2
Separação de concerns

> *"This is what I mean by 'focusing one's attention upon a certain aspect'; it does not mean completely ignoring the other ones, but temporarily forgetting them to the extent that they are irrelevant for the current topic. Such separation, even if not perfectly possible, is yet the only available technique for effective ordering of one's thoughts that I know of." (Dijkstra, 1976)*

O trabalho de qualquer engenheiro é gerenciar a complexidade durante as diferentes fases do desenvolvimento de sistemas. Isso deve ocorrer em especial na engenharia de software: muitas pesquisas na área tratam de como gerenciar a complexidade de sistemas de software, oferecendo melhores métodos e técnicas de estruturação. A *Separação de concerns* [60] está no centro da engenharia de software, em geral, como um princípio fundamental que tenta resolver as limitações da cognição humana ao lidar com a complexidade do software. Um *concern* é uma parte do problema que queremos tratar como uma unidade conceitual única na solução do software [60].

O princípio da separação de concerns defende que, para superar a complexidade, deve-se resolver uma questão (ou concern) importante por vez [60]. Na engenharia de software, esse princípio está relacionado à *modularização* e à *decomposição* de sistemas [193]. Os sistemas de software complexos devem ser decompostos em unidades modulares menores e claramente separadas, cada uma lidando com um único concern [193]. Se bem realizada, a separação de concerns pode oferecer muitos benefícios cruciais [60, 234]. Ela oferece suporte a uma melhor manutenibilidade com alterações aditivas, em vez de invasivas, melhor compreensão e redução da complexidade, melhor reusabilidade e uma integração de componentes simplificada.

Os concerns podem ser gerais, dependentes do domínio e dependentes da aplicação. Os *concerns gerais* estão presentes em quase todos os sistemas de software, independente do domínio e da funcionalidade da aplicação. Alguns

exemplos típicos de concerns gerais são tratamento de erro, distribuição, persistência, auditoria etc. Os *concerns dependentes do domínio* estão relacionados a um domínio de aplicação específica [55], como o domínio de sistemas multiagentes (SMAs). Alguns exemplos típicos de concerns específicos a SMAs são autonomia, adaptação, conhecimento, mobilidade etc. Os *concerns dependentes da aplicação* referem-se a funcionalidades específicas de um sistema de software.

A separação dos concerns depende muito da adeqüabilidade das *abstrações* e *métodos* usados ao longo do ciclo de vida do software. Ela também depende de *mecanismos de composição* para oferecer suporte à composição de concerns separados. A engenharia de software orientada a objetos (OO) foi durante muitos anos o paradigma de desenvolvimento dominante, e seus benefícios são amplamente reconhecidos. Os mecanismos de composição, os métodos e as abstrações OO evoluíram para permitir que fosse alcançada a separação de concerns no código-fonte e nos níveis superiores. As classes, os objetos e os métodos são exemplos de abstrações clássicas na engenharia de software orientada a objetos. Por exemplo, um único concern pode ser modularizado como uma classe ou um método único. A herança e o polimorfismo são exemplos de mecanismos que permitem a modularização e a composição de concerns de software. Com o aumento da complexidade do sistema, fica mais difícil conseguir a separação explícita de todos os concerns dos sistemas com base apenas em abstrações e mecanismos OO. Nos últimos anos, surgiram muitas técnicas de suporte para auxiliar e melhorar a separação de concerns no desenvolvimento de sistemas OO. Os *padrões de software* (Seção 2.1) são um exemplo dessas técnicas que conseguiu alcançar a maturidade. É uma técnica muito aceita no setor e no meio acadêmico. A *separação avançada de concerns* (Seção 2.2) denota muitas técnicas que estão surgindo para oferecer suporte à separação de concerns especiais, que não são capturados por abstrações da engenharia de software orientada a objetos.

2.1
Padrões

Os padrões de software [35, 79] são um veículo importante para a construção de softwares de alta qualidade. Os padrões foram introduzidos na última década como *abstrações nomeadas* que exibem soluções bem conhecidas para problemas recorrentes em um contexto de desenvolvimento. A solução de padrões oferece suporte à separação de concerns oferecendo orientação sobre como decompor e compor os componentes do problema.

Os padrões são retirados de experiências recorrentes e existem em diferentes fases de desenvolvimento, como arquitetura [35], projeto detalhado [79] e implementação [35]. Os *padrões arquiteturais* definem a estrutura básica de uma arquitetura e de sistemas que implementa essa arquitetura [35]; enquanto *padrões de projeto* são mais orientados à implementação do que os padrões arquiteturais e são aplicados em etapas posteriores do projeto. Os *Idiomas* são padrões que resolvem problemas no contexto de uma linguagem de programação específica [35]. A Seção 2.1.1 discute os padrões arquiteturais e as arquiteturas de software. A Seção 2.1.2 apresenta padrões de projeto. Já a Seção 2.1.3 discute as linguagens de padrões.

