Jornada de Atualização em Informática – JAI 2004

Desenvolvimento de Software Orientado por Aspectos

Fabio Tirelo[1] **, Roberto da Silva Bigonha**[2] **,**
Mariza Andrade da Silva Bigonha[2] **, Marco Túlio de Oliveira Valente**[1]

[1]Departamento de Ciência da Computação
Pontifícia Universidade Católica de Minas Gerais
Av. D. José Gaspar, 500 – Prédio 34 – 30.535-610 – Belo Horizonte, MG

[2]Departamento de Ciência da Computação
Universidade Federal de Minas Gerais
Av. Antônio Carlos, 6627 – 31.270-000 – Belo Horizonte, MG

{ftirelo,mtov}@pucminas.br, {bigonha,mariza}@dcc.ufmg.br

Resumo. *O desenvolvimento de software orientado por aspectos é uma técnica nova cujo objetivo é permitir a definição separada de requisitos transversais às classes de um sistema orientado por objetos. Por atravessarem todo o código, tais requisitos são, em geral, de difícil modularização em linguagens orientadas por objetos puras. Com a orientação por aspectos, requisitos transversais, tais como geração de registros de operações, controle sincronização e comunicação, podem ser implementados de maneira elegante, eficiente e modular, aumentando o nível de reutilização de código em sistemas.*

Abstract. *Aspect-oriented software development is a new technique whose objective is to modularize crosscutting concerns in object-oriented systems. Modularizing such concerns in pure object-oriented languages is difficult since their implementation can be scattered throughout many program classes. By using aspect-oriented programming, one can implement crosscutting concerns such as logging generation, synchronization and communication control in an elegant, efficient, and modular manner, which may improve system reusability.*

Conteúdo

1. Introdução

A Programação Orientada por Aspectos (AOP), proposta por Gregor Kiczales em 1997 [23] tem por objetivo modularizar decisões de projeto que não podem ser adequadamente definidas por meio da programação orientada por objetos (POO). Isto se deve ao fato de que alguns requisitos violam a modularização natural do restante da implementação [22].

Na Programação Orientada por Aspectos, requisitos de sistemas são modelados por meio de *classes*, que implementam os objetos do mundo real, e *aspectos*, que implementam requisitos transversais do sistema, tais como geração de registro de operações, tratamento de erros, segurança e comunicação.

Na definição de classes, o uso de herança permite a criação de relações do tipo *é-um* e o uso de composição permite a criação de relações do tipo *é-parte-de*. Por exemplo, um objeto da classe Carro de um sistema *é-um* objeto da classe Veículo, se for definida uma hierarquia de classes em que a classe Carro é subclasse de Veículo. Neste mesmo sistema, um objeto Roda *é-parte-de* um objeto Carro se for considerada a composição das classes Carro e Roda. No entanto, certos elementos de projeto, tais como Aerodinâmica ou Conforto, não são definidos adequadamente com herança ou composição, pois estes requisitos atravessam as decisões de implementação de diversos itens do carro. Tais requisitos são implementados na AOP por meio de *aspectos*.

Algumas linguagens permitem a implementação de programas orientados por aspectos, dentre elas, destacam-se as implementações para as linguagens: Java, representada por *AspectJ* [22] e *HyperJ* [32, 33]; C++, com a implementação de *AspectC++* [41]; C, com a implementação de *AspectC* [11]. Existem também propostas para a AOP independente de linguagem [25].

Para a modelagem de sistemas, há estudos de adaptações da Linguagem Unificada de Modelagem (UML) [37, 38] para o projeto de sistemas orientados por aspectos [7, 8, 42]. Padrões de composição [6, 39] podem ser destacados como metodologias de projeto para a separação de requisitos transversais. A orientação por aspectos pode auxiliar também na reestruturação (*refactoring*) de código [17, 18], tornando-o mais legível e reutilizável.

Os mecanismos de AOP são também poderosas ferramentas de instrumentação de código, permitindo inclusive alterar a estrutura interna das classes, como na implementação de padrões de projetos [15, 19].

Dentre as aplicações de desenvolvimento de software orientado por aspectos, destacam-se a reestruturação do núcleo do sistema operacional *FreeBSD* [9, 10, 11], a implementação de sistemas distribuídos [35, 40], a implementação de *frameworks* [12] e implementação de persistência como requisito transversal [36, 40].

Este minicurso tem por objetivos: apresentar os princípios e as técnicas do desenvolvimento de software orientado por aspectos; mostrar a aplicação de seus conceitos na linguagem AspectJ; realizar um estudo comparativo entre o desenvolvimento orientado por aspectos e o desenvolvimento puramente orientado por objetos; exibir estudos de caso que mostrem a aplicabilidade da orientação por aspectos.

Seção 2 deste texto define requisitos transversais e justifica, por meio do levantamento das limitações da POO, a necessidade de se criar construções próprias para a sua modularização. Seção 3 enfoca os elementos metodológicos do desenvolvimento orientado por aspectos. Seção 4 tem por objetivo apresentar a linguagem AspectJ. Seção 5 contém exemplos de implementações utilizando orientação por aspectos. Seção 6 apresenta as conclusões do minicurso.

2. Implementação de Requisitos de Sistemas

2.1. Introdução

Um módulo é um artefato de programação que pode ser desenvolvido e compilado separadamente de outras partes que compõem o programa [34, 13, 44]. Na programação estruturada, a modularização se limita à implementação de abstrações dos comandos (procedimentos) necessários à realização de uma tarefa. Com o advento da orientação por objetos [30], pode-se obter um grau mais elevado de modularização, por ser possível realizar abstrações de estruturas de dados, tipos e de suas operações por meio da implementação de interfaces. Além disso, com o polimorfismo, é possível definir comportamentos distintos para interface comuns a objetos relacionados, sem a necessidade de acessar elementos básicos da implementação.

Desenvolvedores criam sistemas a partir de requisitos que podem ser classificados como *requisitos funcionais*, que constituem o objetivo final do sistema, e *requisitos não-funcionais*, que compreendem elementos específicos de projeto, muitas vezes sem relação direta com o problema em questão. Por exemplo, em um sistema de processamento de cartões de crédito, um exemplo de requisito funcional é o processamento de pagamentos, ao passo que geração de registro de operações (*logging*), garantia de integridade de transações, autenticação de serviços, segurança e desempenho são requisitos não-funcionais.

Por facilitar a expressão dos objetos do problema em solução, a orientação por objetos permite a boa modularização dos requisitos funcionais do sistema [30]. Entretanto, um problema que este modelo não é capaz de resolver adequadamente está relacionado à modularização de requisitos não mapeáveis diretamente em uma ou poucas classes de um sistema, pois estes tendem a se espalhar por todo o código do programa [21, 23]. Estes requisitos são denominados *requisitos transversais*.

2.2. Programação Orientada por Aspectos

A Programação Orientada por Aspectos (AOP) [23] foi desenvolvida a partir da constatação de que certos requisitos não se encaixam em um único módulo de programa, ou pelo menos em um conjunto de módulos altamente relacionados. Em sistemas orientados por objetos, a unidade de modularização é a classe, e os requisitos transversais se espalham por múltiplas classes. Estes requisitos são também denominados *aspectos*. Ainda nesse trabalho, defende-se a tese de que tais requisitos são difíceis de modularizar por serem transversais aos requisitos funcionais do sistema.

A falta de tratamento adequado aos requisitos transversais pode resultar no baixo grau de modularização do sistema. Por estar espalhado em diversos módulos do sistema, torna-se difícil conceber, implementar e modificar código relacionado à implementação de um requisito transversal.

Figura 1 ilustra a relação entre alguns dos requisitos de um carro. Nesta representação, percebe-se que os componentes de um carro formam uma dimensão e cada requisito transversal forma uma dimensão distinta, cuja evolução deve ser independente das demais. Entretanto, em linguagens orientadas por objetos, a implementação destes elementos, naturalmente multidimensionais, deve ser feita em uma única dimensão: a dimensão de implementação dos requisitos funcionais. Em outras palavras, o espaço de requisitos, que é multidimensional, deve ser projetado no espaço de implementação que é unidimensional.

Este problema é exemplificado na classe definida a seguir, extraída de [16]:

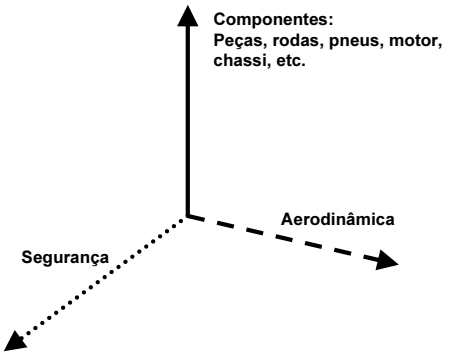

Figura 1: Espaço multidimensional dos requisitos de um carro

```
public class SomeBusinessClass extends OtherBusinessClass {

    "Dados membros do módulo"
    "Outros dados membros" // stream de logging, flag de consistência de dados

    "Métodos redefinidos da superclasse"

    public void performSomeOperation(OperationInformation info) {
        "Garante autenticidade"
        "Garante que info satisfaça contrato"
        "Bloqueia o objeto para garantir consistência caso outras threads o acessem"
        "Garante que a cache está atualizada"
        "Faz o logging do início da operação"
        "REALIZA A OPERAÇÃO OBJETIVO"
        "Faz o logging do final da operação"
        "Desbloqueia o objeto"
    }

    "Outras operações semelhantes à anterior"

    public void save(PersistenceStorage ps) { "..." }
    public void load(PersistenceStorage ps) { "..." }

}
```

Neste código, alguns problemas podem ser identificados:

- a parte de “Outros dados membros” não pertence ao objetivo principal da classe;
- o método performSomeOperation faz mais operações do que realizar a operação objetivo: registra e autentica a operação, sincroniza *threads*, valida de contrato e gerencia cache;
- não é claro se as operações save e load, que realizam a gerência de persistência, são métodos da parte principal da classe.

Outro problema é que a modificação em alguns dos requisitos, como por exemplo, a alteração da política de registro de operações pode originar modificações em diversas partes do programa, o que é uma tarefa desagradável e propensa a erros.

O problema da modularização de requisitos transversais pode ser dividido em *espalhamento* e *intrusão* [23]. O *espalhamento* (*scattering*) diz respeito a código para

implementação de um requisito transversal estar disperso no programa; por exemplo, o código de geração de registro de operações ou o código para verificar coerência de cache está distribuído ao longo de toda a implementação. A *intrusão* (*tangling*) diz respeito à confusão gerada por códigos de mais de um requisito transversal estarem presentes em uma única região do programa, como ocorre no método performSomeOperation do exemplo anterior, que não realiza apenas operações relacionadas ao seu objetivo principal, mas também geração de registro de operações, validação de contrato e coerência de cache.

As implicações destes problemas são diversas. Primeiramente, observa-se que elas dificultam o rastreamento do programa, pois a implementação simultânea de vários requisitos em um único método torna obscura a correspondência entre os requisitos e suas implementações, resultando em um mapeamento pobre entre os dois.

Além disso, obtém-se *baixa produtividade*, pois a implementação simultânea de vários requisitos em um módulo pode desviar a atenção do desenvolvedor do requisito principal para os requisitos periféricos, podendo levar à ocorrência constante de erros. Outro ponto importante é que este modelo possui *baixo grau de reúso*, pois, dado que um único módulo implementa vários requisitos, outros sistemas que precisarem de implementar funcionalidades semelhantes podem não ser capazes de utilizar prontamente o módulo, diminuindo ainda mais a produtividade do desenvolvedor.

Por estes pontos, pode-se perceber ainda que o código apresenta *pouca qualidade interna*, com baixa coesão modular e alto grau de acoplamento de módulos [31], uma vez que a intrusão pode produzir código com problemas ocultos, pois ao se preocupar com diversos requisitos ao mesmo tempo, o desenvolvedor pode não dar atenção suficiente a um ou mais requisitos.

Todos estes problemas levam à produção de código de *difícil evolução* ou *baixa extensibilidade* [30], visto que modificações posteriores no sistema podem ser dificultadas pela pouca modularização. Por exemplo, modificações na política de consistência da cache ou na política de validação de contratos pode levar à reorganização de diversos módulos do sistema.

2.3. Exemplo I: Geração de Registro de Operações

Um exemplo de código que se espalha ao longo de todo o programa é a geração de registro de operações. A implementação exibida nesta seção é utilizada ao longo do texto para exemplificação dos conceitos apresentados.

