Projeto e Implementação de um Mecanismo de Tratamento de Exceções Coordenadas para Arquiteturas de Componentes de Serviços

Este exemplar corresponde à redação final da Dissertação devidamente corrigida e defendida por Douglas Siqueira Leite e aprovada pela Banca Examinadora.

Campinas, 27 de agosto de 2010.

Cecília Mary Fischer Rubira

Profa. Dra. Cecília Mary Fischer Rubira - IC/UNICAMP (Orientadora)

Dissertação apresentada ao Instituto de Computação, UNICAMP, como requisito parcial para a obtenção do título de Mestre em Ciência da Computação.

FICHA CATALOGRÁFICA ELABORADA PELA BIBLIOTECA DO IMECC DA UNICAMP
Bibliotecária: Maria Fabiana Bezerra Müller – CRB8 / 6162

Leite, Douglas Siqueira
L536p Projeto e implementação de um mecanismo de tratamento de exceções coordenadas para arquiteturas de componentes de serviços/ Douglas Siqueira Leite-- Campinas, [S.P. : s.n.], 2010.

Orientador : Cecília Mary Fischer Rubira.
Dissertação (mestrado) - Universidade Estadual de Campinas, Instituto de Computação.

1.Arquitetura orientada a serviços. 2.Tratamento de exceções. 3.Tolerância a falha. I. Rubira, Cecília Mary Fischer. II. Universidade Estadual de Campinas. Instituto de Computação. III. Título.

Título em inglês: Design and implementation of a coordinated exception handling mechanism for service component architecture

Palavras-chave em inglês (Keywords): 1. Service-oriented architecture. 2. Exception handling. 3. Fault tolerance.

Área de concentração: Sistemas de Informação

Titulação: Mestre em Ciência da Computação

Banca examinadora: Profª. Drª. Cecília Mary Fischer Rubira (IC – UNICAMP)
Prof. Dr. Fernando José Castor de Lima Filhos (CI – UFPE)
Prof. Dr. Luiz Eduardo Buzato (IC - UNICAMP)

Data da defesa: 27/08/2010

Programa de Pós-Graduação: Mestrado em Ciência da Computação

TERMO DE APROVAÇÃO

Dissertação Defendida e Aprovada em 27 de agosto de 2010, pela Banca examinadora composta pelos Professores Doutores:

Prof. Dr. Fernando José Castor de Lima Filho
Centro de Informática / UFPE

Prof. Dr. Luiz Eduardo Buzato
IC / UNICAMP

Profª. Drª. Cecília Mary Fischer Rubira
IC / UNICAMP

Instituto de Computação
Universidade Estadual de Campinas

Projeto e Implementação de um Mecanismo de Tratamento de Exceções Coordenadas para Arquiteturas de Componentes de Serviços

Douglas Siqueira Leite

Agosto de 2010

Banca Examinadora:

- Profa. Dra. Cecília Mary Fischer Rubira - IC/UNICAMP (Orientadora)
- Prof. Dr. Fernando José Castor de Lima Filho - Centro de Informática/UFPE
- Profa. Dra. Eliane Martins - IC/UNICAMP
- Prof. Dr. Luiz Eduardo Buzato - IC/UNICAMP (Suplente)
- Prof. Dr. Ivan Luiz Marques Ricarte - FEEC/UNICAMP (Suplente)

Resumo

Arquitetura Orientada a Serviços (*Service-Oriented Architecture* - SOA) é um modelo arquitetural que visa melhorar a eficiência, agilidade e a produtividade de aplicações empresariais através do uso de serviços e composições de serviços, as quais podem ser executadas tanto de forma síncrona quanto assíncrona. Diferentes tecnologias de *software* podem ser usadas para implementar SOA, tais como *Web services* e Arquitetura de Componentes de Serviços (*Service Component Architecture* - SCA). A primeira é baseada em padrões XML, ao passo que a segunda provê um modelo de componentes para implementação de serviços e composições de serviços. Em particular, quando composições de serviços assíncronos são executadas, um ou mais erros podem ocorrer concorrentemente nos diferentes serviços, possivelmente ao mesmo tempo, afetando a dependabilidade da composição. Dessa forma, mecanismos de tolerância a falhas são necessários a fim de prevenir que um defeito se manifeste na composição. Neste trabalho, apresentamos o projeto e implementação de um mecanismo de tratamento de exceções coordenadas para arquiteturas orientadas a serviços que permite a criação de composições de serviços assíncronos tolerante a falhas de uma forma flexível. Mais especificamente, nossa solução é baseada em um mecanismo de tratamento de exceções global, definido pelo modelo *Guardian*, já que este oferece uma solução mais geral e flexível quando comparado com outras abordagens, tais como soluções baseadas em ações atômicas coordenadas. Nosso *framework*, denominado *Guardian-SCA*, foi implementado como parte do projeto *Apache Tuscany SCA*, usando o modelo de extensão do Tuscany e programação orientada a aspectos, aumentando assim a flexibilidade do *framework*.

Abstract

Service-Oriented Architecture (SOA) is an architectural model that aims to enhance the efficiency, agility, and productivity of an enterprise by structuring services in terms of services compositions, which can be executed either synchronously or asynchronously. Different software technologies can be used to implement SOA, such as Web services and Service Component Architecture (SCA). The former is based on XML-based standards, while the latter provides a component model for implementing services and service compositions. In particular, when asynchronous services compositions are executed, one or more errors can occur concurrently, possibly at same time, affecting the composition's dependability. In this way, fault tolerance mechanisms are necessary in order to prevent the services compositions from reaching a failure state. In this work, we present the design and implementation of a coordinated exception handling mechanism, applicable to service-oriented architectures, which allows the creation of fault-tolerant asynchronous service compositions. More specifically, our solution is based on a global exception handling mechanism defined by the Guardian model, since it is more general and flexible when compared to other approaches, like CA Actions-based solutions. Our framework, named Guardian-SCA, was implemented as a part of the Apache Tuscany SCA project, using the Tuscany extension model and aspect-oriented programming with the aim to increase the framework's flexibility.

Sumário

Resumo **vii**

Abstract **ix**

1 Introdução **1**
1.1 Contexto 1
1.2 Definição do Problema 2
1.3 Solução Proposta 3
1.4 Trabalhos Relacionados 4
1.5 Estrutura da Dissertação 6

2 Fundamentação Teórica **7**
2.1 Arquitetura Orientada a Serviços 7
2.2 Arquitetura de Componentes de Serviços 9
2.3 Tolerância a Falhas em Sistemas de Software 12
2.3.1 Conversação 13
2.3.2 Transação 15
2.3.3 Ações Atômicas Coordenadas 15
2.3.4 Classificação dos Mecanismos de Tratamento de Exceções em Sistemas Distribuídos 18
2.4 Modelo Guardian 19
2.4.1 Contexto Excepcional 20
2.4.2 Modelo de Tratamento Exceções Globais 21
2.4.3 Guardian Primitives 23
2.4.4 Regras de Recuperação 25
2.5 Considerações Finais 28

