6
Mecanismo de Tratamento de Exceções Sensível ao Contexto

Este capítulo apresenta a proposta de nosso mecanismo de tratamento de exceções sensível ao contexto. Tal mecanismo foi desenvolvido de acordo com o modelo e a arquitetura apresentados nos capítulos anteriores. Inicialmente, introduzimos o uso de aspectos para um melhor entendimento da implementação do mecanismo. Posteriormente, apresentamos seu projeto detalhado. A seguir, apresentamos alguns detalhes de implementação do mecanismo usando MoCA.

6.1.
Uso de Aspectos no Mecanismo

No Capítulo 5, utilizamos componentes aspectuais para especificar elementos de nossa arquitetura de tratamento de exceções sensível ao contexto. Os componentes aspectuais são implementados como um conjunto de aspectos e classes auxiliares. O conceito de aspectos é introduzido a seguir.

Aspecto é o termo usado para denotar uma abstração 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. 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[6] e adendos[7]; a estrutura estática é alterada através de declarações intertipos[8].

[6] Do inglês: *pointcuts*.
[7] Do inglês: *advices*.
[8] Do inglês, *intertype declarations.*

Para definir conjuntos de junção, é necessária a introdução do conceito de pontos de junção[9]. *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.

[9] Do inglês, *join points*.

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). A notação de ASideML para representação de aspectos no nível de projeto é introduzida na próxima seção.

6.2. Projeto Detalhado dos Componentes

Esta seção apresenta o projeto detalhado de nosso mecanismo de tratamento de exceções sensível ao contexto. As Figuras de 13 a 18 apresentam o projeto de cada um dos componentes através da linguagem de modelagem ASideML (Chavez & Lucena, 2001; Chavez, 2004). 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.

6.2.1. Componente ContextualException

A Figura 13 ilustra o projeto detalhado do componente `ContextualException`.