Para a implementação desta funcionalidade, define-se uma classe Logger, contendo métodos para escrever na saída padrão informações a respeito da entrada e do término da execução de um método da classe.

```
public class Logger {
    public static void logEntry(String message) {
        System.out.println("Entering " + message);
    }
    public static void logExit(String message) {
        System.out.println("Exiting " + message);
    }
}
```

Os métodos cujas execuções deseja-se acompanhar devem possuir chamadas aos métodos desta classe, como mostrado a seguir.

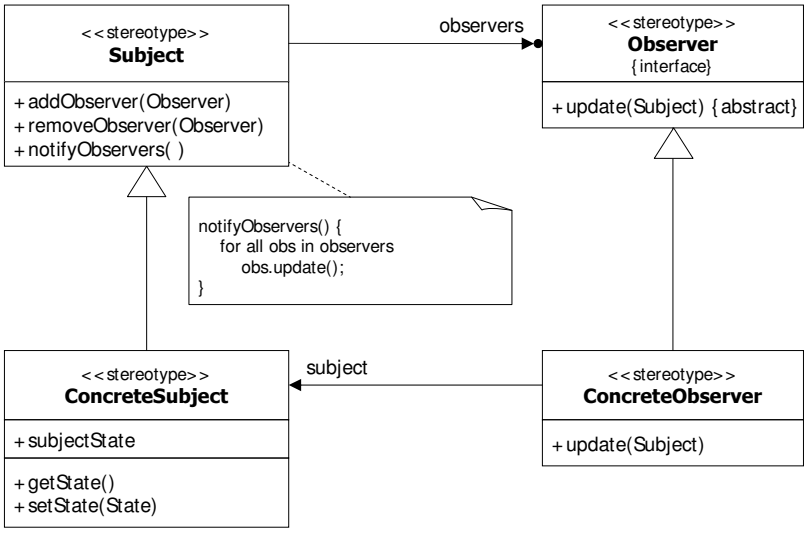

Figura 2: Diagrama UML do Padrão Observador

```
public class SomeClass {
    public void doSomething(int aParameter) {
        Logger.logEntry("doSomething(" + aParameter + ")");
        // Realiza a operação principal do sistema
        Logger.logExit("doSomething");
    }
}
```

A geração de registro de operações é um exemplo de espalhamento, devido às chamadas dos métodos da classe Logger por todo o programa, e de intrusão, visto que as chamadas de métodos da classe são colocadas no mesmo local de código relacionado aos objetivos principais dos módulos.

2.4. Exemplo II: Padrão Observador

O padrão de projeto *observador* tem por objetivo definir uma dependência de um para muitos entre objetos, de modo que, quando o estado de um objeto for alterado, todos os seus dependentes serão notificados e atualizados automaticamente [15].

O padrão observador é utilizado para desacoplar o objeto alvo de observação de seus observadores, permitindo maior grau de reúso. Por exemplo, em bibliotecas de implementação de interfaces gráficas com o usuário, os objetos alvo de observação são as componentes de interface e os observadores são responsáveis pela realização das operações de tratamento da manipulação da interface pelo usuário. O padrão observador permite, portanto, a separação entre dados e apresentação.

Figura 2 mostra a estrutura do padrão observador, utilizando a linguagem UML [37, 38]. Sua implementação consiste geralmente na definição da classe Subject, que representa os elementos alvos de observação, e de uma interface Observer, que representa os observadores de Subject. A classe Subject define também a interface de anexação (método Subject.addObserver) e remoção (método Subject.removeObserver) de obser-

vadores e implementa o protocolo de notificação dos observadores em caso de alteração no estado interno do objeto alvo (método Subject.notifyObservers).

```
import java.util.Iterator;
import java.util.LinkedList;
import java.util.List;

public class Subject {
    private List observers = new LinkedList();
    public void addObserver(Observer observer) {
        observers.add(observer);
    }
    public void removeObserver(Observer observer) {
        observers.remove(observer);
    }
    public void notifyObservers() {
        Iterator it = observers.iterator();
        while (it.hasNext())
            ((Observer) it.next()).update(this);
    }
}
```

A interface Observer define o protocolo de notificação de objetos que observam a mudança de estado interno (método Observer.update).

```
public interface Observer {
    public void update(Subject s);
}
```

Um pequeno exemplo do padrão observador é a implementação de termômetros para graus Celsius e Fahrenheit, que são observadores de uma classe que registra temperatura. Toda vez que a temperatura é alterada, os observadores são notificados e as ações apropriadas são tomadas.

A classe Temperature deve ser subclasse de Subject, para possuir os métodos de registro e notificação de observadores.

```
public class Temperature extends Subject {
    private double value;
    public double getValue() {
        return value;
    }
    public void setValue(double value) {
        this.value = value;
        notifyObservers();
    }
}
```

As classes TermometerCelsius e TermometerFahrenheit são observadores de temperatura e, portanto, são notificados toda vez que uma alteração da temperatura é realizada.

```
public class TermometerCelsius implements Observer {
    public void update(Subject s) {
        double value = ((Temperature) s).getValue();
        System.out.println("Celsius: " + value);
    }
}

public class TermometerFahrenheit implements Observer {
    public void update(Subject s) {
        double value = 1.8 * ((Temperature) s).getValue() + 32;
        System.out.println("Fahrenheit: " + value);
    }
}
```

A implementação do padrão observador é um exemplo de intrusão, visto que código específico do padrão fica misturado com o código da funcionalidade principal do sistema. Por exemplo, é necessário indicar que a classe Temperature é uma subclasse de Subject, o que não está de acordo com os objetivos do programa, mas sim com a implementação do requisito transversal.

Conforme identificado em [19], a implementação direta de padrões de projeto pode produzir diversos efeitos colaterais não desejados. Um dos principais efeitos relatados é o fato de o padrão “desaparecer no código”, perdendo assim a sua modularidade. Além disso, adicionar ou remover um padrão no sistema é geralmente uma substituição invasiva e de difícil retrocesso. Assim, enquanto a especificação do padrão é reusável, a sua implementação em geral não pode ser totalmente modularizada.

2.5. Conclusão

Nesta seção foram relatados alguns problemas encontrados no desenvolvimento de sistemas orientados por objetos. Dentre os mais importantes, pode-se citar a intrusão e o espalhamento de código gerados na implementação de requisitos transversais.

A existência de código de mais de um requisito em um mesmo módulo pode gerar alto acoplamento quando é necessário definir interfaces complexas para módulos. Além disso, a implementação de mais de um requisito em um mesmo módulo pode implicar no desenvolvimento de módulos com baixa coesão interna. Estes problemas geram códigos com baixo grau de reúso e alto custo para extensão.

3. Desenvolvimento de Software Orientado por Aspectos

3.1. Introdução

Nesta seção, mostra-se como a separação de requisitos transversais proposta na orientação por aspectos pode ser aplicada nas fases de desenvolvimento de sistemas. Inicialmente, são mostrados os passos de desenvolvimento de programas orientados por aspectos. Em seguida, mostram-se os elementos da programação orientada por aspectos. No final, é feita uma breve descrição sobre a modelagem de aspectos por meio da UML.

3.2. Metodologia de Orientação por Aspectos

Segundo [16], o desenvolvimento de software orientado por aspectos é realizado em três fases: a decomposição, a implementação e a recomposição de requisitos.

A *decomposição de requisitos* consiste na seperação dos requisitos em funcionais e transversais. No exemplo da classe SomeBusinessClass da Seção 2.2, pode-se identificar os seguintes requisitos: operação objetivo, autenticação e registro de operação, sincronização de *threads*, validação de contrato, coerência de cache e persistência. A operação objetivo deve ser implementada na classe, por representar um de seus requisitos funcionais. Todos os demais requisitos são transversais para este módulo, devendo, portanto, ser implementados em outros módulos.

Em [5], os requisitos transversais são classificados em requisitos de infra-estrutura, de serviços e de paradigmas. Requisitos de infra-estrutura consistem em requisitos responsáveis por coordenação (escalonamento e sincronização), distribuição (tolerância a falhas, comunicação, serialização e replicação) e persistência. Os requisitos de serviços são representados pelos requisitos de segurança, recuperação de erros, operações de retrocesso (*undo*), rastreamento e registro de operações. Exemplos de requisitos de paradigma são os padrões *visitor* e *observer* [15].

Após a decomposição, realiza-se a *implementação independente dos requisitos*. No exemplo da Seção 2.2, deve-se implementar as unidades de lógica do negócio, registro de operações, autorização, etc., em módulos separados. Para a implementação de cada módulo, utiliza-se os recursos da POO padrão. Por exemplo, a geração de registro de operações pode ser feita por meio da implementação de classes e interfaces específicas para o problema, possivelmente organizadas por meio de padrão de projetos. É possível, ainda, utilizar bibliotecas externas, tal como *Log4J* [27].

Após a implementação separada de cada requisito, especificam-se *as regras de recomposição do sistema*. Estas regras são implementadas em módulos denominados *aspectos*. Os aspectos definem como os requisitos são compostos para formar o sistema, em um processo denominado *costura* (*weaving*). Ainda no exemplo da Seção 2.2, um aspecto pode conter regras que definem os pontos do programa onde chamadas de métodos de registro de operações devem ser inseridas.

3.3. Recursos da Programação Orientada por Aspectos

Na programação orientada por aspectos, definem-se *pontos de junção* (*joinpoints*) como sendo pontos da execução do programa. Exemplos de pontos de junção são as chamadas de métodos de uma classe. Associados aos pontos de junção mais dois conceitos são importantes: *conjuntos de junção* e *regras de junção*.

As construções do programa que contêm pontos de junção são denominadas *conjuntos de junção* (*pointcuts*) e têm a função de reunir informações a respeito do contexto destes pontos. Os códigos relativos aos requisitos transversais que devem ser executados em pontos de junção são denominados *regras de junção* (*advices*).

Linguagens orientadas por aspectos devem possuir como elementos básicos: um modelo de definição de *pontos de junção*, uma forma de identificar pontos de junção, isto é, definir conjuntos de junção, e uma forma de interferir na execução de pontos de junção. Estes elementos são encapsulados em módulos denominados *aspectos*, que podem ser definidos como unidades de implementação modular de requisitos transversais e contêm definições de conjuntos e regras de junção.

Considere, por exemplo, a seguinte definição em AspectJ do aspecto LoggingAspect, contida no arquivo LoggingAspect.java:

```
public aspect LoggingAspect {

    pointcut publicMethods(): execution(public * *(..));
    pointcut logObjectCalls() : execution(* Logger.*(..));
    pointcut loggableCalls() : publicMethods() && ! logObjectCalls();

    before(): loggableCalls() {
        Logger.logEntry(thisJoinPoint.getSignature().toString());
    }

    after(): loggableCalls() {
        Logger.logExit(thisJoinPoint.getSignature().toString());
    }

}
```

O aspecto LoggingAspect define os seguintes conjuntos de junção:

- publicMethods: engloba os pontos de junção de execução dos métodos públicos do programa;
- logObjectCalls: engloba os pontos de junção de execução dos métodos da classe Logger;
- loggableCalls: engloba os pontos de junção de execução dos métodos públicos do programa que não pertençam à classe Logger.

São também definidas as regras de junção before e after, contendo os códigos para serem executados antes e depois das execuções dos métodos identificados nos conjuntos de junção indicados. Na regra before, define-se o código a ser executado antes da execução dos métodos públicos, que corresponde à geração do registro de entrada no método; este código corresponde à chamada do método doEntry da classe Logger, passando-lhe a assinatura do ponto de junção [1]. Da mesma forma, na regra after, define-se o código a ser executado após a execução dos métodos públicos, que corresponde à geração do registro de saída do método.

O compilador de aspectos é denominado *weaver* e sua função é realizar o processo de costura de código (*weaving*), entrelaçando os códigos das regras de junção com os códigos dos pontos de junção especificados nos aspectos. O processo de costura consiste na instrumentação do código das classes para executar o código definido nos aspectos.

