2
Programação Orientada a Aspectos

Este capítulo apresenta uma visão geral da programação orientada a aspectos (POA).

O princípio da separação de *concerns* é usado como um framework geral que caracteriza a POA e suas principais abordagens relacionadas – *Programação Adaptativa*, *Filtros de Composição*, *Programação Orientada a Sujeitos* e *Separação Multidimensional de Concerns* – com o objetivo de definir o contexto técnico no qual construiremos esta tese.

2.1
Visão Geral

A Programação orientada a aspectos (POA) é uma tecnologia de programação em evolução que oferece suporte a novos mecanismos de abstração e composição com o objetivo de permitir alcançar uma melhor separação de *concerns* no nível da implementação.

A POA é evolucionária porque considera contribuições reconhecidas da separação de *concerns* proporcionadas por tecnologias anteriores (principalmente, mas não apenas, a programação orientada a objetos), enquanto oferece suporte a uma nova abstração – o aspecto – e novos mecanismos de composição para lidar com alguns *concerns* especiais que não são tratados adequadamente por essas tecnologias. Esses *concerns* especiais são chamados de *concerns* transversais (*crosscutting concerns*) porque afetam naturalmente os limites de outros *concerns*. Em geral, esses *concerns* transversais cuidam de requisitos que envolvem restrições globais, propriedades sistêmicas e protocolos, como sincronização, persistência, tratamento de erros, mecanismos de auditoria, entre outros. Sem os meios apropriados para a separação, os *concerns* transversais tendem a ficar espalhados e entrelaçados com outros *concerns*. As conseqüências naturais são uma menor compreensibilidade, uma menor evolutibilidade e menos reusabilidade dos artefatos do código.

O termo "*programação orientada a aspectos*" e o acrônimo POA foram cunhados em 1996 no Centro de Pesquisa da Xerox em Palo Alto (Xerox Palo Alto Research Center - PARC). Depois de quase uma década de pesquisa ativa [146, 151, 147, 148, 149, 139, 8], a POA está alcançando sua maturidade entre pesquisadores e praticantes. Da mesma forma, outros pesquisadores desenvolveram mecanismos e abstrações inovadores para melhorar (ou *avançar*) a separação de *concerns* oferecida pelo paradigma de objetos. Alguns trabalhos de pesquisa relacionados à POA incluem *Programação Adaptativa* [82], *Filtros de Composição* [3], Programação Orientada a Sujeitos [59], Separação Multidimensional de Concerns [132] e Programação Gerativa [21]. Cada abordagem trata de diferentes subconjuntos de problemas e explora um ponto diferente da solução, mas todas elas compartilham o objetivo comum de promover uma melhor separação de *concerns* no nível da implementação.

Os termos Separação Avançada de Concerns (SAdC) [153, 150, 152, 154], Programação Pós-Objeto (PPO) [40, 60] e, mais recentemente, *Desenvolvimento de Software Orientado a Aspectos* (DSOA) [8] são normalmente usados para caracterizar a nova geração de tecnologias de separação de *concerns* à qual a POA e as abordagens relacionadas oferecem suporte.

2.2 Separação de Concerns

Em sucessivas gerações de linguagens de programação, os mecanismos de composição e abstração existentes evoluíram para promover a expressão de soluções para problemas do mundo real de forma mais natural, assim como para possibilitar alcançar a separação de *concerns* no desenvolvimento de programas de software complexos. Cada geração representativa reinventa a prática da programação e, como conseqüência, a prática do desenvolvimento de software.

Nesta seção, apresentamos duas abordagens atuais usadas na programação de sistemas grandes: *programação estruturada* e *programação orientada a objetos*. Cada abordagem adota um determinado tipo de decomposição e usa a estrutura resultante como o framework para expressar outras perspectivas. As duas oferecem suporte a diversas melhorias para a separação de *concerns* que se mostraram úteis e que ainda permanecem parte da prática de software de hoje em dia. Entretanto, elas têm limitações ao lidar com alguns *concerns* especiais, o que pode impactar os benefícios

esperados fornecidos pela separação de *concerns* ; para contorná-las, foram propostas novas abordagens de PPO (como a POA).

2.2.1 Programação estruturada

No início da história da programação, os primeiros programas de software eram escritos em linguagem de máquina e eram desenvolvidos basicamente para realizar cálculos numéricos em projetos militares. Mais tarde, com a introdução de Fortran e de outras linguagens, a programação e o desenvolvimento de software passaram por algumas mudanças evolucionárias, o que permitiu novas oportunidades para a criação de sistemas intensivos de software mais complexos e maiores.

Por volta dos anos 60, alguns projetos grandes de software eram conhecidos por sua dificuldade de fornecer resultados a tempo, dentro do orçamento disponível e com fatores de qualidade necessários, como a confiabilidade e a adaptabilidade. Uma resposta a esse problema é a *programação estruturada*, cujo manifesto foi uma curta carta de Edsger W. Dijkstra para o editor, chamada "Go to statement considered harmful" [37].

A programação estruturada oferecia suporte à decomposição algorítmica (dividindo o problema de acordo com seus verbos relevantes). Os programas eram construídos a partir de rotinas – procedimentos e funções – que têm um único ponto de entrada e uma única saída. Evitava-se o uso do comando `goto` (que permite pular para dentro e para fora de uma rotina, gerando o código *sphagetti*) e recomendavam-se três tipos básicos de estruturas de controle: *seqüenciamento* (faça A, depois B, depois C), *alternância* (faça A ou faça B) e *iteração* (repetir A até que seja satisfeita alguma condição).

A programação estruturada também trouxe algumas boas práticas e princípios. O *projeto top-down* e o *refinamento por etapas* orientavam o programador a fim de definir as instruções de um procedimento primeiro para depois preencher os detalhes. O suporte à *modularidade* preconizou unidades independentes com interfaces simples entre elas. A *separação de concerns* foi proposta para lidar com algumas questões importantes (ou *concerns*), uma de cada vez, conforme retirado do livro de Dijkstra "Uma disciplina de programação" [36]:

> Eu costumo me referir a isso como "separação de concerns", porque o objetivo é tentar lidar com as dificuldades, as obrigações, os desejos e as restrições, um a um. [..] A escolha

> crucial é em relação a que aspectos serão estudados "isoladamente", como desentrelaçar o nó amorfo original de obrigações, restrições e objetivos globais em um conjunto de "concerns" que admitem uma separação bastante eficaz.

A modularidade está estreitamente relacionada ao princípio de separação de *concerns*. Entretanto, de acordo com Parnas, a eficácia de uma "modularização" também depende dos critérios usados para a decomposição do sistema em um conjunto de módulos [114]. A programação estruturada usa a decomposição algorítmica; cada módulo no sistema denota uma etapa importante em um processo geral [16].

As operações entrelaçadas e espalhadas que são replicadas em várias partes do código podem ser devidamente modularizadas por um procedimento que as implementa; contudo, apesar de cada corpo de operação ser substituído por uma chamada de procedimento correspondente, essas chamadas ainda são espalhadas e entrelaçadas no corpo de outros procedimentos. Além disso, os procedimentos e os dados são separados, mas interdependentes; uma vez que os procedimentos (módulos) são eficazes na modularização de "etapas", os detalhes da implementação dos dados podem ser *espalhados* por vários processos não relacionados e *entrelaçados* dentro de cada corpo de procedimento. Portanto, o *concern de dados* não é adequadamente modularizado.

De acordo com Booch [16], os programas resultantes da decomposição algorítmica tendem a ser maiores, menos flexíveis a mudanças e mais difíceis de desenvolver e reutilizar ao longo do tempo.

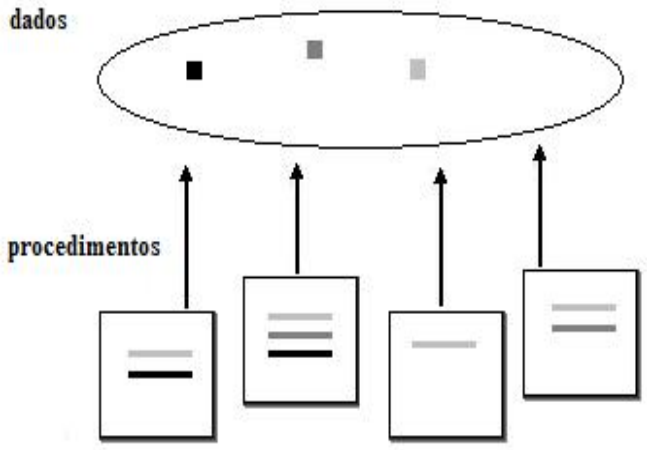

Figura 2.1: Espalhamento e entrelaçamento em programas estruturados.

Exemplo

Considere um programa procedimental (estruturado) para a manipulação de figuras geométricas simples.

Uma figura consiste em alguns elementos, que podem ser pontos ou linhas. O programador deve escrever uma série de procedimentos com nomes como desenhar, girar, escalonar, etc. Cada um desses procedimentos relacionados a verbos pode ser aplicado a todos os tipos de figuras; as figuras são como substantivos incorporados em estruturas de dados separadas dos procedimentos.

Os detalhes da implementação sobre a representação de figuras podem estar *espalhados* por vários procedimentos. Ademais, os detalhes de implementação da representação e manipulação de todos os tipos de figuras podem estar *entrelaçados* em cada corpo de procedimento. A introdução de novos tipos de figuras ou a modificação de representações existentes podem exigir grandes alterações no programa. Por exemplo, se um círculo for descrito em termos de um centro e um raio em vez de um conjunto de pontos, todos os procedimentos devem tratar os círculos como um caso especial.

2.2.2 Programação Orientada a Objetos

Nos anos 60, surgiram as primeiras idéias básicas sobre objetos e classes em Simula 67, uma linguagem de programação desenvolvida por Ole-Johan Dahl e Kristen Nygaard. Nos anos 70, Alan Kay criou a linguagem Smalltalk nos laboratórios da Xerox PARC. Uma década depois, já estavam em uso diversas linguagens orientadas a objetos, em especial C++ de Bjarne Stroustrup e, mais tarde, Java. Características de orientação a objetos também foram incorporadas a novos dialetos de linguagens mais antigas, como Lisp. Desde os anos 80, a prática do desenvolvimento de software passou a ser influenciada cada vez mais pela programação orientada a objetos.

A *programação orientada a objetos* oferece suporte à decomposição orientada a objetos (dividindo o problema de acordo com seus nomes relevantes). O ideal é que cada módulo no sistema denote classes e objetos derivados diretamente do vocabulário do domínio do problema [16] (Figura 2.2). A programação orientada a objetos modulariza os procedimentos e os dados – nomes e verbos – em uma unidade independente e auto-contida, o objeto. Os dados e os procedimentos não são mais separados, e sim acoplados de forma coesa; e os detalhes da implementação são omitidos. As

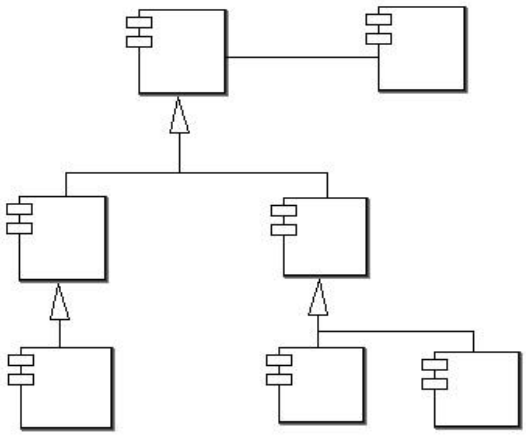

Figura 2.2: Decomposição orientada a objetos.

partes comuns dos objetos são organizadas em classes. Os programas são criados a partir de objetos que se comunicam enviando e recebendo mensagens. Evita-se o compartilhamento global de dados. As classes de objetos são organizadas em uma hierarquia em forma de árvore, permitindo que os dados e os procedimentos definidos em uma classe sejam herdados por suas subclasses. A herança e o polimorfismo oferecem suporte à economia de expressão e à reutilização. O ideal é que uma classe ou um conjunto de classes relacionadas encapsulem o código necessário para algum *concern*.

