologia baseada em mainframes, procedimental e orientada a lotes (*batch*) dos anos 60 para a tecnologia baseada em estações de trabalho distribuídas, orientada a interfaces gráficas e a objetos de hoje, e para agentes computacionais pervasivos, embutidos e móveis no futuro próximo é fundamentalmente uma mudança de algoritmos para interação.

Os sistemas de software atuais oferecem serviços tangíveis ao usuário, não só transformação de dados, e a interação é pervasiva. Os *concerns* podem variar de noções de alto nível como segurança e qualidade de serviço até noções de baixo nível como *caching* e *buffering*. Eles podem ser funcionais, como *features* ou regras de negócio, ou não-funcionais (sistêmicos), como sincronização e gerenciamento de transações.

A necessidade de novas notações para a modelagem de software surge a partir da complexidade cada vez maior dos sistemas de software e dos tipos de *concerns* de que tratam. A ênfase da programação estruturada no trio seqüência-alternância-iteração de estruturas de controle resultou em *fluxogramas estruturados*. O design *top-down* resultou na *diagramação hierárquica baseada em módulos*. A tecnologia de banco de dados promoveu os *diagramas entidade-relacionamento* para a modelagem de dados.

A perspectiva orientada a objetos é a escolha atual para a modelagem de software, na qual os principais blocos básicos dos sistemas de software são *objetos* encapsulados que interagem a fim de conseguir realizar algumas tarefas e suas *classes*, organizadas em hierarquias de classes. Com a programação orientada a objetos vieram *diagramas de classes* e um conjunto de diagramas para oferecer suporte à visão de um sistema de "objetos que interagem": *diagramas de caso de uso*, *diagramas de seqüência* e *diagramas de colaboração* modelam interações internas e externas entre atores e objetos que são importantes para caracterizar o comportamento dinâmico do sistema.

A principal diferença entre essas abordagens de modelagem não reside apenas nos tipos de elementos que capturam e representam. De acordo com [109], a diferença é de *foco*, *representação*, *dedicação*, *visualização* e *seqüência*, de forma que uma linguagem de modelagem "orientada" normalmente prescreve que:

- Uma dimensão seja adotada como fundamental para a modelagem, enquanto outras dimensões são cobertas principalmente para definir o contexto daquela fundamental (foco);
- Uma dimensão seja representada explicitamente, outras apenas implicitamente (representação);
- Alguns elementos sejam tratados por construções de modelagem dedicadas, enquanto outros são menos precisamente tratados pelos genéricos (dedicação);
- Alguns elementos sejam visualizados em diagramas, outros somente registrados textualmente (visualização);
- Algumas dimensões sejam capturadas antes de outras (seqüência).

A abordagem orientada a aspectos para o desenvolvimento de software promete vários benefícios com base em benefícios já oferecidos pela abordagem orientada a objetos. Contudo, como ela influencia a prática atual da modelagem de software? Quais são os avanços introduzidos?

4.1.1 Modelagem Orientada a Aspectos

A modelagem orientada a aspectos incorpora as idéias promovidas pela POA: para qualquer perspectiva e blocos básicos usados, os *concerns* transversais podem estar espalhados por diferentes modelos e entrelaçados com outros modelos. Logo, a MOA propõe o uso de outros elementos que nomeiam e localizam esses *concerns* transversais e o uso de outras visões a fim de melhorar a compreensão, além de facilitar a reutilização e a evolução.

A fim de oferecer suporte à MOA (orientada a objetos ou não), os mecanismos de composição e os blocos básicos que lidam explícita e separadamente com os *concerns* transversais de um sistema devem coexistir com blocos básicos convencionais e mecanismos de composição.

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

Ao tratar dessas novas questões relacionadas a foco, representação, dedicação e visualização, a abordagem orientada a aspectos para a modelagem promove avanços nas abordagens de modelagem convencionais.

4.1.2 Uma Visão Conceitual de Aspectos e Crosscutting

Modelagem conceitual pode ser definida como o processo de organização de nosso conhecimento sobre um domínio de aplicação em *rankings* hierárquicos ou ordenações de abstrações a fim de obter uma melhor compreensão dos fenômenos em questão. Os princípios segundo os quais esse processo ocorre são normalmente referidos como *princípios de abstração* [19].

Um dos benefícios da programação orientada a objetos é que, diferente de outros paradigmas de programação modernos, ela fornece suporte para cada um dos mais importantes princípios de abstração: classificação/instanciação, agregação/decomposição e generalização/especialização [130].

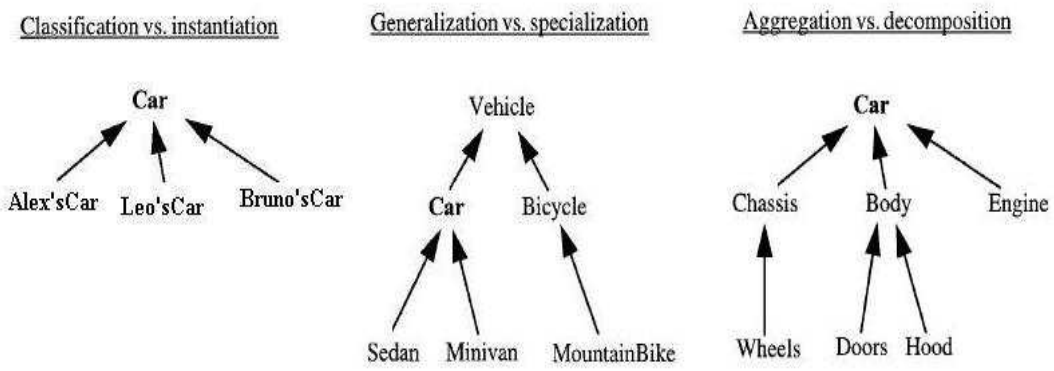

Figura 4.1: Princípios de abstração usados na modelagem orientada a objetos [130].

Modelos de objetos representam aplicações como um conjunto de objetos e classes. *Objetos* representam instâncias de entidades tangíveis e intangíveis enquanto *classes* representam conjuntos de objetos semelhantes. O *relacionamento classificação/instanciação* relaciona uma classe a suas instâncias. As classes são organizadas em hierarquias de *generalização/especialização*, em que as subclasses herdam a estrutura e o comportamento de suas superclasses, e *hierarquias parte-de*, nas quais as classes agregadas são definidas como a agregação de outras classes (Figura 4.1).

Modelos de aspectos representam aplicações como um conjunto de aspectos e componentes (classes e objetos). *Aspectos* representam *propriedades* que afetam a estrutura e/ou o comportamento de componentes existentes. Os aspectos podem ser organizados em *hierarquias de tipo/subtipo*, nas quais os *subaspectos* herdam as propriedades de seus *superaspectos*.

Entretanto, o principal relacionamento genérico presente em modelos de aspectos é o *relacionamento crosscutting*. Na Seção 3.1.3, discutimos algumas questões relacionadas a *crosscutting*. Nesta Seção, discutimos como os aspectos e *crosscutting* podem ser tratados na modelagem conceitual.

Crosscutting e Herança

Crosscutting pode ser considerado (i) um mecanismo de modelagem que relaciona aspectos e componentes de forma transparente, (ii) um mecanismo para reutilização e compartilhamento de código, no qual a estrutura e o comportamento localizados nos aspectos possam ser repetidamente combinados em vários pontos de combinação quantificados ou (iii) um mecanismo para a extensão incremental que permite que componentes existentes me-

lhorem progressivamente sem a necessidade de editá-los.

Quando componentes são *classes* e *objetos*, o mecanismo herança também é fornecido como uma facilidade para a modelagem conceitual, como um mecanismo para a reutilização e compartilhamento de código e como um mecanismo para a extensão incremental das classes existentes. Nesse contexto, eles têm muitas semelhanças, mas algumas distinções importantes, conforme ilustrado na Tabela 4.1.

Propriedades	Herança	Crosscutting
Comportamento da Base?	superclasse	classe
Comportamento adicional?	subclasse	aspecto
Direção?	da subclasse para superclasse	do aspecto para classe
Base modificada pela adição?	não	sim, aspectos modificam classes base implícitamente
Cardinalidade?	1:1 (simples)	1:m
Comportamento reutilizável?	superclasse	aspecto, classe

Tabela 4.1: Crosscutting e Herança.

A herança oferece um mecanismo de extensão entre a classe base e uma classe de extensão (subclasse) que não afeta a implementação nem o comportamento da primeira. A classe base original permanece a mesma. O comportamento adicionado pode ser observado instanciando a subclasse. Ela oferece suporte à reutilização das classes existentes como a base para a definição de novas classes [130].

Crosscutting também oferece um mecanismo de extensão entre uma classe base e um aspecto de extensão, mas, dessa vez, o comportamento da classe base é afetado pela extensão. A classe base original é modificada no local (*in-place*). *Crosscutting* dá suporte à reutilização de aspectos uma vez que os mesmos podem ser combinados com outras classes; ele também facilita a reutilização de classes menos entrelaçadas em outros contextos.

A visão clássica da herança na modelagem orientada a objetos é a de um mecanismo de estruturação hierárquica que dá suporte à *especialização conceitual*. Os objetos são vistos como entidades. Todavia, o que podemos dizer sobre a visão de *crosscutting* e aspectos na modelagem orientada a aspectos?

Aspectos e Crosscutting

O modelo de aspectos e a dicotomia conceitual aspecto-base introduzem um nível de assimetria na modelagem conceitual que faz a distinção entre objetos (com propriedades intrínsecas) e aspectos (as propriedades extrínsecas), ou seja, entre *existência* e *percepção*. As propriedades intrínsecas são propriedades sem as quais o objeto não pode existir, enquanto as propriedades extrínsecas são aquelas que somente devem ser percebidas em determinados contextos (consulte também POS na Seção 2.3.2).

Enquanto a modelagem orientada a objetos se concentra nos conceitos e nos fenômenos em si, a MOA se concentra na representação de aspectos importantes dos conceitos e fenômenos. Os aspectos refletem diferentes facetas dos componentes do software que eles melhoram. Eles podem refletir os *pontos de vista* de diferentes *stakeholders* ou diferentes *papéis* que um componente pode exercer em diferentes relacionamentos.

Crosscutting é um relacionamento entre um aspecto e um conjunto de componentes, os *componentes base* (`Aspecto` → `set of Componente`). Uma possível visão de *crosscutting* na modelagem orientada a aspectos poderia ser um mecanismo de estruturação com o objetivo de *misturar* (*blending*).