[1] A expressão thisJoinPoint.getSignature().toString() obtém uma cadeia de caracteres que corresponde à assinatura do método que contém o ponto de junção ao qual a regra se aplica (thisJoinPoint)

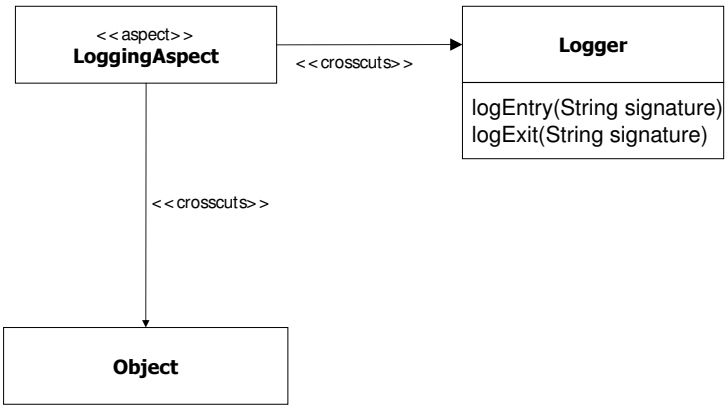

Figura 3: Diagrama UML para a geração de registro de operações por meio de aspectos

3.4. Modelagem de Aspectos

Para a modelagem de sistemas, há estudos de adaptações da UML [37, 38] para o projeto de sistemas orientados por aspectos [7, 8, 42]. Destes, o estudo de padrões de composição [6, 39] pode ser destacado como uma metodologia de projeto para a separação de requisitos transversais. Um aspecto pode ser representado no diagrama de classes da UML de maneira semelhante a uma classe. Utiliza-se o designador <<aspect>> para denotar um aspecto. Além disso, no relacionamento de entidades, utiliza-se o designador <<crosscuts>> para indicar que um aspecto é transversal a uma classe.

Por exemplo, Figura 3 mostra o diagrama de classes para o problema de geração de registros de operações. Nele, define-se que o aspecto de geração de registros (LoggingAspect) implementa a transversalidade entre a classe Logger, responsável pela geração dos registros, e ao estereótipo de classe BusinessClass.

Figura 4 apresenta o diagrama de classes para o padrão observador, utilizando aspectos. O modelo indica um aspecto abstrato que é transversal às interfaces Subject e Observer, que representam um alvo de observação e seu observador, respectivamente. Este aspecto define métodos para adicionar e remover um observador de um alvo de observação (addObserver e removeObserver). Além disso, define também um método abstrato updateObservers, que é chamado quando o estado de um alvo de observação for alterado. O conjunto de junção abstrato subjectChange (ver Seção 4) designa os pontos do programa que representam alterações no estado interno do alvo. Para tornar os objetos de uma classe do sistema ConcreteSubject observada por objetos de outra classe ConcreteObserver, cria-se um aspecto ConcreteObserverProtocol que estende ObserverProtocol.

Este aspecto é transversal às classes da aplicação e define os pontos do programa que representam alterações no estado interno do objeto alvo e a ação a ser realizada para atualizar os observadores. Além disso, este aspecto define ainda que as classes ConcreteObserver e ConcreteSubject implementam as interfaces Observer e Subject respectivamente. Observe que as classes alvo e observadora estão completamente disassociadas, sendo que o acoplamento entre elas está somente no aspecto.

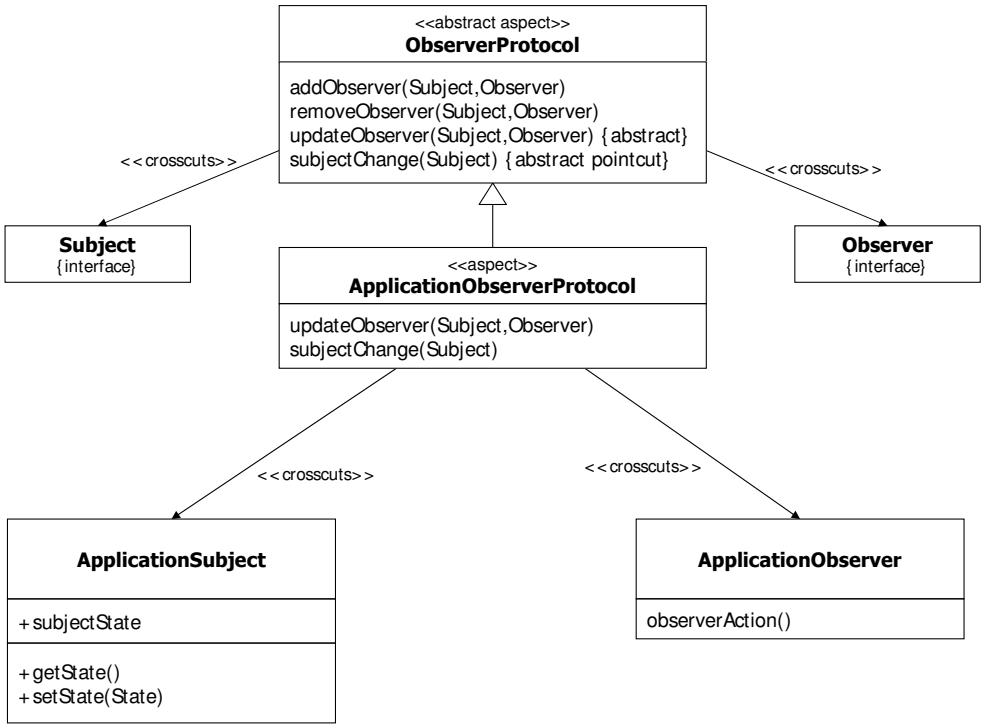

Figura 4: Diagrama de classes para o padrão observador utilizando aspectos

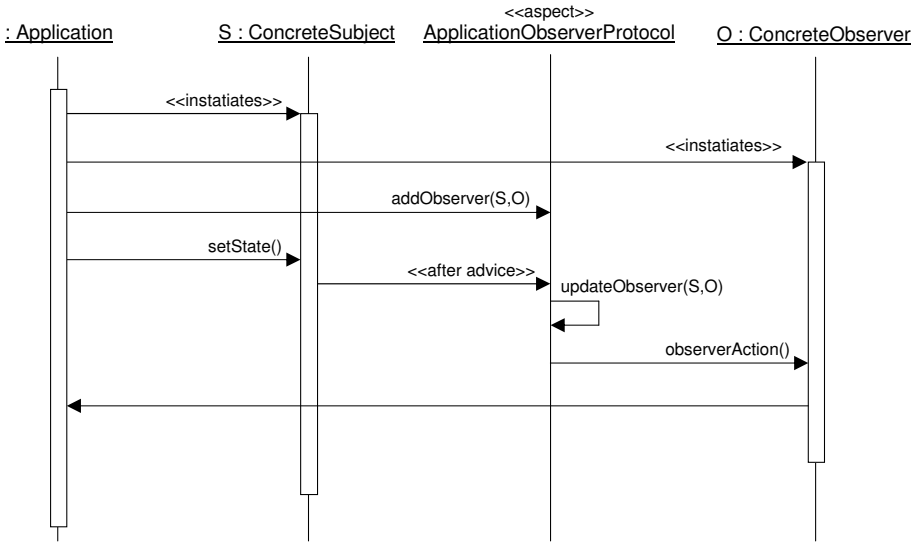

Figura 5: Diagramas de seqüência para a notificação de observadores.

Além do diagrama de classes, o diagrama de seqüências da UML pode ser utilizado para modelar a relação temporal entre as classes e os aspectos de um sistema. Por exemplo, Figura 5 mostra o diagramas de seqüências para a instanciação de alvo e observador, anexação de observador e alteração do estado do alvo de observação durante a execução de um programa. Assim como no diagrama de classes, pode-se observar no diagrama de seqüências que o objeto alvo de observação e o objeto observador são desacoplados, visto que toda interação é feita por meio do aspecto, de modo que não há chamadas diretas de métodos de um a partir de outro.

3.5. Conclusão

O desenvolvimento de sistemas orientado por aspectos possui semelhanças com o desenvolvimento de sistemas orientados por objetos. A proposta metodológica consiste em definir e implementar os requisitos do sistema por meio de orientação por objetos e, em seguida, definir regras de costura que façam o entrelaçamento dos requisitos em diferentes módulos.

Para modelagem de aspectos, pode-se utilizar a própria linguagem UML, com alguns estereótipos apropriados para os novos elementos introduzidos pela metodologia. Dentre os diagramas da UML, os mais utilizados são os diagramas de classes, de seqüência e de colaboração. Como poucos conceitos são introduzidos nestes diagramas, conclui-se que a orientação por aspectos, no nível de projeto do sistema, é bastante acessível aos desenvolvedores de sistemas de software.

4. A Linguagem de Programação AspectJ

4.1. Introdução

Nesta seção, é apresentada a linguagem AspectJ [21, 16, 24, 3], uma das linguagens orientadas por aspectos mais importantes da literatura. Seus aspectos sintáticos e semânticos são discutidos e, no fim da seção, apresenta-se um panorama das demais linguagens orientadas por aspectos seguida de um estudo comparativo com AspectJ.

4.2. A Linguagem AspectJ

A linguagem *AspectJ* [21] é uma extensão de Java com suporte a dois tipos de implementação de requisitos transversais: a *transversalidade dinâmica*, que permite definir implementação adicional em pontos bem definidos do programa, e a *transversalidade estática*, que afeta as assinaturas estáticas das classes e interfaces de um programa Java.

Atualmente, o desenvolvimento do ambiente de execução da linguagem é um projeto Eclipse [14, 3], cujo *website* contém ferramentas para o desenvolvimento de programas orientados por aspectos.

4.2.1. Pontos de Junção

Pontos de junção são posições bem definidas da execução de um programa, como por exemplo, chamadas de métodos ou execução de blocos de tratamento de exceções. Pontos de junção podem também possuir contexto associado. Por exemplo, o contexto de um ponto de junção de chamada de método contém o objeto alvo da chamada e a lista de argumentos da chamada do método.

4.2.2. Conjuntos de Junção

Um conjunto de junção é formado por um conjunto de pontos de junção e as informações de contexto destes pontos. Eles permitem a especificação de coleções de pontos de junção e a exposição de seus contextos para a implementação de regras de junção. Conjuntos de junção podem ser vistos como registros de identificação de pontos de junção em tempo de execução. Por exemplo, o designador de conjunto de junção **call**(**void** Point.setX(**int**)) identifica todos os pontos de junção formados por chamadas do método de assinatura **void** Point.setX(**int**).

A definição de conjuntos de junção em AspectJ obedece à seguinte sintaxe:

pointcut nome-conjunto(arg1, ..., argn): definição;

define um conjunto de junção de nome nome-conjunto com os parâmetros arg1, ..., argn para identificar os pontos de junção representados por definição.

O ponto de junção a seguir define todas as chamadas do método Point.setX(**int**), em que o valor passado como parâmetro seja menor que zero:

pointcut conjunto(**int** x): **call**(**void** Point.setX(**int**) && **args**(x) && **if**(x $<$ 0);

A linguagem AspectJ permite a coleta dos seguintes pontos de junção: (i) chamadas de métodos e construtores; (ii) execuções de métodos e construtores; (iii) acesso a campos de classes e objetos; (iv) execuções de tratadores de exceções; (v) execuções de inicialização de classes.

Chamadas de Métodos e Construtores

Os conjuntos de junção de chamadas de métodos e construtores descrevem os pontos de execução após a avaliação dos argumentos no ponto de chamada de um método. Para definir este tipo de conjunto de junção, utiliza-se o operador **call**, parametrizado com a assinatura do método. Por exemplo, o designador **call**(**void** Point.getX(**int**)) representa os pontos de junção que são chamadas do método getX da classe Point sem valor de retorno e com um parâmetro inteiro.

Para identificar chamadas de construtores, utiliza-se a palavra **new** no lugar do nome do método e não se utiliza o tipo de retorno. Por exemplo, os pontos de junção que são chamadas do construtor da classe Point que recebe dois parâmetros inteiros são representados pela construção **call**(Point.new(**int**,**int**)).

Execuções de Métodos e Construtores

Os conjuntos de junção de execução de métodos e construtores capturam as suas execuções. Em contraste com os conjuntos de junção de chamada, conjuntos de junção de execução representam o corpo de um método ou construtor.

Para definir este tipo de conjunto de junção, utiliza-se o operador **execution**, parametrizado com a assinatura do método. Por exemplo, para representar todas as execuções do método getX da classe Point sem valor de retorno e com um parâmetro inteiro, pode-se utilizar a construção **execution**(**void** Point.getX(**int**)).

A definição de conjuntos de junção de execuções de construtores é semelhante à definição de chamadas. Por exemplo, a construção **execution**(Point.new(**int**,**int**)) designa os pontos de junção de execução do construtor da classe Point que recebe dois parâmetros inteiros.

