2
Desenvolvimento de Software Orientado a Aspectos

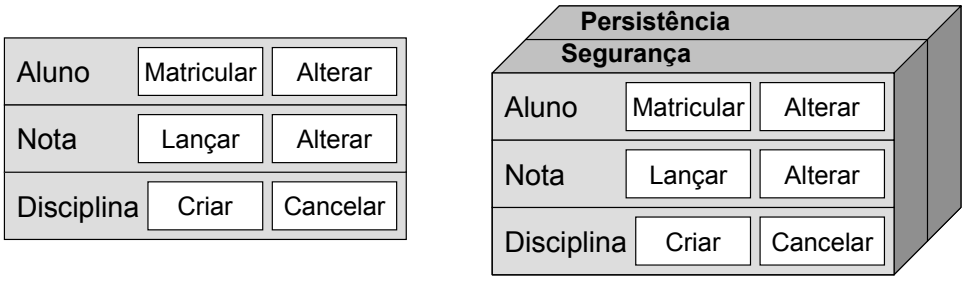
A divisão em partes é um importante instrumento para se reduzir a complexidade de sistemas de software. É muito difícil para o ser humano compreender um sistema de grande porte se este for monolítico, sem fronteiras claras que definam suas funcionalidades. Neste sentido, Edsger Dijkstra [18] em 1974 foi o primeiro a falar em *separação de interesses*[1] para denotar o princípio que guia a divisão em partes. Todo sistema de software lida com diferentes interesses, sejam eles dados, operações ou outros requisitos do sistema. O ideal seria que a parte do programa dedicada a satisfazer a um determinado interesse estivesse concentrada em uma única localidade física, separada de outros interesses. Desta forma, cada interesse pode ser estudado e compreendido com mais facilidade.

No desenvolvimento estruturado, a separação de interesses (SI) ocorre através das diferentes funcionalidades oferecidas pelo software. Cada função é implementada em um único módulo, ou procedimento. Neste paradigma de desenvolvimento, apesar dos interesses relativos a funcionalidades ficarem separados, interesses relativos a dados ficam distribuídos em diversos módulos. Para sanar esta deficiência, o desenvolvimento orientado a objetos (OO) apresenta uma forma mais eficiente para SI. Neste paradigma a separação ocorre em duas dimensões, ou seja, em termos dos dados e das funções que utilizam cada tipo de dados. A Figura 1 (a) ilustra como ocorre SI no desenvolvimento OO.

Na verdade, interesses são modularizados por meio de diferentes abstrações providas pelas linguagens e paradigmas de programação. As abstrações básicas do desenvolvimento OO são as classes, os métodos e os atributos. No entanto, essas abstrações podem não ser suficientes para separar certos tipos de interesses que inerentemente atravessam componentes responsáveis pela modularização de outros interesses. Alguns exemplos tradicionais de tais interesses, chamados

[1] Do termo em inglês *separation of concerns*.

interesses transversais[2], são: persistência, segurança, auditoria e tratamento de exceções. Sem meios apropriados para sua separação em um sistema OO, os interesses transversais tendem a ficar *espalhados* e *entrelaçados*[3] a outros interesses. Um interesse é dito espalhado quando este afeta vários componentes do sistema e entrelaçado quando se mistura com outros interesses dentro de um módulo. Estas duas características dos interesses são indesejáveis porque causam maior dificuldade de entendimento, evolução e reuso dos artefatos de software.



(a) Separação em dados e funções

(b) Separação em dados, funções e aspectos

Figura 1 – Separação de interesses (a) bidimensional e (b) tridimensional

O Desenvolvimento de Software Orientado a Aspectos (DSOA) [42] [43] [66] tem emergido como um novo paradigma, complementar aos existentes, cujo objetivo é prover suporte à separação adequada de interesses transversais em módulos fisicamente separados. Pensando em termos abstratos, a orientação a aspectos introduz uma terceira dimensão de decomposição representada na Figura 1 (b). Além de compor o sistema em dados e funções, o sistema é decomposto de acordo com os interesses transversais em módulos denominado aspectos. Aspectos são unidades modulares de interesses transversais que se associam aos outros módulos do sistema. A Figura 2 (a) apresenta graficamente um interesse transversal espalhado e entrelaçado pelos módulos de um sistema orientado a objetos, enquanto o diagrama da Figura 2 (b) representa a modularização deste interesse em um aspecto.

[2] Do termo em inglês *crosscutting concerns*.

[3] Dos termos em inglês *scattering* e *tangling*.