2.1.1
Padrões arquiteturais e arquiteturas de software

A *arquitetura de software* é uma descrição de alto nível da organização do sistema em termos dos *componentes arquiteturais* e seus inter-relacionamentos [214]. Os componentes arquiteturais são partes físicas e substituíveis de uma arquitetura; a cada componente, é atribuído um concern específico. Eles possuem um conjunto de responsabilidades a fim de implementar o concern que lhe foi atribuído. Além disso, eles devem interagir entre si usando regras preestabelecidas para cumprir com suas responsabilidades, conforme imposto pela arquitetura. Cada componente deve fornecer a realização de um conjunto de *interfaces* e estar em conformidade com elas, de forma a possibilitar os serviços implementados pelo componente.

Os *padrões arquiteturais* definem a estrutura básica de uma arquitetura e dos sistemas que a implementam [35]. A solução de um padrão arquitetural define os

componentes arquiteturais e suas regras de comunicação. Uma arquitetura de software pode incorporar um padrão arquitetural ou mais. Há diversos padrões arquiteturais, incluindo o padrão Camadas, o padrão Reflexão e o padrão Blackboard [35]. Por exemplo, os componentes arquiteturais nos padrões Camadas são *camadas* que interagem entre si em um fluxo de controle de duas vias. Cada camada implementa um concern específico, e sua comunicação está limitada às camadas adjacentes. Kendall et al. [137] propõem uma arquitetura de agentes que segue a estrutura geral do padrão arquitetural Camadas. Cada camada tem o objetivo de cuidar de um concern individual em uma arquitetura de agentes. Discutimos as vantagens e desvantagens da arquitetura de agentes na Seção 3.7.1.

2.1.2 Padrões de projeto

Os padrões de projeto são aplicados em etapas posteriores do projeto e refinam os componentes das arquiteturas de software. Em geral, a escolha dos padrões de projeto sofre a influência do padrão arquitetural escolhido ou da arquitetura de software. A descrição dos padrões segue um *modelo* ou uma *forma* a fim de facilitar o compartilhamento da solução. A descrição de padrões de projeto segue modelos específicos, como o modelo Alexandrino [4], o modelo GoF [79] e o modelo POSA [35]. Os padrões apresentados nesta tese (Capítulo 5) seguem um modelo que mescla as propostas GoF e POSA. A Tabela 1 descreve os elementos do modelo de padrão usado; os itens mais importantes estão em negritos.

Os elementos mais relevantes no modelo de padrão são: (i) nome, (ii) contexto, (iii) problema e (iv) solução. O *nome* do padrão carrega a essência do padrão de forma sucinta. A escolha de um bom nome é vital, porque ele fará parte do vocabulário do projeto dos engenheiros de software [35, 79]. O *contexto* descreve as situações às quais o padrão se aplica. É importante porque normalmente inclui comprovações de generalidade. O *problema* descreve questões do projeto enfrentadas pelo desenvolvedor. A *solução* tem uma importância especial porque descreve como resolver o problema. Ela define os elementos (*participantes*) que criam o projeto em

termos das abstrações do paradigma da programação em uso. Cada concern no problema é, em geral, representado por um elemento da solução.

Como a orientação a objetos é o paradigma de desenvolvimento dominante, muitos padrões de projeto existentes seguem um estilo OO. As soluções são fornecidas em termos de mecanismos e abstrações OO. Um *padrão de projeto orientado a objetos* oferece uma solução de implementação e projeto a um problema recorrente, definindo um conjunto de classes e seus relacionamentos [79]. O uso de padrões de projeto foi muito defendido no desenvolvimento de software OO, com ênfase na reusabilidade e manutenibilidade [79]. A solução do padrão estrutura e disciplina a composição de classes separadas, garantindo que o sistema só possa mudar ou se desenvolver de formas previsíveis e específicas.

Elemento do modelo	Descrição
Nome	O nome, uma expressão ou um nome descritivo e familiar.
Intenção	Frases curtas que respondem às perguntas a seguir: O que o padrão de projeto faz? Qual é o seu princípio? De que questão de projeto ou problema particular ele trata?
Também conhecido como	Outros nomes para o padrão, se houver algum conhecido.
Contexto	As situações às quais o padrão pode ser aplicado. Normalmente inclui discussões e informações gerais de por que o padrão existe.
Exemplo de motivação	Um exemplo real que demonstre a existência de um problema e a necessidade do padrão. Ele compreende um cenário que ilustra o problema do projeto. O cenário ajudará a entender a descrição abstrata do problema e a solução subseqüente.
Problema	Uma declaração do problema resolvido pelo padrão. O problema pode ser colocado como uma pergunta. Inclui uma descrição das forças ou restrições relevantes.
Solução	Oferece a idéia principal subjacente à solução do projeto. Descrição de alto nível dos relacionamentos estáticos e das regras dinâmicas.
Estrutura	Uma especificação detalhada das facetas estruturais da solução do padrão em termos dos participantes.
Participantes	Os componentes (por exemplo as classes e/ou os objetos) que participam do padrão do projeto e suas responsabilidades.
Dinâmicas	Como os participantes colaboram para cumprir com suas responsabilidades? Descrição dos cenários típicos que descrevem o comportamento em tempo de execução do padrão.
Conseqüências	Como o padrão oferece suporte à sua intenção e às suas forças? Quais são as conclusões e os resultados do uso do padrão?
Variantes	Descrição dos variantes ou das especializações de um padrão.
Usos conhecidos	Exemplos do padrão encontrado em sistemas reais.
Padrões relacionados	Que padrões de projeto estão intimamente relacionados a este? Quais são as diferenças importantes? Com quais outros padrões este padrão pode ser usado?