3 Suporte para Tratamento de Exceções Coordenadas no Apache Tuscany SCA **31**
3.1 Apache Tuscany SCA 31

3.1.1 Ambiente de Execução do Tuscany Java SCA 32
3.1.2 Arquitetura de Invocação do Tuscany 39
3.2 Guardian-SCA: Modelo Guardian Aplicado ao Apache Tuscany SCA . . . 41
3.2.1 Guardian Group 41
3.2.2 Guardian Primitives 42
3.2.3 Contextos Excepcionais e Exceções Globais 43
3.2.4 Regras de Recuperação 43
3.2.5 Árvore de Resolução de Exceções 46
3.2.6 Guardian Members 49
3.3 Considerações Finais 51

4 Estudo de Caso: Exemplo de Utilização do modelo Guardian-SCA 53
4.1 Estrutura da Aplicação 54
4.2 Utilização do Modelo Guardian-SCA 57
4.3 Definição das Regras de Recuperação 63
4.4 Definição as Árvores de Resolução 67
4.5 Execução da Aplicação 67
4.6 Considerações Finais 71

5 Modelo de Programação Orientada a Aspectos para o Guardian-SCA 73
5.1 Programação Orientada a Aspectos 73
5.2 Tratamento de Exceções e Programação Orientada a Aspectos 75
5.2.1 Catálogo de Cenários de Tratamento de Exceções 76
5.3 Modelo Guardian-SCA com Aspectos 80
5.4 Considerações Finais 83

6 Conclusões e Trabalhos Futuros 85

Bibliografia 87

Lista de Tabelas

3.1 Características associadas aos aspectos configuráveis de uma implementação 38

4.1 Exceções globais associadas a cada contexto e as situações excepcionais abstraídas por essas. 60

5.1 Cenários de tratamento de exceções de acordo com a classificação proposta. 79
5.2 Descrição dos cenários de tratamento de exceções. 79
5.3 Pontuação para os fatores que dificultam a aspectização do comportamento excepcional de um cenário. 79

Lista de Figuras

2.1 Uso de serviços em Arquitetura Orientada a Serviços. 8
2.2 Relação entre serviços e componentes num contexto SOA e SCA, respectivamente. 10
2.3 Representação visual das partes que compões um componente. 10
2.4 Relacionamentos *promotion* e *wire* entre componentes e composições. . . . 11
2.5 Exemplo de um composição descrita através da Linguagem de Definição de Componentes de Serviços. 12
2.6 Representação gráfica de uma conversação. 14
2.7 Representação gráfica de uma transação. 15
2.8 Representação gráfica de uma CA action. 16
2.9 Diagrama de estados demonstrando os estados assumidos por um sistema ao utilizar um mecanismo de CA actions. 17
2.10 Visão conceitual do modelo *Guardian*. 20
2.11 Estrutura geral de uma regra de recuperação para *single exception*. 26
2.12 Exemplo de uma regra de recuperação para *single exception*. 27
2.13 Estrutura geral de uma regra de recuperação para exceções concorrentes. . 28
2.14 Exemplo de árvore de resolução de exceções. 29

3.1 Estrutura principal do ambiente de execução do Tuscany SCA. 32
3.2 *Pseudo-schema* de uma *intent*. 35
3.3 Exemplo de *intent* descrevendo uma política de autenticação. 36
3.4 Exemplo de um conjunto de política descrevendo uma política de autenticação com uso do *WS-Security*. 36
3.5 Relacionamento entre implementação, instância da implementação, componente e *component type*. 38
3.6 *Pseudo-schema* de um *component type*. 39
3.7 Exemplo de um *component type*. 39
3.8 Criação dos *runtime wires* na comunicação referência/serviço entre dois componentes. 40
3.9 *Pseudo-schema* para o tipo de implementação *implementation.guardian*. . . 42

3.10 *Guardian Primitives* representadas como uma interface de serviço. 42
3.11 Dependências entre `Context`, `GlobalException` e `java.lang.Exception`. 43
3.12 *Pseudo-schema* para a definição das regras de recuperação. 44
3.13 Máquina de estado para representação das expressões regulares válidas. . . 45
3.14 *Pseudo-schema* para a definição das árvores de resolução de exceções. . . . 46
3.15 Equação que expressa o funcionamento do algoritmo de Bender e Farach para computação do menor ancestral comum em uma árvore. 47
3.16 Árvore de resolução de exceções, sendo `N0` a raiz e `N1` a `N6` nodos intermediários. 48
3.17 Definição da *intent* `guardianExceptionHandling` para abstração dos *guardian members*. 50
3.18 Representação interna do *guardian member* nas cadeias de invocação do Tuscany SCA. 50

4.1 Estrutura da aplicação de servidor com N réplicas. 54
4.2 Estrutura da aplicação de servidor com N réplicas utilizando o Tuscany SCA com a extensão do modelo *Guardian*. 55
4.3 Descrição da aplicação de servidor com N réplicas via SCDL. 56
4.4 `Node.java`: Interface de serviço dos componentes `Participant`. 57
4.5 Estrutura geral da utilização do modelo *Guardian* na classe `NodeImpl`. . . 58
4.6 Estrutura mais detalhada da classe `NodeImpl`. 59
4.7 Descrição do método `execute()`. 61
4.8 Descrição do método `primaryService()`. 62
4.9 Descrição do método `backupServiceNodeImpl`. 63
4.10 Descrição da regra `Rule1`. 64
4.11 Descrição da regra `Rule2`. 65
4.12 Descrição da regra `Rule3`. 65
4.13 Descrição da regra `Rule4`. 66
4.14 Árvore de resolução de exceções para aplicação de servidor com N réplicas. 67

5.1 Processo de combinação em que o combinador é descrito na forma de um compilador. 74
5.2 Exemplo de um aspecto descrito em AspectJ. 75
5.3 Categorias utilizadas para a classificação do código de tratamento de erros. 77
5.4 Principais elementos adicionados na extensão `implementation.guardian` para suporte a AspectJ e sua interação com uma classe que representa a implementação de um componente. 81

Capítulo 1

Introdução

1.1 Contexto

Arquitetura Orientada a Serviços (*Service-Oriented Architecture* - SOA) é um modelo arquitetural que visa melhorar a eficiência, agilidade e a produtividade de aplicações empresariais através do uso de serviços e composições de serviços [37]. Um serviço é definido como uma unidade distribuída, auto-contida, composta de dois elementos fracamente acoplados: especificação e implementação. Esta última define a lógica do serviço, de forma concreta, em alguma linguagem de programação, enquanto a primeira provê uma interface abstrata para descrição das operações do serviço. Serviços podem ser agrupados e executados em uma ordem específica, tanto de forma síncrona quanto assíncrona, resultando em uma composição de serviços.

Diferentes tecnologias de *software* podem ser usadas para implementar SOA, sendo *Web services* [1,27,28] uma das mais populares. Tal tecnologia baseia-se em uma série de padrões XML, tais como *Simple Object AccessProtocol* (SOAP) e *Web Service Definition Language* (WSDL). SOAP define um protocolo de comunicação para troca de informação entre serviços, ao passo que WSDL é utilizada na especificação da interface abstrata de um serviço. Composições de serviços baseadas no padrão *Web services* podem ser implementadas utilizando a *Web Services Business Process Execution Language* (WS-BPEL) [2]. Outra forma de se implementar SOA se dá através de uma Arquitetura de Componentes de Serviços (*Service Component Architecture* - SCA) [20], onde um modelo de componentes é definido para implementação de serviços e composição de serviços. A principal vantagem de se adotar uma implementação SOA baseada em SCA está no suporte para interoperabilidade que esta oferece, já que SCA engloba e integra várias tecnologias SOA, tais como *Web services*, WS-BPEL, *Java Message Service* (JMS), *JavaScript Object Notation-Remote Procedure Call* (JSON-RPC), *Common Object Request Broker Architecture* (CORBA), e *Enterprise JavaBeans* (EJB).