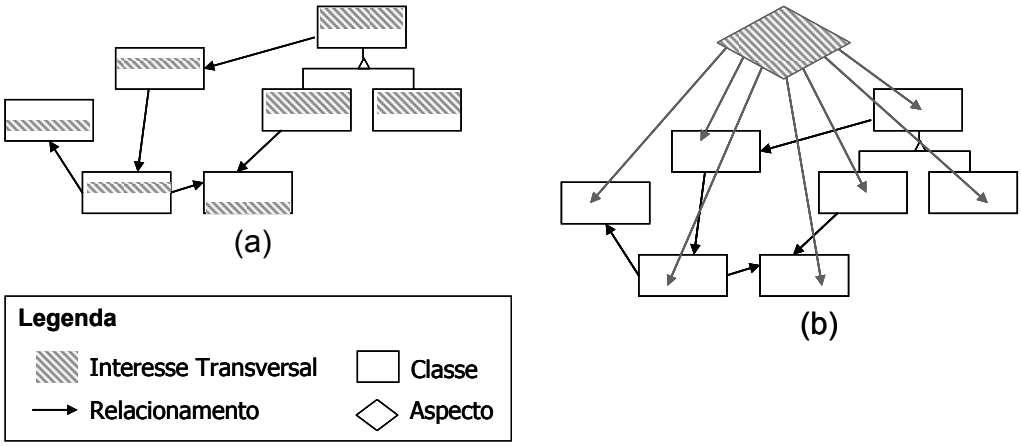


Figura 2 – Interesse transversal em sistemas (a) OO e (b) OA

Além desta nova unidade de modularização, as linguagens para o DSOA provêm mecanismos para compor classes e aspectos. Neste sentido, o paradigma de aspectos envolve duas etapas principais: (i) decomposição do sistema em partes não entrelaçadas; (ii) juntar essas partes novamente de forma significativa para se obter o sistema desejado. O processo de juntar as partes é chamado *combinação*[4]. Outro conceito importante neste paradigma são os *pontos de junção*[5] que especificam pontos específicos nos quais os módulos podem ser combinados. Um ponto de junção é um ponto bem definido na definição estática ou na execução do programa. Outros conceitos importantes deste paradigma são abordados na seção a seguir. Neste momento, torna-se importante ressaltar que as traduções utilizadas neste documento para os termos técnicos de DSOA foram especificadas em um workshop brasileiro da área e encontram-se disponíveis em [68].

2.1. AspectJ: uma Extensão de Java para Orientação a Aspectos

No contexto de DSOA, AspectJ [45] [66] é a linguagem de propósito geral mais conhecida e usada atualmente. AspectJ estende Java [17] com novas abstrações e mecanismos para combinar elementos destas duas linguagens. A combinação de elementos Java e AspectJ é feita pelo compilador de AspectJ (*ajc*). O compilador *ajc* transforma um programa escrito em AspectJ em *bytecode* Java, que pode ser executado por qualquer máquina virtual Java (JVM). Além deste

[4] Do termo em inglês *weaving*.

[5] Do termo em inglês *join point*.

compilador, AspectJ acrescenta novos conceitos e construções à linguagem Java, tais como: aspectos, *conjuntos de pontos de junção*[6], *adendos*[7] e *declarações intertipos*[8]. Estas construções são apresentadas a seguir.

Aspecto

O elemento principal de AspectJ é o aspecto. Cada aspecto assemelha-se a uma classe Java e pode ter tudo que uma classe tem, ou seja, definições de variáveis, métodos, etc. Além desses elementos, os aspectos podem conter outros elementos específicos de AspectJ, como conjunto de pontos de junção, adendos e declarações intertipos. Um aspecto pode afetar a estrutura estática ou dinâmica de uma ou mais classes e objetos de diferentes maneiras. A estrutura dinâmica é alterada pela especificação de conjuntos de pontos de junção e adendos, enquanto a estrutura estática é alterada através de declarações intertipos.

Conjunto de Pontos de Junção

Como mencionado anteriormente neste capítulo, um ponto de junção é qualquer ponto identificável e bem definido durante a execução de um programa. Diversos tipos de pontos de junção são permitidos na linguagem AspectJ, tais como: entradas e saídas de métodos, tratamento de exceções, acessos a variáveis de objetos, construtores, entre outros. Um conjunto de pontos de junção, ou conjunto de junção, é uma construção sintática de AspectJ para agrupar pontos de junção. Esta estrutura não tem nenhum comportamento, ela apenas denota onde e quando os aspectos terão efeito no programa. Para especificar um conjunto de junção podem ser usadas expressões regulares, permitindo grande flexibilidade em sua definição. Sua sintaxe básica é ilustrada no exemplo a seguir:

```
public pointcut printObject() : call (String *.toString());
```

O exemplo acima mostra os quatro elementos principais de um conjunto de pontos de junção. O primeiro é a declaração de restrição de acesso, nesse caso público (`public`), mas conjuntos de junção também podem ser privados

[6] Do termo em inglês *pointcut*.