Os pontos de junção de chamada e de execução de métodos e construtores têm como diferenças básicas:

- o contexto no qual estão inseridos: por exemplo, a chamada do método f no corpo do método g leva como informação de contexto para o **call** os elementos de g, ao passo que, para **execution**, leva os elementos de f;
- o local onde o código do aspecto é costurado: os códigos das regras são espalhados no ponto de chamada para conjuntos de junção **call** e no ponto de definição do método para conjuntos de junção **execution**.

Acessos a Campos de Classes e Objetos

Os conjuntos de junção referentes a operações com campos de classe capturam acessos de leitura e escrita a atributos estáticos e não-estáticos de uma classe. A captura da leitura de um campo utiliza o operador **get** e a de escrita, o operador **set**, ambos parametrizados com a assinatura do campo. Por exemplo, a construção **get**(PrintWriter System.out) representa os acessos de leitura do campo out da classe System, ao passo que **set**(int MyClass.x) representa as operações de escrita ao campo x da classe MyClass.

Execuções de Tratadores de Exceções

Os conjuntos de junção referentes a tratamento de exceção capturam a execução dos blocos **catch**. Eles podem ser definidos por meio do operador **handler**, parametrizado pelo nome da classe de exceção. Por exemplo, para representar os blocos **catch** que tratam a exceção NumberFormatException, utiliza-se **handler**(NumberFormatException).

Execuções de Inicialização de Classes

Os conjuntos de junção referentes à inicialização de classes capturam a execução do código especificado no bloco static dentro das definições de classes. A identificação é feita por meio do operador **staticinitialization**, parametrizado pelo nome da classe.

4.2.3. Outros Recursos Para Transversalidade Dinâmica

Combinações de Conjuntos de Junção

Conjuntos de junção podem ser combinados utilizando-se os operadores de interseção (&&), união (||) e complemento (!). Por exemplo, o seguinte designador composto representa todas as chamadas dos métodos getX e getY da classe Point:

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

Símbolos Curinga

É possível ainda utilizar símbolos curinga (*wildcard*) para identificar todos os métodos que casam com definições parciais.

O símbolo ".." pode ser utilizado para designar que se devem ser casados os métodos com quaisquer números e tipos de argumentos. Por exemplo, a construção **void** MyClass.myMethod(..) representa todos os métodos de nome myMethod que pertençam à classe MyClass e que retornem **void**.

O símbolo "*" pode ser utilizado para designar:

- Qualquer tipo de retorno, como em: * MyClass.myMethod(..)
- Qualquer classe, como em: * *.myMethod(..)
- Qualquer método, como em: * MyClass.*(..)
- Nomes parciais de métodos, como em: * MyClass.get*(..)
- Nomes parciais de campos, como em: * MyClass.id*

Por exemplo, a construção

```
execution(public * *(..)) && !execution(* Logger.*(..))
```

define todos os pontos de junção de execuções de métodos públicos que não pertençam à classe Logger.

Utiliza-se o símbolo "+" para indicar o tipo e suas subclasses. Por exemplo, a construção **handler**(IOException+) representa os blocos **catch** que tratam IOException ou suas subclasses. Outro exemplo, pode-se representar construtores de subclasses por meio do símbolo "+" , como em MyClass+.**new**(..).

Os símbolos curinga podem ser combinados para formar conjuntos de junção complexos, como em **call**(**void** Figure+.set*(..)), que descreve todos os pontos de junção de chamadas de métodos cujos nomes iniciem com set e que pertençam à classe Figure ou a uma de suas subclasses.

Definições Baseadas em Modificadores

É possível ainda definir conjuntos de junção baseados em modificadores: **public**, **private**, **static**, etc. Por exemplo, a construção **call**(**public** * *(..)) representa todos os métodos públicos do programa.

Conjuntos de junção podem ser definidos também por meio da negação de modificadores, utilizando-se o símbolo "!" . Por exemplo, a construção **call**(!**static** * *(..)) representa todos os métodos não estáticos do programa, ao passo que todos os métodos públicos não-estáticos são representados pela construção **call**(**public** !**static** * *(..)).

Definições Baseadas na Estrutura do Programa

A estrutura do programa pode ser utilizada na definição de conjuntos de junção. Em AspectJ, pode-se definir conjuntos de junção que ocorram dentro de classes ou dentro de métodos.

O operador **within**, é utilizado para definir pontos de junção que ocorrem no escopo da classe cujo nome é dado como parâmetro. Por exemplo, a construção **within**(Point) captura todos os pontos de junção que ocorrem dentro da classe Point.

O operador **withincode** permite capturar pontos de junção no escopo de métodos de classe, passando no lugar de um nome de classe a assinatura de um método. Por exemplo, a construção **withincode**(**void** Device.update()) captura todos os pontos de junção que ocorrerem dentro do método **void** Device.update().

Definições Baseadas no Contexto

Contextos podem ser utilizados na definição de conjuntos de junção. Em AspectJ, pode-se definir conjuntos de junção nos seguintes elementos de contexto: tipo do objeto no qual o ponto de junção ocorre; tipo do objeto receptor de uma chamada de método; fluxo de controle a que pertence um ponto de junção; argumentos de métodos e construtores.

A construção **this**(algum-tipo) é utilizada para definir pontos de junção que ocorrem durante a execução de métodos cujo objeto receptor é do tipo algum-tipo. Por exemplo, a construção **this**(Figure) captura todos os pontos de junção que ocorrem quando o objeto corrente (**this**) for do tipo Figure.

Os contextos **this** e **within** representam, respectivamente, os tipos dinâmico e estático de um objeto. Por exemplo, considere que Point seja subclasse de Figure e defina um método de assinatura **void** setX(**int**). Um ponto de junção que ocorre dentro deste método pode ser identificado por **this**(Figure), pois o tipo do objeto corrente, Point, é também da classe Figure. Entretanto, estes mesmos pontos de junção não podem ser identificados por **within**(Figure), se o método não estiver definido dentro desta classe.

A construção **target**(algum-tipo) é utilizada para definir pontos de junção de cha-

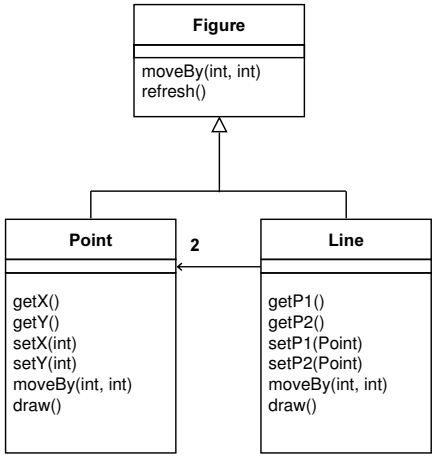

Figura 6: Exemplo utilizado na diferenciação de operadores de escopo e contexto

madas de métodos em que o tipo do objeto receptor é algum-tipo. Por exemplo, a construção **target**(Point) captura todos os pontos de junção de chamadas de métodos que têm Point como tipo dos objetos receptores.

A construção **args**(arg1, ..., argn) é utilizada para definir pontos de junção que utilizam os valores dos argumentos de uma chamada de método. Por exemplo, a construção **call**(**void** Device.move(Figure,**int**,**int**)) && args(Point,**int**,**int**) captura todos os pontos de junção de chamada do método Device.move, em que o primeiro parâmetro passado seja uma instância do tipo Point, subclasse de Figure.

Definições Baseadas no Fluxo de Execução

A construção **cflow**(algum-ponto-de-junção) é utilizada para definir pontos de junção que ocorrem no fluxo de controle gerado a partir de um ponto de junção. Por exemplo, a construção **cflow**(**execution**(**void** Device.update())) captura todos os pontos de junção que ocorrem no fluxo de controle da execução do método Device.update.

Pode-se utilizar ainda o especificador **cflowbelow**, para designar os pontos de junção que ocorrem no fluxo de controle a partir de um ponto de junção, excluindo-se o ponto de junção gerador do fluxo.

Para diferenciar os especificadores de escopo e contexto, considere as classes definidas na Figura 6. Considere ainda o seguinte trecho de código:

```
public class ClientClass {
    public void someMethod() {
        //...
        Line L = new Line(new Point(10,6),new Point(30,5));
        //...
        L.moveBy(10,16);
        //...
    }
}
```

A chamada do método Line.moveBy no corpo do método ClientClass.someMethod pode ser representada pelo conjunto de junção **within**(ClientClass), pois ocorre dentro do

corpo de ClientClass. Da mesma forma, este ponto de junção pode ser representado por **withincode**(**void** ClientClass.someMethod()), por ocorrer no corpo deste método.

Com relação ao contexto, este ponto pode ser identificado também por meio da construção **this**(ClientClass), pois ocorre durante a execução do método someMethod, cujo objeto receptor é da classe ClientClass, ou ainda como **target**(Line), pois o seu objeto receptor, L, é da classe Line. É possível também identificá-lo como **target**(Figure), pois o objeto L é também do tipo Figure, superclasse de Line.

Por outro lado, durante a execução do método Line.moveBy, os pontos de junção que ocorrerem durante esta execução podem ser identificados por **within**(Line) ou então por **withincode**(**void** Line.moveBy). Porém, não é possível identificá-los por meio da construção **within**(Figure) ou **withincode**(**void** Figure.moveBy), pois não ocorrem no corpo da classe Figure. Estes pontos de junção podem ser representados também por meio das construções **this**(Line) ou **this**(Figure).

A construção **cflow**(**execution**(**void** ClientClass.someMethod())) pode ser utilizada para identificar também todos os pontos de junção deste exemplo, pois ocorrem no fluxo de controle de execução do programa gerado a partir da execução do método ClientClass.someMethod.

Condicionais

A construção **if**(expressão-lógica) pode ser utilizada para identificar pontos de junção em que a condição expressa por uma expressão lógica seja verdadeira.

Por exemplo, a construção **call**(**void** Point.setX(**int**)) && **args**(x) && **if**(x $<$ 0) captura todos os pontos de junção de chamada do método Point.setX, em que o valor passado como parâmetro seja menor que zero.

4.2.4. Reflexão Computacional

A linguagem AspectJ suporta uma forma limitada de reflexão, que permite examinar informações nos pontos de execução de algum ponto de junção. Cada corpo de regra de junção possui um objeto especial, thisJoinPoint, que contém informações sobre o ponto de junção. Este tipo de reflexão é importante, por exemplo, para definição de aspectos de registro de operações e depuração.

4.2.5. Regras de Junção

Regras de junção representam códigos que devem ser executados em pontos de junção. Em AspectJ, é possível definir regras que são executados *antes* (before), *após* (after) ou *ao redor de* (around) pontos de junção.

Uma regra before é executada imediatamente antes da execução do ponto de junção, ao passo que uma regra after é executada imediatamente após a execução do ponto de junção.

Por exemplo, considere as seguintes definições de conjuntos de junções e regras:

```
pointcut publicMethods(): execution(public * *(..));
pointcut logObjectCalls() : execution(* Logger.*(..));
pointcut loggableCalls() : publicMethods() && ! logObjectCalls();
```

```
before(): loggableCalls() {
    Logger.doEntry(thisJoinPoint.getSignature().toString());
}

after(): loggableCalls() {
    Logger.doExit(thisJoinPoint.getSignature().toString());
}
```

O conjunto de junção loggableCalls representa todos os pontos de junção de execuções de métodos públicos que não pertencem à classe Logger. A regra before indica que, antes de um ponto de junção pertencente ao conjunto loggableCalls, deve-se emitir uma mensagem indicando que o método está iniciando sua execução; isto é feito por meio do método Logger.doEntry. Da mesma forma, a regra after indica que, após a execução de um ponto de junção pertencente ao conjunto loggableCalls, deve-se emitir uma mensagem indicando que o método está terminando sua execução; isto é feito por meio do método Logger.doExit. Observe que o objeto thisJoinPoint é utilizado para obter a assinatura do método.

É possível, ainda, definir regras after para serem executadas após o retorno normal de um método, utilizando-se para isso a construção after returning. Para definir regras a serem executadas em caso de retorno com lançamento de exceção, utiliza-se a construção after throwing.

Por exemplo, a regra after do programa anterior pode ser reescrita como:

```
after() returning (): loggableCalls() {
    Logger.doExit(thisJoinPoint.getSignature() + " with normal return");
}
after() throwing (Exception exc): loggableCalls() {
    Logger.doExit(thisJoinPoint.getSignature() + " with " + exc + " thrown");
}
```

A primeira regra indica que deve ser feito o registro do retorno normal do método no ponto de junção. A segunda regra indica que, se a saída do método foi feita por um lançamento de exceção, então o registro da saída deve exibir a exceção lançada.