A programação orientada a objetos também traz algumas boas práticas e princípios. O *projeto orientado a objetos* orienta o programador na busca pela decomposição "certa" em classes e objetos e na sua organização em hierarquias. O princípio de *encapsulamento*, ou ocultação de informação (*information hiding*), requer que a visão interna de um objeto permaneça oculta, de forma que futuras mudanças não afetem outros objetos. O princípio de *abstração* enfoca na concentração em similaridades e ignora algumas diferenças. A *hierarquia* impõe um ordenamento de abstrações para promover a compreensibilidade. Princípios como o *princípio open closed* (um módulo deve ser aberto para extensão, mas fechado para modificação), o *princípio de substituição de Liskov* (uma classe derivada pode substituir uma classe base) e o *princípio da inversão da dependência* (dependa das abstrações; não dependa das concretizações) ajudam a gerenciar as dependências dos módulos e a complexidade global.

De acordo com Booch [16], a decomposição orientada a objetos resulta em sistemas menores, mais robustos em relação a mudanças e, portanto, mais fáceis de evoluírem ao longo do tempo.

Exemplo

Considere um programa orientado a objetos para a manipulação de figuras geométricas simples.

Uma figura consiste em alguns elementos, que podem ser pontos ou linhas. Os objetos encapsulam dados e procedimentos. Em um objeto que representa um ponto, os dados são as coordenadas x e y, e os procedimentos, ou métodos, operam nessas coordenadas. A representação de dados está oculta, e os dados só podem ser tratados por métodos dos objetos. Novos tipos de figuras surgem pela especialização. Além disso, como os dados internos estão ocultos, e, enquanto as interfaces estiverem estáveis, a modificação de representações existentes fica localizada em classes e não requer grandes alterações no programa.

Agora, suponha que um novo requisito peça a atualização de uma janela de exibição sempre que uma figura for movida ou modificada. A solução OO direta é incluir uma chamada `display.update()` em todos os métodos que modificam a aparência de uma figura (como desenhar, girar ou escalonar), de forma que quando esses métodos forem chamados nos objetos das figuras, o método de atualização (update) é chamado no objeto de exibição. A implementação do novo requisito – atualizar a exibição (*display update*) – fica entrelaçada com alguns métodos básicos definidos para objetos de figuras (veja a Figura 2.3).

2.2.3 Programação Pós-objeto

A engenharia de software orientada a objetos conseguiu oferecer suporte ao desenvolvimento de sistemas de software de alta qualidade. Contudo, após uma década de criação de sistemas com POO e com o aumento crescente da complexidade das aplicações de software, foram identificados alguns problemas. Por exemplo, a POO apresenta algumas limitações no tratamento de *concerns* que cuidam dos requisitos que envolvem restrições globais e propriedades sistêmicas, como a sincronização, a persistência, o tratamento de erros, os mecanismos de auditoria, entre outros. A razão é que esses tipos de *concerns* não se decompõem de forma organizada em comportamentos centrados em um único local (por exemplo, classe) [40]. Eles tendem a ficar entrelaçados ou espalhados com outros *concerns*.

Qualquer realização de decomposição de sistemas – algorítmica ou orientada a objetos – descobrirá que alguns *concerns* são bem localizados dentro de um módulo específico, enquanto outros são espalhados e entrelaçados

em vários módulos. Esses últimos são chamados de *concerns* transversais (*crosscutting concerns*) porque atravessam os limites de outros *concerns*.

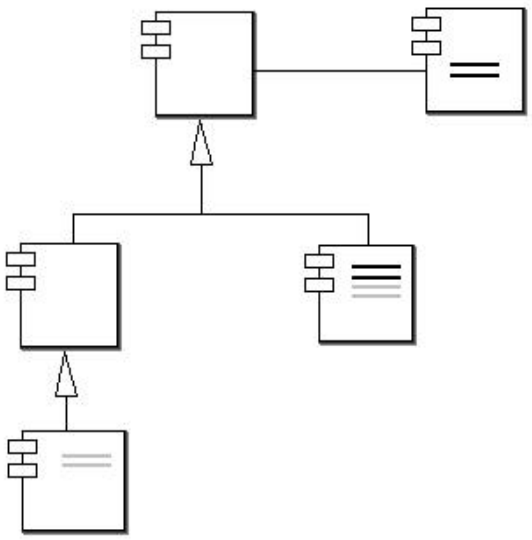

Figura 2.3: Espalhamento e entrelaçamento em programas orientados a objetos.

As tecnologias de programação pós-objeto (PPO) propõem mecanismos e abstrações inovadores para melhorar (ou *avançar*) a separação de *concerns* oferecida pelo paradigma de orientação a objetos. Cada tecnologia de PPO cuida de diferentes subconjuntos de problemas relacionados a POO e se concentra em diferentes pontos do espaço de soluções.

A POA se concentra em mecanismos para a simplificação da realização desses *concerns* transversais . Enquanto a tendência na POO é encontrar pontos comuns entre as classes e empurrá-las para cima na árvore de herança, a POA tenta realizar *concerns* espalhados como abstrações de primeira classe, chamados de *aspectos*, e lançá-los horizontalmente para fora da estrutura do objeto [40]. A POA também oferece novos mecanismos que permitem a composição transparente e flexível de aspectos na estrutura de objetos. A Figura 2.4 apresenta a topologia de um programa orientado a aspectos, em que os losangos representam aspectos, e as setas representam os relacionamentos entre aspectos e classes.

Exemplo

Considere mais uma vez o programa orientado a objetos para a manipulação de figuras geométricas simples.

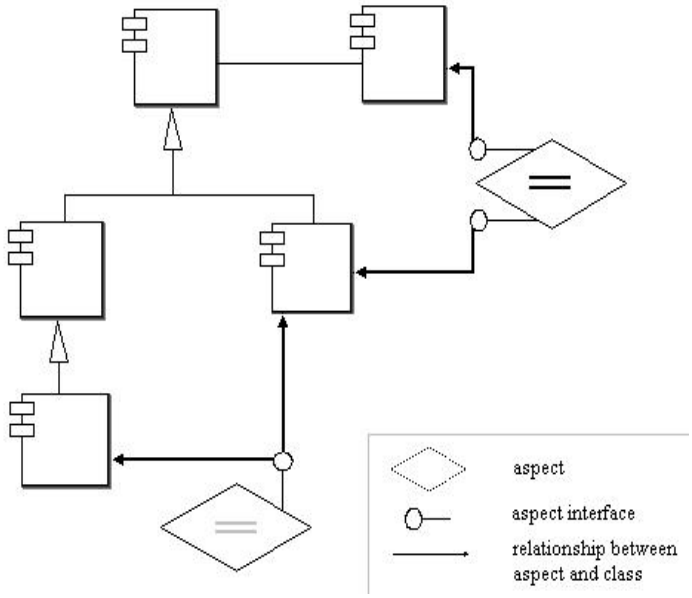

Figura 2.4: Decomposição orientada a Aspectos

Em vez de inserir as chamadas para `display.update()` diretamente nos métodos definidos para as figuras, uma abordagem orientada a aspectos modulariza a chamada `display.update()` em um *aspecto*. O aspecto deve especificar que *depois de uma chamada para desenhar, girar ou escalonar, o sistema deve executar o código definido no aspecto* – nesse caso, a chamada `display.update()`.

As abordagens apresentadas na Seção 2.3 oferecem diferentes soluções para a modularização de requisitos *crosscutting*, como atualizar a exibição (`display.update()`), através de aspectos, sujeitos, filtros, etc., assim como para combinar aspectos e objetos a fim de criar um sistema executável.

2.3 Separação de Concerns Avançada

A programação orientada a aspectos evoluiu a partir de pesquisas de diferentes grupos, construídas sobre diferentes aquisições obtidas a partir da programação orientada a objetos e da programação em metanível (Tabela 2.1).

De 1990 a 1996, muitos pesquisados reconheceram problemas em suas áreas específicas de conhecimento para separar *concerns* individuais e começaram a tentar resolver esse problema. No relatório "Separation of concerns" [63], Hürsch e Lopes identificam e analisam os principais *concerns* de software que foram referenciados na literatura até 1994: recuperação de

Data	Descrição
1967	Simula [34]: Primeira linguagem orientada a objetos
1970s	Separation of Concerns (Dijkstra, Parnas)
	Programação Estruturada
1980s	Programação Orientada a Objetos
	Pesquisa inicial sobre Separation of Concerns Techniques
1992	Adaptive Programming (Lieberherr, Northeastern University)
	Composition Filters (Aksit, Twente University)
1993	Subject-Oriented Programming (Harrison and Ossher, IBM)
1995	Relatório: "Separation of Concerns" (Hürch and Lopes)
	Pesquisa sobre Aspect-Oriented Programming
1996	Aspect-Oriented Programming (Kiczales, Xerox PARC)
1997	Primeiro Workshop sobre Aspect-Oriented Programming (ECOOP 1997)
	Artigo seminal: "Aspect-Oriented Programming" (Kiczales *etal.*)
	`AspectJ`
	Tese: "D: A Language Framework for Distributed Programming" (Lopes)
	Pesquisa sobre Advanced Separation of Concerns
1999	Multi-dimensional Separation of Concerns (Tarr, Harrison and Ossher, IBM)
	`Hyper/J`
2000	Primeiro Workshop sobre Advanced Separation of Concerns (OOPSLA 2000)
	Artigo: "Aspect-Oriented Programming is Quantification and Obliviousness" (Filman and Friedman)
	Pesquisa Aspect-Oriented Software Development
2001	Edição Especial da CACM sobre Aspect-Oriented Programming
2002	First Int'l Conf. on Aspect-Oriented Software Development (AOSD 2002)

Tabela 2.1: Pesquisa sobre POA: Perspectiva Histórica.

falhas, restrições em tempo real, controle de local, sincronização e organização de classes. Da Seção 2.3.1 até a 2.3.4 tratamos de abordagens representativas desenvolvidas no início dos anos 90 (antes do termo "aspecto" ter sido cunhado), discutindo seu suporte à separação de *concerns* no nível da implementação.

De 1996 a 1998, o trabalho de Gregor Kiczales e seu grupo no Xerox PARC ganha destaque. A Seção 2.3.5 cobre as principais idéias que resultaram na POA, e a Seção 2.3.6 apresenta AspectJ [11, 74, 75], a primeira linguagem de programação orientada a aspectos de propósito geral e a linguagem padrão incorporada pela comunidade da POA.

Em 1999, o Centro de Pesquisa T.J. Watson da IBM apresentou *Hyperspaces* [132, 111, 113], uma abordagem para a Separação Multidimensional de Concerns (SMDdC). SMDdC é uma evolução na programação orientada a sujeitos [59]. A Seção 2.3.7 apresenta as principais idéias que resultaram na SMDdC, e a Seção 2.3.8 apresenta Hyper/J [131], uma ferramenta que oferece suporte a SMDdC para Java.

Desde 2001, a POA é considerada uma possível convergência desses caminhos de pesquisa independentes [40], e vários grupos de pesquisa começaram a discutir o papel dos aspectos em outras fases do processo de desenvolvimento de software. A Seção 2.4 trata dessas tendências e das questões relacionadas.

2.3.1 Programação Metanível

A programação metanível é um paradigma de programação baseado na reflexão computacional [122, 93]. A *reflexão computacional* é o processo de raciocinar sobre e atuar sobre o próprio sistema computacional. Um *sistema reflexivo* oferece suporte a uma *arquitetura metanível*, que consiste em um *nível base* e um *metanível*. O metanível incorpora dados concretizáveis que representam ou implementam as facetas computacionais e estruturais do sistema.

