4
Desenvolvimento de Software Orientado a Aspectos

Apesar de ser a tecnologia atualmente dominante no desenvolvimento de software, a orientação a objetos possui algumas limitações nas tarefas de projetar e implementar os interesses que naturalmente *afetam* múltiplas classes e/ou métodos que modularizam outros interesses de um sistema (Kiczales et al., 1997; Tarr et al., 1999). Tais interesses são denominados *interesses transversais*. Exemplos comuns de interesses transversais na literatura são: rastreamento (AspectJ Team), auditoria (AspectJ Team), persistência (Soares et al., 2002), distribuição (Soares et al., 2002) e tratamento de erros (Lippert & Lopes, 2000; Garcia et al., 2001; Garcia & Rubira, 2001).

Existem várias abordagens para a separação avançada de interesses, incluindo programação orientada a sujeitos (Harrison & Ossher, 1993), orientação a aspectos (Kiczales et al., 1997) e separação multi-dimensional de interesses (Ossher & Tarr, 2001). Essas abordagens contribuem para melhorar o desenvolvimento de software orientado a objetos, uma vez que proporcionam a separação de interesses em outras dimensões, além de classes e objetos. A idéia é que, embora as abstrações e os mecanismos de linguagens orientadas a objetos sejam muito úteis, eles são inerentemente incapazes de modularizar todos os interesses em sistemas.

A orientação a aspectos (Kiczales et al., 1997) é a abordagem utilizada neste trabalho para a separação avançada de interesses. *Aspectos* são usados como abstrações capazes de capturar de forma modular os interesses transversais nos sistemas. A linguagem *AspectJ* (Kiczales et al., 2001) é uma extensão orientada a aspectos da linguagem de programação Java. *AspectJ* é considerada a linguagem orientada a aspectos mais madura, uma vez que possui uma grande comunidade de usuários, evoluiu muito nos últimos anos e já foi aplicada em projetos da indústria (AspectJ Team).

Muitos estudos encontrados na literatura exploram a programação orientada a aspectos (POA) em *AspectJ* (Lippert & Lopes, 2000; Soares et al., 2002; Rashid

& Guitchyan, 2003; Garcia et al., 2004). Além disso, a POA está evoluindo da etapa de implementação em direção às fases de arquitetura e projeto de sistemas, a fim de fornecer um caminho completo de desenvolvimento ao longo do ciclo de vida do software. O Desenvolvimento de Software Orientado a Aspectos (DSOA) é uma área de pesquisa emergente cujo objetivo é promover a separação avançada de interesses em todas as etapas do desenvolvimento de software.

Como afirmado na seção introdutória deste trabalho, a mobilidade em SMAs é considerada um interesse transversal (Ubayashi & Tamai, 2001; Garcia et al., 2004). A intrusão da mobilidade em outros interesses tem forte impacto na estrutura básica do agente, bem como no projeto de outros interesses como colaboração e interação entre agentes. A solução proposta neste trabalho para o problema de modularização de mobilidade envolve o DSOA.

Neste capítulo são introduzidos os conceitos de DSOA. São apresentados conceitos de POA e, em seguida, detalhes sobre os níveis de projeto e arquitetura do processo de desenvolvimento de software seguindo uma abordagem OA. Antes, porém, é introduzido um exemplo clássico de interesse transversal na literatura de aspectos: um sistema com tratamento de erros (AspectJ Team; Soares et al., 2002). O objetivo inicial da descrição deste exemplo é apontar problemas de espalhamento e entrelaçamento em seu código orientado a objetos. Posteriormente, o sistema com tratamento de erros é utilizado para a introdução dos conceitos envolvidos no desenvolvimento de software OA.