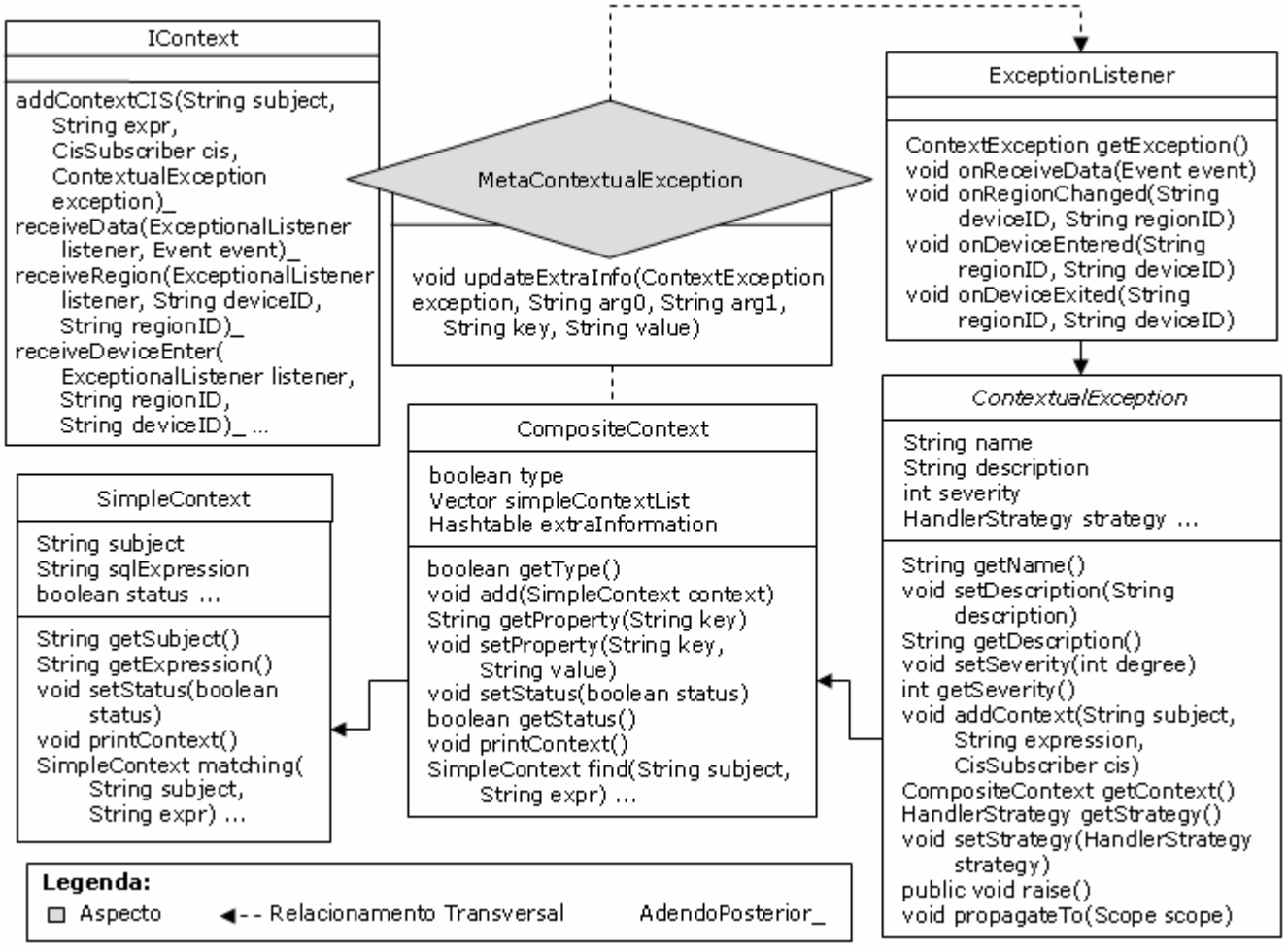

Figura 13. Projeto do Componente ContextualException

O componente `ContextualException` foi implementado através das classes `ContextualException`, `SimpleContext`, `CompositeContext`, `ExceptionListener`, e pelo aspecto `MetaContextualException` (Figura 13).

A classe abstrata `ContextualException` permite a especificação de exceções contextuais. Em outras palavras, esta classe permite a especificação de exceções que possuem informações de contexto associadas, as quais correspondem às situações excepcionais que caracterizam uma exceção. O método `addContext` adiciona à especificação de uma exceção as informações de contexto excepcional que caracterizam a ocorrência da exceção. Sempre que a informação de contexto excepcional for adicionada a uma exceção, o contexto excepcional relacionado é registrado como um contexto de interesse no *middleware* MoCA.

As classes `CompositeContext` e `SimpleContext` permitem definir contextos excepcionais complexos e simples, respectivamente. Os contextos complexos são formados por um ou mais contextos simples. Por exemplo, considerando os contextos *C1 = (dispositivo x mudou de região)* e *C2 = (dispositivo x está com nível de bateria < 20)*, um contexto complexo pode ser especificado como a conjunção dos contextos simples C1 e C2. A especificação de que um contexto complexo é formado por uma conjunção ou uma disjunção de contextos simples é realizada através do atributo `type` de `CompositeContext`. Este atributo permite especificar relações `OR` e `AND` entre contextos. O método `find(subject, expression)` da classe `CompositeContext` permite verificar se um contexto complexo é formado por contextos simples (atributo `simpleContextList`).

A classe `ExceptionListener` consiste em um observador de eventos[10] que detecta eventos publicados por MoCA previamente subscritos pelos usuários da API como sendo de interesse para o tratamento de exceções. Para detectar tais eventos, a classe `ExceptionListener` declara a implementação de métodos de interfaces da API de MoCA automaticamente invocados pelo *middleware* quando os eventos relacionados a uma subscrição são executados.

O aspecto `MetaContextualException` implementa a interface transversal `IContext`, que entrecorta: (i) a adição de contextos excepcionais a fim de subscrever interesses excepcionais ao MoCA (especificados pela classe

[10] Do inglês, *listener*.

`ContextualException`); e (ii) a ocorrência de eventos (especificados pela classe `ExceptionListener`) que indicam ou não se os contextos caracterizam uma exceção contextual. Além disso, o aspecto `MetaContextualException` mantém atualizados os atributos de contextos excepcionais. Além disso, o aspecto é responsável pelo levantamento das exceções contextuais a fim de dar início à atividade de busca sensível ao contexto por tratadores de uma exceção.

6.2.2. Componente HandlingScope

Em nosso modelo de tratamento de exceções sensível ao contexto, são definidos escopos para um dispositivo, um grupo de dispositivos, um servidor e uma região (seção 4.2). A Figura 14 apresenta o projeto detalhado do componente `HandlingScope`, que implementa o modelo de escopos do mecanismo.