Descrição

Em sistemas reflexivos, os elementos básicos da linguagem de programação, como classes ou invocação de objetos, são descritos no metanível e podem ser estendidos ou redefinidos pela programação metanível.

Um protocolo metaobjeto (PMO) oferece suporte à arquitetura metanível para o paradigma orientado a objetos; um PMO define um conjunto de regras para a manipulação e comunicação com metaobjetos [70].

Os metaobjetos interceptam os envios e os recebimentos de mensagens para objetos e podem realizar trabalho em nome (no lugar) de *concerns* transversais . Por exemplo, podem realizar o controle de local, verificar as restrições de sincronização, realizar a migração de parâmetros entre máquinas, realizar a persistência de dados, etc. Isso permite que os programas do nível base sejam escritos sem *concerns* transversais , que são programados no metanível. Além disso, ao ter reflexão estrutural (metaconhecimento sobre as relações entre as classes), a programação metanível pode proporcionar separação entre algoritmos e organização de dados [63].

Exemplo

A Figura 2.5 mostra uma implementação fechada e uma aberta para um sistema que requer habilidade de controle de local (*location control*)

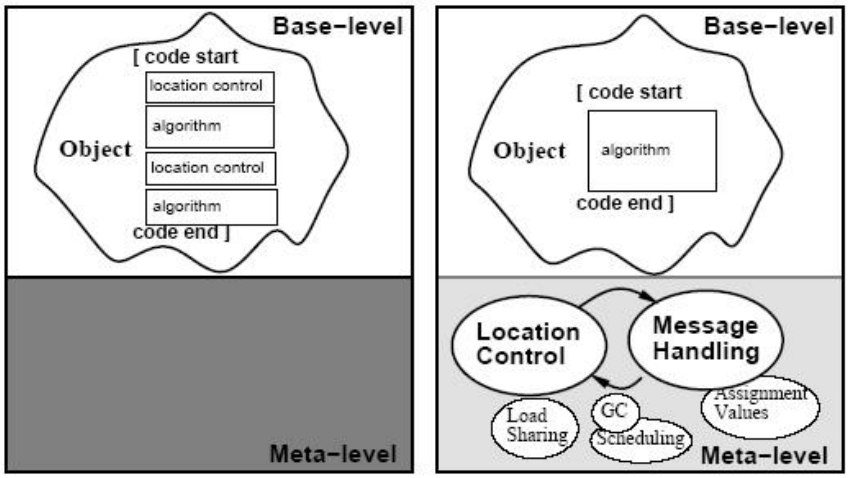

Figura 2.5: Implementação fechada X implementação aberta [108].

[108]. A implementação aberta tem o suporte de uma arquitetura metanível. Na implementação fechada (em que o metanível é fechado ao programador – consulte a figura mais à esquerda), o algoritmo de controle de local está entrelaçado ao programa da aplicação. Na implementação aberta, o algoritmo de controle de local é programado no metanível. Assim, o programa de controle do local e o programa da aplicação são separados.

2.3.2 Programação orientada a sujeitos

A Programação Orientada a Sujeitos (POS) é uma extensão do paradigma orientado a objetos com o objetivo de resolver o problema relacionado ao tratamento de diferentes perspectivas subjetivas definidas sobre os objetos que serão modelados.

A POS tem suas raízes no trabalho de William Harrison e Harold Ossher do Centro de Pesquisa T.J. Watson da IBM, *Subject-Oriented Programming (A Critique of Pure Objects)* [59]. Harrison e Ossher foram os pioneiros na idéia de especificação separada de diferentes hierarquias de classe, cada uma implementando um *concern*, e composição subseqüente de hierarquias apropriadas para construir variantes do sistema.

Descrição

A POS foi proposta para resolver o problema da separação alternativa de "visões" de uma única classe. Outros problemas tratados pela POS são (i) criação de extensões e configurações para o software sem modificar o programa fonte original, (ii) desenvolvimento de sistema multiequipe usando

modelos de domínio independentes, relacionados ou compartilhados e (iii) alinhamento entre os requisitos e o código [59].

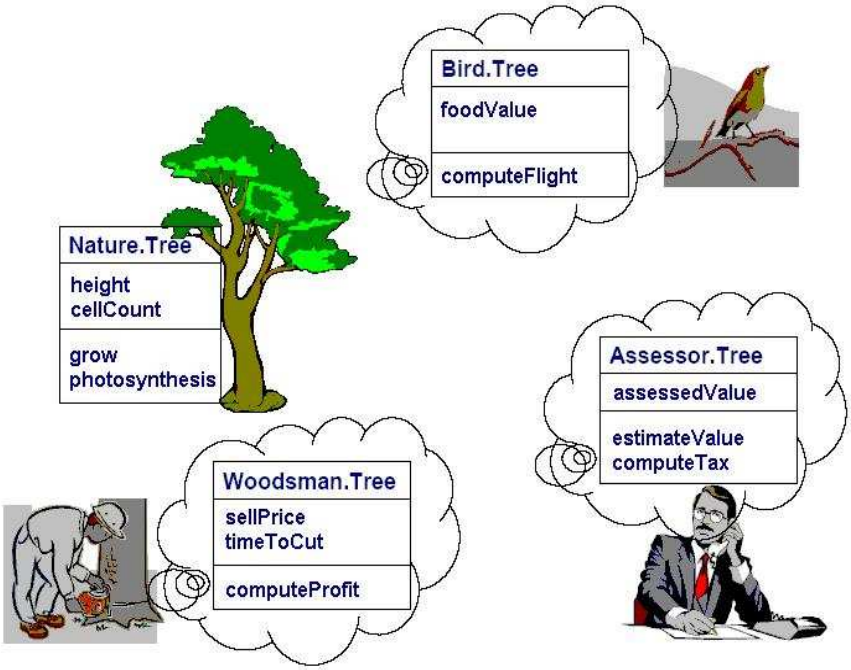

Figura 2.6: Visões subjetivas de uma árvore orientada a objetos.

Sujeitos

Um *sujeito* é uma coleção de classes ou fragmentos de classes cuja hierarquia modela seu domínio de forma subjetiva e própria. Um sujeito pode ser por si só uma aplicação completa ou pode ser um fragmento incompleto que precisa ser composto com outros sujeitos a fim de produzir uma aplicação completa.

Ferramenta de composição

A *composição de sujeito* combina hierarquias de classes a fim de produzir novos sujeitos que incorporam funcionalidade de sujeitos existentes. A POS oferece uma ferramenta de composição a fim de integrar os sujeitos [68].

Regras de correspondência

As composições são especificadas por meio de expressões escritas como coleções de *regras de composição*, normalmente uma regra geral junto com regras que especificam as exceções. As *regras de correspondência* especificam a correspondência, caso exista, entre classes, métodos e atributos de objetos, que pertencem a diferentes sujeitos. As *regras de combinação* especificam como as duas classes devem ser combinadas.

Regras de combinação

A programação orientada a sujeitos estende a programação orientada a objetos com abstrações adicionais (sujeitos) e mecanismos de composição que oferecem suporte ao programador no momento da criação de modelos que encapsulam diretamente um único *concern*, alinhando, assim, o código

com requisitos e removendo a espalhamento e o entrelaçamento de propriedades que afetam a separação de *concerns*.

Exemplo

A Figura 2.6 apresenta um exemplo adaptado de [59] que ilustra a idéia de visões subjetivas. Para muitos seres humanos, o objeto que representa uma árvore incluiria *propriedades intrínsecas* como altura (height), densidade (density) etc, enquanto para um lenhador, incluiria *propriedades extrínsecas* como preço de venda (sellPrice), tempo de corte (timeToCut) etc. Um consultor fiscal também teria sua própria visão das propriedades e dos comportamentos associados à árvore. As características incluem sua contribuição para o valor avaliado da fazenda na qual ela cresce.

Com a POS, cada visão é separada e modularizada como um sujeito. A composição de sujeitos combina hierarquias de classes para produzir a representação completa de uma árvore. A classe resultante incluirá os atributos e métodos independentes; os métodos e os atributos correspondentes serão combinados de acordo com as regras de combinação.

A árvore apresentada nesse exemplo pode ser uma árvore de sintaxe abstrata; o pássaro, um editor; o lenhador, um compilador e o consultor fiscal uma ferramenta de análise semântica [59].

2.3.3 Programação Adaptativa

A *Programação Adaptativa* (PA) [82] é uma abordagem que oferece suporte à separação de *concerns* comportamentais (métodos, estratégias) a partir de *concerns* estruturais (grafos de classes), no contexto de software orientado a objetos.

A PA evoluiu a partir do trabalho de Karl Lieberherr e seu grupo de pesquisa na Northeastern University sobre a Lei de Demeter. A PA oferece suporte a um modelo de programação baseado na idéia de *padrões de código* (*code patterns*) que é considerada uma das mais importantes raízes da programação orientada a aspectos [89].

PA e a Lei de Demeter

Em programas orientados a objetos, há vários pequenos métodos que chamam outros métodos a fim de passar as informações de uma parte do

diagrama de objetos no qual operam para outros métodos que operam em outras partes do diagrama. Isso implica que uma simples mudança no algoritmo de computação pode exigir que se visite novamente muitos métodos.

O grande número de pequenos métodos "que passam informações" nos projetos orientados a objetos é uma conseqüência direta da aplicação da Lei de Demeter [78]. A *Lei de Demeter* (LdD) é um princípio de design que afirma que um método deve conter apenas mensagens enviadas a si mesmo, a variáveis de instâncias locais e/ou argumentos de métodos. Ao seguir a LdD, evitamos longas seqüências de métodos de acesso (por exemplo, `object.op1().op2().op3()`) que amarram grandes estruturas de dados em métodos. No entanto, o resultado é um grande número de pequenos métodos que apenas passam informações.

Para resolver esse problema, a Programação Adaptativa usa *padrões de propagação* (*propagation patterns*) [82].

Descrição

Padrões de propagação
Grafo de classe

Os *padrões de propagação* definem as operações (algoritmos) sobre os dados. As relações entre as estruturas de dados da aplicação são descritas por grafos (chamados de *grafos de dicionário de classe*) aos quais os padrões se aplicam. Um compilador de padrões pega um conjunto de padrões e um grafo de classes e produz um programa orientado a objetos.

diretiva de propagação

Os padrões de propagação identificam os subgrafos de classes que interagem em uma determinada operação. As referências aos dados são feitas de uma maneira independente de estrutura (*structure-shy*) por meio de especificações de subgrafos sucintas, e o código de fato é definido em *wrappers* de código (visitantes adaptativos) ao longo de caminhos transversais (*traversal paths*), definidos por *diretivas de propagação* (*propagation directives*)) [79].

A PA também oferece suporte a outras categorias de padrões. Cada categoria de padrões trata de um *concern* diferente. Os padrões de transporte (*transportation patterns*) [87] são usados dentro dos padrões de propagação a fim de levar parâmetros para dentro e para fora dos subgrafos. Os padrões de sincronização (*synchronization patterns*) [86] definem os esquemas de sincronização entre os objetos em aplicações concorrentes para controlar o acesso dos processos à execução das operações.

Um padrão de propagação encapsula o comportamento de uma operação em um lugar, evitando, assim, o problema de espalhamento, mas

também abstrai na estrutura de classes, evitando, assim, o problema de entrelaçamento.

Exemplo

Em um trabalho mais recente sobre PA [84], os padrões de propagação são chamados de *métodos adaptativos*, e as diretivas de propagação são chamadas de *estratégias transversais*. O comportamento é expresso como uma descrição de alto nível de como alcançar os participantes da computação (uma estratégia transversal), além do que se faz quando cada participante foi alcançado (chamado de *visitante adaptativo*).