Regras podem ser parametrizadas. Por exemplo, as regras after no trecho de código a seguir imprimem valores passados como parâmetro para setX ou retornados de getX.

```
pointcut settingX(int val): execution(void Point.setX(int)) && args(val);
pointcut gettingX(): execution(int Point.getX());

after(int val): settingX(val) {
    System.out.println("Value assigned to x: " + val);
}
after() returning (int val): gettingX() {
    System.out.println("Value of x: " + val);
}
```

Uma regra around envolve um ponto de junção e define se a execução deve prosseguir normalmente, ou com uma lista de parâmetros diferente, ou ainda se outra operação deve ser realizada sem a execução do ponto de junção. A regra *around* define também um tipo de retorno, de modo que é possível especificar um valor de retorno diferente do obtido na execução do ponto de junção.

O exemplo a seguir mostra o uso de regras *around* para garantir os limites do valor de x na classe Point. O conjunto de junção capturado é o conjunto das chamadas do método Point.setX. Na regra, testa-se o valor de x. Se este for válido, então a execução do ponto de junção continua normalmente com o valor passado. Caso contrário, a exceção IllegalArgumentException é lançada.

```
void around(Point p, int x) : target(p) && args(x) && call(void setX(int)) {
    if (x >= 0 && x < 100)
        proceed(p, x);
    else
        throw new IllegalArgumentException();
}
```

4.2.6. Aspectos

Um aspecto é uma unidade modular de implementação de requisitos transversais. Aspectos são definidos por declarações de aspectos, que são similares a declarações de classes em Java. Declarações de aspectos podem declarar conjuntos de junção e regras, incluir métodos e campos e podem estender classes ou outros aspectos.

Ao contrário de classes, aspectos não podem ser instanciados. Para obter referência a um aspecto, utiliza-se o método estático aspectOf(), presente em todos os aspectos.

Para ilustrar, o exemplo a seguir mostra a declaração do aspecto LoggingAspect, que define os conjuntos de junção publicMethods, logObjectCalls e loggableCalls, e regras after e before para loggableCalls.

```
public aspect LoggingAspect {

    pointcut publicMethods(): execution(public * *(..));
    pointcut logObjectCalls() : execution(* Logger.*(..));
    pointcut loggableCalls() : publicMethods() && ! logObjectCalls();

    before(): loggableCalls() {
        Logger.logEntry(thisJoinPoint.getSignature().toString());
    }

    after(): loggableCalls() {
        Logger.logExit(thisJoinPoint.getSignature().toString());
    }

}
```

O controle de visibilidade de membros de aspectos em AspectJ é igual ao controle de visibilidade de membros de classes em Java, sendo possível utilizar os modificadores **public**, **private** ou **protected**, ou ainda definir visibilidade de pacotes, se nenhum dos três modificadores for utilizado.

Aspectos podem ser estendidos ou especializados, sendo neste caso denominados subaspectos, formando uma hierarquia de aspectos. Não é permitida a definição de hierarquias múltiplas de aspectos. As regras para redefinição de conjuntos e regras de junção em subaspectos são similares às regras de redefinição de métodos em Java.

Um aspecto pode ser abstrato, ou pode possuir conjuntos de junção abstratos. A semântica de conjuntos de junção abstratos é semelhante a de métodos abstratos: as decisões de implementação são adiadas para aspectos concretos que estendam aspectos abstratos. O exemplo da Seção 5.4 mostra uma implementação de aspecto e conjunto de junção abstratos.

Em AspectJ, é possível ainda declarar conjuntos estáticos de junção dentro de classes. Entretanto, não é permitido a uma classe declarar regras; apenas aspectos podem conter regras. Aspectos internos a classes devem ser estáticos, pois não pertencem às suas instâncias. Por exemplo, a classe a seguir define um aspecto estático interno SetterEnforcement, que obriga o uso dos métodos setX e setY em todos os pontos do programa, evitando acesso direto aos campos x e y exceto nos corpos dos métodos de alterações (ver Seção 4.2.7).

```
public class Point {
    private int x, y;
    public Point(int x, int y) { this.x = x; this.y = y; }
    public int getX() { return x; }
    public int getY() { return y; }
    public void setX(int x) { this.x = x; }
    public void setY(int y) { this.y = y; }
    static aspect SetterEnforcement {
        declare error: set(int Point.*) && !withincode(void Point.set*(int)):
            "Use setter method, even inside Point class.";
    }
}
```

4.2.7. Transversalidade Estática

Algumas implementações de aspectos, como por exemplo a implementação de padrões de projeto, necessitam de recursos para alterar tanto o comportamento das classes em tempo de execução, quanto a sua estrutura estática.

A transversalidade dinâmica, obtida a partir de regras de junção, permite modificar o comportamento da execução do programa, ao passo que a transversalidade estática permite redefinir a estrutura estática dos tipos – classes, interfaces ou outros aspectos – e o seu comportamento em tempo de execução.

Em AspectJ, é possível implementar os seguintes tipos de transversalidade estática: (i) introdução de campos e métodos em classes e interfaces; (ii) modificação da hierarquia de tipos; (iii) declaração de erros e advertências de compilação; (iv) enfraquecimento de exceções.

Introdução de Campos e Métodos em Classes e Interfaces

A linguagem AspectJ possui um mecanismo denominado *introdução*, que permite a inclusão de atributos e métodos em classes ou interfaces de forma transversal. É possível também adicionar campos e métodos não-abstratos em implementações de interfaces de Java, permitindo adicionar um comportamento *default* às classes que as implementem. Isto permite que o desenvolvedor evite duplicidade de código em cada classe, visto que tal adição informa ao compilador de aspectos que este código deve ser incluído nas classes que implementem as interfaces.

Sem AspectJ, a estratégia usual consiste na criação de uma classe com a implementação *default* da interface, de modo que uma classe que implemente a interface possa ser definida como subclasse da classe com a implementação *default*. Por exemplo, a classe MouseAdapter é uma implementação *default* da interface MouseListener, fornecendo uma implementação vazia para cada um dos seus métodos. Uma classe que precisar, por exemplo, definir somente o comportamento do método mouseClicked pode estender esta classe, sem a necessidade de implementar os demais métodos. Entretanto, isto só pode ser utilizado se a classe não precisar estender outra classe do programa. O uso simplesmente de classes abstratas no lugar de interfaces impede o reúso de outras classes via herança. Este problema pode também ser resolvido por meio de composição. Entretanto, esta solução consiste na implementação de diversos métodos de uma única linha, dando origem a espalhamento.

Por exemplo, considere a interface a seguir cujo objetivo é modelar entidades que possam possuir um nome:

```
public interface Nameable {
    public void setName(String name);
    public String getName();
}
```

Cada classe que implementa esta interface deve definir os seus dois métodos, como no exemplo a seguir:

```
public class Entity implements Identifiable, Nameable {
    private String name;
    public void setName(String name) {
        this.name = name;
    }
    public String getName() {
        return name;
    }
}
```

Código semelhante deve ser inserido em todas as classes que implementarem esta interface. Para evitar esta duplicidade de código, é possível definir um aspecto que introduza a implementação *default* dos métodos para a interface. Para incluir atributos e métodos em interfaces e classes em Java, qualifica-se o seu nome com o nome da classe ou interface onde será inserido, seguido de um ponto.

```
public interface Nameable {
    public void setName(String name);
    public String getName();
    static aspect DefaultImplementation {
        private String Nameable.name;
        public void Nameable.setName(String name) {
            this.name = name;
        }
        public String Nameable.getName() {
            return name;
        }
    }
}
```

Com esta nova implementação, as classes que implementam a interface Nameable

não precisam mais definir os métodos, se a implementação *default* estiver adequada para a classe. Se alguma classe que implementa uma interface com comportamento *default* precisar personalizar algum método, basta fazer a sua implementação direta, pois estes irão redefinir a implementação *default* introduzida no aspecto.

A nova versão da classe *Entity* é mostrada a seguir:

```
public class Entity implements Identifiable, Nameable {
}
```

A solução tradicional de Java consiste em criar uma classe DefaultNameable, contendo as implementações *default*. A classe Entity deve ser, portanto, subclasse de DefaultNameable. Entretanto, esta abordagem funciona adequadamente somente para a implementação de uma única interface. Considere, por exemplo, a interface Identifiable definida a seguir:

```
public interface Identifiable {
    public long getId();
    public void setId(long id);
}
```

Se Entity implementar as interfaces Nameable e Identifiable, sem o uso de AspectJ, será necessário estender uma das classes de implementação *default* e implementar a outra interface, isto pode gerar duplicidade de código. Entretanto, se a mesma abordagem de comportamento *default* em AspectJ for utilizada para a interface Identifiable, a classe Entity não precisará implementar os seus métodos, podendo utilizar as implementações *default* de ambas interfaces.

```
public interface Identifiable {
    public long getId();
    public void setId(long id);
    static aspect DefaultImplementation {
        public long Identifiable.id;
        public long Identifiable.getId() {
            return id;
        }
        public void Identifiable.setId(long id) {
            this.id = id;
        }
    }
}
```

Com isto, a classe Entity implementa ambas as interfaces e o efeito é o mesmo de estender as duas implementações *default* se herança múltipla fosse permitida em Java.

Modificação da Hierarquia de Tipos

Em AspectJ, é possível modificar a hierarquia de classes, de modo a definir superclasses e implementação de interfaces para classes e interfaces já existentes, desde que isto não viole as regras de herança de Java. Como conseqüência, obtém-se o desacoplamento dos aspectos e das classes específicas da aplicação, aumentando, assim, o grau de reúso dos aspectos.

Tais declarações têm a forma:

```
declare parents: tipo-filho implements lista-interfaces;
declare parents: tipo-filho extends classe-ou-lista-interfaces;
```

Por exemplo, a declaração a seguir define que a classe Temperature implementa a interface Subject:

```
declare parents: Temperature implements Subject;
```

A declaração de *parents* deve obedecer às regras de hierarquias de classes em Java. Por exemplo, não é possível declarar superclasses para interfaces, ou então definir superclasses que resultem em herança múltipla.

Declaração de Erros e Advertências de Compilação

Em AspectJ, é possível declarar erros e advertências de compilação. Com este mecanismo, obtém-se comportamento similar ao obtido por meio das diretivas de compilação #error e #warning.

A construção **declare error** permite definir erros de compilação para serem exibidos quando o compilador detectar a presença de pontos de junção identificados por um dado conjunto de junção. O compilador de AspectJ então lança um erro, imprime a mensagem especificada para cada uso indevido e termina o processo de compilação. Esta construção tem a forma:

```
declare error: conjunto-junção : mensagem-erro;
```

Da mesma forma, é possível definir advertências de compilação, em que uma mensagem de erro é gerada, mas o processo de compilação não é interrompido:

```
declare warning: conjunto-junção : mensagem-erro;
```

O exemplo abaixo permite controlar a operação de escrita no campo x da classe Point, permitindo que somente o método setX lhe faça o acesso direto. Isto obriga todos os demais métodos a utilizarem setX, mesmo dentro da classe Point:

```
declare error: set(int Point.*) && !withincode(void Point.set*(int)):
    "Use setter method, even inside Point class.";
```

Enfraquecimento de Exceções

A linguagem Java obriga que todas as exceções, com exceção das subclasses de RuntimeException e Error, sejam declaradas na assinatura dos métodos ou capturadas e tratadas. Esta abordagem é útil por lembrar o desenvolvedor de tratar condições de erro no programa.

Entretanto, em algumas situações, é possível ter certeza de que uma exceção nunca ocorre, de modo que o código necessário para tratar a exceção torna-se inútil e polui a implementação. Por exemplo, considere que um método escreva informações em um arquivo temporário e, logo em seguida, abra este arquivo para leitura. A operação de abertura do arquivo deve estar contida em um trecho de código que trate a exceção de arquivo não encontrado, FileNotFoundException. Entretanto, o código de tratamento ou

de relançamento da exceção é inútil, pois tem-se certeza de que o arquivo existe – ele acabou de ser criado pelo próprio programa.