Misturar (to *blend*) é definido como "combinar ou associar de forma que os constituintes separados ou a linha de demarcação não possam ser distinguidos" e "combinar em um todo integrado" [99]. *Mistura conceitual* (*conceptual blending*) é um termo usado no campo da Ciência Cognitiva [44]. Ela denota "a capacidade de a mente pegar dois conceitos diferentes, formar uma ligação cognitiva entre eles e produzir um terceiro e novo conceito que é uma mistura dos outros dois". Uma faceta importante dessa definição é aquela de "conceitos diferentes". A mistura conceitual denota a mistura entre conceitos e não entre conceitos e suas propriedades. Consideramos que um tipo de mistura definido entre conceitos e aspectos (*mistura aspectual*) que também observa o contexto, é um princípio de abstração candidato para descrever *crosscutting* na modelagem orientada a aspectos.

Por exemplo, ao misturar a propriedade Segurança (*Security*) e o conceito Carro (Figura 4.2), em geral, temos a idéia de um "carro protegido" (*Protected Car*) – com alarme de proteção, por exemplo. Ao combinar Carro e Cobrança (*Billing*), temos a idéia de um carro usado em um serviço cobrado (um táxi). Pela própria natureza dos aspectos, o resultado da mistura aspectual afeta o conceito (um "carro protegido"), nunca o oposto ("segurança do carro").

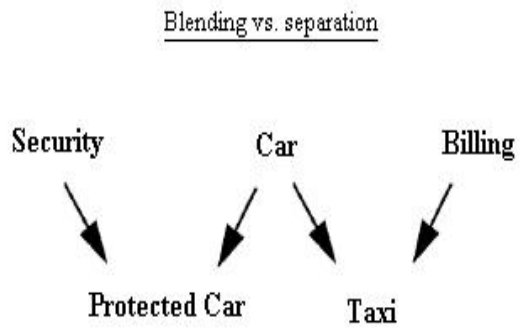

Figura 4.2: Crosscutting na modelagem orientada a aspectos.

A MOA enriquece a modelagem orientada a objetos ao oferecer suporte a modelos hierárquicos e a modelos de crosscutting separados; ela também representa uma melhoria nas abordagens de modelagem atuais quanto ao gerenciamento de várias perspectivas e papéis de componentes, diminuindo, portanto, o gap semântico. A mistura aspectual como um princípio de abstração para a MOA é uma nova idéia que merece mais pesquisa, assim como as questões relacionadas à mistura de aspectos.

4.1.3 Uma Visão Pragmática de Aspectos e Crosscutting

Aspectos têm sido usados para resolver vários problemas. A Seção 2.3.6 listou os principais usos para aspectos relatados pela comunidade de usuários que adotou AspectJ.

Esses aspectos podem ser organizados em diferentes categorias: aspectos independentes de domínio *versus* aspectos específicos ao domínio, aspectos de desenvolvimento *versus* aspectos de produção *versus* aspectos reusáveis [10] etc. Em relação à MOA, organizamos esses aspectos nas categorias a seguir, não necessariamente separadas: aspectos *sistêmicos*, *colaborativos*, *subjetivos* e *evolutivos*. Cada categoria coloca alguns desafios específicos à prática atual da modelagem de software que podem ser resolvidos pela modelagem orientada a aspectos.

Aspectos sistêmicos

Os aspectos sistêmicos são usados para tratar da modularização de *concerns* transversais sistêmicos, ou seja, os *concerns* que tratam de

requisitos não-funcionais, como a sincronização, a persistência, o tratamento de erros, os mecanismos de auditoria, entre outros.

```
aspect TraceMyClasses {
    pointcut myClass(): within(A)||within(B)||within(C);
    pointcut myMethod(): myClass() && execution(* *(..));

    before (): myMethod() {
        Trace.traceEntry("Entry: " +
             thisJoinPointStaticPart.getSignature());
    }
    after(): myMethod() {
        Trace.traceExit("Exit: " +
             thisJoinPointStaticPart.getSignature());
    }
}
```

Figura 4.3: Exemplo de aspecto sistêmico [10].

Muitas vezes, os aspectos sistêmicos descrevem *concerns* não-funcionais e oferecem suporte a *crosscutting* vertical, afetando, possivelmente, várias classes de forma homogênea. Os aspectos não-sistêmicos descrevem *concerns* funcionais e podem oferecer suporte a *crosscutting* horizontal, afetando, possivelmente, um pequeno número de classes.

Encontramos na literatura muitos exemplos de aspectos sistêmicos; por exemplo, [124] apresenta os aspectos que lidam com os *concerns* de distribuição e persistência. A Figura 4.3 apresenta o aspecto TraceMyClasses, codificado em AspectJ [10]. Esse aspecto realiza as chamadas de rastreamento (*tracing*) nos momentos apropriados. De acordo com esse aspecto, o rastreamento é realizado na entrada e na saída de cada método definido dentro das classes `A`, `B` e `C`. Os métodos `traceEntry` e `traceExit` imprimem a assinatura do método que está executando.

Como aspectos sistêmicos afetam vários lugares de forma homogênea, eles requerem mecanismos de quantificação mais poderosos, envolvendo o uso de expressões regulares a fim de facilitar a especificação dos pontos de combinação de interesse.

Aspectos colaborativos

Aspectos colaborativos são usados para cuidar da necessidade de localizar a interação do componente, ou a interação multiobjeto, como

aquela necessária para a implementação de padrões de projeto, validação de contrato etc.

Um aspecto pode afetar duas ou mais classes por meio de *crosscutting*, introduzindo novas responsabilidades ou redefinindo as existentes, de forma a estabelecer uma colaboração entre as classes participantes. Dessa forma, um aspecto deve ter a capacidade de coordenar mais de um componente base, descrevendo cada um deles como um participante separado. Os aspectos colaborativos [83] se parecem com colaborações UML e conectores de arquitetura de software [119]; eles localizam um protocolo e requerem interfaces claras.

Aspectos subjetivos

Aspectos subjetivos podem ser usados para cuidar da necessidade de visões segregadas para objetos, introduzindo novos atributos e métodos em um elemento base ou restringindo o uso de outros, por exemplo.

O comportamento dependente do contexto também pode ser modelado por aspectos subjetivos, como a necessidade de registrar ou monitorar algum comportamento quando observado sob determinada perspectiva ou disparado por um ator específico.

```
aspect SubjectiveView {

    private void Viewed.aMethod();

    pointcut myClass(): within(Viewer);
    pointcut myMethod(Viewed v):
        myClass() && call(public Viewed.*(..)) && target(v);

    after(Viewed v): myMethod(v) {
        v.aMethod();
    }
}
```

Figura 4.4: Exemplo de aspecto subjetivo.

A Figura 4.4 apresenta o aspecto `SubjectiveView`, codificado em AspectJ [10]. Esse aspecto introduz de forma privada o método `aMethod()` na classe `Viewed`. O método introduzido `Viewed.aMethod()` é invocado depois de qualquer chamada aos métodos públicos de `Viewed`, mas apenas quando são emitidos a partir de métodos definidos dentro da classe `Viewer`. Outras chamadas para os métodos públicos de `Viewed` não são afetadas.

Aspectos evolutivos

Aspectos evolutivos podem ser usados para resolver a necessidade de evolução das classes, ao modificar sua superclasse, implementar uma interface ou introduzir novos atributos e métodos.

Os aspectos evolutivos se assemelham aos *papéis* [13]; eles capturam os aspectos evolucionários de objetos que não podem ser modelados por valores de atributos dependentes do tempo e que, de outra forma, seriam rigidamente expressos por classes e pelo relacionamento de generalização [35].

```
aspect CloneablePoint {
   declare parents: Point implements Cloneable;

   public Object Point.clone()
          throws CloneNotSupportedException {
      makeRectangular();  // defined in class Point
      makePolar();        // defined in class Point
      return super.clone();
   }
}
```

Figura 4.5: Exemplo de aspecto evolutivo [10].

A Figura 4.5 apresenta o aspecto `CloneablePoint`, codificado em AspectJ [10]. Esse aspecto modulariza um conjunto de modificações que tratam da evolução de um `Point` para um `CloneablePoint`. O aspecto é responsável por fazer `Point` implementar a interface `Cloneable`. Ele declara que `Point` implementa a interface `Cloneable` com uma forma `declare parents`; também declara um método especializado de `Point`, `clone()`. Em Java, todos os objetos herdam o método `clone` da classe `Object`, mas um objeto não é clonável a menos que sua classe também implemente a interface `Cloneable`.

4.2 Questões da Modelagem Orientada a Aspectos

Nesta Seção, fornecemos nossa visão sobre as principais questões envolvidas na modelagem orientada a aspectos.

A MOA requer que, além de compreender e lidar com a estrutura e o comportamento de aspectos individuais, o projetista possa compreender os relacionamentos entre aspectos e classes, as possíveis interações entre as-

pectos, as possíveis interferências entre os novos mecanismos de composição e os convencionais, a estrutura e o comportamento do sistema depois da combinação de classes e aspectos, e assim por diante.

Organizamos a discussão em torno das questões relacionadas à especificação, visualização e construção de modelos de aspectos.

4.2.1 Especificação: conceitos e propriedades

Especificar significa construir modelos – possivelmente com elementos omitidos ou ausentes – que tendem a se tornar precisos, não-ambíguos e completos ao longo do tempo [17].

A fim de identificar as questões de modelagem relacionadas à especificação de sistemas orientados a aspectos, usamos uma abordagem *bottom-up* e identificamos os principais conceitos e propriedades de cada técnica da POA. Esses elementos foram apresentados no Capítulo 3.

Defendemos que qualquer abordagem de propósito geral à MOA deve considerar um framework conceitual unificador para POA, como o modelo de aspectos apresentado no Capítulo 3, com propriedades, conceitos e terminologia associados. Neste caso, seremos mais permissivos com respeito a propriedades como a dicotomia aspecto-base, de modo a considerar Hyper/J uma abordagem orientada a aspectos. A Tabela 4.2 resume os conceitos e as propriedades que compreendem o modelo de aspectos e relaciona-o ao modelo de objetos.

Modelo de objetos	classe, objeto, atributo, método, interface, mensagem, herança, agregação, etc.
Modelo de pontos de combinação	chamada de método, execução de método, criação de objetos, etc.
Modelo do processo de combinação	combinador de aspectos, processo de combinação, 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 4.2: Conceitos e propriedades definidos no modelo de aspectos.