Método adaptativo
Estratégia transversal
Visitante adaptativo

A Figura 2.7 mostra um pequeno método adaptativo (`sumSalaries`) escrito em Java usando a biblioteca DJ [110, 38]. A finalidade do código é somar os valores de todos os objetos `Salary` alcançáveis pelos relacionamentos `has-a` de um objeto `Company`.

```
import edu.neu.ccs.demeter.dj.ClassGraph;
import edu.neu.ccs.demeter.dj.Visitor;

class Company {
  static ClassGraph cg = new ClassGraph(); // class structure

  Double sumSalaries() {
     String s = "from Company to Salary"; // traversal strategy

     Visitor v = new Visitor() { // adaptive visitor
        private double sum;

        public void start() { sum = 0.0 };
        public void before(Salary host) { sum += host.getValue(); }
        public Object getReturnValue() { return new Double(sum); }
     };
     return (Double) cg.traverse(this, s, v);
  }
// ... rest of Company definition ...
}
```

Figura 2.7: O método adaptativo `sumSalaries` [84].

A variável `cg` é um objeto de tipo grafo de classe que representa a estrutura de classes do programa. Um grafo de classe descreve os relacionamentos `is-a` e `has-a` entre as classes. A estrutura de classes é computada em um construtor de `ClassGraph` usando a reflexão. É usada pelo método

transversal como um contexto no qual se interpretam *estratégias transversais*.

O método transversal começa em um determinado objeto (`this`) e atravessa os caminhos especificados ("from Company to Salary") executando quaisquer métodos de visitante aplicáveis que surjam pelo caminho (nesse caso, no início, para inicializar a soma; quando se alcança cada objeto `Salary`, para adicionar o valor à soma; e no final, para criar o valor de retorno).

Os métodos adaptativos como `sumSalaries` oferecem suporte à implementação modular de um *concern* transversal .

2.3.4 Filtros de Composição

A abordagem de Filtros de Composição (CFs) é uma extensão ortogonal e modular ao modelo de objetos a fim de lidar com os problemas da modelagem orientada a objetos e a fim de aumentar a adaptabilidade e a reusabilidade em sistemas orientados a objetos [3, 15].

O modelo de Filtros de Composição tem suas raízes no trabalho de pesquisa de Mehmet Aksit e do grupo TRESE na University of Twente, no contexto de arquitetura de software e tecnologia de objetos composicionais. O modelo FC foi primeiramente adotado na linguagem orientada a objetos Sina [1]. Uma visão geral técnica e histórica do modelo de FC pode ser encontrada em [14].

Descrição

Filtros

Na abordagem de FC, cada *concern* é expresso como um *filtro*. Um filtro é definido como uma função que manipula as mensagens recebidas e enviadas pelos objetos. Cada filtro especifica as condições de estado para a aceitação ou recusa de mensagens e determina a ação resultante apropriada.

Uma mensagem é processada pelos filtros antes de o método correspondente ser executado. Quando uma mensagem é recebida, ela tem de passar por um conjunto de filtros de entrada e, antes de ser enviada, tem de passar por um conjunto de filtros de saída.

Um filtro estende uma abstração de classe de forma modular (Figura 2.8). Uma extensão modular significa que um filtro pode ser acoplado a uma classe sem modificar a definição da mesma. Cada extensão do filtro

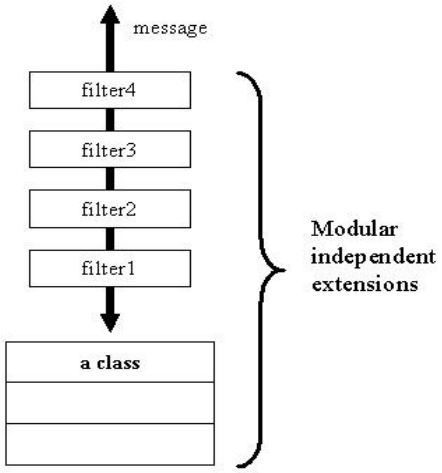

Figura 2.8: Filtros múltiplos [5].

a uma classe é independente das demais extensões de filtros. Isso permite que vários filtros sejam compostos de forma simples.

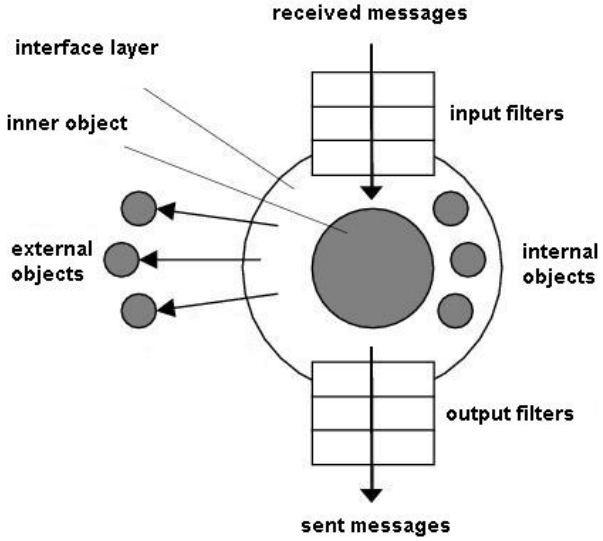

Figura 2.9: Um objeto no modelo de Filtros de Composição [14].

A Figura 2.9 mostra os principais elementos de um objeto no modelo FC. O objeto de filtro consiste em uma camada de interface (*interface layer*) e um objeto interno (*inner object*). O objeto interno é um objeto regular definido em uma linguagem de programação orientada a objetos. A camada da interface contém um número arbitrário de filtros de mensagens de saída e entrada. As mensagens recebidas passam pelos filtros de entrada e as mensagens enviadas passam pelos filtros de saída.

Objeto interno

Camada da interface

Os filtros podem modificar as mensagens alterando o seletor de mensagens ou o objeto alvo. Os filtros podem ser usados para redirecionar as mensagens para outros objetos, ou seja, os objetos internos, que são encap-

sulados na camada de interface ou alguns objetos externos (*external objects*) referenciados a partir da camada de interface e para traduzir as mensagens (modificando os seletores das mensagens). Eles também podem descartar ou armazenar em buffer mensagens ou levantar uma exceção.

Os Filtros de Composição alcançam a separação de *concerns* definindo um filtro para cada tipo de *concern*. Há um conjunto predefinido de tipos de filtros, como *Dispatch* [2], *Meta* [3], *Wait* e *Error* [2], para *concerns* como visões múltiplas, restrições de sincronização, transações atômicas etc.

Se necessário, novos tipos de filtros podem ser introduzidos; em [4], é definido um novo tipo de filtro *RealTime* a fim de expressar as restrições de sincronização nas execuções de mensagens.

2.3.5 Programação Orientada a Aspectos

Em sua breve perspectiva história sobre POA [90], Cristina Lopes ressalta:

> Não consigo me lembrar da data exata em que decidimos chamar nosso trabalho de "Programação orientada a aspectos", mas lembro que o termo foi sugerido por Chris Maeda [..]. Em janeiro de 1996, meu notebook indicou que estávamos usando os termos "implementação aberta" (*Open Implementation*) e POA ao mesmo tempo, ainda que em diferentes partes do trabalho do grupo. Por volta de junho de 1996, enviamos uma proposta para DARPA chamada "Programação orientada a aspectos". No final de 1996, as referências a "implementação aberta" em meu notebook já tinham desaparecido.

A *Programação orientada a aspectos*, conforme proposta (e cunhada) por Gregor Kiczales e seu grupo de pesquisa do Xerox PARC, é uma nova metodologia de programação que promove a separação avançada de *concerns* ao introduzir uma nova unidade modular, chamada *aspecto*, para a modularização dos *concerns* transversais .

Em 1995, o grupo de Kiczales no Xerox PARC trabalhava com *concerns* de desempenho em tempo de execução (por exemplo, otimização da memória, *loop fusion* [98] e tempo de avaliação), implementações abertas [72] e protocolos de metaobjetos [70]. Nesse ano, Cristina Lopes juntou-se ao grupo PARC e trouxe conhecimentos em programação adaptativa obtidos a partir do seu trabalho com Karl Lieberherr (seu orientador de doutorado) na Northeastern University.

Em 1997, Kiczales apresentou um trabalho chamado "*Aspect-Oriented Programming*" [73] na conferência ECOOP'97 que forneceu o primeiro framework conceitual para a POA: um conjunto inicial de termos e conceitos a fim de oferecer suporte a projetos de sistemas baseados em POA. Além disso, apresentou uma análise dos problemas que a POA pretendia resolver, assim como vários exemplos expressos em linguagens orientadas a aspectos específicas do domínio. As idéias paradigmáticas introduzidas por esse trabalho seminal são apresentadas a seguir.

Análise do problema

O foco em *crosscutting* e *concerns* transversais foi uma característica diferenciadora da POA do grupo Xerox PARC, que, mais tarde, foi incorporada às tecnologias de separação de *concerns* prévias. Em [73], Kiczales afirma que:

> Apresentamos uma análise de por que algumas decisões de projeto sempre foram muito difíceis de ser capturadas no código. Chamamos de *aspectos* as questões tratadas por essas decisões e mostramos que a razão pela qual sempre foram difíceis de ser capturadas é que elas atravessam (*crosscut*) a funcionalidade básica do sistema. Apresentamos a base para uma nova técnica de programação, chamada de programação orientada a aspectos (POA), que possibilita expressar claramente programas que envolvem esses aspectos, incluindo o isolamento, a composição e a reutilização apropriados do código do aspecto.

Sua análise preliminar organiza-se em torno de um exemplo expresso em Lisp (ou seja, não orientado a objetos), com base em um projeto desenvolvido no Xerox PARC chamado `RG` [98]. O *concern* que se tentava alcançar nesse projeto era a otimização do uso da memória ao compor as funções que contêm laços aplicados sobre matrizes. Cada função era um filtro que recebia como entrada várias imagens e produzia uma única imagem de saída.

Crosscutting

Kiczales explica o fenômeno de *crosscutting* em termos do entrelaçamento (*tangling*) observado entre a implementação dos *concerns* básicos e funcionais (um conjunto de funções que recebiam muitas imagens e produziam uma única imagem de saída) e a implementação do *concern* de otimização – nesse caso, a fusão de laços (*loops*) para melhorar o uso da memória.

O termo *procedimento generalizado* (*generalized procedure*)é fornecido para denotar "qualquer abstração-chave usada para encapsular uma unidade funcional do sistema geral (objeto, método, procedimento, API etc.)". *Componentes* são definidos como "propriedades de um sistema, no qual a implementação pode ser encapsulada de forma limpa em um procedimento generalizado", como os filtros de imagens do exemplo. *Aspectos* são definidos como "propriedades de um sistema, no qual a implementação não pode ser encapsulada em um procedimento generalizado", como a otimização do uso da memória por meio da fusão de laços.

O objetivo da POA é então: "oferecer suporte ao programador para separar de forma limpa os componentes e aspectos uns dos outros, fornecendo mecanismos que possibilitam abstraí-los e compô-los a fim de produzir o sistema geral".

Outros candidatos a aspectos são listados: *concerns* de tratamento de falhas, sincronização e desempenho, porque "costumam explorar as informações sobre o contexto de execução que se estende pelos componentes". Depois da apresentação da análise do problema, a solução de POA é apresentada.

A Solução da POA

A solução de Kiczales é fornecer outra linguagem para a definição de aspectos junto com o suporte a linguagens já existentes para a definição de componentes. Os componentes da POA denotam *unidades funcionais* expressas em alguma *linguagem de componentes*. Os aspectos de POA denotam *unidades não-funcionais* expressas em uma ou mais *linguagem de aspectos*. A idéia de uma linguagem de aspectos de propósito geral não havia sido vislumbrada.