Em particular, quando composições de serviços assíncronos são executadas, um ou mais erros podem ocorrer concorrentemente nos diferentes serviços, possivelmente ao mesmo tempo, afetando a dependabilidade[1] da composição. Aumentar a dependabilidade em sistemas SOA é de fundamental importância para que esta tecnologia possa ser usada amplamente em sistema reais complexos e de missão crítica [11, 19]. Uma das formas de se atingir isso se dá através do uso de mecanismos de tolerância a falhas, que consistem em permitir que erros sejam tratados em tempo de execução, evitando que a aplicação atinja um estado inconsistente. Há duas formas de se prover recuperação de erros: por retorno[2] e por avanço[3]. A recuperação de erros por retorno consiste em reverter o estado errôneo dos componentes do sistema para um estado anterior livre de erros, ao passo que, em uma recuperação de erros por avanço, o estado do sistema é transformado para um estado livre de erros através do uso de mecanismos de tratamento de exceções.

Os mecanismos de tratamento de exceções fornecem um arcabouço necessário para estruturar atividades de tolerância a falhas. Como abordado por Souchon et al. [33], um sistema de tratamento de exceções deve prover estruturas de controles que permitam a um programador: lançar exceções; definir tratadores de exceções; bem como meios de colocar o sistema em um estado coerente através de ações a serem executadas quando uma situação excepcional é detectada. Este último pode ser alcançado pela execução da ação associada a um tratador de exceção, ou através da delegação do tratamento para outro tratador, tanto propagando a exceção quanto lançando uma nova.

Contudo, o uso de mecanismos de tratamento de exceções para ambientes em que há uma série de elementos trabalhando de forma concorrente, e possivelmente cooperativamente (como é o caso de composições de serviços assíncronos), requer que algumas características específicas sejam atendidas.

1.2 Definição do Problema

Considerando que, no contexto SOA, nem sempre é possível reverter o estado de serviços, já que estes, por definição, são unidade autônomas e auto-contidas, os mecanismos de recuperação de erro por avanço são mais adequados para prover tolerância a falhas neste tipo de ambiente. Entretanto, um mecanismo de tratamento de exceções para composições de serviços assíncronos deve considerar que: 1) uma exceção lançada por um determinado serviço da composição pode implicar na ocorrência de uma exceção (possivelmente diferente da primeira) em outro serviço da mesma composição; 2) diferentes tipos de exceções podem ser lançadas concorrentemente por diferentes serviços. Assim sendo, um erro ocor-

[1] Do inglês, *dependability*
[2] Do inglês, *backward error recovery*
[3] Do ingês, *forward error recovery*

rido em um serviço deve ser tratado levando em consideração o estado dos demais serviços presentes na composição, uma vez que os serviços executados assincronamente trabalham de forma cooperativa para estabelecer o comportamento da composição. Em particular, um ou mais serviços podem manifestar um erro ao mesmo tempo. Isto significa que, diferentes combinações de tipos de exceções lançadas concorrentemente podem implicar na execução de diferentes combinações de conjuntos de tratadores de erros. Tal cenário de tratamento de erros complexo requer flexibilidade e dinamismo, em tempo de execução, do mecanismo de tratamento de exceções. De tal forma, faz-se necessário o uso de um mecanismo de tratamento de exceções coordenadas, onde um coordenador guia o fluxo excepcional entre os serviços participantes da composição, bem como lida com o problema de múltiplas exceções de diferentes tipos sendo lançadas concorrentemente.

1.3 Solução Proposta

Este trabalho apresenta o projeto e implementação de um mecanismo de tratamento de exceções coordenadas, para arquiteturas orientadas a serviços, que permite a criação de composições de serviços assíncronos tolerante a falhas de uma forma flexível. Mais especificamente, nossa solução é baseada em um mecanismo de tratamento de exceções globais, definido pelo modelo *Guardian* [25,26], onde para cada exceção lançada, uma regra de recuperação determina, em tempo de execução, qual exceção deve ser lançada para cada serviço participante da composição. Nosso *framework*, denominado *Guardian-SCA*, foi implementado como parte do projeto *Apache Tuscany SCA*, usando o modelo de extensão do Tuscany e programação orientada a aspectos, aumentando assim a flexibilidade do *framework*.

As implementações atuais de SCA não oferecem nenhum recurso semelhante. Mais especificamente, na implementação provida pela Apache, via o projeto *Apache Tuscany SCA*, os únicos recursos de tolerância a falhas para a composição de serviços que podem ser utilizados sãos aqueles provido pelo mecanismo de tratamento de exceções oferecido pela linguagem de orquestração e composição WS-BPEL. No entanto, tal mecanismo não oferece suporte adequado para recuperação de erros em composições de serviços assíncronos. Quando um serviço assíncrono lança uma exceção dentro de uma composição WS-BPEL, a invocação dos demais serviços é interrompida assim que a exceção é capturada, e tratada por um único tratador. Não há suporte para tratamento de exceções coordenadas.

1.4 Trabalhos Relacionados

Garcia e Toledo [40] propõem uma arquitetura para *Web service*s tolerante a falhas baseada uma extensão do protocolo *Universal Description, Discovery and Integration* (UDDI) desenvolvida para monitorar os atributos de qualidade de serviço (*Quality of Service* - QoS) em *Web services*. O UDDI é um padrão de protocolo, mantido pela OASIS, que especifica um método para publicar e descobrir diretórios de serviços em uma arquitetura orientada a serviços. A solução proposta se dá através da inserção de dois elementos na arquitetura: o monitor e o mediador. O primeiro é responsável por detectar, notificar e confinar os erros, interceptando mensagens com o objetivo de analisar e testar os serviços. É de responsabilidade do monitor também transferir as invocações para serviços réplicas, quando necessário. Já o mediador cria e gerencia um grupo de réplicas usando o conceito do modelo técnico de UDDI.

Chen e Romanovsky [11] descrevem uma arquitetura de *Web service* que utiliza um mediador, denominado *WS-Mediator*, para aumentar a dependabilidade na integração de serviços. O mediador é responsável por aplicar políticas, monitorar os *Web services* e gerar metadados representando seus atributos de confiabilidade, como tempo de resposta e taxa de falha.

Ambos os trabalhos contribuem para o aumento da dependabilidade de aplicações desenvolvidas com o uso de uma abordagem SOA, porém não focam em mecanismos de tolerância a falhas baseados em tratamento de exceções coordenadas, diferentemente da proposta deste trabalho.