Implementação	Você deve estar alerta em relação a quais imprevistos, dicas ou técnicas ao implementar o padrão? Há questões específicas à linguagem?
Código de exemplo	Os fragmentos de código que ilustram como alguém pode implementar o padrão em uma determinada linguagem de programação.

Tabela 1. O modelo de uma descrição de padrão

No contexto de agentes de software, há vários padrões de projeto orientados a objetos [19, 137]. Um padrão de projeto é usado como uma metáfora da implementação e do projeto para estruturas orientadas a objetos recorrentes para sistemas multiagentes. Os padrões orientados a objetos gerais, como os padrões Observador e Composto, e os padrões específicos a SMAs [137, 221] são usados em uma construção de SMAs. O Capítulo 3 discute por que os padrões de projeto OO normalmente não são suficientes para modularizar alguns concerns no domínio dos SMAs. O Capítulo 5 apresenta alguns exemplos de padrões de projeto OO que não conseguem alcançar a separação de concerns específicos a SMAs. Apesar de os padrões de projeto terem surgido da comunidade orientada a objeto, eles podem ser associados a diferentes paradigmas de desenvolvimento. Nesse sentido, o Capítulo 5 também apresenta um conjunto de *padrões de projeto orientados a objetos* para tratar da separação explícita de concerns específicos a SMAs.

2.1.3 Linguagens de padrões

Os padrões são os módulos dos sistemas de software de larga escala, que provavelmente incluirão as instâncias de mais de um desses padrões, compostos de várias maneiras. Em cada ponto de decisão, o projetista seleciona o padrão apropriado de um catálogo. Cada padrão leva a outros padrões, resultando em um projeto final como uma rede de padrões. Um catálogo estruturado de padrões que oferece suporte a esse estilo de projeto é chamado de *linguagem de padrões* [5, 50]. Uma linguagem de padrões é mais do que simplesmente um catálogo de padrões; ela incorpora uma metodologia de projeto e fornece ao projetista um advice específico ao domínio.

Uma linguagem de padrões orienta um projetista ao fornecer soluções trabalháveis para todos os problemas conhecidos em um determinado domínio. Cada

padrão de projeto trata de um concern específico recorrente no domínio da linguagem. A linguagem é uma seqüência de pequenos conhecimentos escritos em um estilo e organizados em uma ordem que leva o projetista a saber como usar os padrões juntos. Uma linguagem de padrões define um *conjunto de padrões* e as *regras* que os combinam. Elas também descrevem arquiteturas de software ou famílias de sistemas relacionados. O Capítulo 5 descreve uma linguagem de padrões voltada para os desenvolvedores de SMAs que desejam criar seus agentes de software com separação explícita de concerns.

2.2 Separação avançada de concerns

Apesar de a programação orientada a objetos ter sido a tecnologia de programação dominante durante anos, a orientação a objetos possui algumas limitações no tratamento de concerns que cuidam dos requisitos naturalmente envolvidos em diversas operações e componentes do sistema. A *separação avançada de concerns* denota as técnicas cujo objetivo é oferecer suporte à modularização desses concerns especiais, chamados de crosscutting concerns. A Seção 2.2.1 define e ilustra os crosscutting concerns. A Seção 2.2.2 discute as técnicas para a separação avançada de concerns, em especial, a programação orientada a aspectos.

2.2.1 Crosscutting Concerns

As abstrações orientadas a objetos normalmente não são suficientes para a separação de *crosscutting concerns* [139, 234]. Esses concerns são chamados de crosscutting concerns porque afetam[2] naturalmente classes e/ou métodos que modularizam outros concerns. Alguns exemplos de crosscutting concerns gerais são rastreamento [12], auditoria [12], persistência [226], distribuição [226] e tratamento de erros [87, 96, 97, 161]. Eles introduzem dois problemas básicos: espalhamento e entrelaçamento. O *espalhamento* é o fenômeno em que as linhas de código que cuidam de um único concern são distribuídas por várias classes [234]. O

[2] Do inglês “crosscut”.

entrelaçamento é o fenômeno em que uma única classe contém as linhas de código que cuidam de vários concerns [234].

A Figura 3 apresenta um exemplo de um crosscutting concern encontrado na literatura [12, 226]. O tratamento de erros é normalmente espalhado e entrelaçado com outros concerns em um sistema orientado a objetos. A classe Server está relacionada à implementação de serviços básicos. Suponha que essa classe seja da classe facade [79], o único ponto de entrada que oferece serviços públicos aos clientes. Cada método implementa um concern específico do servidor, ou seja, de um determinado serviço público. A implementação das atividades normais dos serviços não deve estar intermisturada às atividades excepcionais, ou seja, o concern de tratamento de erros. A separação entre o código normal e excepcional é um requisito impulsionador em aplicações robustas [53, 87, 97, 151, 192], em especial em sistemas tolerantes a falhas [52, 54, 86, 97]. Nesses sistemas, o código dedicado à detecção e ao tratamento de erros é normalmente longo e complexo. Como conseqüência, até dois terços de um programa podem ser usados para o tratamento de erros [53, 85, 102].