Combinador de aspectos

A POA introduz o termo *combinador de aspectos* (*aspect weaver*) para denotar um processador de linguagem especial que oferece suporte à composição entre os componentes e os aspectos, "ao gerar a representação de um ponto de combinação do programa do componente e, em seguida, executar (ou compilar) os programas de aspectos em relação a ele" (Figura 2.10). Essa definição leva ao conceito de *ponto de combinação* (*join point*), outro termo introduzido pela POA.

Pontos de combinação

Os pontos de combinação são definidos como "elementos da semântica da linguagem de componentes com os quais os programas de aspectos coordenam". No exemplo do processamento de imagens, os pontos de combinação são os fluxos de dados do programa de componentes. Conforme

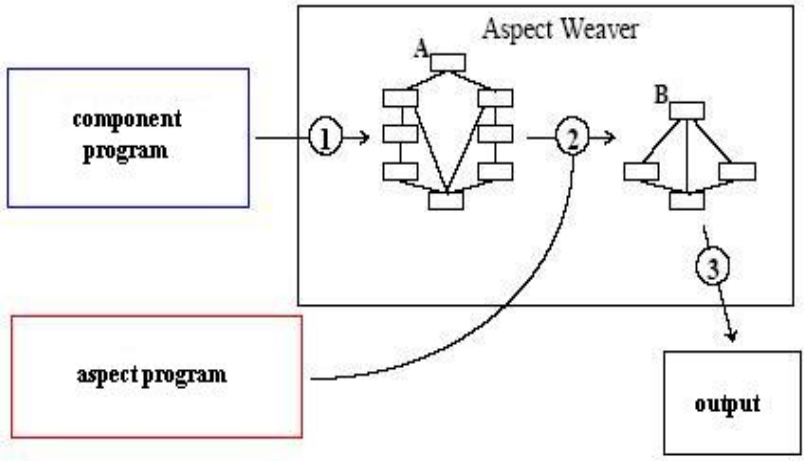

Figura 2.10: O combinador de aspectos [73].

ilustrado por outro exemplo apresentado no trabalho – uma versão simplificada de uma biblioteca digital distribuída [89] – pontos de combinação podem ser invocações de método em tempo de execução em um programa orientado a objetos. Kiczales refere-se a esses exemplos a fim de enfatizar que pontos de combinação "não são necessariamente construções explícitas da linguagem de componentes", mas sim que "são elementos claros, mas talvez implícitos, da semântica do programa de componentes". Essa definição de pontos de combinação é mais tarde incorporada no modelo de pontos de combinação de AspectJ (consulte a Seção 2.3.6).

Finalmente, Kiczales argumenta que "apesar de pedir ao programador para tratar explicitamente de aspectos de implementação poder soar como um retrocesso", o uso adequado de POA significa que os programadores "estão expressando a estratégia de implementação em um nível abstrato apropriado, por meio de uma linguagem de aspectos adequada, com localidade correta".

Os principais elementos introduzidos pelo trabalho seminal sobre POA – componentes, aspectos, *crosscutting*, pontos de combinação e processo de combinação – foram incorporados ao projeto de AspectJ, a primeira linguagem orientada a aspectos de propósito geral.

2.3.6 AspectJ

AspectJ [11, 74, 75] é uma extensão à linguagem de programação Java que oferece suporte à programação orientada a aspectos de propósito geral

baseada nos principais conceitos da POA apresentados em [73].

A linguagem AspectJ foi desenvolvida pelo grupo de POA do Xerox PARC como uma base para uma avaliação empírica da programação orientada a aspectos em Java. A equipe de AspectJ relata que a *compatibilidade* era a principal questão no projeto da linguagem [74]: *compatibilidade superior* (todos os programas legais em Java devem ser programas legais em AspectJ), *compatibilidade de plataformas* (todos os programas legais em AspectJ devem ser executados também nas máquinas virtuais Java), *compatibilidade de ferramentas* (as ferramentas existentes, inclusive as ferramentas de projeto, documentação e ambientes de desenvolvimento integrados, podem ser usadas) e *compatibilidade de programador* (os programadores de Java devem se sentir confortáveis programando em AspectJ).

Esses requisitos de compatibilidade foram motivados por outro objetivo principal: o desenvolvimento e o suporte adequado a uma comunidade substancial de usuários. De fato, muitas características incorporadas nas versões evolutivas de AspectJ surgiram a partir de feedbacks recebidos de usuários e de discusão surgidas em listas de discussão de usuários de Aspect [12].

Como uma linguagem de propósito geral, AspectJ permite a modularização adequada de uma grande variedade de *concerns* transversais como verificação e tratamento de erros, sincronização, distribuição, comportamento sensível ao contexto, otimizações de desempenho, monitoramento e auditoria, depuração, protocolos multiobjetos etc.

Em 2002, o projeto de AspectJ mudou-se do Xerox PARC para um projeto desenvolvido abertamente em `http://eclipse.org/aspectj` [11].

Visão Geral

A linguagem AspectJ oferece um conjunto de novos elementos de programação para concretizar os principais conceitos da POA: *pointcuts*, *advice*, *inter-type declarations* e aspectos.

Crosscutting em AspectJ pode ser estrutural ou comportamental. *Crosscutting* estrutural pode afetar as definições de classe e a estrutura estática das hierarquias de classes. *Crosscutting* dinâmico pode afetar o comportamento das classes; é restrito a um conjunto predefinido de pontos especificados por um *modelo de pontos de combinação* (*join point model*). Esse modelo de pontos de combinação concentra-se na execução dinâmica de programas em Java.

O compilador de AspectJ gera código Java simples que realiza *cross-cutting* de acordo com as regras definidas dentro dos aspectos. Esse processo de geração chama-se processo de combinação (*weaving*).

Esta Seção apresenta uma breve ilustração dos conceitos de AspectJ por meio de uma aplicação simples adaptada de [75] que se assemelha ao exemplo de manipulação de figuras apresentado na Seção 2.2. Uma visão geral mais detalhada de características de AspectJ pode ser encontrada em "The AspectJ Programming Guide" [10].

A Figura 2.11 mostra um diagrama de classes que descreve figuras, pontos e linhas. Uma figura (`Figure`) consiste em alguns elementos (`FigureElement`), que podem ser pontos (`Point`) ou linhas (`Line`). A classe `Figure` também é uma fábrica de elementos da figura. Há uma única tela (`Display`) na qual os elementos da figura são desenhados.

Agora, considere um novo requisito que peça a atualização de um objeto de exibição, instância de `Display`, sempre que uma figura for movida. Um retângulo tracejado é desenhado em volta dos métodos que implementam o comportamento que move os elementos da figura. O retângulo não se encaixa em nenhuma classe no diagrama, nem em volta de nenhuma delas – em vez disso, atravessa várias classes.

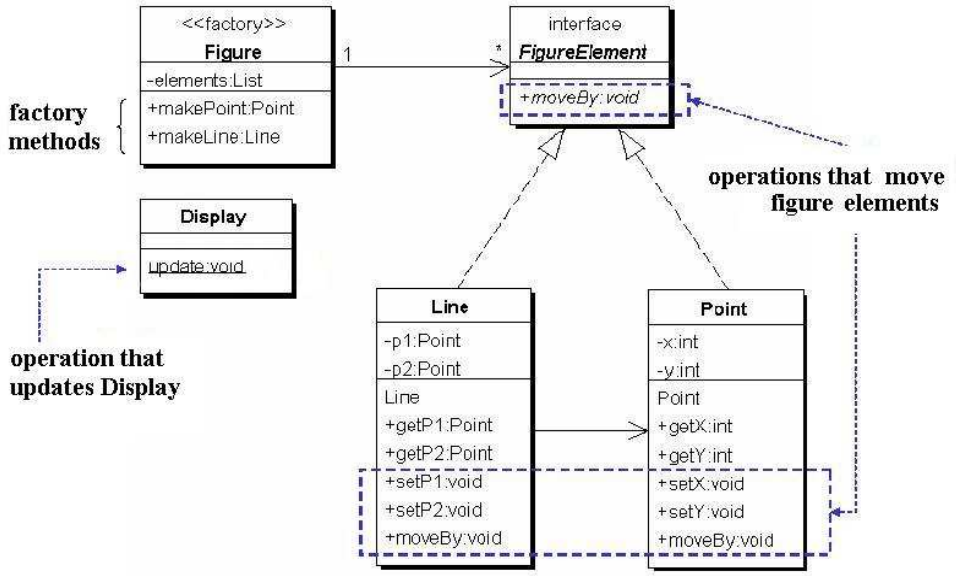

Figura 2.11: Diagrama de classes do exemplo das figuras.

Pontos de Combinação

AspectJ adota um modelo no qual um ponto de combinação é um ponto bem-definido na execução dinâmica de um programa em Java [74].

Os pontos de combinação de AspectJ incluem a chamada ou a execução de métodos, construtores ou tratadores de exceção, acesso ou modificação do valor de um atributo etc.

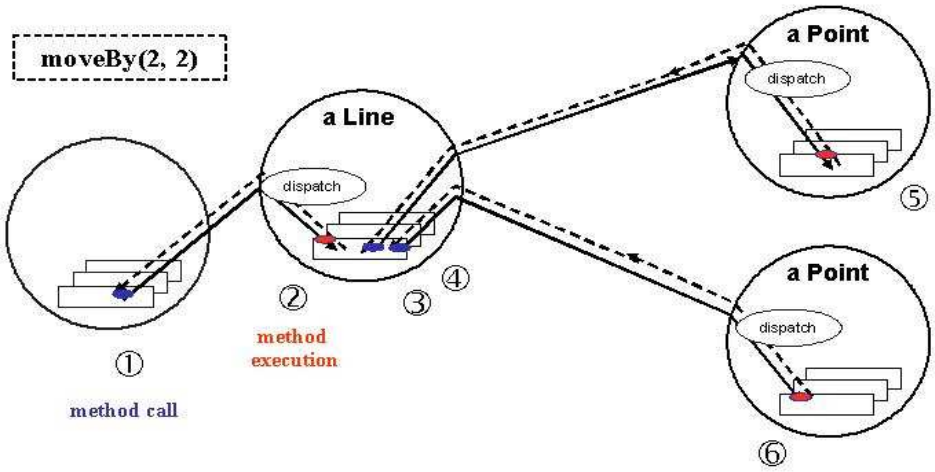

Figura 2.12: Grafo de chamadas para Line.moveBy(2,2) [74].

No modelo de pontos de combinação de AspectJ, os pontos de combinação devem ser considerados nós em um gráfico de chamadas de objetos em tempo de execução (Figura 2.12). O controle passa por cada ponto de combinação duas vezes [74], por exemplo, quando uma mensagem é enviada e quando essa mensagem é respondida. As chamadas de métodos e as execuções de métodos estão representadas por elipses.

Pointcuts

Um *pointcut* é um elemento de programa que seleciona pontos de combinação e expõe dados do contexto de execução desses pontos de combinação. Eles podem ser compostos por operadores booleanos `||`, `&&` e `!`, a fim de criar outros pointcuts.

Os *designadores de pointcuts* identificam pontos de combinação particulares ao filtrar um subconjunto de todos os pontos de combinação no fluxo do programa. Por exemplo, o designador de pointcuts apresentado a seguir identifica qualquer chamada aos métodos `setX` ou `setY` definidos por `Point`:

```
call(void Point.setX(int)) || call(void Point.setY(int))
```

Os programadores podem definir designadores de pointcuts e atribuir nomes. O código a seguir define um pointcut chamado `move` que designa qualquer chamada de método que move elementos da figura:

```
pointcut move():
   call(void FigureElement.moveBy(int, int)) ||
   call(void Point.setX(int)                 ||
   call(void Point.setY(int)                 ||
   call(void Line.setP1(Point)               ||
   call(void Line.setP2(Point);
```

AspectJ fornece um conjunto de pointcuts primitivos predefinidos. Alguns deles selecionam todos os pontos de combinação de um determinado *tipo* (`call`, `execution`, `initialization`, `get`, `set`, `handler`), enquanto outros selecionam todos os pontos de combinação que aparecem no *contexto* de uma determinada classe ou definição de método, instância de classe ou fluxo de controle (`within`, `withincode`, `this`, `target`, `args`, `cflow`, `cflowbelow`, `if`).