Tartanoglu et al. [36] propõe uma solução baseada em recuperação de erro por avanço com o objetivo de prover composições confiáveis[4] baseadas em *Web services*. A solução é proposta em termos de Ações Atômicas Coordenadas (*Coordinated Atomic Actions* - CA actions). Na abordagem proposta é feita uma adaptação de ações atômicas coordenadas para a composição de *Web services* em termos de ações confiáveis, culminando na chamada *Web Service Composition Action* (WSCA). O processo de recuperação de uma WSCA é fortemente baseado no uso de ações de compensação, isto é, operações associadas a cada operação que definem o comportamento normal de um serviço, destinadas a desfazer o que é realizado pela operação a qual se associa. Em um serviço de agência de viagens, por exemplo, a operação `cancel` seria uma operação de compensação para uma operação de reserva de passagem. A descrição de uma WSCA se dá por meio de uma linguagem baseada em XML denominada *Web Service Composition Action* (WSCAL).

O trabalho de Gorbenko et al. [15] propõe uma maneira de modelar *Web services* confiáveis que podem ser compostos por componentes não-confiáveis. O trabalho é baseado em WSCA, que permite que *Web services* sejam estruturados em termos de Ações Atômicas

[4]Do inglês, *dependable*

Coordenadas. A partir desse modelo, é implementado um exemplo de agência de viagens, através do qual os autores afirmam que a abordagem proposta aumenta significativamente a razão de requisições de *Web services* bem sucedidas.

O trabalho de Silva et al. [31] propõe uma abordagem baseada na composição de contratos para a composição tolerante a falhas de aplicações concorrentes. Um contrato de composição estende a noção de contratos de coordenação de forma a incluir o tratamento de exceções baseado em ações atômicas coordenadas. Um contrato de coordenação é uma conexão que pode ser estabelecida entre um grupo de objetos, a qual é regida por regras e restrições necessárias para que ocorra a colaboração.

As soluções baseadas em ações atômicas coordenadas, mais especificamente utilizando a proposta de Tartanoglu et al. [36] com WSCA, possuem algumas desvantagens quando comparadas com a nossa proposta. Primeiro, para cada exceção lançada por um serviço na composição, a mesma exceção é entregue aos demais serviços. Nossa solução é mais flexível, uma vez que, permite a especificação do comportamento do fluxo de tais exceções. Assim sendo, o lançamento de uma exceção `E1` pode implicar no lançamento de diferentes tipos de exceções para os demais serviços. Segundo, o processo de recuperação adotado em uma WSCA é fortemente baseado no uso de ações de compensação, o que não é uma característica obrigatória presente na implementação de um serviço. Na nossa abordagem, o usuário é livre para definir qualquer ação de recuperação, inclusive ações de compensação, caso estas sejam disponibilizadas por um serviço. Terceiro, a solução baseada em WSCA permite apenas o uso da tecnologia de *Web services*, ao passo que nossa solução, baseada em na Arquitetura de Componentes de Serviços implementada no projeto *Apache Tuscany SCA*, engloba e integra várias tecnologias SOA, inclusive *Web services*.

Souchon et al. [32, 33] apresenta um modelo para tratamento de exceções em sistemas multi-agentes. Tal modelo foca em dois problemas: preservar o paradigma de agentes, e dar suporte a concorrência cooperativa entre entidades comunicantes. O modelo proposto explora tanto o tratamento de exceções de atividades internas de um agente, quanto as geradas no nível de comunicação entre agentes. Devido à natureza do sistema, em que atividades são executadas de forma concorrente e a comunicação se dá por meio de troca de mensagens assíncronas, é possível que exceções sejam lançadas concorrentemente, gerando a necessidade de um mecanismo de resolução de exceções. Para tal necessidade, o modelo adota o uso de funções de resolução.

O modelo proposto por Souchon et al. [32, 33], similarmente ao encontrado em ações atômicas coordenadas, possui regras de recuperação fixas, ou seja, toda vez que uma ou mais exceções são lançadas por um agente, o mesmo fluxo excepcional é executado. Além disso, o modelo proposto não é baseado em um padrão de tecnologia SOA, tal como *Web services*, CORBA ou SCA, diminuindo assim a interoperabilidade do sistema.

1.5 Estrutura da Dissertação

Este trabalho está organizado da seguinte forma. O Capítulo 2 apresenta a fundamentação teórica necessária para o entendimento da solução proposta. O Capítulo 3 apresenta detalhes da solução proposta, onde o modelo *Guardian* é adaptado e implementado no framework *Apache Tuscany SCA*, resultando no modelo denominado *Guardian-SCA*. O Capítulo 4 traz um estudo de caso em que o modelo *Guardian-SCA* é aplicado para implementar um mecanismo de tolerância a falhas baseado em réplicação de servidores. O Capítulo 5 apresenta uma extensão do modelo *Guardian-SCA* em que o uso de programação orientada a aspectos é utilizada para oferecer um modelo de programação em que o código dedicado a tolerância a falhas é separado do código normal da aplicação. Por fim, as conclusões e alguns trabalhos futuros referentes a este trabalho são apresentados no Capítulo 6.

Capítulo 2

Fundamentação Teórica

Este capítulo apresenta alguns conceitos chaves para o entendimento da solução proposta: Arquitetura Orientada a Serviços (seção 2.1), Arquitetura de Componentes de Serviços (seção 2.2), e mecanismos de tolerância a falhas (seção 2.3). A seção 2.4 apresenta um detalhamento maior com realação ao mecanismo de tolerância a falhas denominado modelo *Guardian*. As considerações sobre o capítulo são apresentadas na seção 2.5.

2.1 Arquitetura Orientada a Serviços

Arquitetura Orientada a Serviços (*Service-Oriented Architecture* - SOA) é uma maneira de se organizar um sistema, baseada na exposição de unidades de *software* customizadas, denominadas serviços [20]. Tais unidades podem ser executadas individualmente ou agrupadas e executadas em uma ordem específica, resultando nas composições de serviços. A adoção de SOA oferece uma solução reutilizável e flexível, possibilitando mudanças rápidas e a um baixo custo, impulsionadas por mudanças de requisitos e tecnologia.

Um serviço é uma unidade distribuída, auto-contida, dividida em duas partes: implementação e especificação. A implementação do serviço é a concretização da lógica de negócio abstraída por esse, em uma linguagem de programação, tal como Java [3], C++ [34], Ruby ou Python. Um serviço pode ter várias implementações, em diferentes linguagens, referentes a uma mesma especificação. Esta define os detalhes de acesso e o contrato, associados aos serviços.

Os detalhes de acesso a um serviço levam em consideração dois fatores. Primeiro, a localização do serviço, isto é, o endereço em que ele pode ser encontrado. Segundo, o protocolo de comunicação usado para transmitir informações entre o fornecedor e consumidor do serviço.

O contrato especifica o comportamento que um serviço assumirá, sendo independente de detalhes de implementação. Um contrato provê uma interface que informa as operações

oferecidas pelo serviço, permitindo que as mesmas sejam acessadas independentemente da linguagem em que tal serviço foi implementado. Além disso, atributos de QoS, descrevendo regras de interação com os serviços, também estão associados a um contrato.