Os métodos e os atributos afetados pelo código de tratamento de erros estão descritos na Figura 3. Esse concern de tratamento de erros está entrelaçado com os serviços do servidor durante a implementação da classe. O comportamento excepcional e normal são misturados na implementação da classe Server. O concern 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 de tratamento de erros, podemos mover esses elementos do código para uma nova classe ExceptionalServer. Entretanto, isso não resolve o problema; as chamadas para as atualizações e métodos de tratamento de erros nos atributos continuariam na implementação da classe Server.

Além disso, o problema de crosscutting concerns não se limita à modularidade no nível da classe. Um segundo e ainda mais sério problema é que o concern de tratamento de erros também afeta a modularidade de diversos métodos em Server. A implementação de todos os serviços públicos normais está intermisturada ao código

para os manipuladores de exceção. A Figura 3 ilustra esse problema na implementação do método getComplaints(). Apesar de a figura mostrar como o tratamento de erros afeta uma única classe e seus métodos internos, esse concern normalmente afeta várias classes e métodos do sistema. Como conseqüência, há uma diminuição na reusabilidade e na manutenibilidade dos concerns de tratamento de erros e de outros concerns no sistema.

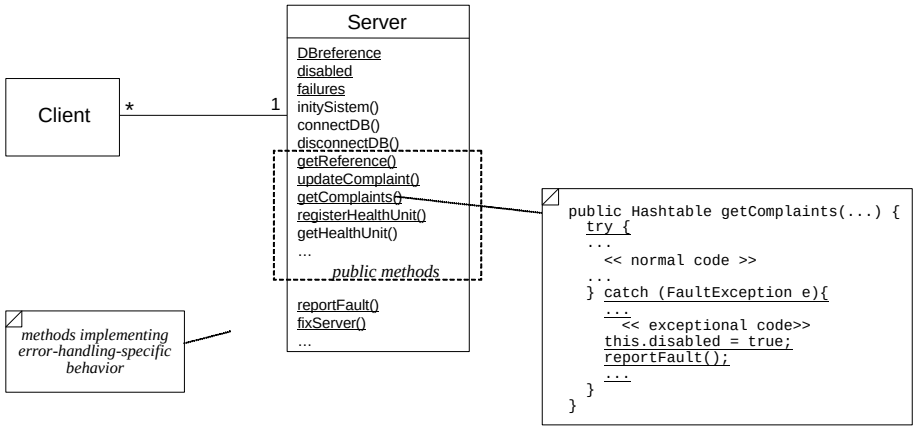


Figura 3. Tratamento de erros: um exemplo de Crosscutting Concern.

Portanto, sem os mecanismos e as abstrações adequados para a separação e a composição, os crosscutting concerns tendem a ficar espalhados e entrelaçados com outros concerns. O problema não se limita à implementação. Os crosscutting concerns são responsáveis pela produção de representações entrelaçadas e espalhadas ao longo do ciclo de vida do software. A separação efetiva de crosscutting concerns é essencial para melhorar a manutenibilidade e a compreensibilidade de artefatos em várias etapas de desenvolvimento do software.

2.2.2 Aspectos e técnicas de programação

Há várias abordagens recentes à separação avançada de concerns, incluindo a programação orientada a aspectos [139], separação multidimensional de concerns [188, 235], programação orientada a sujeitos [116], programação adaptativa [159, 160] e filtro de composição [3, 21]. Essas abordagens contribuíram para melhorar a programação orientada a objetos, uma vez que proporcionavam a separação de concerns em outras dimensões, além das classes e dos objetos. A *Programação*

orientada a aspectos (POA) e a *Separação multidimensional de Concerns* (SMDC) são as técnicas mais conhecidas. Elas introduzem novas abstrações de modularização e mecanismos de composição para melhorar a separação de crosscutting concerns no nível da implementação.

Cada uma dessas abordagens citadas descreve seu próprio modelo de programação para oferecer suporte a crosscutting concerns. Esses modelos oferecem abstrações e mecanismos de implementação que possibilitam a separação e a composição de concerns a fim de produzir um sistema geral. A idéia é que, embora os mecanismos de modularidade hierárquica das linguagens de implementação orientadas a objetos sejam muito úteis, eles são inerentemente incapazes de modularizar todos os concerns de interesse em sistemas complexos. AspectJ [104] e Hyper/J [235] são duas linguagens orientadas a aspectos, em desenvolvimento, reconhecidas como as principais representantes da POA e da SMDC, respectivamente. AspectJ é a linguagem mais madura, uma vez que possui uma grande comunidade de usuários, evoluiu muito nos últimos sete anos e já foi aplicada em projetos no setor [12]. Muitos estudos de caso encontrados na literatura exploram a programação orientada a aspectos e AspectJ [98, 138, 226]. Este trabalho baseia-se na terminologia da POA [12, 139, 140].

Aspectos

Aspecto é o termo usado para denotar a abstração cujo objetivo é oferecer suporte a um melhor isolamento de crosscutting concerns. Eles são unidades modulares de crosscutting concerns associados a um conjunto de classes e objetos. Um aspecto pode *afetar* uma ou mais classes e/ou objetos de várias formas. Ele pode alterar a estrutura estática (*static crosscutting*) ou a dinâmica (*dynamic crosscutting*) de classes e objetos.