Esses novos conceitos e propriedades devem usar e melhorar os elementos que compõem o modelo de objetos sem entrar em contradição com

a semântica padrão. Por exemplo, aspectos não são classes, portanto, a abstração de aspectos deve ser especificada por um novo elemento de modelagem.

Além desse framework conceitual, o projetista precisa especificar os relacionamentos entre aspectos que surgem quando aspectos são compostos.

Relacionamentos entre aspectos

O ideal é que os aspectos não dependam de outros aspectos. Na prática, os aspectos podem precisar *conhecer* (*know about*) outros aspectos quando são compostos a um conjunto de classes na mesma aplicação. Eles podem interagir ou se sobrepor, e devem ser fornecidos relacionamentos e interfaces explícitos a fim de expressar esse tipo de dependência:

- *Interação entre aspectos*
  - Se um aspecto afeta uma operação definida por outro aspecto, dizemos que esses aspectos *interagem*, esse relacionamento de dependência interaspecto deve ser explicitamente modelado a fim de tornar o design mais robusto.
  - Se dois ou mais aspectos afetam a mesma operação, dizemos que esses aspectos *interagem*; esse tipo de acoplamento é análogo ao problema de acoplamento indireto por meio de compartilhamento de dados descrito em [100] (acoplamento indireto por meio de compartilhamento de ponto de combinação, no caso de aspectos). A *precedência* entre os aspectos que estão interagindo também deve ser explicitamente modelada.
- *Sobreposição de aspectos*
  - Se dois aspectos definem um comportamento similar, relacionado às mesmas classes, dizemos que os aspectos se *sobrepõem* (*overlap*).

Portanto, um projetista pode desejar especificar situações em que:

- Um aspecto requer a presença de outro aspecto para sua implementação ou funcionamento correto (requisito ou *requirement*);
- Entre um conjunto de aspectos, apenas um pode ser aplicado em determinadas circunstâncias, possivelmente porque se sobrepõem (exclusão mútua);

– Dois ou mais aspectos interagem, e um ordenamento deve relacionar explicitamente sua composição ao mesmo elemento base (precedência).

4.2.2 Visualização: diagramas e visões

A apresentação visual mostra informações semânticas em uma forma que pode ser vista, navegada e editada por seres humanos. A organização de elementos de apresentação em diagramas é fundamental para permitir a compreensão humana de um modelo e sua comunicação eficaz.

A modelagem orientada a aspectos deve lidar com novos problemas relacionados à visualização, como a visualização de (i) aspectos sistêmicos e os pontos em que ocorrem as melhorias, (ii) aspectos colaborativos e suas interfaces, (iii) aspectos que estão interagindo e se sobrepondo, (iv) os resultados da combinação de aspectos e classes etc.

Aspectos sistêmicos

A apresentação visual de relacionamentos entre aspectos sistêmicos e classes afetadas em diagramas de classes (usando o relacionamento de *cross-scutting*) é um desafio para a MOA. Se todos os relacionamentos estiverem presentes, o número de símbolos que denotam os relacionamentos pode se tornar impossível de gerenciar, prejudicando assim a compreensibilidade. Entretanto, devemos fornecer alguns meios de expressar aspectos, classes e seus relacionamentos, de forma que a visão arquitetural do sistema torne-se evidente.

Aspectos colaborativos

Aspectos podem afetar as classes de forma heterogênea, ou seja, duas ou mais classes podem ser afetadas por diferentes subconjuntos de características transversais localizadas dentro do mesmo aspecto. A decomposição da interface do aspecto em duas ou mais subinterfaces promove a previsibilidade da composição e melhora a compreensão. Além disso, a apresentação gráfica dessas interfaces facilita a compreensão.

Aspectos que interagem e se sobrepõem

A interação e a sobreposição ocorrem porque dois ou mais aspectos afetam as mesmas classes base. Portanto, a MOA deve fornecer um suporte à visualização que se concentre na descrição de um elemento base e dos aspectos que o afetam, de forma a apresentar explicitamente as informações sobre a dependência entre aspectos.

Elementos combinados

A separação entre aspectos e classes aumenta a compreensibilidade do sistema que está sendo projetado. Todavia, uma visão clara do sistema depois do processo de combinação é desejável a fim de permitir que o projetista faça uma avaliação inicial dos resultados da composição.

A MOA deve fornecer formas para modelar os efeitos de um conjunto de aspectos em um design orientado a objetos, tornando aparente os efeitos da combinação.

4.2.3 Construção: o papel de ferramentas

A MOA depende muito de ferramentas adequadas, principalmente para lidar com a interação entre aspectos, a quantificação e os aspectos sistêmicos, a separação entre aspectos e classes e a transparência.

Devido a possíveis *interações entre aspectos*, os relacionamentos devem ser explicitamente representados a fim de aumentar a compreensão e facilitar a evolução.

Devido a *aspectos sistêmicos*, o projetista deve ser capaz de especificar os pontos de combinação de interesse, sem ter de enumerá-los explicitamente: isto seria um processo improdutivo e tedioso. Os mecanismos de *wildcards* facilitam a tarefa: uma das maiores vantagens de AspectJ é oferecer ao desenvolvedor a capacidade de quantificar grande parte de sua aplicação com poucas linhas de código usando expressões regulares a fim de capturar os pontos de combinação desejados. Contudo, essa característica também introduz duas desvantagens que permanecem entre as mais sérias críticas à POA e que devem ser tratadas com o suporte adequado de ferramentas: (i) a transferência da tomada de decisão da atividade de análise da estrutura de um programa para o nível meramente léxico (dependência do nome) e (ii) a incerteza em relação ao escopo da especificação baseada em

wildcards, principalmente diante da evolução do sistema (novas classes podem ser adicionadas, por exemplo, cujos nomes correspondem a expressões regulares fornecidas por aspectos, mas que não devem ser afetadas pelo comportamento do aspecto).

Devido à *transparência* e o nível adicional de separação (*entre aspectos e classes*), o projetista precisa avaliar os efeitos dos aspectos e *crosscutting* em um projeto base a partir dos elementos combinados que resultam da composição.

Como qualquer abordagem moderna para modelagem, a MOA requer ferramentas que ofereçam suporte à edição gráfica, à visualização, ao gerenciamento de dados, à análise, à refatoração, à reengenharia, à geração de código etc. Além disso, uma *ferramenta de combinação* (*weaving tool*) é necessária para ajudar os projetistas no tratamento da interação de aspectos, na dicotomia aspecto-base e na transparência. Outra ferramenta importante nesse contexto é uma ferramenta de casamento de padrão (*pattern matching tool*) para lidar com a quantificação de um grande conjunto de elementos de modelagem.

4.3 Requisistos para Linguagens de Modelagem Orientadas a Aspectos

– Quais são os requisitos das ferramentas e linguagens de modelagem orientadas a aspectos?

Ter uma *notação expressiva e bem definida* é um requisito importante para qualquer linguagem de modelagem. Uma *notação padrão* possibilita ao projetista formular um design e, em seguida, comunicá-lo aos outros. Nesse contexto, a compatibilidade com UML, a linguagem de modelagem padrão para sistemas orientados a objetos, é fundamental.

Outro requisito geral para as linguagens de modelagem é fornecer uma notação independente da linguagem de programação (talvez personalizável para cada linguagem). Para a MOA, esse também é um requisito fundamental, diante da diversidade das linguagens de programação orientadas a aspectos.

As linguagens e as ferramentas para a modelagem orientada a aspectos devem tratar dos seguintes requisitos para lidar com as questões de modelagem discutidas anteriormente (Seção 1.2):

– Tratar aspectos como cidadãos de primeira classe;

– Oferecer suporte à dicotomia conceitual entre aspectos e classes;

- Oferecer suporte à expressão de interfaces de aspectos;
- Oferecer suporte à decomposição de interfaces de aspectos;
- Oferecer suporte à representação explícita de interações de aspectos;
- Fornecer *composabilidade no nível do design*, com base em uma semântica de composição consistente para *crosscutting*;
- Oferecer suporte à descrição explícita das melhorias sensíveis ao contexto;
- Oferecer suporte para algum tipo de quantificação na estrutura ou no comportamento de classes;
- Oferecer suporte à apresentação de uma visão do sistema depois do processo de combinação.

4.4 A Linguagem UML

A Linguagem de Modelagem Unificada (Unified Modeling Language - UML) [17, 116, 134] é uma linguagem de propósito geral para a especificação, visualização, construção e documentação de artefatos de um sistema de software.

UML 1.0 foi oferecida para padronização ao Object Management Group (OMG) em 1997. O esforço de UML surgiu de três linguagens de modelagem e notações de design proeminentes, a saber os métodos de design de Booch [16], OOSE de Jacobson [65], e OMT de Rumbaugh [115], com o objetivo de fornecer uma linguagem de modelagem padrão que combinasse os pontos fortes das três linguagens e notações e que se adequasse às necessidades da maioria dos domínios da aplicação.

A linguagem UML é uniforme, desde a especificação de requisitos até a fase implantação: o mesmo conjunto de conceitos e notação podem ser usados em diferentes atividades de desenvolvimento [116]. A UML não depende de um processo de desenvolvimento, linguagem de programação ou ferramenta em particular. No entanto, UML permite (mas não exige) um processo incremental, iterativo, centrado na arquitetura e orientado a casos de uso [17].

Como uma linguagem de modelagem, a UML fornece blocos básicos a fim de modelar sistemas intensivos de software. O vocabulário de UML incorpora três tipos de blocos básicos: abstrações que representam elementos comportamentais e estruturais do sistema; relacionamentos, que especificam como essas abstrações se relacionam entre si; e diagramas, que representam

visões complementares de um conjunto de abstrações e seus relacionamentos [17]. Um *metamodelo* subjacente integra essas visões de forma que um sistema autoconsistente possa ser analisado e construído. A UML incorpora o consenso da comunidade orientada a objetos em relação aos principais conceitos de modelagem e permite variações que devem ser expressas em termos de seus *mecanismos de extensão*.

Este Capítulo refere-se à UML, Versão 1.4 [134]. Nas próximas Seções, reproduzimos algumas definições fornecidas em [17, 116, 134], a fim de oferecer um resumo dos elementos que compõem a UML, bem como a arquitetura e o formalismo da linguagem.