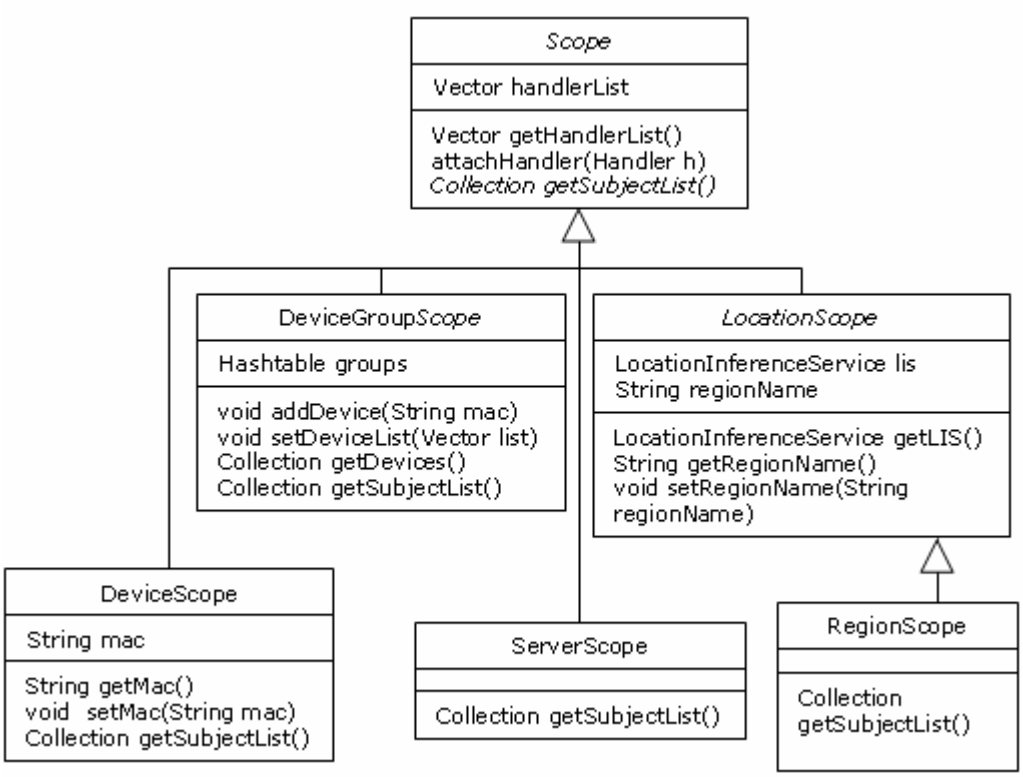

Figura 14. Projeto do Componente HandlingScope

A classe abstrata `Scope` define métodos e atributos comuns a todos os escopos no mecanismo de tratamento de exceções. O método `attachHandler` permite associar um tratador a um escopo. O método `getHandlerList` permite

obter a lista de tratadores para um dado escopo. O método abstrato `getSubjectList` permite recuperar a lista de dispositivos relacionados ao escopo.

A classe abstrata `LocationScope` especifica os escopos que precisam utilizar o serviço de localização de MoCA para identificar dinamicamente os dispositivos que pertencem ao escopo. O atributo `lis` desta classe contém uma referência para o serviço de localização em MoCA, que permite obter informações como a lista de dispositivos de uma região e a localização atual de um dispositivo. Desta forma, a classe `RegionScope`, subclasse de `LocationScope`, utiliza o método `getDevices(String region)` da API de MoCA para identificar os dispositivos que estão em uma região especificada pelo argumento do método.

As classes `DeviceScope` e `ServerScope` especificam escopos simples para um dispositivo móvel e um servidor de aplicação, respectivamente. A classe `DeviceGroupScope` permite que a aplicação especifique um grupo de dispositivos que representam o escopo. O escopo de grupo está diretamente relacionado ao domínio da aplicação e não à localização física dos dispositivos.

6.2.3. Componente Handler

A Figura 15 apresenta o projeto detalhado do componente `Handler`.