A abstração de aspectos possui três propriedades básicas [44, 67, 139]: (i) dicotomia baseada em aspectos, (ii) transparência e (iii) quantificação. A *dicotomia baseada em aspectos* significa a adoção de uma distinção clara entre classes e aspectos [139, 42]. Os sistemas orientados a aspectos são decompostos em classes e aspectos; os aspectos modularizam os crosscutting concerns e as classes modularizam

os não-crosscutting concerns. Os aspectos são explicitamente representados separados das classes e de outros aspectos.

A *transparência* é uma propriedade essencial da programação orientada a aspectos. Ela é o ato ou o efeito de deixar transparente, no sentido de poder ser esquecido ou passar despercebido. A transparência é a idéia de que os componentes não precisam ser especificamente preparados para receber as melhorias proporcionadas pelos aspectos [67]. Quando existe a propriedade de transparência, os componentes não percebem os aspectos que poderão afetá-los [44]. A transparência é considerada a característica que torna a POA especial [71]. A *Quantificação* é a capacidade de escrever declarações unitárias e separadas que afetam muitos lugares não-locais no sistema de programação [67]. Ela proporciona a capacidade de declarar coisas como: “em programas P, sempre que surgir a condição C, faça a ação A” [71]. Quando há a propriedade de quantificação, os aspectos podem afetar um número arbitrário de componentes simultaneamente.

A definição de aspectos incorpora pointcuts, advices, métodos e atributos internos e inter-type declarations.

Join Points e Pointcuts

No centro do processo de composição de aspectos e classes está o conceito de *join points*, os elementos que especificam como as classes e os aspectos se relacionam. Join points são pontos bem definidos na execução dinâmica de um sistema. Alguns exemplos de join points são as chamadas de método, as execuções de método e as leituras e os conjuntos de campo. *Pointcuts* têm um nome e são um conjunto de join points.

Advices e Inter-Type Declarations

Advice é um construto especial do tipo método acoplado a pointcuts. Os advices são recursos dinâmicos de *crosscutting* uma vez que afetam o comportamento dinâmico de classes e objetos. Há vários tipos de advices: (i) *before advice*s são executados sempre que é alcançado um join point e antes do prosseguimento real da computação; (ii) *after advices* são executados depois da conclusão da computação “sob join point”, ou seja, depois que o corpo do método tiver sido executado e logo antes do retorno do controle ao chamador; (iii) *around advices* são executados sempre

que um join point for alcançado e têm controle explícito se a computação sob o join point tiver permissão para ser executada.

Um aspecto também pode conter *inter-type declarations*; essas declarações especificam novos membros (atributos ou métodos) para as classes das quais o aspecto faz parte ou alteram o relacionamento de herança entre as classes. Diferente dos *advices*, que operam primariamente de forma dinâmica, as *inter-type declarations* operam de forma estática, em tempo de compilação. Essas declarações são recursos de *crosscutting* estáticas uma vez que afetam a estrutura estática dos componentes.

Um exemplo de aspecto

A Figura 4 ilustra o aspecto FaultHandler para a modularização do concern de tratamento de erros (Seção 2.2.1). O projeto do aspecto baseia-se na linguagem de modelagem ASideML [41, 43], usada nesta tese. Essa linguagem estende a UML com notações para representar aspectos. A notação fornece uma descrição detalhada dos elementos do aspecto. Na linguagem do projeto, um aspecto é representado por um losango, é composto de uma estrutura interna e de interfaces crosscutting. A *estrutura interna* declara os métodos e os atributos internos. Uma *interface crosscutting* especifica quando e como o aspecto afeta uma ou mais classes [41, 43]. Cada interface crosscutting é apresentada usando o símbolo retangular com compartimentos (Figura 4). Uma interface crosscutting é composta por inter-type declarations, pointcuts e advices. A primeira parte de uma interface crosscutting representa inter-type declarations, e a segunda, os advices acoplados. A notação usa uma seta pontilhada para representar o *relacionamento crosscutting*, que relaciona um aspecto a classes e/ou aspectos.

A estrutura interna do aspecto FaultHandler consiste em dois métodos e um atributo. Eles foram movidos da classe Server para o aspecto uma vez que fazem parte do concern de tratamento de erros. A interface crosscutting IErrorDetection (Figura 4) declara como o aspecto FaultHandler afeta a classe Server. Essa interface introduz o atributo disabled em Server, e dois advices. Há um *before advice* e um *after advice*, os dois associados ao mesmo pointcut chamado services. Observe que o aspecto FaultHandler modulariza o concern de tratamento de erros, e a classe Server não

contém código de tratamento de exceções. Esse aspecto mantém as propriedades de transparência e qualificação dos presentes nos aspectos.