A fim de se obter soluções para problemas mais complexos, serviços podem ser agrupados e executados em um determinada ordem, tanto de forma síncrona, quanto assíncrona, ou uma combinação de ambas, resultando em uma composição de serviços. A composição pode ser guiada por um processo coordenador, resultando em uma orquestração, ou então cada serviço responsabiliza-se por invocar outro serviço, resultando em uma coreografia.

A Figura 2.1 mostra o funcionamento de uma arquitetura SOA. Num primeiro momento, o serviço é disponibilizado em um repositório de serviços ou *broker* de serviços, permitindo que o mesmo seja encontrado (1). Os consumidores de serviços acessam o *broker* de serviços para procurar o serviço que necessitam (2) e, caso encontrem, utilizam-no através do acesso direto ao provedor de serviços (3).

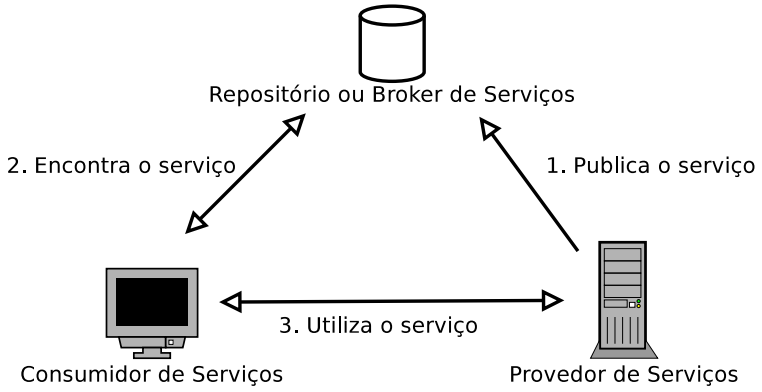

Figura 2.1: Uso de serviços em Arquitetura Orientada a Serviços.

No contexto da tecnologia de *Web services*, as interfaces são publicadas usando a *Web Service Definition Language* (WSDL), o protocolo de comunicação para troca de informações entre computadores é o *Simple Object Access Protocol* (SOAP) e o *Universal Description, Discovery, and Integration* (UDDI) permite que os serviços sejam publicados e descobertos. Todos esses padrões são independentes de plataforma e utilizam descrições em XML.

Um impacto direto do uso de SOA, com serviços bem definidos, está na modularização do sistema. Isso permite que, ao se alterar a forma como um serviço soluciona um problema de negócio, nenhuma alteração seja necessária nos usuários daquele serviço, desde que a interface do contrato seja mantida inalterada.

A modularização favorece o baixo acoplamento entre os serviços, isto é, por mais que um serviço faça uso de outros, mudanças nas lógicas podem ser realizadas sem que isso implique em mudanças em cascata. Deste modo, a aplicação fica apta a atender rapidamente

mudanças de requisitos e tecnologia a um baixo custo, favorecendo a manutenibilidade.

Pelo fato de um serviço oferecer suporte a várias implementações em diferentes linguagens e manter um contrato de comunicação independente de tecnologia, a interoperabilidade é favorecida em uma aplicação SOA. Por interoperabilidade entende-se a capacidade da comunicação entre diferentes tecnologias e linguagens de programação de forma efetiva.

Serviços bem projetados estimulam, ainda, a reusabilidade, evitando a reescrita de código na construção de novos sistemas de *softwares* correlacionados. Eis aqui outro quesito fundamental para melhoria do processo de desenvolvimento de sistemas de *software*.

Todavia, aplicações SOA tendem a apresentar uma perda de desempenho, como menciona Merson [22]. Isso em geral ocorre devido a um *overhead* adicionado pela necessidade de localização dos serviços distribuídos e pela presença de elementos responsáveis por fazer a serialização dos dados, a fim de garantir a interoperabilidade.

O desenvolvimento de testes é outro fator de dificuldade em aplicações SOA. Serviços distribuídos podem dificultar a configuração e rastreio da execução do sistema, além do que o código fonte de um serviço externo, bem como outros arquivos úteis para a etapa de depuração, podem não estar disponíveis para o usuário do serviço. A criação de testes pode complicar ainda mais, caso determinados serviços sejam selecionados e localizados em tempo de execução, dificultando a previsão de qual serviço será utilizado.

2.2 Arquitetura de Componentes de Serviços

Arquitetura de Componentes de Serviços (*Service Component Architecture* - SCA) [20] é um padrão aberto proposto para compor e dispor aplicações orientadas a serviços, através de um modelo de componentes. As especificações da SCA foram originalmente desenvolvidas em uma versão 1.0 por um consórcio de empresas (incluindo IBM, Sun, Oracle, Redhat e Siemens) chamado *Open Service Oriented Architecture*[1] (OSOA). Atualmente o padrão é mantido pela OASIS na seção *Open Composite Services Architecture*[2] (CSA).

Um componente de *software* é uma unidade de modularização composta de duas partes: uma especificação, com interfaces providas e requeridas explícitas, e uma implementação [35]. No contexto SCA, um componente (denominado componente de serviço[3]) é utilizado como unidade base para a implementação de serviços, onde cada componente pode implementar um ou mais serviços. A implementação do serviço faz parte da implementação do componente, ao passo que a especificação do serviço é mapeada para a especificação do componente. A Figura 2.2 mostra a relação de serviços e componentes, num contexto SOA e SCA, respectivamente.

[1] http://www.osoa.org
[2] http://www.oasis-opencsa.org/
[3] Do inglês, *service component*

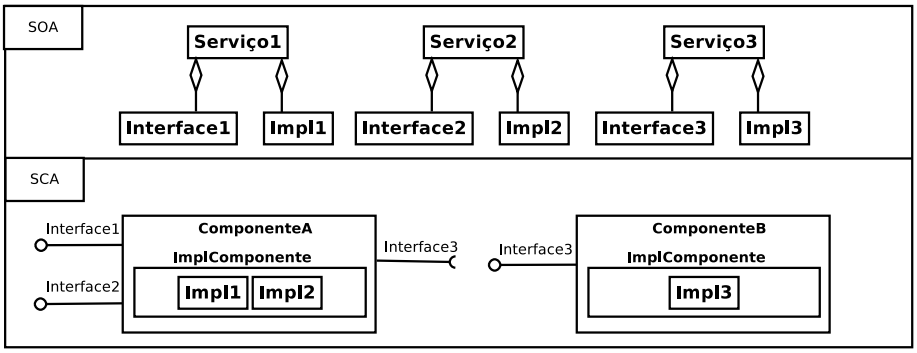

Figura 2.2: Relação entre serviços e componentes num contexto SOA e SCA, respectivamente.

A título de simplificação, analisemos apenas a interface como parte da especificação de um serviço. A Figura 2.2 mostra o `ComponenteA` sendo usado para implementar os serviços `Serviço1` e `Serviço2`, onde a implementação dos serviços faz parte da implementação do componente, e as interfaces são mapeadas para as interfaces providas do componente. Note ainda que o `ComponenteA` possui uma interface requerida `Interface3`, indicando que tal componente faz uso do `Serviço3`, serviço este que é implementado pelo `ComponenteB`.