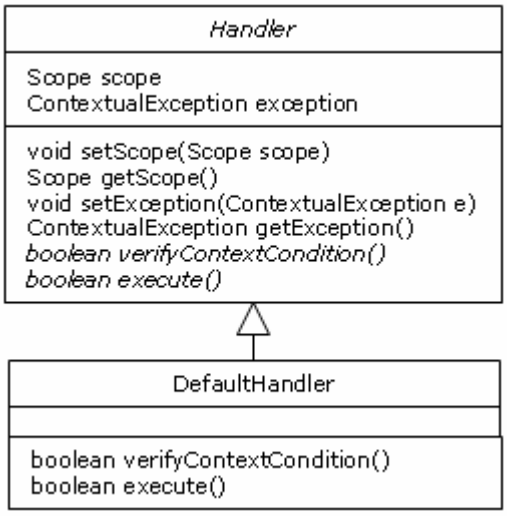

Figura 15. Projeto do Componente Handler

A classe abstrata `Handler` deve ser estendida sempre que uma aplicação necessita implementar um tratador de exceção. São importantes os seguintes

métodos da classe `Handler`: (i) `setException`, associa uma exceção ao tratador, (ii) `setScope`, associa um escopo ao tratador, (iii) `verifyContextCondition`, verifica se as condições de contexto necessárias à execução do tratador são satisfeitas, e (iv) `execute`, executa o tratamento da exceção associada. Os métodos `verifyContextCondition` e `execute` são especificados por aplicações.

Se ao final de uma busca por tratadores em todos os níveis, não for encontrado nenhum tratador definido pela aplicação, o mecanismo executa um tratador genérico previamente definido para todas as exceções, cuja tarefa é apenas tornar acessível aos usuários as informações sobre a ocorrência de uma exceção contextual. O tratador genérico definido pelo mecanismo é implementado através da classe `DefaultHandler`, uma subclasse da classe `Handler`.

6.2.4. Componente ExceptionHandlingStrategy

A Figura 16 apresenta o projeto detalhado do componente `ExceptionHandlingStrategy`.