4.1. Exemplo de Interesse Transversal: Tratamento de Erros

A Figura 15 apresenta um exemplo de interesse transversal encontrado na literatura de aspectos (AspectJ Team; Soares et al., 2002). Trata-se do tratamento de erros, um interesse normalmente espalhado e entrelaçado com outros interesses em um sistema orientado a objetos. A classe `Server` é uma classe *Fachada* (Gamma et al., 1995), isto é, o único ponto do sistema que permite acesso aos seus serviços públicos. Cada método da classe `Server` implementa um serviço público específico. As atividades normais não deveriam estar entrelaçadas com as atividades excepcionais ou com o interesse de tratamento de erros na implementação dos serviços. A separação entre o código normal e o código

excepcional é um requisito importante em aplicações robustas (Cristian, 1989; Lang & Stewart, 1998; Papurt, 1998; Garcia & Rubira, 2001; Garcia, et al., 2002), especialmente em sistemas tolerantes a falhas (Cristian, 1982; Cristian, 1995). Nestes sistemas, o código dedicado à detecção e ao tratamento de erros é normalmente longo e complexo. Até dois terços de um programa são usados para o tratamento de erros (Cristian, 1989; Gehani, 1992).

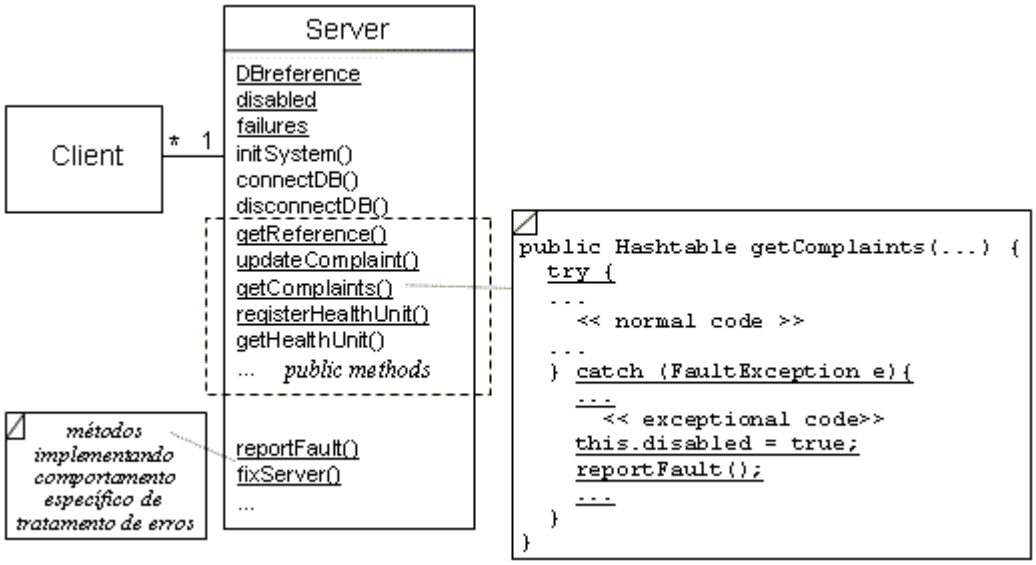

Figura 15. Tratamento de Erros: Exemplo de Interesse Transversal

Na Figura 15, os métodos e os atributos afetados pelo código de tratamento de erros estão sublinhados. O interesse de tratamento de erros está entrelaçado com os serviços públicos do servidor por toda a extensão da classe, isto é, o comportamento excepcional e o comportamento normal estão entrelaçados na implementação da classe `Server`. O interesse de tratamento de erros afeta a modularidade da classe uma vez que requer dois atributos (`disabled` e `failures`) e dois métodos relacionados ao tratamento de erros (`reportFault()` e `fixServer()`) na implementação da classe.

A fim de modularizar o comportamento relacionado ao tratamento de erros, é possível mover estes elementos do código para uma nova classe denominada `ExceptionalServer`. Entretanto, esta alternativa não resolve o problema, uma vez que permanecem as chamadas para atualizações de atributos e para métodos de tratamento de erros definidos na classe `ExceptionalServer` a partir da implementação de `Server`. Além disso, o problema de separação avançada de interesses transversais não se limita à modularização no nível de classes.