4.4.1 Elementos

Elementos estruturais descrevem as partes estáticas dos modelos de UML e representam elementos conceituais ou físicos no sistema; incluem *classes, interfaces, casos de uso, colaborações, componentes* e *nós*.

Elementos comportamentais descrevem as partes dinâmicas de modelos de UML. Há dois tipos de elementos comportamentais: *interações* e *máquinas de estado*.

Elementos de gerenciamento de modelo são elementos para agrupar, as partes organizacionais de modelos de UML. Há um tipo primário de elemento que agrupa: *pacotes*.

Relacionamentos são os blocos relacionais básicos de UML. Há quatro tipos de relacionamento em UML: *dependência, associação, generalização* e *realização*.

Diagramas apresentam visões complementares de um conjunto de elementos de modelagem e seus relacionamentos. UML inclui nove tipos de diagramas: *diagramas de classes, objeto, caso de uso, seqüência, colaboração, estados, atividade, componente* e *implantação*.

4.4.2 Arquitetura

A arquitetura de UML baseia-se em uma estrutura de metamodelo de quatro camadas (Figura 4.6). As metacamadas são numeradas de M3 a M0, sendo que M3 denota a camada de metametamodelo, M2 denota a camada metamodelo, M1 denota a camada modelo (de domínio) e M0 denota a camada de objetos do usuário.

O nível M3 inclui o metametamodelo MOF[1]. O metametamodelo MOF é o metametamodelo para o metamodelo de UML. A camada de metametamodelagem forma a base da arquitetura de metamodelagem. A principal responsabilidade dessa camada é definir a linguagem para especificar um metamodelo.

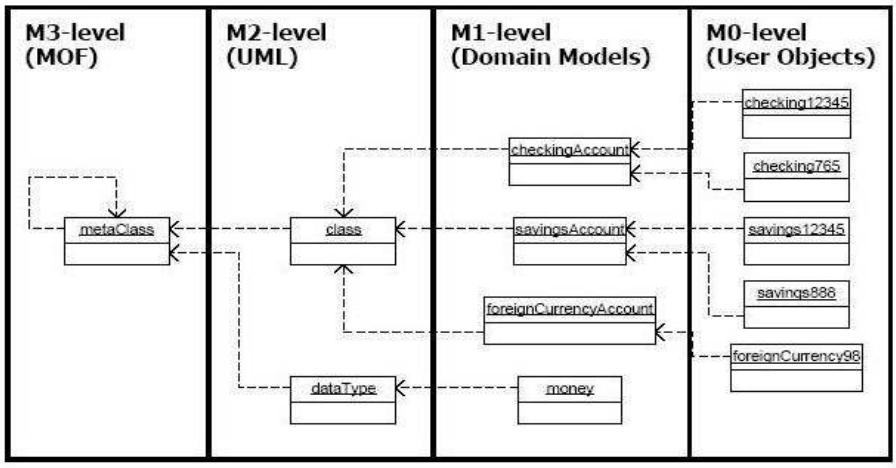

Figura 4.6: A arquitetura de quatro camadas de UML.

O nível M2 inclui o metamodelo de UML. Um *metamodelo* é uma instância de um metametamodelo, o que significa que todo elemento do metamodelo é uma instância de um elemento no metametamodelo. A principal responsabilidade da camada de metamodelo (M2) é definir uma linguagem para especificar modelos. O metamodelo de UML oferece uma definição precisa das construções e regras necessárias para a criação de modelos de UML. Alguns exemplos de metaobjetos na camada de metamodelagem são: `Class`, `Attribute`, `Operation`, e `DataType`.

O nível M1 inclui modelos de domínio. Um *modelo* é uma instância de um metamodelo. A principal responsabilidade da camada de modelo (M1) é definir uma linguagem que descreve um domínio de informações. Alguns exemplos de objetos na camada de modelagem são: `CheckingAccount`, `SavingsAccount`, e `ForeignCurrencyAccount`.

A hierarquia de metamodelo termina em M0, que contém os *objetos do usuário*, as instâncias em tempo de execução de elementos de modelo definidos em um modelo (M1). A principal responsabilidade da camada de objetos do usuário (M0) é descrever um domínio de informações específico. Alguns

[1]MOF (Meta-Object Facility) [92] é a tecnologia adotada pela OMG para a modelagem e a representação de metadados (incluindo o metamodelo de UML) como objetos CORBA.

exemplos de objetos na camada de objetos de usuário são: `checking12345`, `savings12345`, etc.

4.4.3 Formalismo

A especificação de UML usa uma combinação de linguagens – um subconjunto de UML, uma linguagem de restrições (OCL) e linguagem natural – a fim de descrever a semântica e a sintaxe abstrata de UML.

UML é especificada por meio de um *metamodelo*. Para criar o metamodelo de UML, foram usadas diferentes técnicas a fim de especificar construções da linguagem, usando alguns recursos da própria UML. As principais construções da linguagem são descritas por metaclasses no metamodelo. Algumas construções, em essência sendo variantes de outras, são definidas como estereótipos de metaclasses no metamodelo (consulte o nível M2 na Figura 4.6).

A *sintaxe abstrata* de UML é apresentada nos diagramas de classe de UML mostrando as metaclasses que definem as construções da linguagem e seus relacionamentos. A *semântica estática* de metaclasses de UML é definida como um conjunto de regras de boa formação (*well-formedness rules*) de uma instância de metaclasse. Cada regra é expressa em Object Constraint Language (OCL) [134], junto com uma explicação informal da regra.

A *semântica dinâmica* dos elementos de UML é expressa em linguagem natural.

4.4.4 Extensibilidade

A UML pode ser estendida de duas formas:

- as extensões peso leve (*light weight extensions*), nas quais se introduzem novos estereótipos (subclasses de metaclasses existentes) ou se definem perfis ou *profiles* (pacotes estereotipados que contêm estereótipos, restrições e definições rotuladas) que oferecem significado restrito para construções existentes;
- as extensões peso pesado (*heavy weight extensions*), nas quais é possível definir novas metaentidades (por exemplo, novas metaclasses que não são necessariamente derivadas das existentes) ou modifi-

car bastante as existentes (metaatributos, metaoperações, metaassociações etc.).

Os mecanismos peso leve são mais fáceis de usar e têm suporte automático de ferramentas, então devem ser usados sempre que possível. No entanto, se as novas construções da linguagem não puderem ser facilmente expressas restringindo as construções existentes da linguagem base, os mecanismos peso pesado devem ser usados.

Mecanismos de extensão light-weight

Os mecanismos de extensão são os meios para estender UML a fim de oferecer suporte a um domínio específico ou a uma nova tecnologia. Os mecanismos de extensão de UML incluem *estereótipos, definições de tags, restrições* e *valores rotulados*. Um conjunto coerente dessas extensões, definidas para propósitos específicos, constitui um *perfil* (*profile*) de UML [134]. A Figura 4.7 mostra exemplos de um estereótipo (`<<exception>>`), valores rotulados (`version` e `author`) e uma restrição ({`ordered`}).

Estereótipos são usados para introduzir um novo tipo de elemento de modelagem como uma extensão ou uma classificação de elementos base existentes. Um estereótipo define valores adicionais (com base na definição das tags), outras restrições e, opcionalmente, uma nova representação gráfica. O conceito de estereótipo permite que o elemento estendido "se comporte como se tivesse sido instanciado a partir do construto do metamodelo".

Definições de tags (*tag definitions*) especificam novos tipos de propriedades que podem ser associadas aos elementos do modelo. Os valores rotulados especificam pares de valor de palavra-chave de elementos de modelagem a fim de indicar uma nova propriedade para o elemento de modelagem de UML existente ou podem ser aplicados a estereótipos.

Restrições (*constraints*) são um conjunto de regras de formação expressas em (OCL) [134] ou uma linguagem natural. As restrições são os meios pelos quais a nova semântica pode ser introduzida à UML.

Um perfil não estende UML através da adição de um novo conceito. Em vez disso, fornece conexões para aplicar e especializar UML para um determinado domínio. É um requisito obrigatório em qualquer perfil que quando ele for aplicado, a notação resultante deva aderir à sintaxe e à semântica de UML. Em outras palavras, o perfil não deve romper os princípios de modelagem de UML existentes.

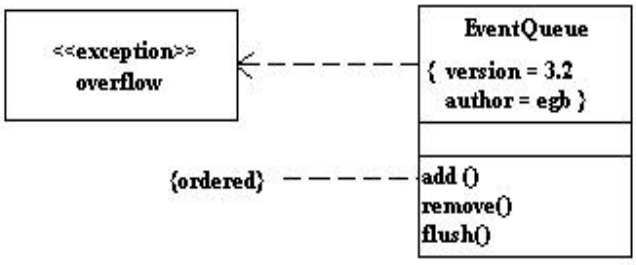

Figura 4.7: Mecanismos de extensão [17].

Mecanismos de extensão heavy-weight

Uma das principais restrições associadas ao uso de estereótipos e perfis é que novas metaclasses não podem ser criadas, uma vez que não há acesso direto à linguagem de metamodelagem. Uma abordagem mais poderosa é instanciar a linguagem de metamodelagem (isto é, o MOF) a fim de modelar novas linguagens ou extensões de linguagem diretamente.

Esse mecanismo é normalmente chamado de mecanismo de extensão "peso pesado" (*heavy weight*) ou de "primeira classe". Essa força expressiva traz um impacto negativo na facilidade de uso. A metamodelagem é, em geral, reconhecida como uma tarefa árdua, devido à necessidade de mover os processos mentais a um nível mais alto de abstração, bem como à necessidade de ter de lidar com um escopo muito maior do que muitos modelos de sistemas.

As hierarquias de classes abstratas podem melhorar esses problemas ao fornecer um núcleo comum usado na definição e extensão de metamodelos de linguagem. Essas hierarquias são usadas extensivamente na definição da própria UML. Se novas construções de linguagem não puderem ser facilmente expressas usando perfis, devem ser usados mecanismos peso pesado [41].

4.5 Modelagem Orientada a Aspectos e UML

É bastante natural investigar a adequabilidade de UML como uma linguagem de modelagem que possa dar suporte à modelagem orientada a aspectos.