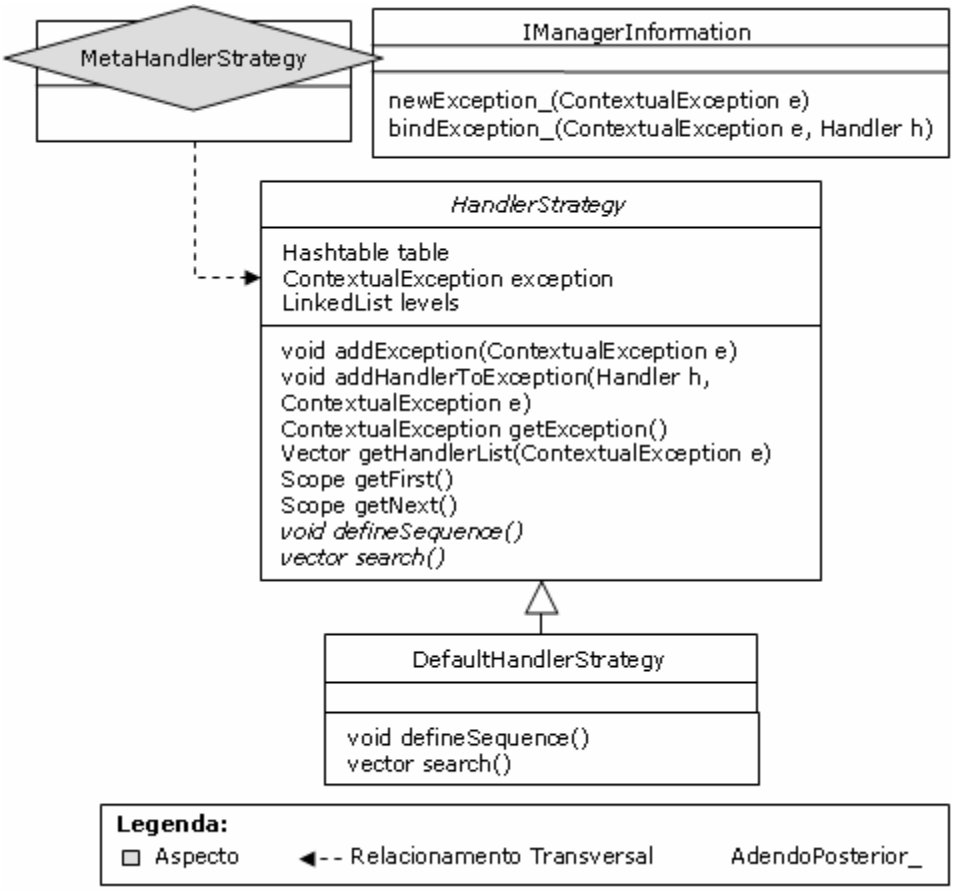

Figura 16. Projeto do Componente ExceptionHandlingStrategy

O componente `ExceptionHandlingStrategy` mantém o controle de todas as exceções contextuais definidas pela aplicação e seus possíveis tratadores. Para isso, o componente define o aspecto `MetaHandlerStrategy`, que entrecorta: (i) a instanciação de exceções contextuais (conjunto de junção `newException`), e (ii) a associação entre tratadores e exceções (conjunto de junção `bindException`), a fim de manter atualizada a relação entre exceções contextuais e tratadores no atributo `table` de classe `HandlerStrategy`.

O mecanismo pode realizar a associação dinâmica entre uma ocorrência de exceção e seu tratador, considerando a estratégia de busca especificada pela aplicação e as restrições de contexto especificadas para os tratadores. Através da classe abstrata `HandlerStrategy`, as aplicações podem definir novas estratégias de tratamento de exceções, especificando os seguintes métodos: (i) `defineSequence`, que permite especificar a seqüência de prioridade entre os diversos tipos de escopo, utilizada pelo mecanismo durante a busca de tratadores, e (ii) `search`, o algoritmo de busca de tratadores. Caso as aplicações não necessitem utilizar estratégias específicas, o mecanismo define uma estratégia geral para todas as exceções através da classe `DefaultHandlerStrategy`.

6.2.5. Componente PropagationManager

O componente `PropagationManager` possui a função de manter uma infra-estrutura *publish-subscribe* a fim de permitir a propagação de exceções contextuais (Seção 5.3). Desta forma, a implementação deste componente segue o funcionamento de uma arquitetura *publish-subscribe*, baseada em clientes e servidores. A Figura 17 apresenta o projeto detalhado de `PropagationManager`.

O componente `PropagationManager` define as classes `Device`, `Client` e `Server` que instanciam os componentes `ECIClient` e `ECIServer`, responsáveis na arquitetura *publish-subscribe* de MoCA pela realização da subscrição e publicação de eventos, respectivamente. Os métodos abstratos `init` e `propagate` da classe `Device` são especificados pelo mecanismo nas classes `Client` e `Server` para a inicialização dos componentes e propagação de exceções.

O componente `PropagationManager` também define os aspectos abstratos `MetaClientDevice` e `MetaServerDevice`, que devem ser estendidos por um usuário do mecanismo para a especificação do conjunto de junção referente ao momento de inicialização de sua aplicação (conjunto de junção `initialization`) a fim de que o mecanismo de tratamento de exceções configure os parâmetros necessários à inicialização de clientes e servidores em MoCA.

