2
Desenvolvimento de Software Orientado a Aspectos

Separação de *concerns* é um princípio bem estabelecido da engenharia de software que diz que, para se dominar a complexidade do desenvolvimento de software, deve-se separar os assuntos de forma a se concentrar em apenas um por vez [13]. A primeira pessoa a usar o termo "separação de *concerns*" foi Dijkstra em seu livro "*A Discipline of Programming*" [14], onde ele diz que a principal característica do pensamento inteligente é ser capaz de estudar em profundidade um aspecto de determinado problema isoladamente, para o bem de sua própria consistência, mas sabendo o tempo todo que outros aspectos estão esperando sua vez, pois a mente humana é tão limitada que não consegue lidar com todos eles simultaneamente sem ficar confusa.

Um *concern* é alguma parte do problema que se deseja tratar como uma unidade conceitual única [3]. No desenvolvimento de software, *concerns* são modularizados por meio de diferentes abstrações providas por linguagens, métodos e ferramentas. As abstrações básicas do desenvolvimento de software orientado a objetos (DSOO) são classes, objetos, métodos e atributos. No entanto, essas abstrações podem não ser suficientes para separar alguns *concerns* especiais existentes em muitos sistemas complexos. Esses *concerns* são chamados de *crosscutting concerns*, já que, inerentemente, atravessam os componentes responsáveis pela modularização de outros *concerns*. Sem meios apropriados para sua separação e modularização, os *crosscutting concerns* tendem a ficar espalhados e misturados a outros *concerns*. As conseqüências naturais são a redução da facilidade de entendimento, o aumento da dificuldade de evolução e a diminuição da reusabilidade dos artefatos de software.

Desenvolvimento de software orientado a aspectos (DSOA) [1, 2, 3] foi proposto como uma técnica para melhorar a separação de *concerns* na construção de software e apoiar ao aumento da reusabilidade e facilidade de evolução. O DSOA apóia a modularização de *crosscutting concerns* por meio de abstrações que possibilitam a sua separação e composição para produzir o sistema. O DSOA

usa aspectos como uma nova abstração e provê um novo mecanismo para compor aspectos e classes. Vale a pena ressaltar que a orientação a aspectos não é uma abordagem restrita apenas ao paradigma orientado a objetos [2], no entanto este trabalho se preocupou com as técnicas orientadas a aspectos dentro do contexto do desenvolvimento de software orientado a objetos.

AspectJ [15, 16] é a linguagem orientada a aspectos de propósito geral mais conhecida atualmente. Ela é uma extensão orientada a aspectos da linguagem de programação Java [17]. AspectJ acrescenta alguns novos conceitos e construções associadas. Esses conceitos e construções são chamados de aspecto, *join point*, *pointcut*, *advice* e *inter-type declaration*[3].

Aspecto é a unidade de modularidade para *crosscutting concerns* em AspectJ. Cada aspecto encapsula uma funcionalidade que atravessa outras classes do programa. Um aspecto é definido por uma declaração de aspecto, que tem uma forma similar a declaração de classe em Java. Similarmente a uma classe, um aspecto contém métodos e atributos e pode também ser especializado por sub-aspectos. Um aspecto é combinado com as classes que ele afeta de acordo com especificações definidas dentro do aspecto. Além disso, um aspecto pode introduzir métodos, atributos, declarações de implementação de interface e declaração de extensão de classe usando a construção chamada de *inter-type declaration*. Membros introduzidos podem ser visíveis a todas as classes e aspectos (*inter-type declaration* pública) ou apenas internamente ao aspecto (*inter-type declaration* privada).

Essencial para o processo de composição entre aspectos e classes é o conceito de *join points*, que são elementos que especificam como as classes e os aspectos estão relacionados. Um *join point* é um ponto bem definido da execução de um programa, como a chamada de um método, o acesso a um atributo, a inicialização de um objeto, o levantamento de uma exceção, etc. *Pointcuts* descrevem e dão nome a conjuntos de *join points*. *Pointcuts* podem ser combinados, e novos *pointcuts* podem ser definidos de acordo com essas combinações. AspectJ provê várias primitivas de *pointcut* que podem ser

[3] A comunidade brasileira de DSOA ainda não encontrou boas traduções para os termos *join point*, *pointcut*, *advice* e *inter-type declaration*, por isso eles são usados em inglês nesta dissertação.

combinadas por meio de operadores lógicos para construir descrições completas de *pointcuts* de interesse. O manual de programação de AspectJ [16] possui a lista completa das primitivas disponíveis.