Em primeiro lugar, a UML [17] é reconhecida pela comunidade da engenharia de software como a *linguagem de modelagem padrão* da indústria. Segundo, é uma *linguagem de modelagem de propósito geral* que pode ser

usada em uma ampla gama de domínios de aplicação. A UML fornece um rico conjunto de notações e diagramas que podem ser usados para descrever visões de sistemas intensivos de software. Finalmente, UML é uma *linguagem de modelagem extensível*. Assim, ela fornece um conjunto de mecanismos de extensão internos que permite sua personalização para processos ou domínios específicos. Além disso, fornece um metamodelo (que também é definido em UML) que descreve a semântica da linguagem e pode ser estendido.

Nesse contexto, algumas questões importantes relacionadas à MOA que serão tratadas nesta Seção são:

- as técnicas e notações existentes de UML são adequadas para a MOA ou precisamos estender a UML?
- se tivermos de estendê-la, os mecanismos de extensão fornecidos por ela são adequados?

As abordagens atuais que tratam da MOA podem ser classificadas de acordo com as respostas fornecidas para as perguntas acima.

1. **Abordagens transformacionais**

   As abordagens convencionais defendem que UML já fornece as notações e as técnicas que podem oferecer suporte à MOA sem precisar de novos conceitos dedicados ou elementos de apresentação, tais como aspectos ou estereótipos para denotar aspectos. Por exemplo, os casos de uso são realizados por colaborações que descrevem o comportamento que, por sua vez, afeta várias classes. As colaborações localizam o comportamento *crosscutting* e, portanto, são "orientadas a aspectos".

   Essas abordagens requerem ferramentas de transformação apropriadas baseadas em UML para a seleção e empacotamento de elementos de *crosscutting* e para sua combinação a fim de gerar novos elementos.

2. **Abordagens baseadas em perfis**

   As abordagens baseadas em perfis defendem que o uso de mecanismos de extensão internos de UML (estereótipos, valores rotulados e restrições) é suficiente para cuidar dos problemas da modelagem orientada a aspectos, sem a necessidade de modificar o próprio metamodelo de UML. Algumas abordagens definiram os mecanismos de extensão independentes [61, 128], enquanto outras empacotaram um conjunto coerente desses mecanismos em um perfil de UML [6]. Por exemplo,

os estereótipos podem ser usados para introduzir um novo tipo de elemento de modelo como uma especialização de algum elemento base existente (por exemplo, ≪aspect≫ como um estereótipo para o elemento `class`).

3. **Abordagens baseadas em metamodelo**

   As abordagens baseadas em metamodelo defendem que os mecanismos de extensão internos fornecidos por UML não são suficientes para tratar da MOA.

   Uma restrição fundamental no uso de mecanismos de extensão peso leve é que as extensões devem ser estritamente aditivas à semântica de UML padrão. Perfis fornecem a especialização de elementos existentes, usando herança de implementação por conveniência (*implementation inheritance for convenience* [130]), ou seja, uma nova classe se torna uma subclasse de uma classe existente simplesmente porque a classe existente parece fornecer o que se deseja. Apesar de "conveniente" por várias razões, as abordagens baseadas em perfis podem não ser conceitualmente consistentes e podem não fornecer suporte adequado para a expressão do modelo de aspectos.

   As abordagens baseadas em metamodelo usam mecanismos de extensão peso pesado a fim de estender a semântica de UML e tratar da MOA [129, 29, 26, 67] .

4.6 Abordagens para Modelagem Orientada a Aspectos

Esta Seção descreve algumas abordagens atuais usadas para construir modelos orientados a aspectos. Podem ser encontradas referências a outros trabalhos de pesquisa da MOA em [9].

4.6.1 UMLaut: uma abordagem transformacional

UMLAUT é um framework originalmente dedicado à manipulação de modelos de UML que pode ser considerado uma abordagem transformacional à MOA [62].

O framework UMLAUT oferece um kit para a construção de "combinadores" específicos para cada aplicação e para a geração de modelos de

design detalhados a partir de modelos de UML orientado a aspectos e de alto nível.

Os modelos UML "orientados a aspectos" de UMLAUT são modelos de objetos em que alguns elementos são usados para modelar *concerns* transversais ou *concerns* não-funcionais. A fim de especificar outras informações não-funcionais (por exemplo, persistência), a abordagem de UMLAUT recorre aos mecanismos de extensão internos de UML. Três mecanismos são normalmente usados para anotar elementos existentes: ocorrências de padrão de projeto (*design pattern occurrences*), estereótipos e valores rotulados (Figura 4.8).

Os estereótipos são usados para criar um subtipo de um determinado tipo de elemento de modelo, por exemplo, para especificar que instâncias de uma classe devem ser *persistentes*. As ocorrências de padrão de projeto são usadas para especificar que um determinado padrão de projeto deve ser aplicado em um determinado lugar no modelo. Os valores rotulados fornecem ao combinador outras informações para orientar o processo de combinação.

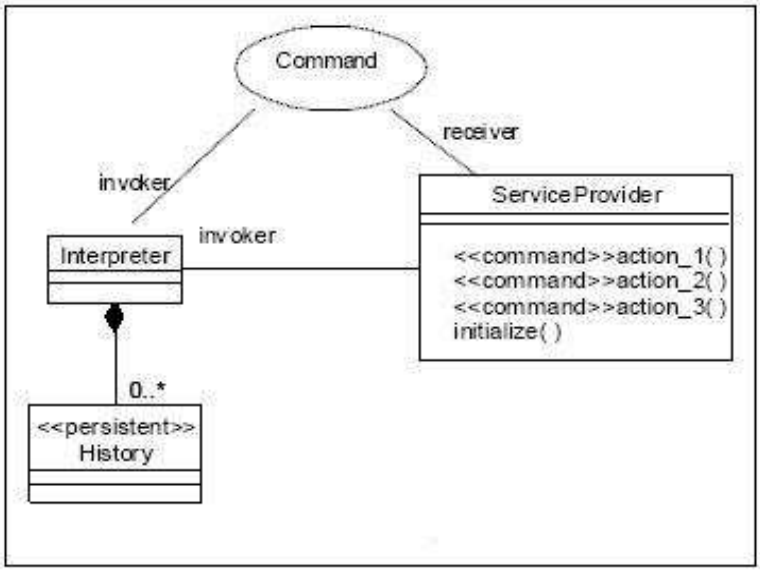

Figura 4.8: Diagrama de classes em UML com estereótipos e ocorrências de padrão [62].

O framework UMLAUT lida com o processo de combinação de designs "orientados a aspectos" ao tratar dos modelos de usuário no nível de metamodelo. Usando UMLAUT, o processo de combinação é implementado como um processo de transformação de modelo: cada etapa do processo de combinação é uma etapa transformacional aplicada ao modelo de UML, em que o resultado final também é um modelo de UML. Esse processo de

combinação, bem como outras características do framework UMLAUT, são descritos em [62].

A abordagem de UMLAUT oferece suporte ao processo de combinação de modelos de UML; no entanto, não oferece suporte aos requisitos das ferramentas e linguagens de modelagem orientadas a aspectos apresentadas na Seção 4.3.

4.6.2 AODM: uma abordagem baseada em perfil

Aspect-Oriented Design Model (AODM) [127, 128] é uma extensão de UML que estende a especificação UML existente com conceitos orientados a aspectos que reproduzem de forma apropriada as características transversais de AspectJ.

Em sua tese [127], Stein compara os conceitos orientados a aspectos de AspectJ com os conceitos orientados a objetos existentes de UML, ressaltando suas semelhanças e diferenças. Em sua abordagem, os *links* de UML são identificados como representações adequadas para os pontos de combinação dinâmicos de AspectJ. Para todos os componentes de um aspecto de AspectJ, como *pointcuts, advice* e *inter-type declarations*, assim como o próprio aspecto, a abordagem deriva as representações de UML a partir das metaclasses de UML existentes usando os mecanismos de extensão de UML.

Stein usa um exemplo retirado do Guia de Programação de AspectJ [10] para apresentar a notação de AODM. O exemplo ilustra como o padrão de projeto Observer apresentado em [46] é codificado em AspectJ. O padrão Observer é usado para realizar um rótulo colorido (*color label*), que exerce o papel de observador e muda sua cor sempre que um botão (*button*), que exerce o papel de sujeito, é clicado. A Figura 4.9 apresenta um aspecto abstrato chamado `SubjectObserverProtocol` que implementa o padrão de projeto Observer, e a Figura 4.10 apresenta um aspecto concreto chamado `SubjectObserverProtocolImpl` que implementa o padrão de projeto Observer para `Button` e `ColorLabel`.

Elementos de design

As categorias das principais construções e características da abordagem de AODM são avaliadas com a ajuda do framework conceitual proposto no Capítulo 3.

```
abstract aspect SubjectObserverProtocol {

   abstract pointcut stateChanges(Subject s); // pointcut

   after(Subject s): stateChanges(s) {  // after advice
      for (int i = 0; i < s.getObservers().size(); i++) {
        ((Observer)s.getObservers().elementAt(i)).update();}
   }

   // inter-type declarations - Subject role
   private Vector Subject.observers = new Vector();
   public  void   Subject.addObserver(Observer obs) {
     observers.addElement(obs);
     obs.setSubject(this);
   }
   public  void   Subject.removeObserver(Observer obs) {
     observers.removeElement(obs);
     obs.setSubject(null);
   }
   public Vector Subject.getObservers() {return observers;}

   // inter-type declarations - Observer role
   private Subject Observer.subject = null;
   public void Observer.setSubject(Subject s) {subject = s;}
   public Subject Observer.getSubject() { return subject; }
}
```

Figura 4.9: `SubjectObserverProtocol` em AspectJ [10].

Aspectos Os aspectos de AspectJ são representados como classes de UML de um estereótipo especial (chamado ≪aspect≫) devido à sua afinidade estrutural com as classes de UML padrão.

Especificação de pontos de combinação AODM reconhece a similaridade estrutural entre as operações orientadas a objetos e os pointcuts orientados a aspectos e define um novo estereótipo (chamado ≪pointcut≫) para operações UML padrão a fim de capturar a semântica dos pointcuts de AspectJ.

Características transversais AODM reconhece a similaridade estrutural entre as operações orientadas a objetos e os *advices* orientados a aspectos e define um novo estereótipo (chamado ≪advice≫) para operações UML padrão a fim de capturar a semântica do *advice* de AspectJ.

```
aspect SubjectObserverProtocolImpl
     extends SubjectObserverProtocol {

   /* concrete pointcut declaration */
   pointcut stateChanges(Subject s):
       target(s) && call(void Subject.click());

   /* introduction # Subject: Button # */
   declare parents: Button implements Subject;

   public Object Button.getData() { return this; }

   /* introduction # Observer: ColorLabel # */
   declare parents: ColorLabel implements Observer;

   public void ColorLabel.update() { colorCycle(); }
}
```

Figura 4.10: `SubjectObserverProtocolImpl` em AspectJ [10].