Em uma Arquitetura de Componentes de Serviços, a implementação de um componente de serviço não inclui apenas linguagens de programação, mas também *frameworks* e outros ambientes, tais como o *framework* Spring e a linguagem de orquestração BPEL. Uma interface provida, referida como "serviço" no contexto SCA, define as operações providas pelo componente de serviço. Além disso, um componente de serviço pode requerer interfaces providas por outros componentes de serviço, através das interface requeridas, denominadas "referências".

A Figura 2.3 apresenta uma representação visual para um componente de serviço com seus principais elementos: serviços, referências, propriedades e *bindings*. A notação utilizada para representação dos componentes de serviço foi baseada na notação padrão adotada pela comunidade de SCA.

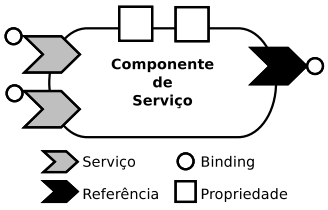

Figura 2.3: Representação visual das partes que compões um componente.

Serviços e referências são descritos através de alguma tecnologia para descrição de interfaces, tal como Java ou WSDL. Propriedades são valores configuráveis que possibili-

tam que um componente de serviço seja configurado em tempo de exceção. Já um *binding* especifica um protocolo de comunicação utilizado no acesso ao componente de serviço.

Através de serviços e referências, componentes de serviço SCA podem ser conectados, tanto manualmente (por um programador) quanto automaticamente (pelo ambiente de execução do SCA), resultando em uma composição. Tais composições podem englobar componentes que estão em uma mesma máquina e em um mesmo processo, ou em uma mesma máquina, mas em processos diferentes, ou então em máquinas diferentes. É possível ainda promover serviços, referências e propriedades, dos componentes que compõem uma composição, para o nível da composição. Assim, pode-se utilizar esta como uma unidade atômica em um sistema SCA, sendo os relacionamentos dos componentes internos à composição irrelevantes para os elementos que estabelecem uma comunicação com esta.

A Figura 2.4 mostra a promoção de serviços, referências, e propriedades, utilizando o relacionamento *promotion*, indicado por linhas tracejadas. O outro relacionamento presente na figura, indicado por linhas cheias, é chamado *wire*, e representa a comunicação entre uma referência e um serviço que satisfaça as necessidades daquela referência.

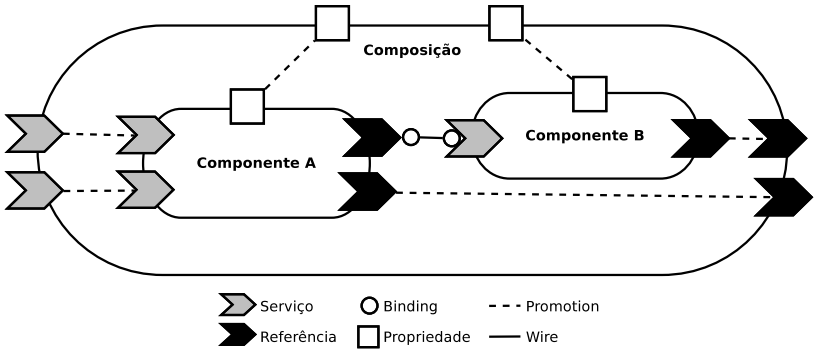

Figura 2.4: Relacionamentos *promotion* e *wire* entre componentes e composições.

A especificação de composições e componentes de serviços se dá através de uma linguagem baseada em XML denominada Linguagem de Definição de Componentes de Serviços (*Service Component Definition Language* - SCDL). A Figura 2.5 apresenta uma composição chamada `MyComposition`, que conecta os componentes de serviço `Calculator-Component` e `AddServiceComponent`, ambos implementados com a tecnologia Java (`implementation.java`, linhas 4 e 12), usando comunicação baseada no padrão de *Web services* (`binding.ws`, linhas 7 e 15). Observe que a referência `addService` (linha 5), presente no componente de serviço `CalculatorComponent`, é igual ao serviço provido pelo componente de serviço `AddServiceComponent` (linha 13), uma vez que ambos possuem a mesma interface `AddService`.

Uma característica importante conferida às aplicações SCA, decorrente do conceito de

```
1  <composite name="MyComposition">
2
3    <component name="CalculatorComponent">
4      <implementation.java class="CalculatorImpl"/>
5      <reference name="addService" >
6        <interface.java interface="AddService"/>
7        <binding.ws uri="..."/>
8      </reference>
9    </component>
10
11   <component name="AddServiceComponent">
12     <implementation.java class="AddServiceImpl"/>
13     <service name="AddService">
14       <interface.java interface="AddService"/>
15       <binding.ws uri="..."/>
16     </service>
17   </component>
18
19 </composite>
```

Figura 2.5: Exemplo de um composição descrita através da Linguagem de Definição de Componentes de Serviços.

bindings, é a separação dos detalhes de comunicação da lógica de negócio, como enfatiza Margolis [20] e Chappell [10]. Deste modo, fica extremamente fácil mudar a forma como um serviço, ou referência, pode ser acessado, sem que isso acarrete mudança alguma na lógica implementada. Basta apenas trocar, ou acrescentar um novo *binding* associado a um serviço ou referência. Nesse sentido, diz-se que uma solução SCA favorece o aumento da interoperabilidade, uma vez que, a comunicação entre diferentes tecnologias é garantida de forma efetiva.

2.3 Tolerância a Falhas em Sistemas de Software

Um sistema é considerado tolerante a falhas se ele continua sendo executado corretamente mesmo após a ocorrência de uma falha [4]. Falhas[4] são problemas estruturais no código que quando ativados causam erros[5]. Se não houver um mecanismo para corrigi-los no tempo especificado, ocorrerá um defeito[6], que se manifestará pela não execução ou mudança indesejada no serviço. Assim, o objetivo das técnicas de tolerância a falhas é evitar que erros acarretem defeitos no sistema. Nesse contexto, exceções podem ser usadas para indicar um erro ou uma condição anormal no sistema, e mecanismos de tratamento de exceções podem ser empregados para prover tolerância a falhas. Um mecanismo de tratamento de exceções permite a definição de exceções que, uma vez disparadas, são

[4]Do inglês, *faults*
[5]Do inglês, *errors*
[6]Do inglês, *failure*

capturadas e tratadas por tratadores. A função dos tratadores está em executar uma ação de recuperação que coloque o sistema em um estado consistente, livre dos efeitos causados pela ativação da falha.

Todavia, como abordado por Xu [39], o tratamento de exceções em sistemas distribuídos de processos assíncronos difere significativamente daquele aplicado em sistemas não distribuídos, já que: uma exceção pode ser lançada assincronamente em um processo devido à ocorrência de outra exceção em outro processo; e múltiplas exceções de diferentes tipos podem ser lançadas concorrentemente, possivelmente ao mesmo tempo.

Tais diferenças podem ser resumidas em um problema chave: no tratamento de exceções em sistemas distribuídos, cada processo envolvido deve invocar o tratador semanticamente correto para uma determinada exceção. O que não significa, necessariamente, que um mesmo tratador deva ser invocado para todos os processos envolvidos, ou que uma mesma exceção deva ser lançada em todos os processos envolvidos. Assim, faz-se necessária a presença de um coordenador para guiar o fluxo de exceções entre os processos assíncronos envolvidos.