Um aspecto pode especificar *advices* que são usados para definir algum código que deve ser executado quando um *pointcut* é atingido. *Advice* é um mecanismo parecido com um método que consiste de código que é executado antes (*advice* do tipo *before*), depois (*advice* do tipo *after*) ou no lugar (*advice* do tipo *around*) de um *pointcut*. Um *advice* do tipo *around* é executado no lugar do *pointcut* indicado, permitindo assim que um método da classe seja substituído.

Um programa em AspectJ pode ser dividido em duas partes: a parte do código base que inclui classes, interfaces e outras construções da linguagem para implementar a funcionalidade básica do programa, e a parte do código de aspecto que inclui os aspectos para implementar os *crosscutting concerns* do programa. Em geral, a comunidade de DSOA adota uma clara distinção entre componentes (componentes do código base) e aspectos, no entanto, neste trabalho, o termo **componente** é usado para denotar tanto classes e interfaces quanto aspectos. O mecanismo responsável por fazer a composição entre o código base e o código de aspectos é chamado de *weaver*.

A Figura 1 mostra um exemplo de definição de um aspecto em AspectJ tirado de [16]. O aspecto *FaultHandler* consiste de uma *inter-type declaration* que introduz um atributo na classe *Server* (linha 03), dois métodos (linhas 05-07 e 08-10), a definição de um *pointcut* (linha 12) e dois *advices* (linhas 14-16 e 17-20). O *pointcut*, chamado de *services*, define como *join points* os pontos da execução do programa onde objetos da classe *Server* têm qualquer um de seus métodos públicos chamados.

A Figura 2 apresenta um exemplo didático para mostrar a diferença entre a implementação em Java e AspectJ de um mesmo programa. Esse exemplo mostra o código de um programa simples de elementos gráficos. A implementação em Java (lado esquerdo da figura) é formada pelas classes *Point*, *Line* e *Display* (essa última não é apresentada na figura). A implementação em AspectJ (lado direito da figura) é constituída das mesmas classes, mais o aspecto *DisplayUpdating*. Esse exemplo mostra que toda vez, após a chamada dos métodos *setX* e *setY* da classe *Point* e dos métodos *setP1* e *setP2* da classe *Line*, é preciso que o método *update* da classe *Display* seja chamado. Na implementação em Java, a chamada ao

método *update* da classe *Display* fica espalhada pelos quatro métodos, pois é feita explicitamente no final de cada um deles (linhas 9, 13, 25 e 29). Na solução em AspectJ, essa chamada fica localizada apenas no aspecto *DisplayUpdating* (linha 37), e é introduzida nas classes pelo pointcut *move* (linhas 30-34).

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

Figura 1 – Exemplo da definição de um aspecto em AspectJ

```
01 class Point {
02  private int x = 0, y = 0;
03
04  int getX() { return x; }
05  int getY() { return y; }
06
07  void setX(int x) {
08    this.x = x;
09    Display.update();
10  }
11  void setY(int y) {
12    this.y = y;
13    Display.update();
14  }
15 }
16
17 class Line {
18  private Point p1, p2;
19
20  Point getP1() { return p1; }
21  Point getP2() { return p2; }
22
23  void setP1(Point p1) {
24    this.p1 = p1;
25    Display.update();
26  }
27  void setP2(Point p2) {
28    this.p2 = p2;
29    Display.update();
30  }
31 }
```

```
01 class Point {
02  private int x = 0, y = 0;
03
04  int getX() { return x; }
05  int getY() { return y; }
06
07  void setX(int x) {
08    this.x = x;
09  }
10  void setY(int y) {
11    this.y = y;
12  }
13 }
14
15 class Line {
16  private Point p1, p2;
17
18  Point getP1() { return p1; }
19  Point getP2() { return p2; }
20
21  void setP1(Point p1) {
22    this.p1 = p1;
23  }
24  void setP2(Point p2) {
25    this.p2 = p2;
26  }
27 }
28
29 aspect DisplayUpdating {
30  pointcut move():
31    call(void Line.setP1(Point))||
32    call(void Line.setP2(Point))||
33    call(void Point.setX(int))  ||
34    call(void Point.setY(int));
35
36  after() returning: move() {
37    Display.update();
38  }
39 }
```

Figura 2 – O mesmo programa em Java (lado esquerdo) e AspectJ (lado direito).

Nesta dissertação, as métricas do framework de avaliação são apresentadas no contexto de AspectJ. A definição das métricas, no entanto, trata dos conceitos básicos de desenvolvimento de software orientado a aspectos e pode, portanto, ser adaptada para outras técnicas orientadas a aspectos.