Cada designador primitivo de pointcuts em AspectJ pode receber restrições adicionais que, em geral, são definidas em termos de padrões textuais para assinaturas de métodos e nomes de tipos. *Wildcards* podem ser usados para abstrair a partir de método individual, tipo, subtipo, classe ou identificadores de conjunto (`*`, `+`) ou um número arbitrário de argumentos (`..`).

Além de definir conjuntos de pontos de combinação, os designadores de pointcuts podem *expor determinados valores no contexto de execução em pontos de combinação*. Os valores expostos são chamados de *parâmetros de pointcuts* e podem ser usados em decalrações de *advice*. Os parâmetros de pointcuts são sempre parâmetros de saída. Por exemplo, em `calls(void Point.setX(int))`, tanto o objeto que recebe a chamada, de tipo `Point`, como o novo valor, de tipo `int`, podem ser expostos. A exposição do contexto de execução só é possível usando os designadores primitivos de pointcuts `this`, `target`, e `args`.

No designador de pointcuts `p1` a seguir, `fe` é um parâmetro de pointcut que expõe o objeto sendo chamado:

```
pointcut p1(FigureElement fe): call(void Point.setX(int)) &&
target(fe)
```

No designador de pointcuts `p2` a seguir, `x` é um parâmetro de pointcut que expõe o valor que é o parâmetro real da chamada `setX`:

```
pointcut p2(int x): call(void Point.setX(int)) && args(x)
```

Advice

Um *advice* compreende o comportamento de aspectos que deve ser executado em cada ponto de combinação especificado por um pointcut. São definidos por declarações *before*, *after* e *around* declarations. Essas declarações associam o código do aspecto com um pointcut e uma relação de ordem, indicando como o código do aspecto será combinado com o código de base.

Before advice executa no momento em que um ponto de combinação é alcançado; *after advice* executa no momento em que o controle retorna através do ponto de combinação; *around advice* executa quando o ponto de combinação é alcançado e tem controle explícito sobre se o próprio método afetado deve ser executado.

O advice a seguir usa o pointcut `move()` a fim de especificar que o método `Display.update()` será chamado depois que algum elemento da figura for movido:

```
after(): move() { Display.update(); }
```

Aspectos

Os *aspectos* de AspectJ são unidades modulares de implementação para código *crosscutting*. Um aspecto é definido como uma classe e pode ter métodos, campos, construtores, inicializadores, pointcuts nomeados e advice [75]. O aspecto a seguir modulariza o requisito para a atualização da exibição:

```
aspect DisplayUpdating {
   pointcut move():
      call(void FigureElement.moveBy(int, int)) ||
      call(void Point.setX(int)                 ||
      call(void Point.setY(int)                 ||
      call(void Line.setP1(Point)               ||
      call(void Line.setP2(Point);

   after(): move() { Display.update(); }
}
```

O modelo de implementação de AspectJ define aspectos como tipos e instâncias de aspectos como objetos regulares que não podem ser explicitamente instanciados.

AspectJ fornece regras para estabelecer a precedência relativa entre aspectos, para situações em que mais de um advice se aplica a um ponto de combinação (por exemplo, o relacionamento *precedence*).

Inter-type declarations

Inter-type declarations são declarações que atravessam classes e suas hierarquias. Os aspectos podem declarar membros que atravessam várias classes ou alteram o relacionamento de herança entre classes.

O aspecto a seguir declara que Point implementa a interface `Cloneable` com uma forma *declare parents*; também declara um método especializado de `Point`, `clone()`. Os objetos de `Point` tornam-se `Cloneable` e seu comportamento é estendido pelo comportamento `clone()`.