Um segundo e ainda mais sério problema é que o interesse de tratamento de erros também afeta a modularidade de diversos métodos em Server. A implementação de todos os serviços públicos normais está entrelaçada com o código de tratadores de exceção. A Figura 15 ilustra este problema na implementação do método `getComplaints()`. Apesar de a Figura 15 mostrar como o tratamento de erros entrecorta uma única classe e seus métodos internos, este interesse normalmente afeta várias classes e métodos do sistema. Como conseqüência, há uma diminuição na reusabilidade e na manutenibilidade do tratamento de erros e de outros interesses no sistema.

Portanto, sem mecanismos e abstrações adequados para a separação e a composição, os interesses transversais tendem a ficar espalhados e entrelaçados com outros interesses no código dos sistemas. O problema não se limita à implementação. Os interesses transversais são responsáveis pela produção de representações entrelaçadas e espalhadas ao longo de todo o ciclo de vida do software. Portanto, a separação efetiva de interesses transversais é essencial para melhorar a legibilidade e a manutenibilidade de artefatos nas várias etapas de desenvolvimento de software.

4.2. Programação Orientada a Aspectos

A POA introduz novas abstrações de modularização e mecanismos de composição para melhorar a separação de interesses transversais. A idéia é que, embora as abstrações e os mecanismos de linguagens orientadas a objetos sejam muito úteis, eles são inerentemente incapazes de modularizar todos os interesses em sistemas complexos. As novas abstrações propostas pela POA são: (1) aspectos[15], (2) pontos de junção[16], (3) conjuntos de junção[17], (4) adendos[18] e (5) declarações intertipos[19]. Estas abstrações são introduzidas a seguir.

15 Do inglês: *aspects*.
16 Do inglês: *joinpoints*.
17 Do inglês: *pointcuts*.
18 Do inglês: *advices*.
19 Do inglês: *inter-type declarations*.

4.2.1. Aspectos

Aspecto é o termo usado para denotar a abstração da POA que dá suporte a um melhor isolamento de interesses transversais. Em outras palavras, um aspecto corresponde a um interesse transversal e constitui uma unidade modular projetada para entrecortar um conjunto de classes e objetos do sistema.

São três as propriedades básicas de aspectos (Kiczales et al., 1997; Chavez et al., 2001; Elrad et al., 2001):

- *dicotomia baseada em aspectos*: diz respeito à adoção de uma distinção clara entre classes e aspectos (Kiczales et al., 1997; Chavez et al., 2001). Os sistemas orientados a aspectos são decompostos em classes e aspectos. Os aspectos modularizam os interesses transversais e as classes modularizam os outros interesses;
- *inconsciência*:[20]: propriedade desejável da programação orientada a aspectos. É a idéia de que os componentes não precisam ser preparados para serem entrecortados por aspectos (Elrad et al., 2001). Pela propriedade de inconsciência, os componentes não percebem os aspectos que poderão afetá-los (Chavez & Lucena, 2003);
- *quantificação*: capacidade de escrever declarações unitárias e separadas que afetam muitos pontos de um sistema (Elrad et al., 2001). Pela propriedade de quantificação, é possível fazer declarações do tipo "em programas P, sempre que surgir a condição C, faça a ação A".

Um aspecto pode afetar uma ou mais classes e/ou objetos de diferentes maneiras. Esta alteração pode afetar a estrutura estática ou dinâmica de classes e objetos. A estrutura dinâmica é alterada através da especificação de conjuntos de junção e adendos; a estrutura estática é alterada através de declarações intertipos. As definições de conjuntos de junção, adendos e declarações intertipos são apresentadas a seguir.

20 Do inglês: *obliviousness*.

4.2.2.
Pontos e Conjuntos de Junção, Adendos e Declarações Intertipos

Para definir conjuntos de junção, é necessária a introdução do conceito de pontos de junção. *Pontos de junção* são pontos bem específicos da execução de um sistema. Alguns exemplos de pontos de junção são: chamadas a métodos, execuções de métodos, leituras de atributos, modificações de atributos, etc. Através do conceito de pontos de junção, torna-se possível especificar o relacionamento entre aspectos e classes. Por exemplo, um aspecto A afeta uma classe C1 *no instante da invocação* de um método M1. Em outro exemplo, o aspecto A afeta uma classe C2 *ao final da execução* de um método M2.