As introduções de AspectJ são representadas com um novo estereótipo para templates de colaboração de UML (chamado ≪introduction≫), usados para descrever as características e os relacionamentos introduzidos às classes base.

Crosscutting Os templates de colaboração desses estereótipos são implicitamente ligados a um conjunto prefixado de definições de classe base. Cada estereótipo definido pela abordagem é fornecido com uma metapropriedade complementar (chamada "base") a fim de manter as instruções dos processos de combinação especificadas em declarações de aspectos, introdução, *advice* e *pointcut*.

No caso de pointcuts, *advice* e aspectos, as metapropriedades devem atribuir um conjunto de *links* (especificados por uma definição de pointcut) que representa os pontos de combinação nos quais os respectivos elementos afetam a hierarquia da classe base. No caso de introduções, a metapropriedade "base" define um conjunto de definições de classe base (especificada por um padrão de tipo) que são afetadas pelas introduções.

Diagramas

Nos diagramas de UML, os novos estereótipos de AODM são interpretados da mesma forma que suas metaclasses base correspondentes – ador-

nados com o nome do estereótipo acima do nome do elemento.

Os aspectos são representados como retângulos com um compartimento de operação, nome e um atributo. Os pointcuts e advices são apresentados como *strings* que especificam suas assinaturas e são listados nos compartimentos de operação. As introduções são representadas como templates de colaboração a partir de UML. Os pontos de combinação podem ser visualizados pelos *links* realçados nos diagramas de colaboração de UML. O comportamento advice é descrito usando diagramas de seqüência de UML.

A Figura 4.11 mostra a especificação AODM para os aspectos em AspectJ mostrados nas Figuras 4.9 e 4.10.

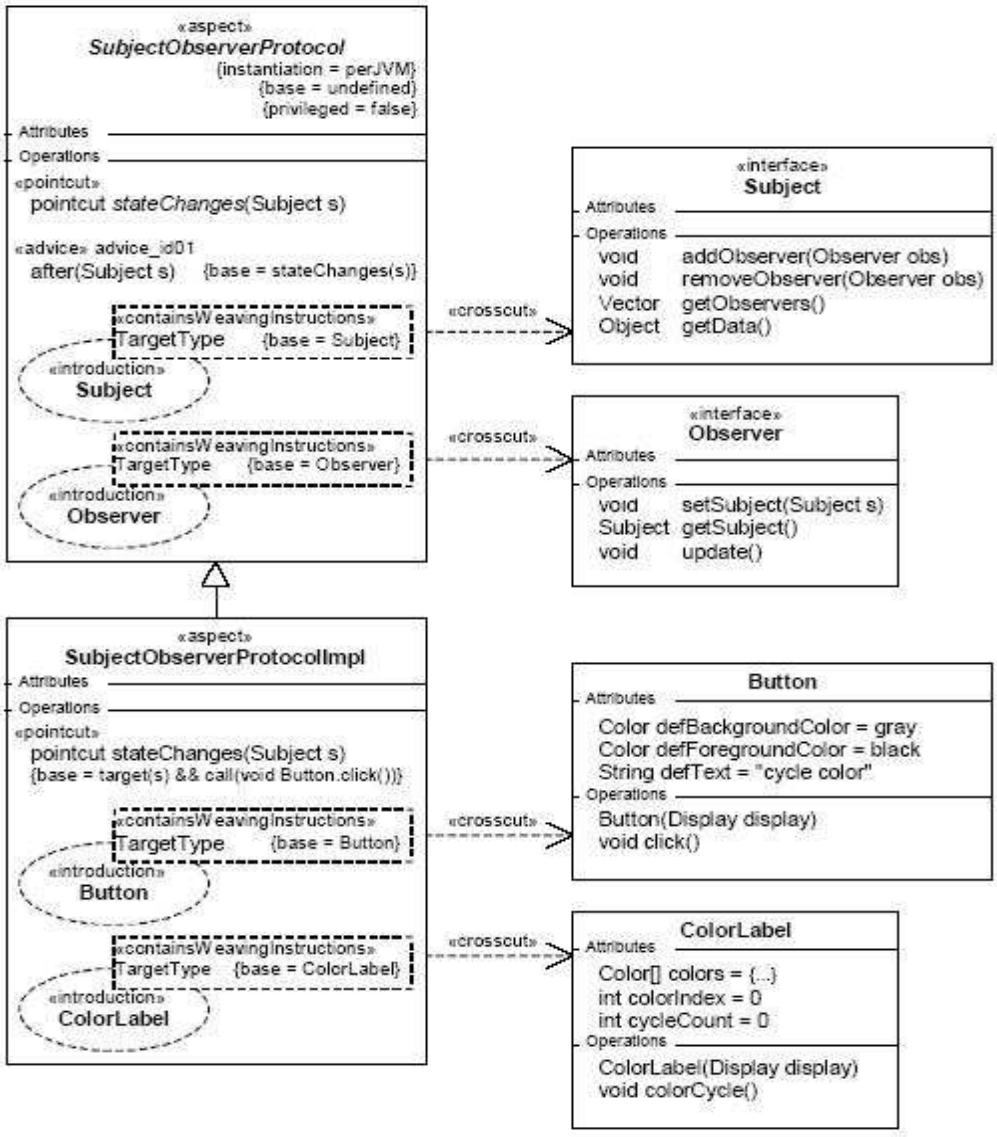

Figura 4.11: AODM: Notação [127].

O aspecto abstrato `SubjectObserverProtocol` define um pointcut

abstrato `stateChanges`, um after advice `advice_id01`, e duas introduções chamadas `Subject` e `Observer`. Além do nome do aspecto, suas outras metapropriedades especificam que o aspecto deve ser instanciado uma vez para o ambiente global (uma vez por Java Virtual Machine). Esse tipo de especificação só é útil para implementações de AspectJ. O aspecto `SubjectObserverProtocol` é estendido por um aspecto concreto chamado `SubjectObserverProtocolImpl`, que atribui o pointcut `stateChanges` a um conjunto concreto de pontos de combinação. Além disso, o aspecto concreto contém duas outras introduções chamadas `Button` e `ColorLabel`.

A Figura 4.11 mostra os relacionamentos *crosscut* que denotam os efeitos de crosscutting a partir das introduções contidas nos aspectos `SubjectObserverProtocol` e `SubjectObserverProtocolImpl` nas classes base e nas interfaces base, `Subject`, `Observer`, `Button`, e `ColorLabel`, de acordo com as instruções dos processos de combinação especificadas pelas expressões "base". Entretanto, os relacionamentos *crosscut* não são usados para expressar os efeitos de crosscutting do advice contidos nos aspectos sobre as classes base, pois argumenta-se que tal informação poderia comprometer a compreensibilidade do design.

A Figura 4.12 mostra uma especificação de um after advice contido no aspecto (abstrato) `SubjectObserverProtocol`. O advice é realizado por uma colaboração, que é completamente definida na Figura 4.13.

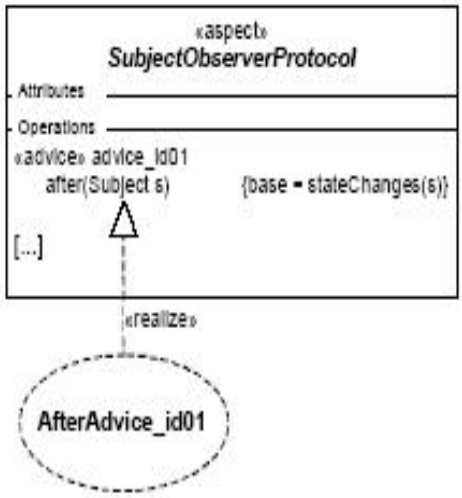

Figura 4.12: AODM: Notação para Advice [127].

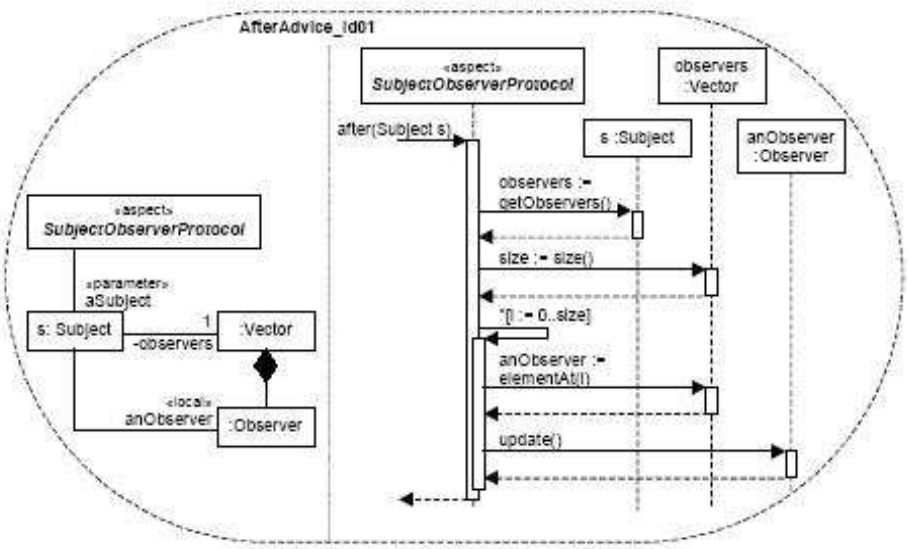

Figura 4.13: AODM: Colaboração para After Advice [127].

Processo de Combinação

A abordagem de AODM apresenta uma implementação de mecanismos de processos de combinação de Aspect em UML. Ela usa instruções de processos de combinação contidas nas metapropriedades complementares dos novos estereótipos de UML para pointcuts, advice, introduções e aspectos.

Os efeitos crosscutting na *estrutura* das classes base são determinados pelas instruções de processos de combinação contidas nas declarações de introdução e podem ser visualizados realçando as definições de classes nos diagramas de classes de UML. Os efeitos crosscutting no *comportamento* dos objetos base são determinados pelas instruções de processos de combinação contidas nas declarações de advice e podem ser visualizados realçando os *links* nos diagramas de colaboração de UML. A Figura 4.14 mostra uma colaboração que define como um objeto base interage com o aspecto depois do processo de combinação.