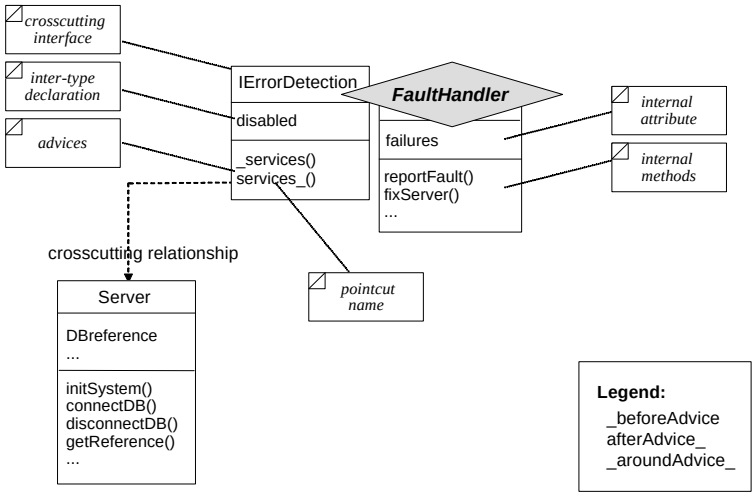
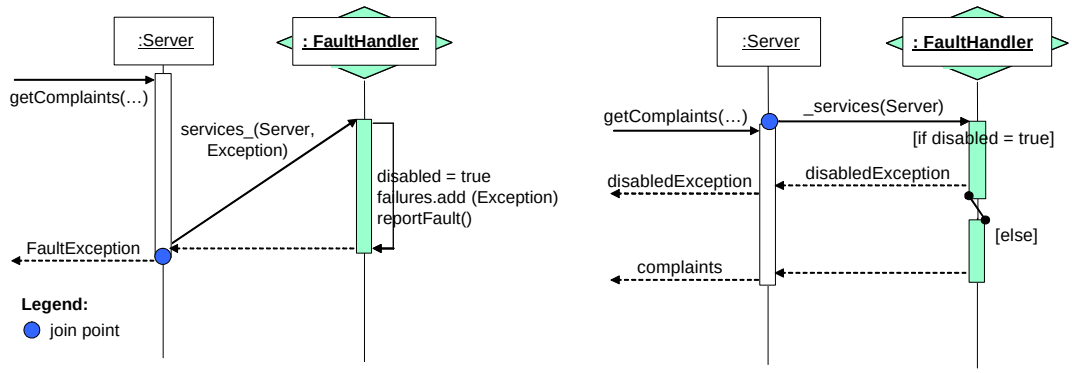


Figura 4. O projeto do aspecto FaultHandler.

A Figura 5 apresenta os diagramas de interação que ilustram quando o aspecto FaultHandler afeta dinamicamente a classe Server. Os losangos representam instâncias de aspectos, e os círculos, join points. A Figura 6 mostra a implementação de AspectJ do aspecto FaultHandler. Esse aspecto possui um atributo interno (linha 3), adiciona um campo à classe Server (linha 5) e define dois métodos internos (linhas 7 a 9 e 11 a 13). O aspecto também define o pointcut (linha 15) e os advices (linhas 17 a 19 e 21 a 24). O campo introduzido indica se a execução de algum método público (identificado pela definição do pointcut) lançou uma FaultException, conforme especificado pelo after advice. O before advice verifica o campo antes da chamada de qualquer método público, evitando chamadas do método para servidores desativados.



(a) desativando o servidor

(b) desativando as chamadas de método para servidores desativados

Figura 5. A dinâmica da classe Server e o aspecto FaultHandler.