O "instante da invocação do método M1" e o "ponto final da linha de execução do método M2" são exemplos de pontos de junção. Tais pontos de junção são os locais no programa onde os aspectos atuam sobre as classes. Eles podem ser especificados em uma linguagem orientada a aspectos através da definição de conjuntos de junção. Um *conjunto de junção* é o mecanismo que especifica os pontos de junção em que aspectos e classes se relacionam.

Para definir o comportamento transversal introduzido por um aspecto nas classes que este afeta, é necessária a especificação de um adendo. *Adendo* é um construtor especial semelhante a um método de uma classe, que define o comportamento dinâmico executado quando são alcançados um ou mais conjuntos de junção definidos previamente. Em outras palavras, adendos são recursos de entrecorte transversal que afetam o comportamento dinâmico de classes e objetos.

São dois os tipos de adendos utilizados neste trabalho: (i) *adendos anteriores*, executados sempre que os pontos de junção associados são alcançados e antes do prosseguimento da computação; e (ii) *adendos posteriores*, executados no término da computação, ou seja, depois que os pontos de junção forem executados e imediatamente antes do retorno do controle ao chamador.

As *declarações intertipos* especificam novos atributos e/ou métodos nas classes que um aspecto entrecorta ou modificam o relacionamento entre classes e entre classes e interfaces. Ao contrário de adendos, que operam de forma dinâmica, as declarações intertipos operam estaticamente, em tempo de compilação. Além de especificarem elementos que entrecortam classes de um sistema, aspectos podem possuir *métodos e atributos internos*, como classes OO.

As linguagens de programação orientadas a aspectos devem suportar a definição de atributos, métodos, conjuntos de junção, adendos e declarações intertipos em aspectos. A linguagem *AspectJ* (Kiczales et al., 2001), extensão orientada a aspectos da linguagem Java, oferece suporte à definição destas abstrações. Nesta linguagem, a composição entre aspectos e objetos de um sistema é realizada através de um processo denominado *combinação*.

Combinador é o mecanismo responsável pela composição de classes e aspectos. Quase todo o processo de combinação de aspectos e classes é realizado como um pré-processamento em tempo de compilação (AspectJ Team). Entretanto, a versão 5 de *AspectJ* já oferece suporte ao processo de combinação dinâmico, pela associação de mecanismos da linguagem com *frameworks* como *AspectWerkz* (AspectWerkz Website).

4.2.3. Exemplo de Aspecto

A Figura 16 mostra o aspecto `FaultHandler` implementado em *AspectJ* para modularizar o interesse de tratamento de erros (Seção 4.1). Este aspecto possui um atributo interno (linha 3), adiciona um atributo à classe `Server` (linha 5) e define dois métodos internos (linhas 7 a 9 e 11 a 13). O aspecto também define o conjunto de junção (linha 15) e os adendos (linhas 17 a 19 e 21 a 25).

O atributo introduzido em `Server` (Figura 16, linha 5) possui a função de indicar se a execução de qualquer método público (a identificação de qualquer método público está presente na definição do conjunto de junção) levantou uma exceção `FaultException`, conforme especificado pelo adendo posterior. O adendo anterior possui a função de verificar o conteúdo do atributo antes da chamada de qualquer método público, evitando que sejam efetuadas chamadas de métodos para servidores desativados.