Em tratamento de exceções em sistemas distribuídos, chamam-se exceções locais, ou interna, aquelas exceções que podem ser tratadas localmente por um processo. Já as exceções que devem ser repassadas para outros processos, e exigem um tratamento coordenado, são denominadas exceções globais, ou externas [26, 39].

As técnicas de tolerância a falhas se enquadram em duas categorias [30]: recuperação de erros por retorno[7], e recuperação de erros por avanço[8]. A primeira consiste em restaurar o sistema para um estado consistente que existia no passado, antes da ativação da falha. Já as técnicas de recuperação de erros por avanço baseiam-se em corrigir ou isolar os efeitos causados pela ativação da falha, corrigindo o estado errôneo do sistema de forma que a execução de seu comportamento normal possa ter continuidade. Alguns esquemas propostos implementam tais técnicas, como conversações [29], transações [16] e ações atômicas coordenadas [38].

2.3.1 Conversação

Conversação [29] é uma técnica de tolerância a falhas que permite a recuperação de erros coordenada em um conjunto de processos comunicantes (e possivelmente, cooperantes). Caso uma exceção seja lançada por um processo dentro da conversação, então a mesma exceção é entregue aos demais processos. Em caso de múltiplas exceções serem lançadas ao mesmo tempo, um mecanismo de resolução deve ser usado para combinar tais exceções em uma única nova exceção resolvida. Assim, todos o processos devem oferecer suporte

[7] Do inglês, *backward error recovery*
[8] Do inglês, *forward error recovery*

a tratadores para todas as possíveis exceções globais lançadas por cada outro processo participante, executando uma recuperação de erros por avanço ou por retorno. Fala-se em concorrência cooperativa entre os processos participantes de uma conversação, já que estes trabalham em prol de uma ação coletiva.

Como apresentado na figura 2.6, uma conversação é formada por três elementos principais: linhas de recuperação, linhas de testes, e um *firewall* duplo.

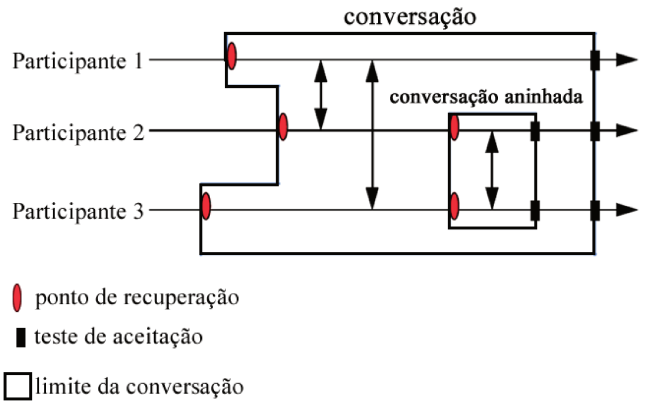

Figura 2.6: Representação gráfica de uma conversação. (Adaptado de [38])

As linhas de recuperação são formadas por um conjunto coordenado de pontos de recuperação. Dada a ocorrência de algum erro durante a execução da ação provida pela conversação, tais pontos permitem a recuperação dos estados dos participantes (recuperação de erros por retorno).

Para cada participante da ação há associada uma linha de testes, responsável por agregar um conjunto de testes de aceitação. Os testes são um mecanismo utilizado para validar o quão correta foi a execução de uma atividade por um participante. Admite-se ainda, a existência de testes globais, envolvendo vários participantes.

Pode-se dizer que uma conversação é executada com sucesso quando todos os testes de aceitação, e possíveis testes globais, são validados com sucesso. Em caso de falhas, exceções são disparadas, requerendo um tratamento adequado a fim de manter o estado da conversação consistente.

A função do *firewall* duplo está em isolar as ações desempenhadas pelos participantes de dentro da conversação com os demais participantes do sistema que estejam fora daquela conversação.

2.3.2 Transação

Transações [16], por sua vez, lidam com concorrência competitiva no acesso a objetos que foram projetados e implementados separadamente das aplicações que fazem uso deles. Garante-se sobre tais objetos as propriedades de atomicidade, consistência, isolamento e durabilidade (ACID).

A Figura 2.7 apresenta uma representação gráfica para uma transação.

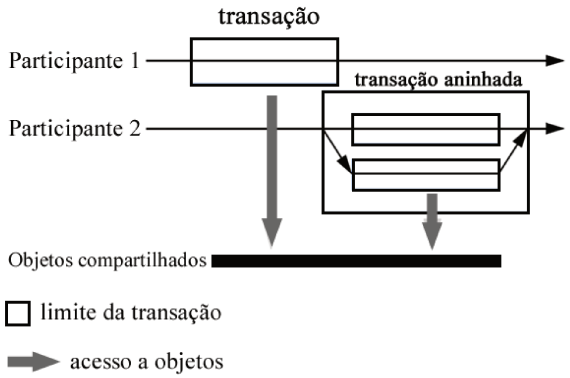

Figura 2.7: Representação gráfica de uma transação. (Adaptado de [38])

2.3.3 Ações Atômicas Coordenadas

O conceito de Ações Atômicas Coordenadas (*Coordinated Atomic actions* - CA actions), originalmente definido por Xu [38], diz respeito a um mecanismo de tolerância a falhas baseado no tratamento coordenado de exceções em sistemas orientado a objetos (OO) concorrentes. Tal mecanismo integra características de dois conceitos complementares: conversação e transação. A Figura 2.8 apresenta uma representação gráfica de uma *CA action*. Pode-se observar como elementos principais os participantes, os papéis que esses desempenham e objetos externos compartilhados.

As características das conversações se evidenciam no relacionamento entre os papéis desempenhados pelos participantes de uma *CA action*. Já as características transacionais estão presentes no acesso aos objetos compartilhados externos pelos papéis. Uma *CA action* conta com os mecanismos de recuperação de erro por retorno e por avanço a fim de tratar exceções lançadas durante a execução da ação.

O mecanismo de resolução de exceções concorrentes de uma *CA action* consiste no uso do modelo de árvores de resolução de exceções. Em tal modelo procura-se a menor sub-árvore que contenha todas as exceções lançadas concorrentemente, sendo a raiz a exceção comum, denominada *resolved exception*, que, por sua vez, é repassada para todos os processos participantes da *CA action*.