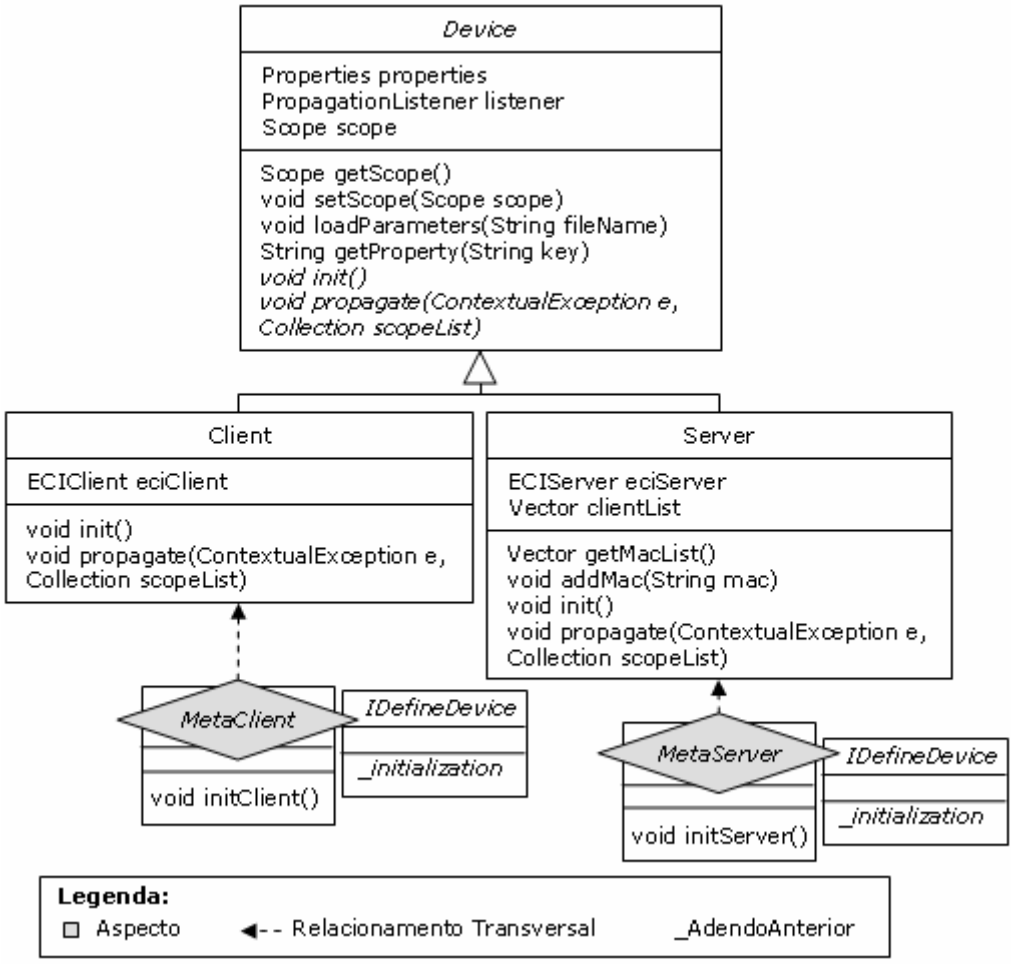

Figura 17. Projeto do Componente PropagationManager

6.2.6. Componente ExceptionPropagation

A Figura 18 apresenta o projeto detalhado do componente `ExceptionPropagation`, o elemento central na tarefa de propagação de exceções de modo totalmente transparente às aplicações. Tal componente se comunica com o componente `PropagationManager` (Seção 6.2.3) em duas direções: (i) na

detecção de exceções contextuais propagadas, e (ii) na solicitação de propagação de exceções.

Na detecção de exceções contextuais propagadas, o observador de eventos `PropagationListener` detecta o evento de propagação de uma exceção enviada por outro dispositivo e é entrecortado pelo aspecto `MetaPropagation` para a realização do tratamento da exceção recebida.

Na solicitação de propagação de exceções, o aspecto `MetaPropagation` detecta o final da execução dos tratadores, entrecortando o método `execute` da classe `Handler`, a fim de verificar se existe a necessidade de realizar a propagação de uma exceção (adendo `handlerExecution_`). Se um tratador é executado com sucesso e o escopo associado ao ele referencia outros dispositivos, o método `automaticPropagation` é executado. Caso nenhum tratador tenha sido executado com sucesso, o método `propagateNextLevel` é executado.