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

    public Object Point.clone()
             throws CloneNotSupportedException {
      /* .. code .. */
    }
}
```

Da pesquisa para a prática

Depois de meia década de avaliação dos usuários, AspectJ é um padrão para POA. Os principais usos para aspectos relatados pela comunidade de usuários que adotou AspectJ são [90]:

- passagem de parâmetros remotos, questões de configuração, restrições em tempo real, tratamento de falhas, segurança, depuração, etc.;
- desempenho em tempo de execução: otimização de memória, fusão de laços e tempo de avaliação;
- depuração e instrumentação: rastreamento, auditoria, teste, *profiling*, monitoramento e *asserting*;
- construção de programa: mixins, herança múltipla e visões (*views*);
- garantia e verificação: garante que os tipos de um framework são usados de forma adequada, validação de contrato entre componentes, garantia de melhores práticas de programação;

– configuração: gerenciamento das especificidades do uso de diferentes plataformas e a escolha de espaços de nome apropriados para o gerenciamento de propriedades;
– aspectos operacionais: sincronização, caching, persistência, gerenciamento de transações, segurança e balanceamento de carga;
– tratamento de falhas: redirecionamento de uma chamada com falha para um outro serviço.

A capacidade de conectar e desconectar diferentes aspectos de acordo com uma diretiva de compilação foi relatada como a maior vantagem de AspectJ.

2.3.7 Separação Multidimensional de Concerns

A *Separação Multidimensional de Concerns* (SMDdC) é uma abordagem para a separação de *concerns* que evoluiu dos primeiros trabalhos sobre programação orientada a sujeitos de William Harrison e Harold Ossher no Centro de Pesquisa T.J. Watson da IBM [59]. Com Peri Tarr, evoluíram a POS para a SMDdC a fim de permitir várias e simultâneas decomposições do mesmo software e a extração de *concerns* de softwares existentes [113].

O termo "multidimensional" denota "vários critérios para a decomposição"; assim, a SMDdC pôde ser redefinida como "separação de *concerns* por meio de vários critérios de decomposição".

Análise do Problema

Um dos principais objetivos da SMDdC é superar a *tirania* da decomposição dominante. De acordo com [132], muitos dos formalismos de software atuais oferecem suporte, até certo ponto, à separação de *concerns*, mas como uma tendência na direção de uma única dimensão:

> Qualquer critério para decomposição e integração será apropriado para alguns contextos e requisitos, mas não para todos. Por exemplo, a decomposição de dados em sistemas orientados a objetos facilita muito a evolução dos detalhes da estrutura de dados, porque estão encapsulados dentro de classes únicas (ou de algumas estreitamente relacionadas), mas impede a adição ou evolução de características (*features*) [..] Dessa forma, a modularização de acordo com diferentes dimensões de *concerns* é

> necessária para diferentes fins: às vezes por classe, algumas por características e outras por ponto de vista, aspectos, papel ou outros critérios.

A SMDdC permite que os desenvolvedores decomponham o software de forma que possam encapsular todos os tipos relevantes (dimensões) de *concerns* ao mesmo tempo, sem que uma domine as outras. A SMDdC também inclui uma poderosa capacidade de composição, que permite aos desenvolvedores integrar as partes separadas.

Um outro objetivo importante da SMDdC é a capacidade de oferecer novas possibilidades de decomposição em softwares existentes (ou seja, decomposição em *concerns* com uma nova dimensão), sem refatoração explícita, reengenharia ou outra mudança invasiva. Essa capacidade chama-se *remodularização por demanda* [112].

Hyperspaces [132, 111, 113] é a abordagem da IBM para realizar a SMDdC.

A Abordagem Hyperspaces

Hyperspaces é a abordagem da IBM para a separação multidimensional de *concerns*. As principais características propostas são:

- Identificação explícita de dimensões e *concerns* em qualquer etapa do desenvolvimento de software;
- Encapsulamento de *concerns* ;
- Identificação e gerenciamento de relacionamentos entre *concerns* ; e
- Composição de *concerns*.

Identificação de Concerns

Hyperspace

Um *hyperspace* é definido como um conjunto de unidades de software que compõem um sistema de software, organizado em uma *matriz de concerns* multidimensional em que:

- Cada eixo representa uma *dimensão de concern*. Cada dimensão de *concern* pode ser vista como uma partição do conjunto de unidades: uma decomposição de software particular

  Veja a dimensão `Class` e a dimensão `Feature` na Figura 2.13.

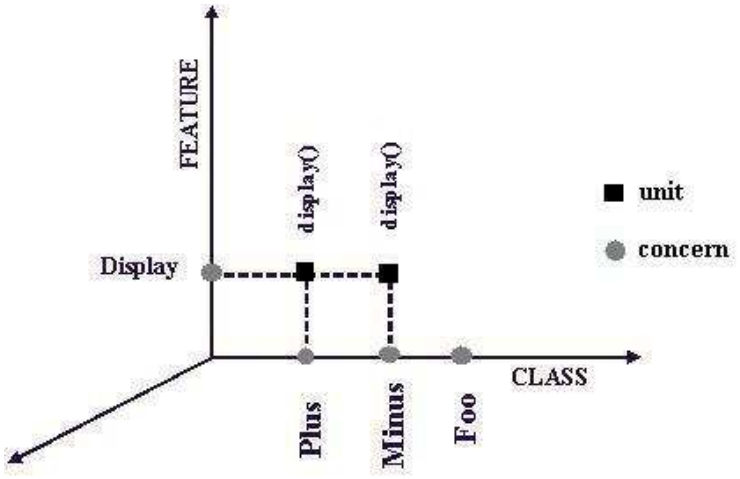

Figura 2.13: Um Hyperspace.

– Cada ponto em um eixo representa um *concern nessa dimensão*. Consulte `Display` (uma característica) na dimensão `Feature` e `Plus`, `Minus` e `Foo` na dimensão `Class` (Figura 2.13).

– As coordenadas de uma unidade de software indicam todos os *concerns* que ela afeta.

  Há duas unidades chamadas `display()` no hyperspace apresentado na Figura 2.13; cada unidade afeta um *concern* na dimensão `Class` (`Plus` ou `Minus`) e um *concern* na dimensão `Feature` (`Display`).

Na terminologia de Hyperspaces, os *concerns* `Plus` e `Display` se sobrepõem (*overlap*) porque cada par possui unidades em comum (o mesmo vale para `Minus` e `Display`). Os *concerns* na mesma dimensão devem ser separados, ou seja, eles não têm unidades em comum e não podem se sobrepor. Por exemplo, `Minus` e `Plus` são *concerns* na dimensão `Class`; são separados e não se sobrepõem. Entretanto, *concerns* na mesma dimensão podem estar relacionados (em hierarquias de classes, por exemplo).

O hyperspace (ou matriz de *concerns*) facilita a identificação de *concerns* e organiza as unidades de acordo com as dimensões e *concerns*; por exemplo, `Plus` pode ser identificado como `Class.Plus` (dimensão `Class`, *concern* `Plus`). Contudo, para encapsular *concerns*, introduz-se o conceito de "hyperslice".

Encapsulamento de Concerns

Qualquer *concern* único dentro de algumas dimensões define um *hyperplane*. Um hyperplane contém um conjunto de unidades que representam

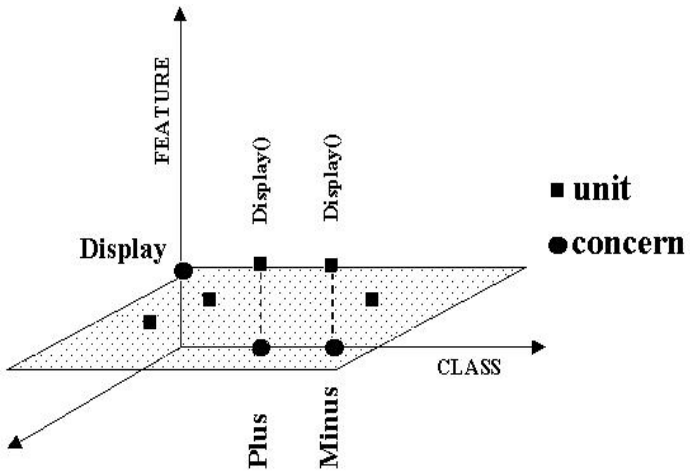

Figura 2.14: Hyperslice.

um *concern*. Um *hyperslice* é um hyperplane que é declarativamente completo; é tratado como um módulo, encapsulando um *concern*. A *completude declarativa* significa que um hyperslice deve declarar tudo a que se refere. Esse requisito elimina o acoplamento entre hyperslices. *Hyperslice*

A Figura 2.14 apresenta uma visão gráfica de um hyperslice. Esse hyperslice encapsula cinco unidades que incorporam a característica `Display`; ele modulariza a característica `Display` apresentada na dimensão `Feature` (eixo vertical). Os hyperslices generalizam o conceito de *sujeito* da programação orientada a sujeitos.

Os hyperspaces e hyperslices são usados para a identificação e modularização de *concerns*. Além disso, *hypermodules* são definidos para permitir a *integração de concerns*.

Integração de Concerns

Um *hypermodule* compreende um conjunto de hyperslices sendo integrados e um conjunto de relacionamentos de integração. Os relacionamentos de integração (*integration relationships*) especificam como os hyperslices se relacionam entre si e como devem ser integrados. A *correspondência* é um relacionamento de integração importante, indicando quais unidades específicas dentro de diferentes hyperslices devem ser integradas a outras (análogo a *regras de correspondência*, descritas em programação orientada a sujeitos). *Hypermodule* *relacionamentos de integração*

As *unidades correspondentes* (*corresponding units*) são unidades (pacotes, classes, métodos, atributos etc.) que pertencem a diferentes hyperslices e que se correspondem entre si de acordo com alguns critérios. As unidades correspondentes são compostas (semântica de *merge* ou *override*) *Corresponding units*

em uma nova unidade de software que contém algumas ou todas as funcionalidades das unidades originais.

A especificação dos relacionamentos de integração em Hyperspaces segue uma abordagem em que os desenvolvedores primeiro especificam uma estratégia geral de composição para identificar unidades correspondentes em hyperslices e, em seguida, definir as exceções, ou especializações, dessa estratégia para os casos em que a estratégia não se aplica. Hyperspaces oferecem suporte a três estratégias gerais: *mergeByName*, *nonCorrespondingMerge*, e *overrideByName*. A estratégia *mergeByName*, por exemplo, declara que unidades em hyperslices diferentes com o mesmo nome devem ser correspondentes e devem ser mescladas em uma nova unidade.

Estratégia de composição

O relacionamento *order* indica que a ordem das unidades relacionadas é importante e descreve as restrições de ordenação. O relacionamento *equate* indica que um conjunto de unidades devem ser correspondentes entre si, mesmo se os nomes forem diferentes.

O relacionamento *bracket* indica que um conjunto de métodos deve ser agrupado – sua chamada deve ser precedida ou seguida de outros métodos. Esse relacionamento é usado para compor o comportamento *crosscutting* antes ou depois de alguns métodos.

Hyper/J [131] é uma ferramenta que oferece suporte a Hyperspaces em Java.

2.3.8 Hyper/J

Hyper/J [131] é uma ferramenta desenvolvida no Centro de Pesquisa T.J. Watson da IBM para oferecer suporte a Hyperspaces [64]. Hyper/J oferece suporte à identificação, encapsulamento e integração de *concerns* a partir de programas em Java.

Um dos principais objetivos de Hyper/J era não modificar ou estender a linguagem Java. Em vez disso, Hyper/J oferece suporte à SMDdC para softwares em Java desenvolvidos usando qualquer metodologia e ferramentas [113].

Descrição

Hyper/J permite que um desenvolvedor componha uma coleção de *hyperslices* separados. Cada *hyperslice* encapsula um *concern* através da

definição e implementação de uma hierarquia de classes (parcial) apropriada para esse *concern*.

Antes de usar a ferramenta Hyper/J em um conjunto de classes Java, o desenvolvedor deve fornecer três entradas:

1. Um arquivo *hyperspace*

   O arquivo *hyperspace* é semelhante a uma descrição de projeto. Ele lista todos os arquivos Java com os quais um desenvolvedor está trabalhando e aos quais ele deseja aplicar Hyper/J.

2. Um ou mais arquivo de mapeamento de *concerns* (*concern mapping*)

   O(s) arquivo(s) de mapeamento de *concerns* descreve(m) como as diferentes partes dos arquivos Java tratam os diversos *concerns* em um *hyperspace*.

3. Um arquivo *hypermodule*

   O arquivo *hypermodule* descreve quais *hyperslices* devem ser integrados e como essa integração deve ser feita, incluindo a estratégia de composição geral, as cláusulas de ordem (`order`) etc. O relacionamento `bracket` oferece suporte à especificação de quais métodos devem ser executados antes e/ou depois de um método que será afetado por outro comportamento.

Assim que os arquivos descritos anteriormente tiverem sido especificados, a ferramenta de composição de Hyper/J pode ser aplicada para produzir um novo conjunto de unidades integradas – um novo *hyperslice* composto.

Exemplo

Considere o exemplo apresentado na Seção 2.3.6. Para trabalhar com Hyper/J, os dois *concerns* – o *concern* de dados, para os elementos da figura, e o *concern* de atualização de exibição – devem ser organizados em dois pacotes diferentes: `Figure` e `DisplayUpdate`. O pacote Figure contém as classes Java apresentadas no diagrama de classes da Figura 2.11.

O pacote DisplayUpdate e suas classes são apresentados na Figura 2.15. A hierarquia de classes é a mesma apresentada no pacote Figure, mas apenas métodos que movem figuras proporcionam implementação. A implementação consiste em uma chamada a `Display.update()`. O pacote DisplayUpdate é escrito deliberadamente para escapsular um *concern*. Como

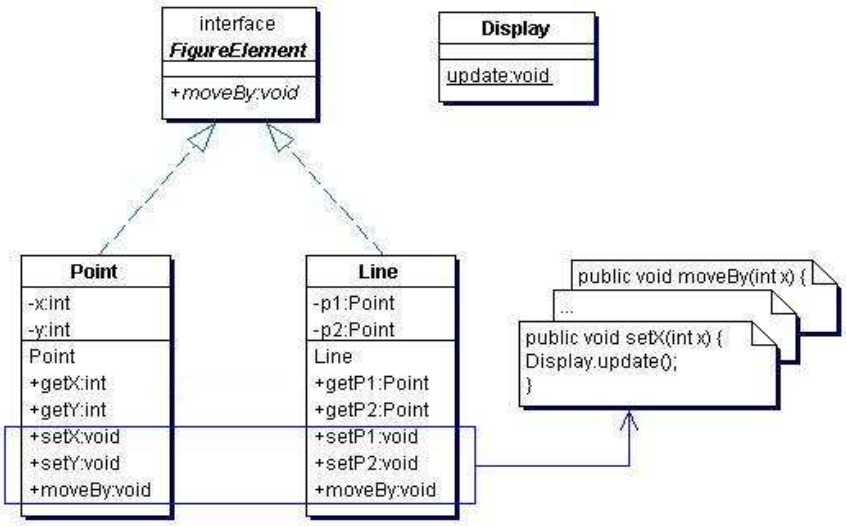

Figura 2.15: O pacote DisplayUpdate.

todos os métodos nesse pacote oferecem a mesma implementação, a solução não é boa. Entretanto, ela é usada para ilustrar os conceitos e as estratégias de composição de Hyper/J.

1. Arquivo *Hyperspace*

   Os dois pacotes são considerados como estando no espaço de trabalho atual e são definidos em um arquivo *hyperspace*:

   ```
   hyperspace TrackFigureMoves
      composable class DisplayUpdate.*;
      composable class Figure.*;
   ```

   Esse arquivo especifica que todas as classes dentro dos pacotes Figure e DisplayUpdate devem ser incluídas no hyperspace. HyperJ criará automaticamente uma dimensão (Class File) e um *concern* nessa dimensão para cada arquivo de classes que carrega (`Line`, `Point` etc.).

2. Mapas de *concerns*

   Os mapeamentos de *concerns* podem ser definidos como:

   ```
   package DisplayUpdate : Feature DisplayUpdate
   package Figure : Feature Figure
   ```

   Esse arquivo especifica como as unidades existentes no hyperspace tratam os *concerns* em uma nova dimensão chamada `Feature`. HyperJ processa o arquivo de mapeamento de *concerns*, adicionando a dimensão `Feature` à matriz de *concerns*, e os *concerns* `DisplayUpdate` e `Figure` à dimensão `Feature`.

3. Arquivo *Hypermodule*

   Ao usar a estratégia *mergeByName*, as unidades nos dois hyperslices com o mesmo nome devem ser correspondentes e devem ser mescladas em uma nova unidade. A ordem de composição segue a ordem na qual os hyperslices são listados no arquivo hypermodule.

```
hypermodule MobileFigures
   hyperslices:
     Feature.Figure,
     Feature.DisplayUpdate;

   relationships:
     mergeByName;

end hypermodule
```


Atualização de tela (display update) com `bracket`

A more efficient solution would be the use the `bracket` relationship to relate methods that move figures to the invocation of the Display.update() method after their own execution.

Uma solução mais eficiente seria usar o relacionamento *bracket* para relacionar métodos que movem figuras com a chamada do método `Display.update()` depois de sua própria execução.