A idéia por trás do pointcut services no aspecto FaultHandler é que o comportamento relacionado ao tratamento de falhas deve ser disparado nas chamadas a métodos públicos. Por exemplo, o servidor pode não conseguir dar prosseguimento à solicitação devido a alguma falha. Essas chamadas de método são, portanto, eventos interessantes para esse aspecto, porque, caso eles ocorram, serão realizadas determinadas ações relacionadas a falhas.

```
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 6. Exemplo de AspectJ.

Parte do contexto no qual ocorrem os eventos está exposta pelos parâmetros formais do pointcut. Nesse caso, consiste em objetos do tipo Server. O parâmetro formal está então sendo usado no lado direito da declaração a fim de identificar a quais eventos o pointcut se refere. Nesse caso, um pointcut que escolha join points em que um Server seja o alvo de alguma operação (target(s)) está sendo composto (&&, significando ‘e’) com um pointcut escolhendo join points (call(...)). As chamadas são identificadas por assinaturas que incluem *wild cards*. Nesse caso, há *wild cards* na posição do tipo retorno (primeiro *), na posição do nome (segundo *) e na posição da lista de argumentos (..); a única informação concreta dada é o qualificador public.

Processo de combinação e AspectJ

AspectJ [140] é a extensão orientada a aspectos prática à linguagem de programação Java. A programação com AspectJ usa objetos e aspectos para separar concerns. Os concerns bem-modelados como objetos são separados dessa forma; já os concerns que afetam os objetos são separados usando unidades chamadas aspectos, compostas por objetos de um sistema por um processo chamado *weaving* (processo de combinação). AspectJ oferece suporte à definição de aspectos, advices, join points e pointcuts.

Weaver (processador de combinação) é o mecanismo responsável pela composição de classes e aspectos. Ao construir aspectos de AspectJ com código Java padrão, obtemos uma nova aplicação de AspectJ. Até a versão atual de AspectJ, quase todo o processo de combinação é realizado como uma etapa de pré-processamento em tempo de compilação [12]. Há alguns frameworks sendo desenvolvidos que oferecem suporte ao processo de combinação dinâmico, como AspectWerkz [13], JBoss [76] e Prose [195].

2.2.3
Desafios no desenvolvimento orientado a aspectos

A programação orientada a aspectos está alcançando a maturidade depois de quase uma década de pesquisas [236] com um número crescente de aplicações, ferramentas e usuários [12]. Uma edição de 2001 da *Technology Review*, a revista de inovação do MIT, dedicou-se à seleção das “dez áreas de tecnologia emergentes que, em breve, terão um grande impacto na economia e na forma como vivemos e trabalhamos” [236]. Uma das dez áreas escolhidas foi a programação orientada a aspectos, reconhecida como uma tecnologia promissora na era da programação pós-objeto.

A evolução de um novo estilo de programação normalmente evolui da programação na direção da arquitetura e do projeto a fim de proporcionar um caminho completo ao longo do ciclo de vida do desenvolvimento do software. O *Desenvolvimento de software orientado a aspectos* (DSOA) é uma área emergente

cujo objetivo é promover a separação avançada de concerns ao longo de todo o ciclo de vida do desenvolvimento de software. Muitas questões devem ser resolvidas antes de alcançar o desenvolvimento de software orientado a aspectos. Em especial, observamos os problemas a seguir no estado da técnica na área do DSOA, enquanto eram avaliadas as abordagens existentes das aplicações orientadas a aspectos:

- **A necessidade de aplicação da POA em concerns específicos a domínios**:

Em geral, a pesquisa de DSOA dedicou-se a crosscutting concerns gerais ou triviais, como auditoria [12], rastreamento [12], distribuição [226], persistência [226], concorrência [141] e tratamento de exceções [96, 97, 161]. Além disso, esses concerns foram quase sempre estudados de forma isolada. Há pouca compreensão sobre o uso de aspectos com vários concerns em um sistema de software.

Os crosscuting concerns gerais são mais fácies de identificar porque normalmente estão ligados por construtos de linguagens de programação especiais. Por exemplo, o concern de tratamento de erros (Seção 2.2.2) é definido no código usando construtos para o levantamento de exceções (por exemplo `throw` em Java), para especificar contextos de tratamento (por exemplo `try` em Java), para especificar os manipuladores (por exemplo `catch` em Java) e para declarar a propagação das exceções externas nas assinaturas de método (por exemplo `throws` em Java).

O objetivo de AspectJ e Hyper/J é ser linguagens orientadas a aspectos de propósito geral e, como conseqüência, elas devem ser investigadas em relação à modularização de outros tipos de crosscutting concerns. Os crosscutting concerns específicos a domínios são mais difíceis de serem identificados porque não estão baseados em construtos de linguagem de programação específicos. Para identificar esses aspectos, é necessário um conhecimento de domínio extensivo. Há poucas experiências envolvendo aspectos no domínio de SMAs [91]. Kendall et al ressaltam o uso dos aspectos para a modularização de papéis [136].

- **A necessidade de suporte arquitetural:**

Há a necessidade de mudança do foco de mecanismos de implementação orientados a aspectos para abordagens de projeto e arquitetura a fim de estabelecer a

POA como um novo paradigma para o desenvolvimento de software. As arquiteturas orientadas a aspectos exibem diferentes características porque são compostas por componentes e aspectos. As técnicas de DSOA devem oferecer meios sistemáticos para a identificação e representação de aspectos na etapa arquitetural. Recentemente, muitos grupos de pesquisa começaram a discutir a função dos aspectos nas primeiras etapas do processo de desenvolvimento de software [11, 63, 64, 239]. Contudo, a maioria dos trabalhos existentes é dedicada à engenharia de requisitos [63, 111, 200, 201, 229] e linguagens de modelagem [41, 47, 48, 231].

Se os crosscutting concerns não estiverem efetivamente modularizados no nível da arquitetura, não é possível discutir seus efeitos no sistema ou entre si em uma etapa inicial do projeto [64]. Além disso, a falta de modularização desses concerns pode resultar em um grande efeito de ondulação nos componentes arquiteturais no momento da evolução. A disposição de meios efetivos para o tratamento de aspectos arquiteturais possibilita o estabelecimento de conclusões críticas logo no início do ciclo de vida do software.