A idéia por trás do conjunto de junção `services` (Figura 16, linha 15) é que o comportamento relacionado ao tratamento de erros deve ser disparado em quaisquer chamadas a métodos públicos. Por exemplo, o servidor não deve dar prosseguimento a uma solicitação devido à ocorrência de um erro. Estas chamadas de método são, portanto, eventos interessantes para o aspecto, porque, caso ocorram, devem ser realizadas ações associadas ao tratamento de erros.

```
1: aspect FaultHandler {
2:
3: private Vector failures = new Vector();
4:
5: private boolean Server.disabled = false;
6:
7: private void reportFault() {
8:    System.out.println("Failure! Please fix it.");
9: }
10:
11: public static void fixServer(Server s) {
12:   s.disabled = false;
13: }
14:
15:  pointcut  services(Server  s):  target(s)  &&  call(public  *
*(..));
16:
17: before(Server s): services(s) {
18:   if (s.disabled) throw new DisabledException();
19: }
20:
21: after(Server s) throwing (FaultException e): services(s) {
22:   s.disabled = true;
23:   failures.addElement(e);
24:   reportFault();
25: }
26: }
```

Figura 16. Implementação do Aspecto FaultHandler

Parte do contexto no qual ocorrem os eventos é exposta pelos parâmetros formais do conjunto de junção. No exemplo, o contexto consiste em objetos do tipo `Server`. O parâmetro formal está sendo usado no lado direito da declaração a fim de identificar a quais eventos o conjunto de junção se refere. Neste caso, o conjunto de junção especifica como pontos de junção eventos em que um objeto `s` do tipo `Server` seja o alvo de alguma operação (`target(s)`) e, simultaneamente, (&&, significando 'e') os eventos correspondem a chamadas de métodos (`call(...)`). As chamadas são especificadas por assinaturas que incluem *wildcards*. No exemplo, existem *wildcards* (i) na posição do tipo retorno (primeiro *), (ii) na posição do nome (segundo *) e (iii) na posição da lista de argumentos (..). A única informação concreta no conjunto de junção é o qualificador `public`.

4.3.
Projeto Orientado a Aspectos

Abstrações e mecanismos orientados a aspectos também devem ser suportados em fases preliminares do processo de desenvolvimento de software. Na etapa de projeto detalhado, linguagens de modelagem são essenciais para a especificação de aspectos, bem como para a especificação do relacionamento existente entre aspectos e entre aspectos e classes.

A Figura 17 ilustra o projeto do aspecto `FaultHandler` para a modularização do interesse de tratamento de erros (Seção 4.1). O projeto do aspecto baseia-se na linguagem de modelagem ASideML (Chavez & Lucena, 2001; Chavez, 2004), usada neste trabalho. Esta linguagem estende a UML (Booch et al., 1999) com notações para representar aspectos. Tais notações permitem descrever detalhadamente os elementos de um aspecto. No nível de projeto detalhado, um aspecto é representado por um losango, composto de estrutura interna e de interfaces transversais. A *estrutura interna* especifica os métodos e os atributos internos do aspecto. Uma *interface transversal* especifica quando e como o aspecto afeta uma ou mais classes, através de declarações intertipos, conjuntos de junção e adendos. Cada interface transversal é apresentada por um retângulo dividido em compartimentos. O primeiro compartimento do retângulo representa as declarações intertipos de um aspecto, e o segundo compartimento representa os conjuntos de junção e os adendos associados. A notação de ASideML utiliza uma seta pontilhada para representar o *relacionamento transversal* entre um aspecto e uma classe ou entre um aspecto e outro aspecto.