[7] Do termo em inglês *advice*.

[8] Da expressão em inglês *inter-type declaration*.

(`private`) ou protegidos (`protected`). Em seguida, a palavra reservada de AspectJ `pointcut` denota que estamos declarando um conjunto de pontos de junção. O terceiro elemento desta estrutura é um identificador (`printObject`, em nosso exemplo), que pode ou não ser acompanhado por parâmetros. Finalmente, depois dos dois (`:`), vem o tipo e uma expressão regular que representa os pontos de junção. Nesse caso, os pontos de junção são todas as chamadas a métodos `toString()` do sistema. É importante ressaltar que este documento não tem a intenção de oferecer um completo entendimento desta e outras construções sintáticas de AspectJ, mas apenas familiarizar o leitor com a sintaxe da linguagem. Para maiores detalhes devem-se consultar livros [45] e manuais [66] de programação AspectJ.

Adendo

Adendo, ou comportamento transversal, é uma estrutura que denota o que um aspecto deve fazer, ou seja, qual o comportamento do aspecto. Em termos mais formais, esta estrutura designa a semântica comportamental do aspecto. Adendos são comportamentos e estão associados a conjuntos de junção. Eles especificam não só o que será feito, na forma de uma seqüência de operações Java, mas também o momento em que serão feitas estas operações. Há três tipos de adendos:

1. *Anterior*: executa antes do ponto de junção;
2. *Posterior*: executa depois do ponto de junção;
3. *De contorno*: executa "em volta" do ponto de junção.

O adendo do tipo "anterior" é o mais simples. Ele simplesmente executa antes do ponto de junção. Um detalhe a observar é que se este adendo levantar exceção, o seu respectivo ponto de junção não será executado. A seguir um exemplo de adendo do tipo "anterior":

```
before() : printObject() {
   System.out.println("Antes de imprimir!");
}
```

O adendo do tipo "posterior" executa após o ponto de junção. Existem três variações desse adendo que dependem de como terminou a execução do ponto de junção: se terminou normalmente, com uma exceção, ou de qualquer forma. Os exemplos a seguir ilustram estas variações:

```
after() : printObject() {
   // executa independente de como retornar
   // o ponto de junção
}

after() returning : printObject() {
   // executa se o ponto de junção terminar normalmente
}

after() throwing : printObject() {
   // executa se o ponto de junção sair com uma exceção
}
```

O adendo do tipo "de contorno" é utilizado para substituir a execução do ponto de junção pela execução do adendo. No entanto, o ponto de junção pode ser executado de dentro do corpo do adendo usando-se o comando `proceed()`. Todo adendo de contorno deve declarar um tipo a ser retornado. O exemplo a seguir apresenta este tipo de estrutura:

```
String around() : printObject() {
  String s = proceed();
  return s.toLowerCase();
}
```

Declaração Intertipo

Além de modificar o comportamento das classes de um sistema pelo uso de conjuntos de junção e adendos, um aspecto pode alterar a estrutura estática dos módulos do sistema. Isto é feito pelo mecanismo chamado de declaração intertipo que é uma forma do aspecto introduzir mudanças em classes, interfaces e outros aspectos. Exemplos de elementos que podem ser adicionados são variáveis, métodos e construtores. Um aspecto pode ainda acrescentar relacionamentos de herança a classes existentes de duas formas: (i) fazendo com que classes implementem (uma ou mais) interfaces; (ii) fazendo com que classes estendam outras classes. O exemplo de código abaixo mostra como a declaração intertipo pode ser utilizada para adicionar, respectivamente, uma variável, um método e uma superclasse a classe `Aluno`.

```
private int Aluno.senha;

public boolean Aluno.verificar(int s) {
   return (senha == s);
}

declare parents : Aluno extends Pessoa;
```