Sempre houve um corpo de trabalho significativo na separação de concerns na engenharia de requisitos e nas comunidades de projeto de arquitetura, por exemplo, pontos de vista [73], casos de uso [125], objetivos [157, 242] e modelos de análise de conclusões de arquitetura [18, 134]. No entanto, essas abordagens não se concentram explicitamente em crosscutting concerns. Portanto, os trabalhos relacionados a aspectos devem complementar essas abordagens proporcionando meios sistemáticos para o tratamento desses concerns.

- **A necessidade de padrões de projetos orientados a aspectos:**

Um projeto orientado a aspectos de alta qualidade é uma parte crítica do DSOA que se concentra em orientações e padrões para a identificação, a estruturação e a representação de crosscutting concerns em projetos de software, oferecendo um caminho desde a arquitetura até a implementação com POA. Os padrões de projeto são um veículo importante para o suporte à construção de sistemas orientados a aspectos com reusabilidade e manutenibilidade (Seção 2.1). A necessidade de padrões de projeto é ainda maior para o desenvolvimento de software orientado a aspectos

uma vez que esse paradigma oferece muitas formas de decompor e compor um sistema de software (Seção 2.2.3).

Apesar de o paradigma orientado a aspectos defender o suporte a uma melhor separação de concerns, ainda não está claro como alcançar isso em termos de abstrações orientadas a aspectos. O desafio fica ainda maior ao usar aspectos para mais crosscutting concerns específicos a domínios, como concerns do agente. Entretanto, conforme mencionado anteriormente, a experiência com prática de projetos em desenvolvimento de software orientado a aspectos é bastante limitada e ficou restrita a crosscutting concerns gerais.

- **A necessidade de suporte de avaliação**:

A utilidade de novos paradigmas de desenvolvimento e das práticas de projetos associadas pode ser avaliada por meio de estudos empíricos. As métricas de software [68, 118] são usadas em estudos empíricos como indicadores dos pontos fortes e fracos das abordagens estudadas. Muitas dessas métricas foram propostas [46, 68], usadas e, às vezes, empiricamente validadas [15, 158]. Elas incluem várias linhas de código [68], métricas de CK [46] etc. Contudo, as métricas disponíveis na literatura não se dedicam a DSOA.

- **A necessidade de estudos empíricos quantitativos:**

Muitos estudos empíricos que envolvem a aplicação da tecnologia de aspectos são baseados em uma avaliação qualitativa [82, 115, 138, 141, 226]. Normalmente, para investigar a qualidade das soluções orientadas a aspectos, esses estudos usavam conceitos ainda não bem compreendidos, como *conectividade* e *composabilidade* [115]. Apesar de o objetivo desses conceitos ser capturar várias facetas de manutenção e reutilização de software, suas definições são amplamente vagas e bem fundamentadas na intuição dos proponentes. Há então a necessidade de realizar estudos quantitativos em DSOA baseados em um modelo de qualidade estruturado e métricas efetivas.

Esta tese oferece contribuições nesses quatro campos. Ela oferece soluções e estudos empíricos que lidam com crosscutting concerns específicos a SMAs desde a etapa arquitetural até as etapas de avaliação e implementação.

2.3 Resumo

Um *concern* é uma parte do problema que queremos tratar como uma unidade conceitual única na solução do software. Alguns exemplos de concerns em sistemas multiagentes são *tipo de agente*, *autonomia, adaptação, conhecimento, mobilidade* etc. O princípio da *separação de concerns* precisa ter suporte em todas as fases do ciclo de vida do software. A reusabilidade e a manutenibilidade de sistemas de software dependem muito da capacidade de as abstrações, os mecanismos de composição e os métodos associados oferecem suporte à separação explícita dos concerns durante as etapas de implementação e projeto.

Os padrões de software são abstrações importantes para auxiliar a separação de concerns e a construção de softwares de alta qualidade. Os *padrões arquiteturais* definem a estrutura básica de uma arquitetura e dos sistemas que a implementam. No nível arquitetural, as abstrações precisam oferecer suporte à separação de cada concern de interesse em um determinado componente. Os *padrões de projeto* são mais orientados a problemas do que os padrões arquiteturais e são aplicados nas etapas finais do projeto. Cada padrão de projeto e seus participantes internos na solução lidam com concerns específicos. Os padrões de projeto e as abstrações OO podem não ser capazes de capturar alguns concerns especiais, chamados de *crosscutting concerns*.

Sem os meios apropriados para a separação e a modularização, os crosscutting concerns tendem a ficar espalhados e entrelaçados com outros concerns. As conseqüências naturais são uma menor compreensibilidade, uma menor manutenibilidade e menos reusabilidade dos artefatos do software. A Programação orientada a aspectos (POA) é uma tecnologia em evolução que oferece suporte a um novo tipo de separação de concerns no nível do código-fonte. A POA é evolucionária porque considera as melhorias reconhecidas na separação de concerns proporcionadas

pelas tecnologias anteriores, principalmente a programação orientada a objetos. A idéia central é que embora as abstrações e os mecanismos de composição do paradigma de objetos sejam extremamente úteis, eles são inerentemente incapazes de modularizar crosscutting concerns. Este capítulo tratou dos conceitos básicos relacionados a aspectos. Os aspectos consistem em métodos e atributos ordinários, *pointcut definitions*, *inter-type declarations* e *advice*, sendo que os advices podem ser before, after ou around advice.

A POA e a maioria dos trabalhos existentes sobre a separação avançada de concerns tratavam do fenômeno de crosscutting no nível da implementação (Seção 2.2.3). Esta tese se concentra nos concerns específicos a agentes que não sejam crosscutting concerns clássicos. Contudo, a fim de entender os concerns associados ao desenvolvimento de SMAs, precisamos antes defini-los. O próximo capítulo apresenta o TAO, um framework conceitual que define os concerns de SMAs. Ele também discute por que a separação desses concerns nem sempre é modularizada com base em padrões e abstrações OO.