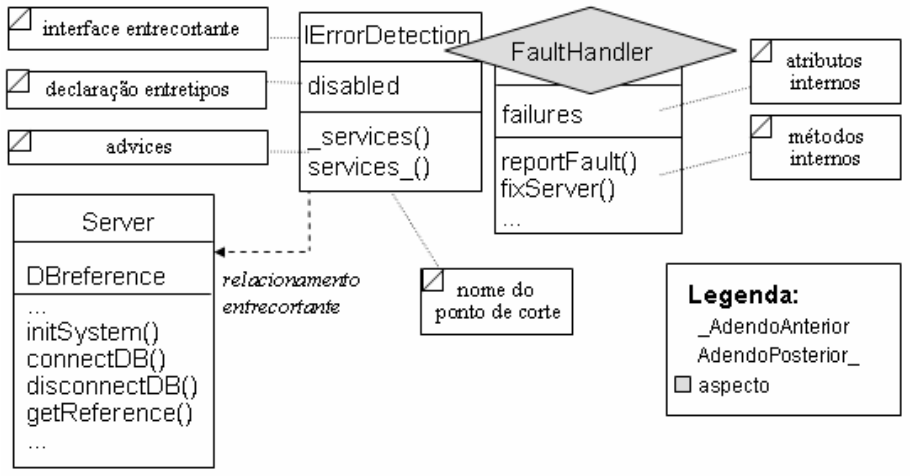

Figura 17. Projeto do Aspecto FaultHandler

A estrutura interna do aspecto `FaultHandler` consiste em dois métodos e um atributo (Figura 17). Estes elementos foram movidos da classe `Server` para o aspecto, uma vez que pertencem ao interesse de tratamento de erros. A interface transversal `IErrorDetection` (Figura 17) declara como o aspecto `FaultHandler` entrecorta a classe `Server`. Esta interface introduz o atributo `disabled` em `Server` e dois adendos, um adendo anterior e um posterior (_AdendoAnterior e AdendoPosterior_, respectivamente, na Legenda da Figura 17), associados ao mesmo conjunto de junção denominado `services`. Observe que o aspecto `FaultHandler` modulariza o interesse de tratamento de erros. A classe `Server` não contém código de tratamento de erros. Além disso, o aspecto `FaultHandler` mantém as propriedades de inconsciência e quantificação presentes em aspectos.

A Figura 18 apresenta os diagramas de interação que ilustram quando o aspecto `FaultHandler` entrecorta dinamicamente a classe `Server`. Os losangos representam instâncias de aspectos e os círculos pontos de junção.