4.6.3 Composition Patterns: uma abordagem baseada em metamodelo

Padrões de Composição (*Composition Patterns*) [32, 30] são a abordagem à modelagem orientada a aspectos com UML mais referenciada na literatura. O interessante é que tem suas raízes no Design Orientado a Sujeitos, o correspondente no nível de projeto à Programação Orientada a

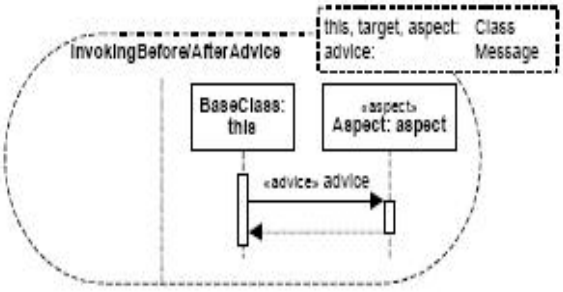

Figura 4.14: AODM: processo de combinação de advice [127].

Sujeitos (POS) [59]. O Modelo de Projeto Orientado a Sujeitos (MPOS) ou Subject-Oriented Design Model (SODM) [32] oferece uma base conceitual consistente para o projeto dos programas orientados a sujeitos. Os Padrões de Composição baseiam-se em uma combinação de MPOS (para a decomposição e a composição separada de projetos potencialmente sobrepostos) e templates de UML, com o objetivo de modelar os requisitos que têm impactos *crosscutting* em várias classes. Em [29], Clarke oferece um mapeamento de Padrões de Composição para os elementos de programação de AspectJ e, assim, ilustra como esses padrões podem ser usados para o projeto requisitos de crosscutting em programas de AspectJ.

O modelo de Padrões de Composição

O modelo de Padrões de Composição baseia-se nos seguintes conceitos principais de MPOS:

Sujeitos de design (*Design subjects*) MPOS oferece suporte a modelos de design separados como visões independentes chamadas de sujeitos de design (*design subjects*), denotados com um estereótipo ≪subject≫ em um pacote de UML. Os sujeitos de design encapsulam todos os elementos de modelo relacionados a um determinado requisito (Figura 4.17). Os Padrões de Composição são *sujeitos de design parametrizados* que encapsulam todos os elementos de modelo relacionados a um requisito transversal (Figura 4.17).

Relacionamentos de composição MPOS introduz o *relacionamento composição* (*composition relationship*) como uma nova construção no nível de design que oferece suporte à especificação de como os modelos de design devem ser compostos. Os relacionamentos de composição indicam elementos que correspondem e como devem ser integrados.

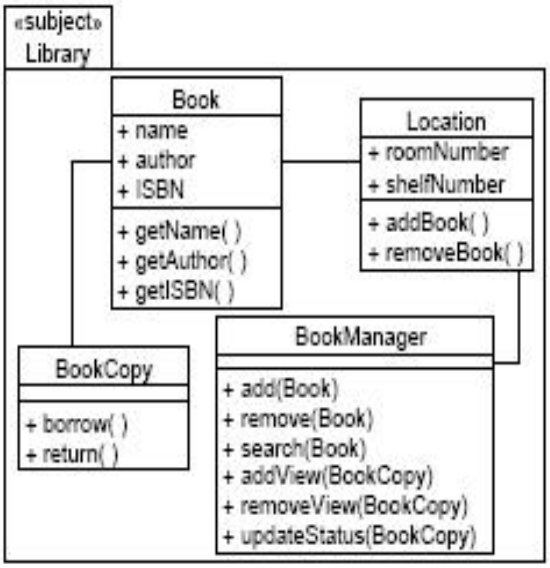

Figura 4.15: Um sujeito no nível de design [32].

Estratégias de integração A composição em MPOS usa um conjunto de *sujeitos de entrada* e integra-os de acordo com a estratégia definida por um (conjunto de) relacionamento(s) de composição, produzindo um *sujeito de saída*. Semânticas de integração diferentes definem como os elementos especificados como correspondentes são compostos. A estratégia de integração particular relevante para o Padrão de Composição é o `merge`. A integração de `merge` combina os sujeitos de entrada. As operações mescladas combinam os comportamentos realizados por cada operação correspondente. Isso pode ser alcançado com a criação de um modelo de interação que realiza a operação composta delegando a cada uma das operações de entrada correspondentes. A Figura 4.16 ilustra dois sujeitos, cada um com uma classe. O relacionamento de composição entre os dois determina que os sujeitos sejam *mesclados* (representado por cabeças de seta em cada extremidade do arco) e que os elementos com o mesmo nome correspondam entre si (representado por `match[name]` no relacionamento). A Figura 4.16 também apresenta o resultado da composição.

As classes `S1.A` e `S2.A` são mescladas porque possuem o mesmo nome. Cada uma dessas classes possui uma especificação `op1()` que é correspondente. A semântica de *merge* define a saída `op1()` como delegando para as duas especificações de entrada de `op1()`, que foram renomeadas ($S1_op1()$ e $S2_op1()$, ao colocar o nome do sujeito de entrada seguido por um traço sublinhado e o nome da operação, a

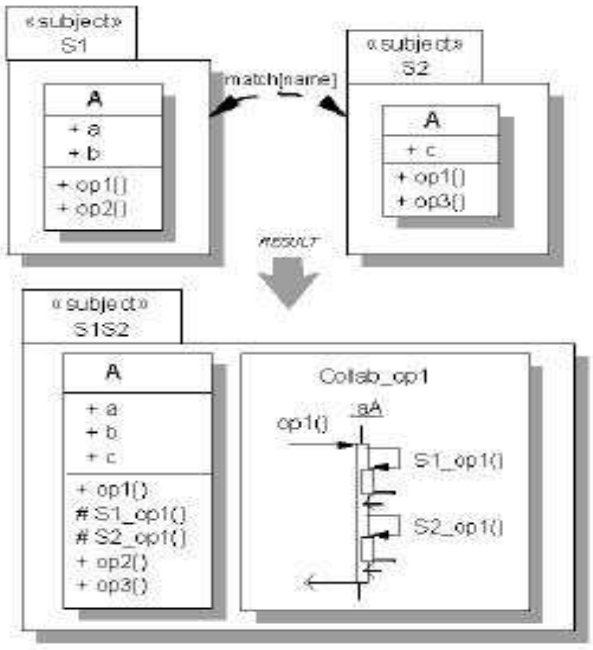

Figura 4.16: Exemplo de integração com Merge [32].

fim de evitar colisões de nomes). Um diagrama de interação é gerado para definir o comportamento de delegação.

Elementos de design

As categorias das construções da abordagem de Padrões de Composição são avaliadas com a ajuda do framework conceitual proposto no Capítulo 3.

Aspectos De acordo com [29], um sujeito de padrão de composição (*composition pattern subject*) é o equivalente a um aspecto no nível de design.

Especificação de pontos de combinação A especificação de pontos de combinação tem o suporte de parâmetros de templates. Os parâmetros de templates de operação podem ser definidos e referenciados dentro de especificações de interação, denotando que são pontos de combinação para o comportamento crosscutting.

Características Transversais Dentro de um diagrama de interação, o comportamento crosscutting pode ser especificado a fim de executar quando um template de operação é chamado. Esse comportamento é equivalente a uma característica transversal comportamental.

Os elementos de design que não são elementos de template podem ser definidos dentro de padrões de composição. São classes, atributos, operações ou relacionamentos. s atributos e operações são equivalentes a uma característica transversal estrutural.

Crosscutting O relacionamento de composição com a semântica de integração *merge* e a semântica de template de UML oferece suporte a crosscutting.

Diagramas

Nos diagramas de UML, um padrão de composição, um sujeito de design parametrizado, é representado como um tipo especial de pacote de UML, estereotipado como ≪subject≫, com todos os parâmetros de templates combinados em uma única caixa de template e colocados na caixa de sujeitos. Os sujeitos de design (não transversais) comuns são apresentados sem a caixa de template. Um padrão de composição pode conter um diagrama de interação para cada operação composta, que descreve seu comportamento crosscutting como uma delegação a cada uma das operações de entrada (renomeadas) correspondentes.

O relacionamento de composição é representado com uma linha tracejada com um complemento `bind[]` que define os elementos (potencialmente múltiplos) que substituem os templates dentro do padrão de composição.

Exemplo: o padrão de composição Observer

No padrão de composição Observer, duas classes de padrão são definidas, Subject e Observer. A interação representada na Figura 4.17 especifica um comportamento crosscutting. O parâmetro de templates de sujeito _aStateChange(..) é complementado com um comportamento relacionado à notificação de observadores de alterações no estado. A operação de substituição real possui um "_", ou seja, _aStateChange(..).

A Figura 4.18 mostra a notação para especificar a composição de Library com o padrão Observer. Um relacionamento de composição é especificado entre os sujeitos, em que BookCopy desempenha o papel de Subject e BookManager exerce o papel Observer, respectivamente. A Figura 4.19 mostra a saída da composição de Observer com Library.

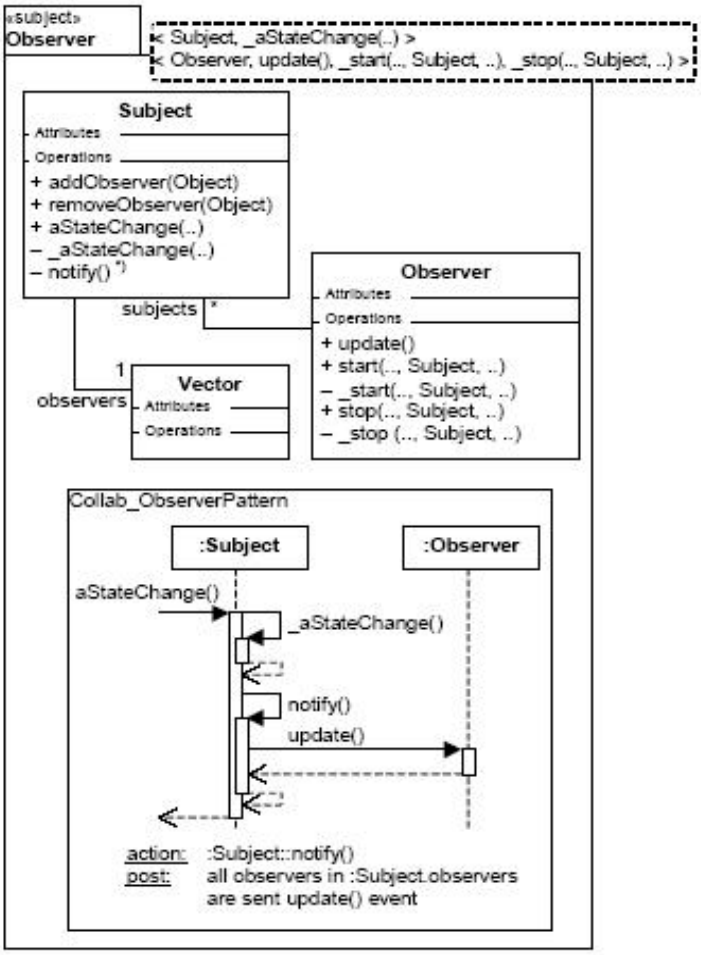

Figura 4.17: Padrões de composição: o padrão de projeto Observer Pattern [29].