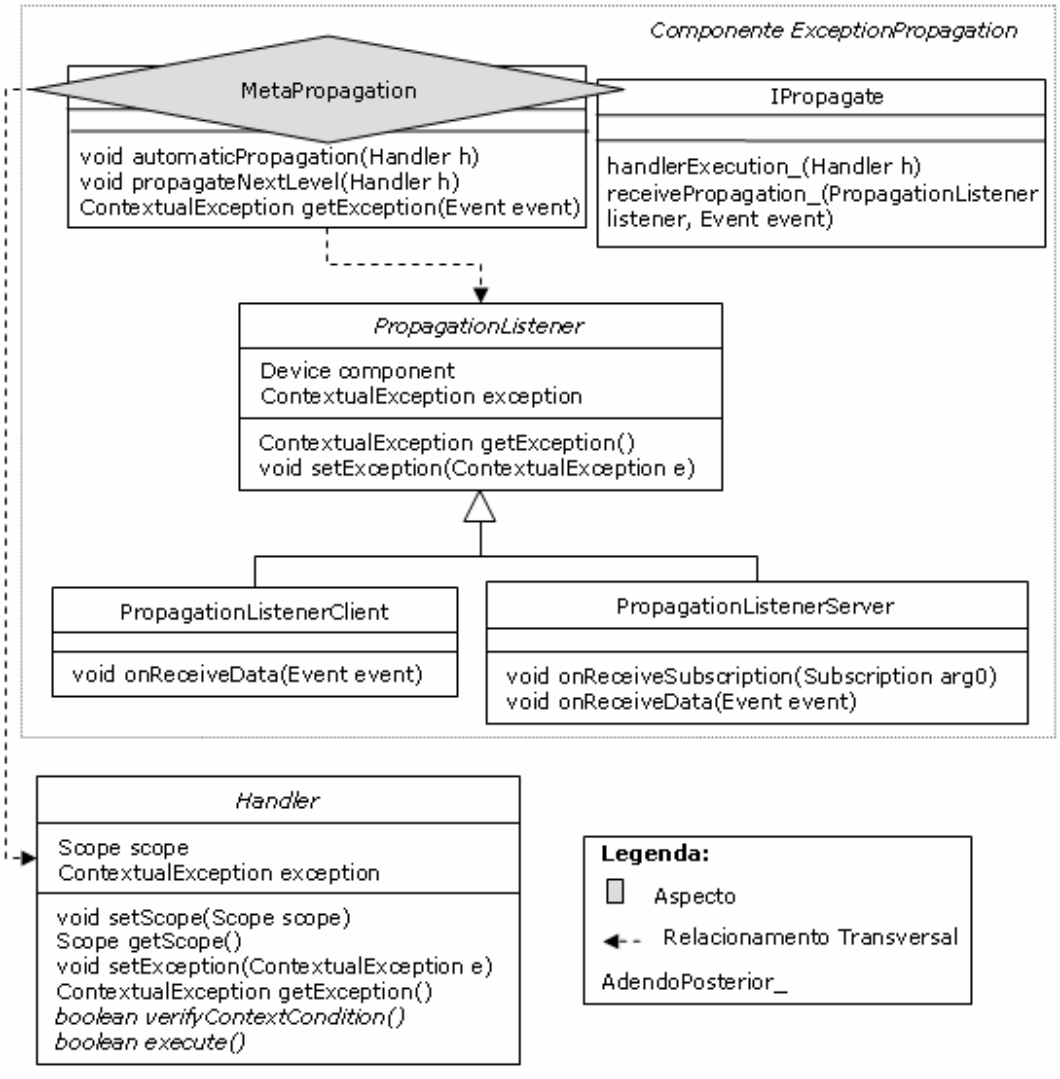

Figura 8. Projeto do Componente ExceptionPropagation

6.3.
Implementação dos Componentes usando MoCA

O mecanismo de tratamento de exceções sensível ao contexto foi implementado fazendo uso das APIs de MoCA. A seguir, mostramos alguns exemplos de utilização destas APIs na implementação das funcionalidades independentes de aplicação do mecanismo.

Por exemplo, o código da Figura 19 mostra que, sempre que há a adição de um contexto excepcional a uma exceção contextual através do método `addContext` (linha 3), é realizada a subscrição do interesse excepcional através do método `subscribe` da API de MoCA (linha 10). Em outras palavras, no adendo (linhas de 7 a 13) associado ao conjunto de junção `addContextCIS` (linhas 1 a 6), o mecanismo realiza as subscrições de interesses excepcionais (linhas 10 a 12).

```
1: pointcut addContextCIS(String subject, String expression,
2:    CisSubscriber cis, ContextualException exception):
3:    (execution(void ContextualException.addContext(String,
4:    String, CisSubscriber)) && args(subject, expression, cis)
5:    && target(exception));
6:
7: after(String subject, String expression, CisSubscriber cis,
8: ContextualException exception) :
9:    addContextCIS(subject, expression, cis, exception) {
10:         Topic topic = cis.subscribe(subject, expression);
11:         cis.addListener((EventListener)new
12:         ExceptionalListener(exception), topic);
13: }
```