2.2. Alguns Exemplos de Utilização de Aspectos

Esta seção apresenta três exemplos de aspectos [66] que ilustram o uso da tecnologia de DSOA. O primeiro exemplo consiste em um aspecto para monitorar as falhas de um servidor. O segundo tem como objetivo permitir que objetos possam ser clonados. E o último exemplo desta seção utiliza um aspecto para registrar tempos de conexão entre clientes e servidor. Todos os três exemplos são demonstrados em trechos de código AspectJ e possuem dois objetivos principais: revisar as estruturas sintáticas de AspectJ em contextos de aplicação e motivar o uso do DSOA em diferentes problemas que envolvem interesses transversais.

```
01  public aspect FaultHandler {
02
03    private boolean Server.disabled = false;
04
05    private void reportFault() {
06      System.out.println("Failure! Please fix it.");
07    }
08
09    public static void fixServer(Server s) {
10      s.disabled = false;
11    }
12
13    pointcut services(Server s):
        target(s) && call(public * *(..));
14
15    before(Server s): services(s) {
16      if (s.disabled) throw new DisabledException();
17    }
18
19    after(Server s) throwing (FaultException e):
        services(s) {
20      s.disabled = true;
21      reportFault();
22    }
23  }
```

Figura 3 – Exemplo de aspecto para tratamento de falhas

O aspecto `FaultHandler` [66] apresentado na Figura 3 possui três das principais construções da linguagem AspectJ: declaração intertipo, conjunto de pontos de junção e adendos. A declaração intertipo (linha 03) é utilizada para introduzir o atributo `disabled` na classe `Server`. Este atributo informa se o servidor se encontra disponível ou não. Todas as chamadas a métodos do servidor são interceptadas pelo conjunto de junção `services` (linha 13). A idéia é que comportamentos de falha podem ser disparados em chamadas de métodos

públicos da classe `Server` e, portanto, estas chamadas são eventos interessantes para o aspecto e devem ser capturadas pelo conjunto de junção. A ocorrência destes eventos causa a execução do adendo anterior (linhas 15-17) e do adendo posterior (linhas 19-22). Além destas três construções de AspectJ, o aspecto `FaultHandler` também possui os métodos `reportFault()` (linhas 05-07) e `fixServer()` (linhas 09-11).

```
public interface Shape { }

public class Point implements Shape {
   ...
}

public class Line implements Shape {
   ...
}

public aspect CloneableShape {

   declare parents: Shape extends Cloneable;

   public Object Shape.clone()
        throws CloneNotSupportedException {
      makePolar();
      return super.clone();
   }
}
```

Figura 4 – Exemplo de aspecto para associação de papel a classes

A Figura 4 apresenta a interface `Shape`, duas classes que implementam esta interface (`Point` e `Line`) e o aspecto `CloneableShape` [66]. Este aspecto é responsável por associar o papel de clonável às classes fazendo uso de declarações intertipo. Lembrando que em Java todo objeto herda o método `clone()` da classe `Object`, entretanto, ele só é clonável se implementar a interface `Cloneable`. Além disso, frequentemente o método `clone()` é reescrito nas classes para adicionar algum comportamento específico, como o método `makePolar()` do exemplo. Desta forma, o aspecto `CloneableShape` associa a interface `Cloneable` e garante a implementação do método `clone` como requerido pelas classes sem replicação de código.

O último exemplo desta seção, apresentado na Figura 5, é um aspecto para monitorar o tempo de total de conexão de cada cliente com o servidor [66]. Neste exemplo, o aspecto `Timing` utiliza uma declaração intertipo para introduzir o atributo `totalConnectTime` (linha 3) na classe `Customer`. Este atributo armazena

o tempo total de conexão do cliente. O aspecto também declara em cada objeto do tipo `Connection` (linha 9) um `Timer` para registrar este tempo de conexão. Além dos mecanismos de declaração intertipo, o aspecto `Timing` também possui conjuntos de junção (linha 19) e adendos (linhas 15-17 e 21-25). Os adendos garantem que a contagem de tempo é iniciada assim que a conexão é completada e pára quando a conexão termina. É interessante notar que o adendo das linhas 15-17 utiliza um conjunto de pontos de junção anônimo enquanto o adendo das linhas 21-25 utiliza um nomeado `endTiming`.

```
01  public aspect Timing {
02
03     public long Customer.totalConnectTime = 0;
04
05     public long getTotalConnectTime(Customer cust) {
06        return cust.totalConnectTime;
07     }
08
09     private Timer Connection.timer = new Timer();
10
11     public Timer getTimer(Connection conn) {
12        return conn.timer;
13     }
14
15     after (Connection c):
        target(c) && call(void Connection.complete()) {
16        getTimer(c).start();
17     }
18
19     pointcut endTiming(Connection c):
        target(c) && call(void Connection.drop());
20
21     after(Connection c): endTiming(c) {
22        getTimer(c).stop();
23        c.getCaller().totalConnectTime +=
        getTimer(c).getTime();
24        c.getReceiver().totalConnectTime +=
        getTimer(c).getTime();
25     }
26  }
```

Figura 5 – Exemplo de aspecto para registrar tempo de conexão com servidor