Em AspectJ, é possível, por meio da construção **declare soft**, silenciar exceções de forma seletiva e relançá-las como exceções não checadas. Esta construção é constituída por uma exceção e por um conjunto de junção. A sua semântica consiste em enfraquecer a exceção para o conjunto de junção especificado, fazendo com que o compilador Java não exija o seu tratamento.

```
declare soft: tipo-exceção : conjunto-de-junção;
```

No exemplo acima considere que o método **void** SomeClass.someMethod() crie um arquivo temporário e, em seguida reabra este arquivo para leitura, de modo que não há dúvidas de que o arquivo exista. A declaração a seguir define que a exceção não precisa ser tratada, o que simplifica a escrita do método:

```
declare soft: FileNotFoundException
    : withincode(void SomeClass.someMethod());
```

4.3. Conclusão

A linguagem AspectJ permite a implementação de programação orientada por aspectos em Java. Seus mecanismos de transversalidade permitem realizar interferências tanto na estrutura estática das classes de um programa Java quanto no comportamento do programa durante a sua execução.

Dentre as linguagens que implementam orientação por aspectos, destacam-se HyperJ [32], Jiazzi [29], AspectC++ [2], DemeterJ [26], AspectWerkz [4].

A linguagem HyperJ [32] foi desenvolvida pela IBM para o suporte à metodologia de separação multi-dimensional e integração de requisitos em Java. O objetivo desta metodologia é quebrar a *tirania da decomposição dominante*, permitindo que cada requisito seja implementado em uma dimensão distinta, de modo que o ambiente de compilação faça a composição, de forma semelhante à feita em AspectJ.

A linguagem Jiazzi [29] é uma extensão de Java que permite a compilação separada de módulos denominados *units*. Estas *units* funcionam de forma semelhante a aspectos em AspectJ, podendo alterar a estrutura e o comportamento de classes.

A linguagem AspectC++ [2] é uma implementação de orientação por aspectos em C++, com projeto baseado na linguagem AspectJ.

DemeterJ [26] é uma implementação da metodologia de *programação adaptativa* em Java. O seu objetivo é modularizar a implementação de requisitos relacionados a travessia de objetos em um sistema.

AspectWerkz [4] implementa transversalidades estática e dinâmica, permitindo que o processo de costura de código seja feito durante a compilação ou durante a execução. A sua implementação é flexível, permitindo que código de transversalidade seja implementada como combinação de código Java com anotações e arquivos XML.

O poder de expressão e recursos especiais destas linguagens para desenvolvimento de sistemas baseados em aspectos situam-se em um mesmo patamar de equivalência. Entretato, AspectJ destaca-se por sua grande difusão e alta aderência à proposta inicial do criador da programação orientada por aspectos.

5. Exemplos de Aplicações de Orientação por Aspectos

5.1. Introdução

Nesta seção, são discutidas implementações de bibliotecas reutilizáveis utilizando orientação por aspectos. Na primeira parte, é mostrada a implementação de um mecanismo modular e não-intrusivo para a geração de registros de operações. Em seguida, mostra-se uma implementação em AspectJ do padrão *singleton* [15], em que o código de controle do padrão é separado do código da classe implementada para se ter instância única. O terceiro exemplo da seção é a implementação em AspectJ de outro importante padrão de projetos, o *observador* [15]; esta implementação é também não-intrusiva e não exige que as classes participantes implementem as interfaces pré-definidas para o padrão.

5.2. Geração de Registros de Operações

Apresenta-se, nesta seção a implementação completa das classes e aspectos para geração de registros de operações. A implementação por meio de aspectos desta funcionalidade foi adaptada da solução apresentada em [16].

A classe responsável pela geração dos registros de operações é a classe Logger, definida a seguir. Esta classe possui métodos para registrar entradas de métodos, saídas de métodos por meio de retorno ou por meio de lançamento de exceções e registro de tratamento de exceções.

```
public class Logger {
    public static void logEntry(String signature) {
        System.out.println("Entering " + signature);
    }
    public static void logNormalExit(String signature) {
        System.out.println("Normal return of " + signature);
    }
    public static void logExitWithException(String signature, Exception exc) {
        System.out.println("Return of " + signature + " with exception " + exc);
    }
    public static void logExceptionHandling(Exception exc) {
        System.out.println("Handling exception " + exc);
    }
}
```

Observando-se os métodos que contêm chamadas dos métodos da classe Logger, observa-se quais pontos do programa são os candidatos a pontos de junção. Pelo exemplo da Seção 2, vê-se que há código de geração de registros espalhados no início e no final dos corpos de cada método. Esta constatação leva à definição do conjunto de junção loggableCalls. Além disso, é preciso registrar também as execuções dos tratadores de exceções, sendo assim necessário definir o conjunto de junção exceptionHandling.

Considere o aspecto definido a seguir:

```
public aspect LoggingAspect {

    pointcut publicMethods(): execution(public * *(..));
    pointcut logObjectCalls() : execution(* Logger.*(..));
    pointcut loggableCalls() : publicMethods() && ! logObjectCalls();
    pointcut exceptionHandling(Exception exc): handler(Exception+) && args(exc);

    before(): loggableCalls() {
        Logger.logEntry(thisJoinPoint.getSignature().toString());
    }

    after() returning: loggableCalls() {
        Logger.logNormalExit(thisJoinPoint.getSignature().toString());
    }

    after() throwing (Exception exc) : loggableCalls() {
        Logger.logExitWithException(thisJoinPoint.getSignature().toString(), exc);
    }

    before(Exception exc): exceptionHandling(exc) {
        Logger.logExceptionHandling(exc);
    }

}
```

O aspecto LoggingAspect define os conjuntos de junção para captura dos métodos cujas execuções devem ser registradas (loggableCalls) e para captura de tratamento de exceções (exceptionHandling). O conjunto de junção exceptionHandling é parametrizado para permitir a impressão do nome da exceção na geração do registro de operações.

As regras de junção do aspecto chamam os métodos da classe Logger com os parâmetros apropriados para fazer os registros da execução. A regra before no conjunto de junção loggableCalls indica que a chamada de Logger.logEntry deve ser incluída antes da execução de todos os métodos públicos do programa. Da mesma forma, as regras after para o mesmo conjunto de junção definem os métodos a serem chamados ao final de cada execução de método. O uso de returning e throwing permite que o retorno normal e o retorno com lançamento de exceção sejam tratados diferentemente.

Com o aspecto LoggingAspect é possível modularizar toda a política de geração de registros de operação do programa. No momento do *weaving*, as chamadas dos métodos da classe Logger são adicionadas nos pontos de junção identificados por loggableCalls, de modo que o comportamento do programa é idêntico ao obtido no exemplo da Seção 2.3.

5.3. Padrão Singleton

O padrão *singleton* tem por objetivo definir classes das quais se pode ter apenas uma instância. Exemplos de uso do padrão *singleton* são fábricas de sessões, que possuem, geralmente, uma única instância no programa. Em geral, as subclasses podem criar instâncias da classe *singleton*.

Nesta seção apresentam-se o padrão *singleton* [15] e as dificuldades em definir este padrão em linguagens orientadas por objetos, em especial quando as classes participantes não implementam as interfaces pré-definidas para o padrão. Mostra-se a partir daí como a definição de um pequeno conjunto de aspectos permite tornar qualquer classe

participante do padrão, independentemente do seu intuito inicial. A implementação por meio de aspectos para este padrão foi adaptada da solução apresentada em [19].

O exemplo ilustrado nesta seção é uma implementação do padrão *singleton* para uma classe Printer. Para garantir instância única deve-se definir o construtor da classe como privado ou protegido. O uso de protegido garante, ainda, que as subclasses podem criar instâncias da superclasse. Os objetos do programa que desejarem obter a instância única da classe devem utilizar o método estático instance, que retorna a instância única caso exista, ou cria uma nova instância se ainda nenhuma tiver sido criada. Utiliza-se um atributo estático, uniqueInstance para armazenar a instância única da classe.

```
// Implements a Printer as a singleton
public class PrinterSingleton {
    private static int objectsSoFar = 0; // Counts the instances of this class
    private static PrinterSingleton uniqueInstance = null; // Stores the printer unique instance
    private int id; // Each instance has a distinct id
    protected PrinterSingleton() { // Creates a new printer: note it is protected
        id = ++objectsSoFar;
    }
    public static PrinterSingleton instance() { // Factory method used to create a new instance
        if (uniqueInstance == null) {
            uniqueInstance = new PrinterSingleton();
        }
        return uniqueInstance;
    }
    public void print() { // Prints the instance of the printer
        System.out.println("\tMy ID is " + id);
    }
}
```

Os seguintes problemas de modularização podem ser vistos neste código: (i) o código de definição do padrão é intrusivo no código da classe; (ii) o código de definição do padrão não pode ser reaproveitado em outras classes que precisarem fazer controle de única instância; (iii) é possível violar o protocolo *singleton* dentro de pacotes ou subclasses em Java. Para evitar estes problemas, pode-se criar o aspecto abstrato SingletonProtocol, definido a seguir.

```
import java.util.Hashtable;

// Defines the general behavior of the Singleton design pattern.
public abstract aspect SingletonProtocol {
    private Hashtable singletons = new Hashtable(); // Stores the singletons instances
    public interface Singleton {} // Defines Singleton interface, implemented by singleton classes
    public pointcut protectionExclusions(); // Defines which points are allowed to create new instances
    // Protects the constructor of a singleton class
    Object around() : call((Singleton +).new (..)) && !protectionExclusions() {
        Class singleton = thisJoinPoint.getSignature().getDeclaringType();
        if (singletons.get(singleton) == null)
            singletons.put(singleton, proceed());
        return singletons.get(singleton);
    }
}
```

Este aspecto usa uma tabela, singletons, para conter as instâncias únicas de cada classe, e define uma interface Singleton, que é utilizada nos subaspectos de SingletonProtocol para garantir consistência de tipos com a implementação. O conjunto de junção protectionExclusions pode ser utilizado para definir pontos que têm a permissão de criar instâncias de uma classe *singleton*; por exemplo, subclasses de uma classe singleton podem criar novas instâncias da classe.

A garantia de unicidade de instância de uma classe é feita pela regra around, que é executada ao redor das chamadas de construtor das classes que implementam a interface Singleton. O código da regra tenta localizar na tabela singletons a instância única da classe cujo construtor está sendo chamado. Se nenhuma instância da classe for encontrada na tabela, então armazena-se uma nova instância criada por meio da chamada de proceed(). A instância na tabela é então retornada como resultado da chamada do construtor.

A seguir, apresentam-se a classe Printer, que não possui código de implementação do padrão, e uma subclasse PrinterSubclass, que não implementa o padrão *singleton*.

```
public class Printer {
    private static int objectsSoFar = 0; // Counts the instances of the class
    private int id; // Each instance has a distinct id
    public Printer() { // Constructor: note it is not protected
        id = ++objectsSoFar;
    }
    public void print() { // Prints the id of the instance
        System.out.println("\tMy ID is " + id);
    }
}

// This class has access to the singleton constructor
public class PrinterSubclass extends Printer {
    public PrinterSubclass() { // Creates an instance of this class
        super();
    }
}
```

Para garantir a existência de uma única instância da classe Printer, define-se o aspecto SingletonInstance, que estende SingletonProtocol. Define-se neste aspecto que a classe Printer implementa a interface Singleton. Além disso, redefine-se o conjunto de junção protectionExclusions, que captura todas as chamadas de construtor de PrinterSubclass; isto permite que esta classe e suas subclasses possam instanciar Printer.

```
// Implements a concrete instance of the Singleton pattern. It declares
// Printer to be Singleton and defines an exception to the constructor
// protection: PrinterSubclass (a subclass of the Singleton) can still
// access Printer's constructor.
public aspect SingletonInstance extends SingletonProtocol {

    // Gives Printer the Singleton's protection
    declare parents : Printer implements Singleton;

    // Allows Printer subclasses to access the constructor
    public pointcut protectionExclusions(): call((PrinterSubclass +).new (..));

}
```

A implementação do padrão *singleton* via aspectos é não-intrusiva, pois não é necessário alterar a interface da classe para que esta seja adequada ao padrão. Além disso, o código do padrão pode ser completamente reaproveitado em outras implementações; com efeito, para definir que outra classe qualquer possui instância única, basta acrescentar outra declaração de implementação da interface Singleton no aspecto SingletonInstance. Por meio da redefinição do conjunto de junção protectionExclusion, é possível ainda permitir que subclasses possuam múltiplas instâncias, sem a necessidade de definir o construtor da classe de única instância como protegido (**protected**). Desta maneira, não é necessário violar o princípio aberto-fechado[2] [28, 30] para permitir múltiplas instâncias das subclasses de uma classe de única instância.