Figura 19. Adição de Contextos Excepcionais

As APIs de MoCA também são utilizadas na especificação de observadores de eventos que detectam a ocorrência de contextos excepcionais nos dispositivos. A Figura 20 apresenta a especificação do observador de eventos `ExceptionalListener` para a detecção de um contexto excepcional. O método `onReceiveData` é automaticamente executado quando ocorre um evento com informações de contexto relacionadas ao CIS ou aos serviços assíncronos da API MoCA (linha 3). Por sua vez, o método `onRegionChanged` é executado automaticamente quando um dispositivo muda de região (linha 5). Finalmente, os métodos `onDeviceEntered` e `onDeviceExited` são executados automaticamente

quando é detectada a entrada de um dispositivo em uma região (linha 7) e a saída de um dispositivo de uma região (linha 9), respectivamente.

```
1: public class ExceptionalListener implements
2:          EventListener, DeviceListener, RegionListener {
3:  public void onReceiveData(Event event) {
4:  ... )
5:  public void onRegionChanged(String deviceID, String regionID){
6:  ... }
7:  public void onDeviceEntered(String regionID, String deviceID){
8:  ... }
9:  public void onDeviceExited(String regionID, String deviceID) {
10: ... }
11: }
```

Figura 20. Observador de Eventos para Detecção de Contextos

A API de MoCA também é utilizada para a recuperação dinâmica dos dispositivos pertencentes a um determinado escopo de localização. Por exemplo, a Figura 21 mostra a definição do escopo de região. O método `getSubjectList` retorna todos os dispositivos que estão em uma região no momento em que ocorreu a exceção contextual (linhas de 4 a 9). O método `getRegionName` permite identificar a região associada ao escopo (linha 6). Esta região pode ser a região em que o dispositivo se encontra ou uma outra região especificada pela aplicação. A identificação dos dispositivos que estão presentes na região anterior é realizada através do método `getDevices` da API MoCA (linha 7).

```
1: public class RegionScope extends LocationScope {
2:    ...
3:    public static RegionScope getInstance(){...}
4:    public Collection getSubjectList() {
5:      ...
6:      String region = getRegionName();
7:      devices = getLIS().getDevices(region);
8:      return Arrays.asList(devices);
9:    }
10: }
```

Figura 21. Recuperação dos Dispositivos do Escopo de Região

As APIs de MoCA também são utilizadas para especificar a infra-estrutura de propagação de exceções do mecanismo. Diversas tarefas devem ser realizadas: (i) a subscrição de interesse em eventos que correspondem a exceções propagadas, (ii) a especificação de observadores de eventos para tratar a propagação de exceções contextuais, e (iii) inicialização dos componentes cliente e servidor na arquitetura *publish-subscribe*.

A Figura 22 apresenta um trecho de código da definição da classe `Client`. No construtor de `Client`, é instanciado um observador de eventos específico para a detecção da propagação de exceções contextuais (linha 4). No método `init`, o cliente ECI é iniciado (linhas 8 e 9) e, em seguida, subscreve-se o interesse em eventos de propagação de exceções (linha 10). O observador de eventos instanciado anteriormente é associado ao cliente ECI (linha 11). O método `propagate` (linhas de 13 a 19) utiliza a arquitetura *publish-subscribe* para enviar as exceções contextuais aos diversos escopos, publicando esta informação aos outros dispositivos (linha 17).

```
1: public class Client extends Device {
2:    private ECIClient eciClient = null;
3:    public Client(){ ...
4:       listener = new PropagationListenerDevice(this);
5:       init();
6:    }
7:    public void init(){ ...
8:       eciClient = new ECIClient(
9:          ECIConstants.TCP, serverAddress, clientAddress);
10:      Topic topic = eciClient.subscribe(getProperty("EH_MAC"));
11:      eciClient.addListener(listener, topic);
12:   }
13:   public void propagate(ContextualException e, Collection
14:   scopes){. . .
15:      for (Iterator i = scopes.iterator(); i.hasNext();){
16:         String subject = (String)i.next();
17:         this.eciClient.publish(subject, e);
18:      }
19:   }
20: }
```

Figura 22. Propagação de Exceções

As APIs de MoCA podem ser utilizadas para a implementação de funcionalidades dependentes de aplicação. Como exemplo, é possível realizar consultas síncronas a informações de contexto na implementação do método abstrato `verifyContextCondition` da classe `Handler`, que verifica as condições de contexto dos tratadores de exceções. Trechos de código referentes a funcionalidades dependentes de aplicação são apresentados no próximo capítulo.