```
bracket "setX" with
  (after Feature.Figure.Display.update,
  "Point");
bracket "setY" with
  (before Feature.Figure.Display.update,
  "Point");
```

Por exemplo, no relacionamento *bracket* descrito anteriormente, os métodos setX e setY devem ser agrupados pelo método `Feature.Figure.Display.update`. Dessa forma, quando o software composto chama um método chamado setX(), a chamada resultará primeiro na execução de setX() a partir de `Figure.Point` e, em seguida, update() a partir de `Figure.Display`. No entanto, o suporte de Hyper/J ao relacionamento *bracket* não era estável no momento em que implementamos essas soluções.

2.4 Desenvolvimento de Software Orientado a Aspectos

A programação orientada a aspectos oferece suporte à separação e à composição de *concerns* transversais na fase de implementação. Esse foco inicial centrado em programação foi natural e é análogo ao desenvolvimento inicial da POO.

Depois do lançamento de AspectJ e Hyper/J, diversos experimentos e estudos de casos que envolviam a POA e a SMDdC [98, 101, 69, 136, 121, 80, 102, 22, 23, 51, 124] foram realizados a fim de avaliar sua utilidade, e muitas abordagens novas de POA foram desenvolvidas (consulte [8] para obter uma compilação). Como as ferramentas disponíveis tornaram-se cada vez mais robustas e ofereciam suporte IDE razoável e compatibilidade com Java, a POA chegou às empresas de software; os programadores começaram a realizar experimentos com a POA em seus projetos. A categorização proposta pela equipe de AspectJ – aspectos de reusabilidade, produção e desenvolvimento [10] –promove uma transição suave e incremental a partir da POO pura até a POA.

A programação orientada a aspectos também trouxe algumas boas práticas e princípios relacionados. Um princípio básico é representar os aspectos como abstrações de primeira classe na linguagem. Isso permite que sejam compostos, estendidos e reutilizados. Duas importantes propriedades discutidas nos primeiros momentos da POA eram a *transparência* e a *quantificação*, conforme afirmado por Filman e Friedman em seu trabalho "Aspect-Oriented Programming is Quantification and Obliviousness" [42].

Recentemente, muitos grupos de pesquisa começaram a discutir a função dos aspectos em outras fases do processo de desenvolvimento de software. O Desenvolvimento de Software Orientado a Aspectos [139, 143] é uma área emergente cujo objetivo é promover a separação avançada de *concerns* ao longo de todo o ciclo de vida do desenvolvimento de software.

Os aspectos e a orientação a aspectos influenciam a análise e o projeto uma vez que introduzem novos tipos de conceitos e novos tipos de decomposição e modularização. Como conseqüência, os termos *projeto orientado a aspectos* (POaA) [140] e *análise orientada a aspectos* (AOA) foram usados para denotar a adoção de técnicas orientadas a aspectos nas fases de análise e projeto. Ademais, os aspectos podem surgir logo no início do desenvolvimento de software, durante a especificação dos requisitos. A *engenharia de requisitos orientada a aspectos* (EROA) [141, 144] preocupa-se com a identificação de requisitos de *crosscutting*, assim

como o mapeamento entre requisitos de *crosscutting* e outros artefatos de software. A *modelagem orientada a aspectos* (MOA) [142, 145] é uma parte crítica do DSOA que se concentra na notação e nas técnicas para a visualização e comunicação das soluções orientadas a aspectos durante o caminho que conduz dos requisitos à implementação da POA.

Sem a devida atenção aos *concerns* transversais durante as demais fases do desenvolvimento de software, pode ser difícil gerenciar a complexidade, a compreensibilidade, a evolução e a composição de sistemas de software.

2.5 Considerações Finais

A contribuição inicial da POA e das abordagens relacionadas é enfocar na importante limitação das tecnologias de programação atuais: a incapacidade de lidar com alguns *concerns* especiais que podem impactar os benefícios esperados oferecidos pela separação de *concerns*.

Até certo ponto, todas as abordagens apresentadas compartilham os mesmos objetivos comuns: alcançar um melhor alinhamento entre os requisitos e o código, melhorando a adaptabilidade, extensibilidade e reusabilidade, e alcançar uma separação "completa" de *concerns*. Entretanto, cada abordagem oferece uma solução específica para tratar desses objetivos. A área ainda está muito diversificada e aparentemente imatura. Não há consenso ainda em relação a quais linguagens persistirão e quais abordagens estão fadadas a serem consideradas no futuro apenas sob uma perspectiva histórica.

AspectJ e Hyper/J são duas importantes ferramentas, reconhecidas como as principais representantes da POA e da SMDdC, respectivamente. AspectJ foi aceita por uma comunidade mais ampla; seus conceitos e mecanismos sofreram a influência do projeto de várias tecnologias ditas "orientadas a aspectos".

Desafio: a necessidade de um framework conceitual unificador para a POA.

Alguns desafios devem ser alcançados antes que o DSOA possa se beneficiar das contribuições já oferecidas pela POA no nível da programação.

As Figuras 2.16, 2.17 e 2.18 apresentam modelos conceituais para três abordagens da SAdC que apresentamos enquanto tentávamos entender os

detalhes de cada abordagem. Os conceitos de cada modelo foram descritos na Seção 2.3. Conforme mencionado no Capítulo 1, na ausência de uma terminologia e uma semântica consistente para os mecanismos de composição e abstração que lidam com *concerns* transversais , sua essência e benefícios podem ser prejudicados pela diversidade das abordagens da POA que ainda estão em desenvolvimento.

Não somos contra tal diversidade, mas sim em favor da adoção de um framework conceitual unificador para a POA que caracterize o projeto de linguagens orientadas a aspectos e forneça um melhor suporte ao desenvolvimento de software orientado a aspectos.

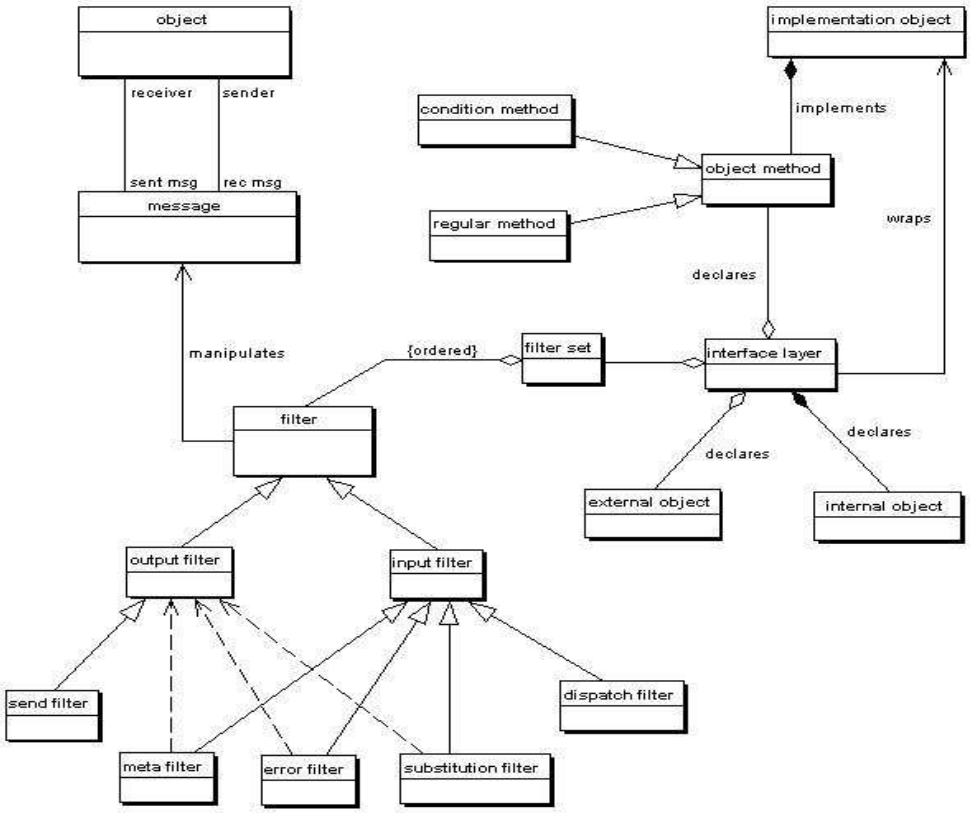

Figura 2.16: O modelo de Filtros de Composição

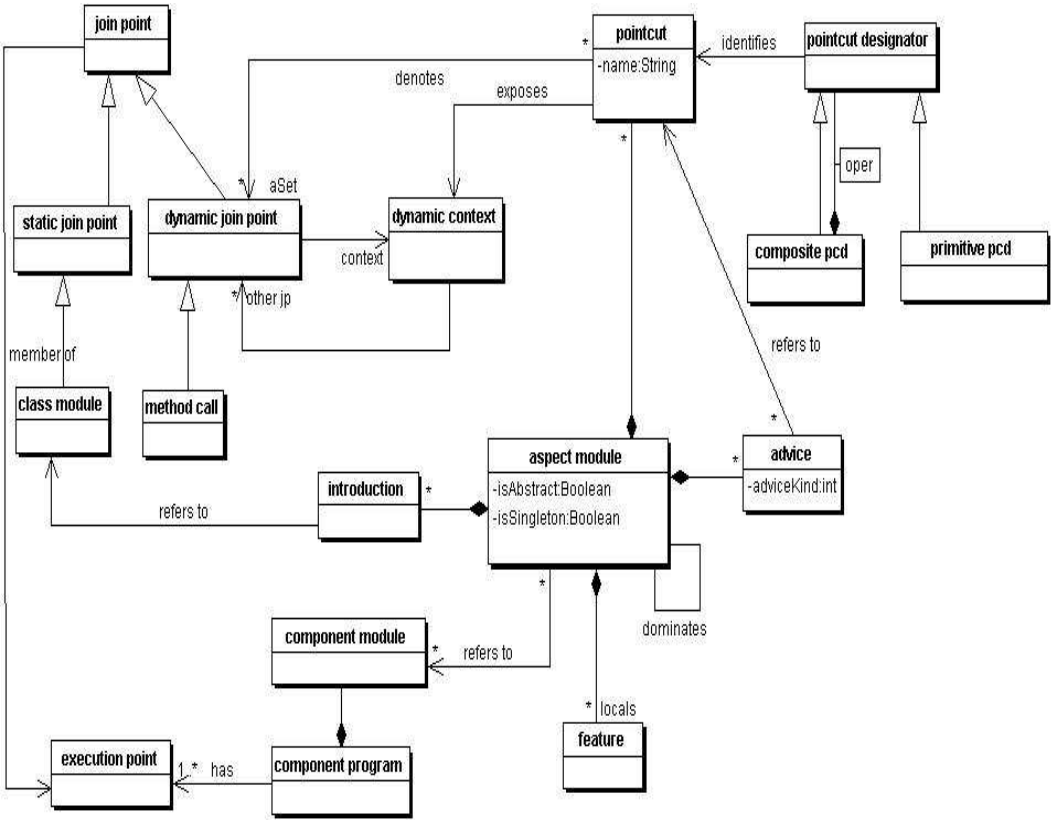

Figura 2.17: O modelo de AspectJ

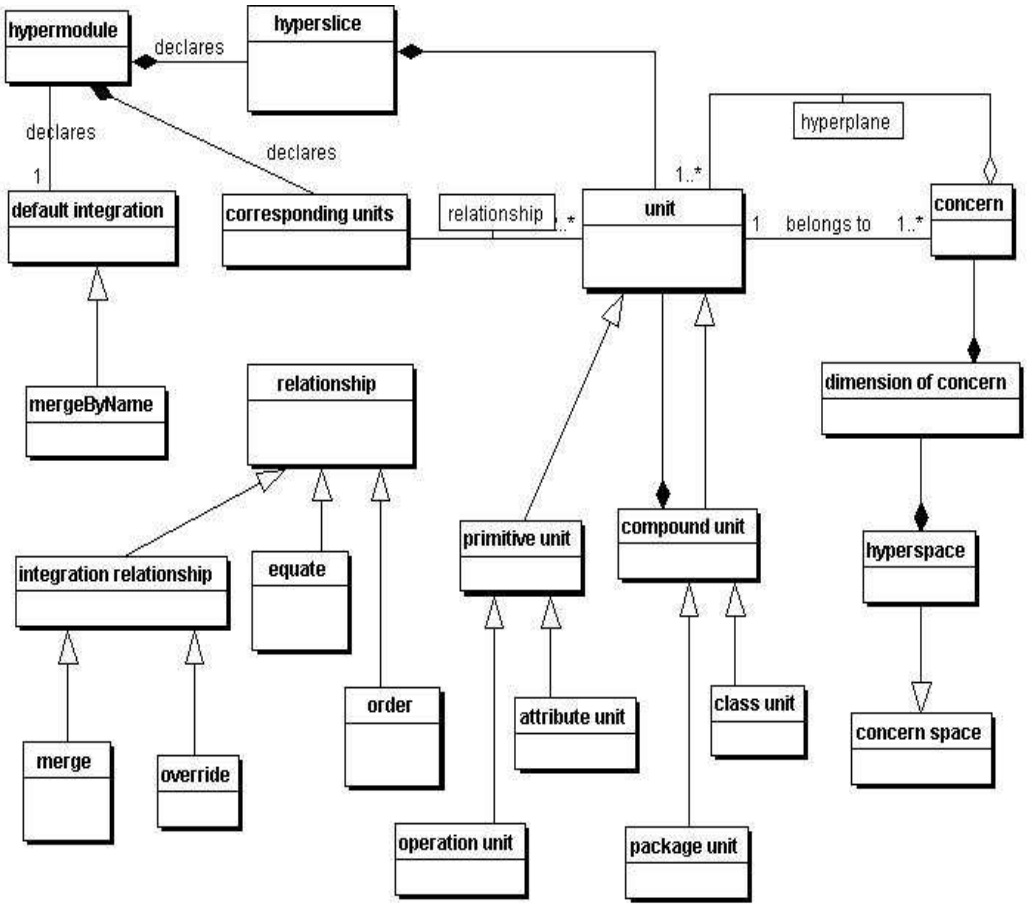

Figura 2.18: O modelo de Hyperspaces