5.4. Padrão Observador

A implementação do padrão observador por meio de linguagens orientadas por objetos puras, como a mostrada na Seção 2.4, apresenta problemas de modularização, devido a código de atualização dos observadores estar espalhado ao longo do código dos objetos observados. Nesta seção, apresenta-se uma implementação do padrão observador utilizando AspectJ. A implementação por meio de aspectos para este padrão foi adaptada da solução apresentada em [19].

O aspecto abstrato ObserverProtocol, definido a seguir, é responsável por mapear objetos em seus observadores. Este aspecto possui os seguintes elementos:

- as interfaces Subject e Observer representam, respectivamente, um objeto alvo de observação, doravante denominado *alvo*, e um observador; estas interfaces são utilizadas para consistências de tipos nos demais elementos do aspecto;
- o mapeamento perSubjectObservers é responsável por armazenar as listas de observadores de cada alvo;
- os métodos addObserver e removeObserver são responsáveis por adicionar e remover um observador da lista de observadores de um alvo, respectivamente;
- o método getObserver é responsável por retornar a lista de observadores de um alvo, armazenada na tabela perSubjectObservers; é responsável também por criar uma nova lista vazia e associá-la a um alvo, caso este ainda não esteja cadastrado na tabela;
- o conjunto de junção abstrato subjectChange deve ser redefinido nos subaspectos de ObserverProtocol; nos subaspectos, ele define os métodos responsáveis por alterar o estado interno dos objetos observados (ver aspecto TemperatureObservation, definido a seguir);
- o método abstrato updateObserver deve ser redefinido nos subaspectos, de modo que defina as ações de notificação dos observadores quando o estado interno do sujeito for alterado;
- a regra after definida para o conjunto de junção subjectChange é responsável por obter a lista de observadores de um alvo e chamar o método de notificação para cada observador; este código é incluído após cada ponto de junção que represente uma mudança de estado do objeto alvo.

A implementação do aspecto é a seguinte:

[2] A classe de única instância é aberta para extensões, mas fechada para modificações.

```
import java.util.List;
import java.util.LinkedList;
import java.util.Iterator;
import java.util.WeakHashMap;

public abstract aspect ObserverProtocol {

    public interface Subject {}
    public interface Observer {}

    private WeakHashMap perSubjectObservers;

    private List getObservers(Subject s) {
        if (perSubjectObservers == null) {
            perSubjectObservers = new WeakHashMap();
        }
        List observers = (List) perSubjectObservers.get(s);
        if (observers == null) {
            observers = new LinkedList();
            perSubjectObservers.put(s, observers);
        }
        return observers;
    }

    public void addObserver(Subject s, Observer o) {
        getObservers(s).add(o);
        updateObserver(s,o);
    }

    public void removeObserver(Subject s, Observer o) {
        getObservers(s).remove(o);
    }

    abstract public pointcut subjectChange(Subject s);

    abstract public void updateObserver(Subject s, Observer o);

    after(Subject s): subjectChange(s) {
        Iterator it = getObservers(s).iterator();
        while (it.hasNext()) {
            updateObserver(s, (Observer) it.next());
        }
    }

}
```

Para implementar o padrão observador utilizando aspectos, deve-se definir: (i) os objetos alvo de observação; (ii) os objetos observadores; (iii) as operações dos alvos que definem as alterações do estado interno; (iv) o algoritmo de atualização de cada observador; (v) os momentos em que a observação sobre um alvo inicia e termina.

Para definir que uma classe é observadora de outra, deve-se implementar um aspecto que estenda ObserverProtocol e que defina o conjunto de junção de modificação do estado interno e o método de notificação de observadores.

Considere, por exemplo, as classes Termometer e Temperature a seguir:

```
public abstract class Termometer {
    public abstract void printTemperature(double value);
}


public class Temperature {
    private double value;
    public Temperature(double initialValue) {
        value = initialValue;
    }
    public double getValue() {
        return value;
    }
    public void setValue(double value) {
        this.value = value;
    }
}
```

O aspecto TemperatureObservation, mostrado a seguir, estende o aspecto ObserverProtocol, definindo que Termometer implementa a interface Observer, Temperature implementa a interface Subject. Definem-se também que o conjunto de junção subjectChange representa todas as execuções do método Temperature.setValue, e que a notificação dos observadores é feita por meio da chamada do método de impressão da temperatura, Termometer.printTemperature.

```
public aspect TemperatureObservation extends ObserverProtocol {

    declare parents: Temperature implements Subject;
    declare parents: Termometer implements Observer;

    public pointcut subjectChange(Subject s)
        : target(s) && execution(void Temperature.setValue(double));

    public void updateObserver(Subject s, Observer o) {
        Temperature temperature = (Temperature) s;
        Termometer termometer = (Termometer) o;
        termometer.printTemperature(temperature.getValue());
    }

}
```

A seguir, definem-se duas subclasses Termometer: TermometerCelsius e TermometerFahrenheit.

```
public class TermometerCelsius extends Termometer {
    public void printTemperature(double value) {
        System.out.println("Celsius: " + value);
    }
}
```

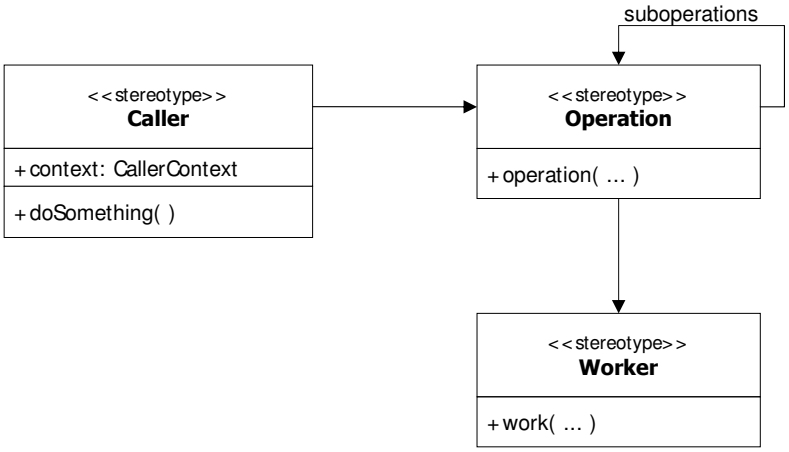

Figura 7: Representação em UML do padrão *wormhole*.

```
public class TermometerFahrenheit extends Termometer {
    public void printTemperature(double value) {
        System.out.println("Fahrenheit: " + value);
    }
}
```

Para associar uma instância de uma classe termômetro como observadora de uma instância de temperatura, utiliza-se o método addObserver do aspecto, como mostrado a seguir:

```
TermometerCelsius term1 = new TermometerCelsius();
TermometerFahrenheit term2 = new TermometerFahrenheit();
TemperatureObservation.aspectOf().addObserver(t,term1);
TemperatureObservation.aspectOf().addObserver(t,term2);
```

O método aspectOf é utilizado para referenciar a instância única de um aspecto.

As principais vantagens da abordagem são:

- a modularização completa do padrão: foram retirados das classes do objeto observado e do observador os códigos necessários para implementação do padrão observador;
- aumento do grau de reúso do padrão: pode-se definir de maneira não-intrusiva qualquer classe como observador ou alvo de observação, definindo-se somente um aspecto que implemente o respectivo protocolo de observação.

5.5. Padrão *Wormhole*

Considere o conjunto de classes definido na Figura 7, em que uma chamada típica de operação gera a execução de diversas operações secundárias, representadas na forma do grafo da Figura 8. A implementação por meio de aspectos para este padrão foi adaptada da solução apresentada em [16].

Se um objeto Worker precisar de informações a respeito do contexto do objeto Caller que iniciou a execução das operações, pode-se definir parâmetros nas chamadas de métodos dos objetos Operation para que este contexto seja propagado até os objetos Worker. Esta implementação gera intrusão, pois os objetos Operation podem não necessitar

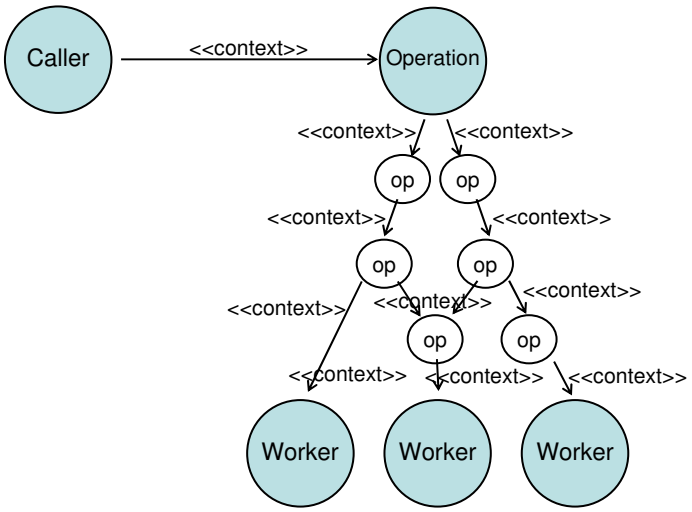

Figura 8: Implementação de transmissão de contexto via passagem de parâmetros (Extraída de [20])

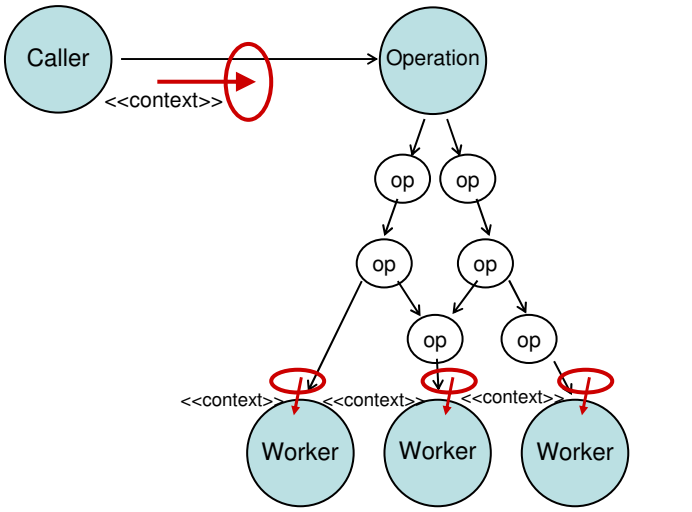

Figura 9: Representação gráfica do padrão *wormhole* (Extraída de [20]).

das informações de contexto. Neste caso, a existência destes parâmetros pode tornar complexas as interfaces das classes Operation, produzindo assim alto grau de acoplamento entre módulos. Outra solução possível é definir uma área de acesso global, onde o objeto Caller armazena seus dados para leitura pelos objetos Worker. Esta situação também gera alto grau de acoplamento entre os objetos Caller e Worker.

O padrão *wormhole* tem por objetivo definir um canal direto de comunicação entre os objetos Caller e Worker, tornando disponível para os objetos Worker as informações de contexto que ele precisar. Este padrão permite encapsular a comunicação entre estes objetos, propiciando uma implementação com grau de acoplamento menor que as implementações que passam o contexto como parâmetro ou o armazenam em uma área de acesso global.

Figura 9 esquematiza a implementação da passagem de contexto por meio do padrão *wormhole*. Nesta implementação, a informação de contexto é transmitida diretamente do objeto Caller para os objetos Worker, sem a necessidade de alterar a interface dos objetos Operation.

O aspecto WormholePattern define o padrão *wormhole*.

```
public aspect WormholePattern {

    pointcut invocations(Caller c):
        this(c) && call(void Operation.doOperation());

    pointcut workPoints(Worker w):
        target(w) && call(void Worker.doWork());

    pointcut perCallerWork(Caller c, Worker w):
        cflow(invocations(c)) && workPoints(w);

    before (Caller c, Worker w): perCallerWork(c, w) {
        w.setCallerContext(c.getContext());
    }

}
```

Neste aspecto, definem-se os seguintes conjuntos de junção:

- invocations: pontos de chamadas de métodos da classe Operation, ou suas subclasses, que ocorrerem em métodos cujo objeto receptor é da classe Caller;
- workPoints: pontos de chamadas do método doWork da classe Worker;
- perCallerWork: pontos de chamadas do método Worker.doWork() ou suas subclasses, que ocorrerem no fluxo de execução de uma invocação de operação a partir de métodos da classe Caller.

A regra before aplicada ao conjunto de junção perCallerWork propaga para o objeto Worker o contexto do objeto Caller cujo fluxo de execução gerou a chamada ao método Worker.doWork().