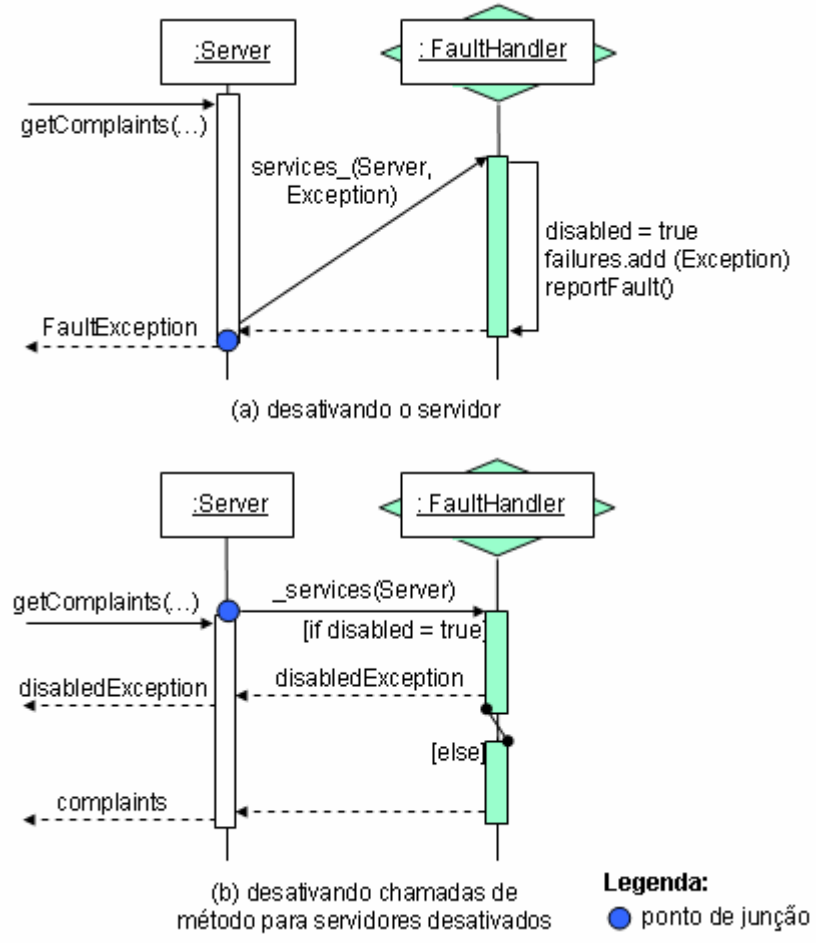

Figura 18. Interação entre Classe Server e Aspecto FaultHandler

4.4. Arquitetura Orientada a Aspectos

Arquitetura de software de um sistema é uma descrição em alto nível da organização do sistema em termos dos componentes arquiteturais que o compõem e dos relacionamentos existentes entre estes componentes (Shaw & Garlan, 1996). Os *componentes arquiteturais* são os principais blocos de construção[21] de uma arquitetura. Cada componente arquitetural corresponde a um interesse específico no sistema. Os componentes obedecem a um conjunto de responsabilidades a fim de implementar o interesse correspondente. Eles devem interagir usando regras previamente definidas a fim de que as responsabilidades impostas pela arquitetura sejam respeitadas. Cada componente possui um conjunto de *interfaces*, as quais tornam acessíveis os serviços implementados pelo componente.

Para o escopo deste trabalho, são importantes apenas as arquiteturas orientadas a aspectos. Particularmente, a noção de arquitetura orientada a aspectos proposta por Garcia (2004) introduz o conceito de *componente aspectual*. Um componente aspectual separa os interesses transversais associados a este componente dos demais componentes arquiteturais do sistema. Cada um dos componentes aspectuais se relaciona com mais de um componente arquitetural, o que decorre de sua natureza transversal. Tais relacionamentos estão associados com as interfaces do componente. As interfaces são categorizadas em dois grupos: (i) interfaces normais, e (ii) interfaces transversais. Uma interface transversal é diferente de uma interface normal. Uma interface normal apenas especifica os serviços fornecidos aos outros componentes. As interfaces transversais, além de fornecerem serviços ao sistema, também especificam quando um aspecto arquitetural afeta outros componentes.

Um componente aspectual pode realizar mais de uma interface transversal uma vez que pode afetar múltiplos componentes de diferentes maneiras. Uma interface de aspecto entrecorta ou elementos internos de um componente ou elementos de outras interfaces. O primeiro caso significa que o aspecto arquitetural entrecorta a estrutura ou comportamento interno do componente. O segundo caso significa que o aspecto afeta um ou mais serviços disponibilizados

21 Do inglês: *building blocks*.

pelas interfaces do outro componente. Uma notação para os elementos da arquitetura orientada a aspectos acima é introduzida a seguir.

A Figura 19 ilustra a arquitetura orientada a aspectos do aspecto `FaultHandler` para a modularização do interesse de tratamento de erros (Seção 4.1). A modelagem da arquitetura é baseada na linguagem ASideML (Chavez, 2004). Nesta linguagem, os componentes aspectuais são representados como diamantes. As interfaces normais são representadas como círculos brancos; as interfaces transversais são representadas como círculos cinzas. A visão de um aspecto arquitetural suprime todas as informações sobre seus elementos internos, exceto os nomes de suas interfaces transversais. Cada interface transversal é ilustrada como um pequeno círculo com o nome da interface localizado próximo ao círculo. O círculo está ligado por uma linha sólida ao diamante representando o aspecto arquitetural.

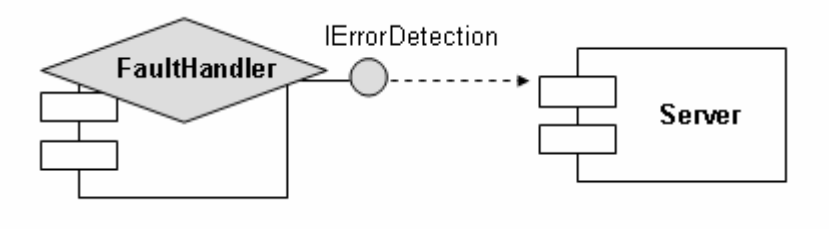

Figura 19. Modelagem do Aspecto Arquitetural FaultHandler

No próximo capítulo, é apresentada a arquitetura orientada a aspectos proposta neste trabalho como parte da solução para o problema de separação avançada dos interesses de mobilidade em SMAs.