O Metamodelo

Os padrões de composição são templates de UML para sujeitos de design que esperam classes e operações como *parâmetros de templates*. As classes de padrão (*pattern classes*) são os placeholders que devem ser substituídos por elementos de classe reais, enquanto *operações de template* são placeholders para operações. A Figura 4.20 descreve o metamodelo de UML que especifica os Padrões de Composição.

As classes de padrão são representadas pela metaclasse `PatternClass`, que descreve o relacionamento entre um sujeito de design parametrizado e classes de padrão contidos nele. As operações de template são representadas pela metaclasse `TemplateParameter`, que descreve o relacionamento entre uma classe de padrões e suas operações de template.

As classes de padrão são usadas a fim de modelar o *crosscutting* estrutural estático. As operações de template, por outro lado, são usadas

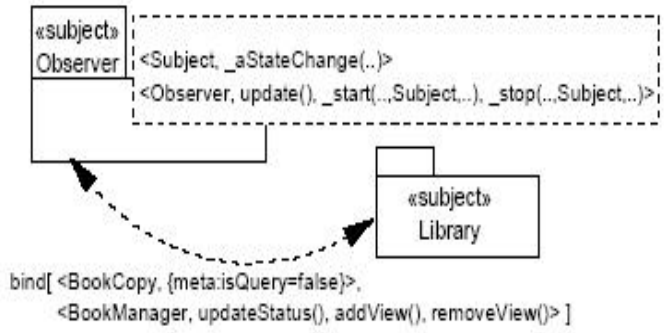

Figura 4.18: Composição de Observer com Library [29].

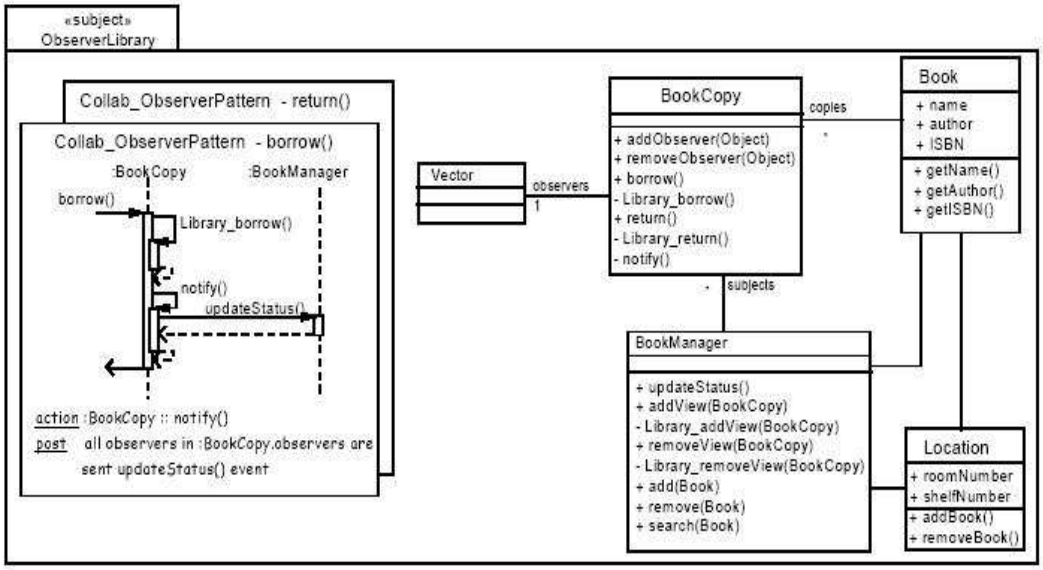

Figura 4.19: Saída de composição: Library com Observer [29].

para modelar o crosscuting comportamental.

4.6.4 Discussão

Elementos de *crosscutting* são tratados em UML como cidadãos de segunda classe, que costumam ser nomeados, mas não explicitamente especificados. Não há suporte para modularizar os *concerns* transversais usando aspectos nem para expressar a semântica *crosscutting* em UML. Portanto, UML deve ser estendida para tratar dos problemas de MOA de forma adequada.

As abordagens existentes para MOA/DOA estendem UML para fornecer soluções adequadas para linguagens orientadas a aspectos, em espe-

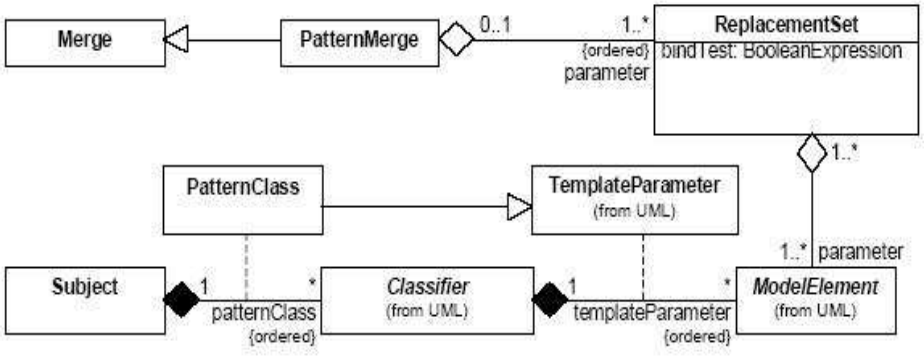

Figura 4.20: Metamodelo de padrões de composição[32].

cial AspectJ. De fato, o projeto da linguagem AspectJ influenciou muito o consenso da comunidade orientada a aspectos em relação aos principais conceitos e semântica para DSOA.

Os Padrões de Composição de Clarke baseiam-se na abordagem de MPOS [32]. O MPOS é uma abordagem bem fundamentada para modelar programas orientados a sujeitos, mas não foi originalmente proposta para lidar com *concerns* transversais . Portanto, os Padrões de Composição especificam *concerns* transversais de uma maneira orientada a sujeitos que é inapropriada para o design de programas orientados a aspectos em AspectJ por várias razões diferentes. Para superar essas limitações, a pesquisa de Clarke sobre a abordagem de Padrões de Composição está evoluindo para Theme/UML [31], com o objetivo de fornecer uma "linguagem de design de DSOA padrão".

O AODM de Stein, por outro lado, apresenta um modelo de design que incorpora a semântica de AspectJ. Ele propõe um conjunto de extensões que complementam UML para o projeto de programas orientados a aspectos apenas com AspectJ.

A Tabela 4.3 apresenta um resumo das características de cada abordagem apresentada nesta Seção.

4.7 Considerações Finais

O número de pesquisadores que estão trabalhando nas diversas questões de modelagem orientada a aspectos [9] está aumentando, mas muitos problemas relacionados à especificação no nível do design, visualização e construção de modelos orientados a aspectos ainda permanecem

sem solução.

No Capítulo 2, defendemos que a essência e os benefícios da POA podem ser obstruídos pela diversidade das abordagens de POA ainda em evolução. Um argumento semelhante pode ser aplicado à MOA e DOA. A menos que um framework conceitual que incorpore o consenso da comunidade orientada a aspectos sobre os principais conceitos para o DSOA seja adotado e uma base rigorosa para a compreensibilidade das linguagens de modelagem orientadas a aspectos seja fornecida, a MOA também estará espalhada em diversidade.

Além disso, sentimos falta de um modelo de aspectos independente da linguagem e de *alto nível* que omita detalhes específicos de implementação e que enfatize os detalhes comportamentais e estruturais importantes no *nível do design*. Sem a devida atenção às propriedades e aos conceitos orientados a aspectos no nível do design, pode ser difícil gerenciar a complexidade e aumentar a compreensibilidade, a evolução e a reusabilidade no desenvolvimento de software orientado a aspectos.

	AODM	CP	UMLAUT
aspectos são de primeira classe	"aspectos" são classes estereótipadas	"aspectos" são pacotes estereótipados	não
interações explícitas entre aspectos	não	não	não
extensões sensíveis ao contexto	sim	não	não
semântica para crosscutting	AspectJ	delegação	espalhada em regras de transformação
quantificação	sim	sim	não
visão do sistema combinado	sim	Subject output	modelos UML
dependência de linguagem	AspectJ	POS (sujeitos)	customizável
ferramenta	não	não	ferramenta de transformação
extensão de UML	Light-weight	Heavy-weight	não

Tabela 4.3: Comparação entre abordagens.

Uma abordagem orientada a modelos para o design orientado a aspectos.

Um modelo de aspectos independente de linguagem no nível de design também pode ser o ponto de partida para uma abordagem orientada a modelos para o desenvolvimento de software orientado a aspectos, em que os modelos de aspectos em conformidade com UML e no nível de design sejam mantidos de forma separada dos modelos de implementação específicos.

A iniciativa da Arquitetura Orientada a Modelos (AOM) da OMG ou Model-Driven Architecture (MDA) [91] é uma abordagem de desenvolvimento baseada em modelos de UML. Uma *especificação MDA* completa consiste em um modelo de UML independente da plataforma (Platform-Independent UML Model), mais um ou mais modelos específicos a plataforma (*Platform-Specific Models* (PSM)) e conjuntos de definição de interface, cada um descrevendo como o modelo base é implementado em uma plataforma diferente.

Nesse contexto, a extensão de UML para oferecer suporte à MOA permite que os desenvolvedores de software orientado a modelos se beneficiem das vantagens da tecnologia orientada a aspectos.