A implementação por meio de aspectos permite obter a propagação direta de informações de contexto, sem aumentar a interface da classe Operation ou aumentar o acoplamento entre as classes Caller e Worker por meio de valores globais. Além disso, todo o acoplamento entre as classes Caller e Worker é controlado pelo aspecto WormholePattern, que funciona como mediador. Quaisquer alterações no contexto geram alterações somente neste aspecto, de modo que as classes participantes não precisam ser alteradas.

É possível definir também um aspecto abstrato que represente o protocolo do padrão *wormhole* de maneira semelhante à feita nas implementações dos padrões *singleton* e observador. Com isso, este padrão também pode possuir uma implementação genérica e reutilizável.

5.6. Conclusão

Os exemplos apresentados nesta seção ilustram o poder de modularização da orientação por aspectos. Todos os exemplos geraram aspectos reutilizáveis e permitiram a produção de módulos com baixo acoplamento e alta coesão.

A geração de registros de operação por meio de aspectos é totalmente independente das classes do sistema. Ao se incluir uma nova classe no sistema, não é necessário que esta classe possua código para gerar registros, pois a característica transversal do aspecto faz com que as operações desta nova classe sejam alvo de registro.

As implementações dos padrões *singleton* e observador por meio de aspectos retirou toda a intrusão de código das classes participantes. Os códigos dos padrões são totalmente reutilizáveis, de modo que a criação de novas classes participantes pode ser feita pela simples definição de um subaspecto do aspecto que define o protocolo do padrão.

A implementação do padrão *wormhole* permite a criação de um canal de comunicação direta entre classes, evitando aumentar a interface de classes intermediárias no fluxo de execução, ou o aumento do acoplamento via uso de dados de acesso global. A solução mostrada pode ser também generalizada, o que a torna totalmente reutilizável.

6. Conclusões

O desenvolvimento de software orientado por aspectos foi proposto com o objetivo de permitir melhor modularização de requisitos transversais de sistemas. A linguagem AspectJ é uma linguagem que permite a implementação orientada por aspectos em Java.

A programação orientada por aspectos não é um substituto para a programação orientada por objetos. Os requisitos funcionais de um sistema continuarão a ser implementados por meio da POO. A orientação por aspectos simplesmente adiciona novos conceitos à POO, facilitando a implementação de requisitos transversais e retirando grande parte da atenção dada a tais requisitos nas fases iniciais de desenvolvimento. Com efeito, o desenvolvedor pode se concentrar na implementação dos módulos de requisitos funcionais do sistema, permitindo que a implementação da distribuição dos requisitos transversais aos módulos seja feita separadamente.

O desenvolvimento orientado por aspectos permite definir claramente as responsabilidades dos módulos individuais, uma vez que cada módulo é responsável unicamente por seu requisito principal. Além disso, o nível de modularização do sistema é melhorado, com baixo acoplamento e alta coesão modular. Com efeito, ao retirar código intruso dos módulos, é possível diminuir a interface de cada módulo e, ao mesmo tempo, aumentar o seu nível de coesão, por tratar somente um requisito. As conseqüências diretas destes fatos são a melhoria do processo de manutenção de sistemas e aumento no grau de reúso.

Além disso, a evolução de sistemas é facilitada, visto que, se novos aspectos forem criados para implementar novos requisitos transversais, as classes do programa podem permanecer inalteradas. Da mesma forma, ao adicionar novas classes ao programa, os aspectos existentes também são transversais a estas classes.

Como principal ponto negativo, tem-se que a orientação por aspectos pode quebrar o encapsulamento de módulos, ao permitir acesso a detalhes de sua implementação e a alteração da estrutura estática de tipos.

Uma limitação importante da orientação por aspectos está na integração de módulos, conforme destacado em [43]. Neste artigo discute-se o fato de que as implementações atuais das linguagens de programação orientada por aspectos não permitem que aspectos sejam instanciados, e isto pode dificultar a integração de certos módulos do sistema.

Após a leitura deste tutorial, os autores recomendam aos interessados em aprofundar seus estudos no assunto a visita ao site [1], que é um dos repositórios mais importantes de material sobre o assunto. Nele, há referências para projetos de pesquisa na área e uma relação bastante extensa das ferramentas existentes.

Programação por aspectos ainda não resolve definitivamente a importante questão da modularidade de sistemas computacionais complexos, mas, sem dúvida, trata-se de área de pesquisa promissora e representa um dos mais importantes avanços desde o advento da programação orientada por objetos.

Referências

[1] Aosd.net. *http://aosd.net/*. Último acesso: 23 de maio de 2004.

[2] AspectC++. *http://www.aspectc.org/*. Último acesso: 23 de maio de 2004.

[3] Aspectj.org. *http://www.aspectj.org*. Último acesso: 23 de maio de 2004.

[4] AspectWerks. http://aspectwerkz.codehaus.org/. Último acesso: 23 de maio de 2004.

[5] Colin Atkinson and Thomas Kühne. Aspect-Oriented Development with Stratified Frameworks. *IEEE Software*, 20:81–89, 2003.

[6] Johan Brichau, Maurice Glandrup, Siobhán Clarke, and Lodewijk Bergmans. Advanced Separation of Concerns. In *Proceedings of 15th European Conference on Object-Oriented Programming (ECOOP) Workshop Reader*. Springer-Verlag, 2001.

[7] Siobhán Clarke. Extending Standard UML with Model Composition Semantics. *Science of Computer Programming*, 44(1):71–100, 2002.

[8] Siobhán Clarke and Robert J. Walker. Towards a Standard Design Language for AOSD. In *Proceedings of the 1st International Conference on Aspect-Oriented Software Development*, 2002.

[9] Yvonne Coady and Gregor Kiczales. Back to the Future: A Retroactive Study of Aspect Evolution in Operating System Code. In *Proceedings of the 2nd Aspect-Oriented Software Development*. Springer-Verlag, 2003.

[10] Yvonne Coady, Gregor Kiczales, Mike Feeley, Norm Hutchinson, and Joon Suan Ong. Structuring Operating System Aspects: Using AOP to Improve OS Structure Modularity. *Commun. ACM*, 44(10):79–82, 2001.

[11] Yvonne Coady, Gregor Kiczales, Mike Feeley, and Greg Smolyn. Using AspectC to Improve the Modularity of Path-specific Customization in Operating System Code. In *Proceedings of the 8th European software engineering conference held jointly with 9th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 88–98. ACM Press, 2001.

[12] Constantinos A. Constantinides, Atef Bader, Tzilla H. Elrad, P. Netinant, and Mohamed E. Fayad. Designing an Aspect-oriented Framework in an Object-oriented Environment. *ACM Computing Surveys*, 32(1es):41, 2000.

[13] Edsger W. Dijkstra. *A Discipline of Programming*. Prentice Hall, 1976.

[14] Eclipse.org. *http://www.eclipse.org*. Último acesso: 23 de maio de 2004.

[15] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[16] Joseph D. Gradecki and Nicholas Lesieck. *Mastering AspectJ: Aspect-Oriented Programming in Java*. Wiley Publishing, Inc., 2003.

[17] Kris Gybels and Johan Brichau. Arranging language features for more robust pattern-based crosscuts. In *Proceedings of the 2nd international conference on Aspect-oriented software development*, pages 60–69. ACM Press, 2003.

[18] Jan Hannemann, Thomas Fritz, and Gail C. Murphy. Refactoring to Aspects: an Interactive Approach. In *Proceedings of the 2003 OOPSLA workshop on eclipse technology eXchange*, pages 74–78. ACM Press, 2003.

[19] Jan Hannemann and Gregor Kiczales. Design Pattern Implementation in Java and aspectJ. In *Proceedings of the 17th ACM SIGPLAN conference on Object-oriented*

programming, systems, languages, and applications, pages 161–173. ACM Press, 2002.

[20] Gregor Kiczales. Aspect-Oriented Programming – Keynote Talk at EclipseCon 2004. Disponível em *http://www.cs.ubc.ca/~gregor/*. Último acesso: 23 de maio de 2004.

[21] Gregor Kiczales and Erik Hilsdale. Aspect-oriented programming. In *Proceedings of the 8th European software engineering conference held jointly with 9th ACM SIGSOFT international symposium on Foundations of software engineering*, page 313. ACM Press, 2001.

[22] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold. An Overview of AspectJ. In *Proceedings of the 15th European Conference on Object-Oriented Programming*. Springer-Verlag, 2001.

[23] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Videira Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In *Proceedings of the 11th European Conference on Object-Oriented Programming*, page 220ff. Springer-Verlag, 1997.

[24] Ramnivas Laddad. *AspectJ in Action: Practical Aspect-Oriented Programming*. Manning, 2003.

[25] Donal Lafferty and Vinny Cahill. Language-independent Aspect-oriented Programming. In *Proceedings of the 18th ACM SIGPLAN conference on Object-oriented programing, systems, languages, and applications*, pages 1–12. ACM Press, 2003.

[26] Karl J. Lieberherr and Doug Orleans. Preventive program maintenance in Demeter/Java (research demonstration). In *International Conference on Software Engineering*, pages 604–605, Boston, MA, 1997. ACM Press.

[27] Log4J. *http://www.log4j.org*. Último acesso: 23 de maio de 2004.

[28] Robert C. Martin. The Open-Closed Principle. Disponível em *www.objectmentor.com/resources/articles/ocp.pdf*. Último acesso: 23 de maio de 2004.

[29] Sean McDirmid and Wilson Hsieh. Aspect-Oriented Programming in Jiazzi. In *Proceedings of the 2nd Aspect-Oriented Software Development*. Springer-Verlag, 2003.

[30] Bertrand Meyer. *Object-Oriented Software Construction*. Prentice Hall, 1997.

[31] Glenford J. Myers. *Reliable Software through Composite Design*. Petrocelli/Charter, 1975.

[32] Harold Ossher and Peri Tarr. Hyper/J: Multi-Dimensional Separation of Concerns for Java. In *Proceedings of the 22nd international conference on Software engineering*, pages 734–737. ACM Press, 2000.

[33] Harold Ossher and Peri Tarr. Using Multidimensional Separation of Concerns to (Re)Shape Evolving Software. *Communications of the ACM*, 44(10):43–50, 2001.

[34] David L. Parnas. On the Criteria to be Used in Decomposing Systems into Modules. *Commun. ACM*, 15(12):1053–1058, 1972.

[35] Rajeev R. Raje, Ming Zhong, and Tongyu Wang. Case Study: a Distributed Concurrent System with AspectJ. *SIGAPP Appl. Comput. Rev.*, 9(2):17–23, 2001.

[36] Awais Rashid and Ruzanna Chitchyan. Persistence as an Aspect. In *Proceedings of the 2nd Aspect-Oriented Software Development*, pages 120–129. Springer-Verlag, 2003.

[37] James Rumbaugh, Ivar Jacobson, and Grady Booch. *The Unified Modeling Language Reference Manual*. Addison-Wesley Longman, 1999.

[38] James Rumbaugh, Ivar Jacobson, and Grady Booch. *The Unified Modeling Language User Guide*. Addison-Wesley Longman, 1999.

[39] Robert J. Walker Siobhán Clarke. Composition Patterns: An Approach to Designing Reusable Aspects. In *Proceedings of the 23rd International Conference on Software Engineering (ICSE)*, 2001.

[40] Sérgio Soares, Eduardo Laureano, and Paulo Borba. Implementing Distribution and Persistence Aspects with AspectJ. In *Proceedings of the 17th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 174–190. ACM Press, 2002.

[41] Olaf Spinczyk, Andreas Gal, and Wolfgang Schröder-Preikschat. AspectC++: An Aspect-oriented Extension to the C++ Programming Language. In *Proceedings of the Fortieth International Conference on Tools Pacific*, pages 53–60. Australian Computer Society, Inc., 2002.

[42] Dominik Stein, Stefan Hanenberg, and Rainer Unland. A UML-based Aspect-oriented Design Notation for AspectJ. In *Proceedings of the 1st international conference on Aspect-oriented software development*, pages 106–112. ACM Press, 2002.

[43] Kevin Sullivan, Lin Gu, and Yuanfang Cai. Non-Modularity in Aspect-Oriented Languages: Integration as a Crosscutting Concern for AspectJ. In *Proceedings of the 1st international conference on Aspect-oriented software development*, pages 19–37. ACM Press, 2002.

[44] Arndt von Staa. *Programação Modular*. Editora Campus, 2000.