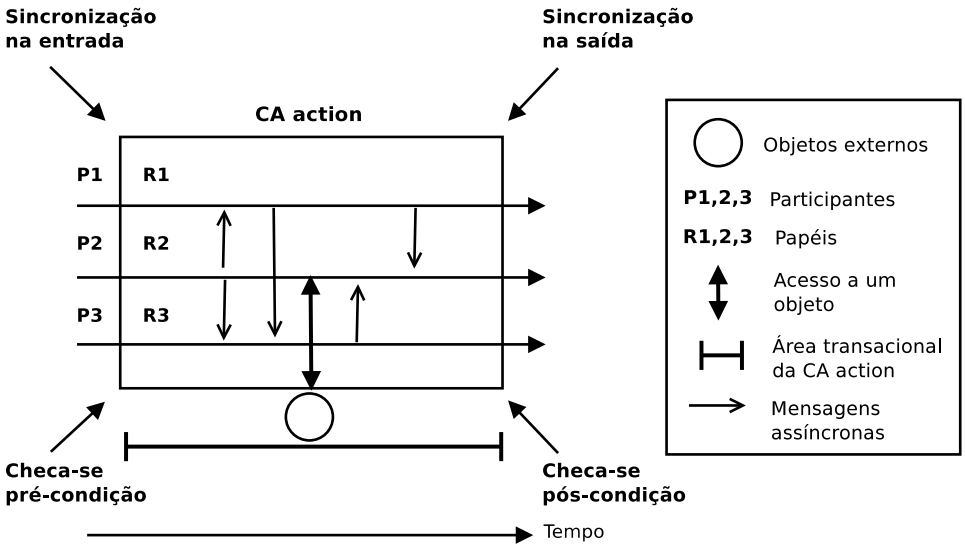

Figura 2.8: Representação gráfica de uma CA action. (Adaptado de [9])

Para toda ocorrência de erro, seja durante a execução de uma atividade, ou então durante a fase de testes de aceitação, um comportamento excepcional é iniciado, entrando em ação os mecanismos de recuperação de erro.

Como apresentado por Capozucca [9], os objetos externos podem prover seu próprio mecanismo de recuperação, sendo classificados como automaticamente recuperáveis, ou então repassar essa atividade para àqueles que fazem uso deles (fala-se em objetos manualmente recuperáveis). Objetos automaticamente recuperáveis podem ser tratados por mecanismos de recuperação de erro tanto por avanço, quanto por retorno, ao passo que os manualmente recuperáveis se restringem a recuperação de erros por retorno.

Dependendo do tipo de objeto externo que é tratado por uma recuperação de erros por retorno, tal mecanismo pode adotar uma estratégia de *roll back*, em que o estado do objeto é restaurado para um estado seguro e consistente (para objetos automaticamente recuperáveis), ou de compensação, em que o mecanismo de recuperação deve ser feito manualmente (para objetos manualmente recuperáveis).

O diagrama de estados apresentado na Figura 2.9 é de grande utilidade para o entendimento do funcionamento de uma *CA action*.

O estado inicial `E0` representa um estado consistente do sistema, existente antes da ativação da *CA action*. Mediante a validação de uma pré-condição, o sistema inicia a execução da ação atômica coordenada, o que fica caracterizado no estado `Serviço` (transição `t1`). Para um cenário em que nenhum comportamento excepcional é disparado, e a atividade refletida pela *CA action* é concluída com sucesso, tem-se o encaminhamento do sistema do estado `Serviço` para o estado `E1` (transição `t2`).

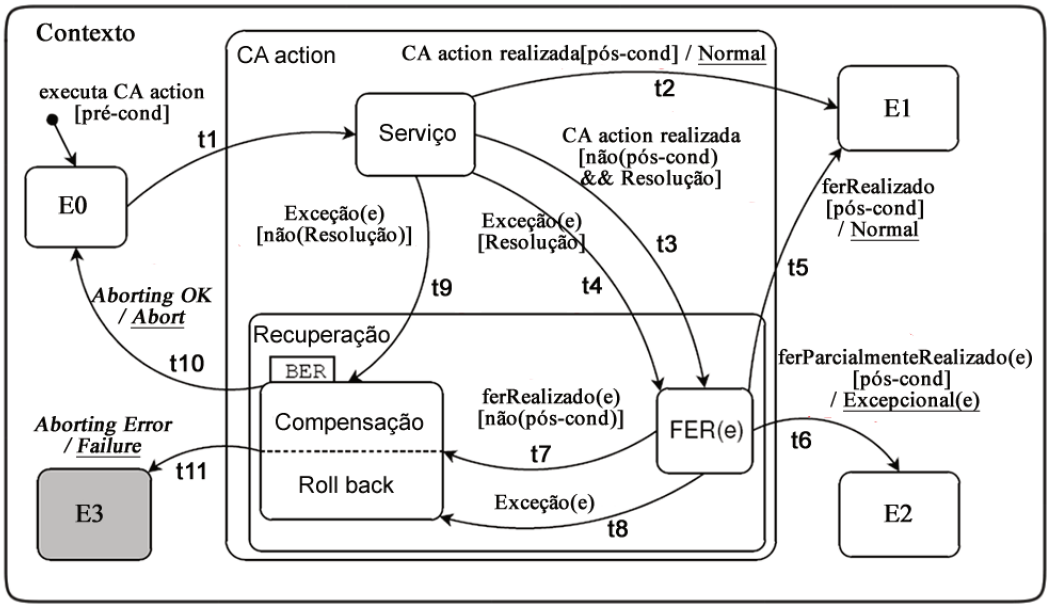

Figura 2.9: Diagrama de estados demonstrando os estados assumidos por um sistema ao utilizar um mecanismo de CA actions. (Adaptado de [9])

Quando uma *CA action* falha, seja por falha em algum teste de aceitação (transição `t3`) ou devido ao fato de exceções terem sido lançadas (transição `t4`), tem-se a necessidade da execução de mecanismos de recuperação de erros, o que fica evidenciado no estado `Recuperação`. A técnica de recuperação de erro por avanço, retratada no estado denominado `FER(e)` (uma abreviação para *Forward Error Recovery*), é utilizada como primeira alternativa para solucionar o problema. Caso a execução dessa seja bem sucedida, e uma pós-condição seja obedecida, pode-se dizer que a falha foi tratada e o sistema mantém-se em um estado consiste `E1` (transição `t5`). Há ainda a possibilidade da exceção gerada ser parcialmente tratada, o que leva o sistema a uma estado parcialmente consiste, denominado `E2` (transição `t6`).

É possível que, mesmo uma exceção sendo tratada no estado `FER(e)`, o teste de aceitação imposto pela pós-condição não seja obedecido (transição `t7`), ou ainda, que alguma exceção seja lançada durante a execução de tal estado (transição `t8`). Ambos os casos levam a *CA action* a ativar o mecanismo de recuperação de erro por retorno, retratado no estado `BER` (uma abreviação para *Backward Error Recovery*). Dependendo do tipo dos objetos externos que estão sendo manuseados, tem-se o uso das técnicas de *roll back* ou de compensação. Para o caso em que a recuperação de erros por retorno é executada com sucesso, o sistema retorna a um estado consistente, existente antes da execução da *CA action* (transição `t10`). O pior caso ocorre quando a falha detectada não pode ser tratada nem pela técnica de recuperação de erro por avanço, nem pela de

retorno, levando o sistema a um estado inconsistente `E3` (transição `t11`).

Quando várias exceções são lançadas concorrentemente, faz-se necessário o uso de um algoritmo de resolução de exceções para procurar uma exceção comum entre elas. Isso é representado pela variável booleana `Resolução`, colocada em teste nas transições `t3`, `t4` e `t9`. Observa-se, assim, outro caminho para se alcançar o estado `BER`: através da transição `t9`. Tal caminho é executado quando o algoritmo de resolução de exceções não consegue achar uma exceção comum para ser tratada no estado `FER(e)`. Deve-se notar que o valor `e` representa justamente a exceção comum encontrada pelo algoritmo.

Destaca-se que, ao iniciar a execução de uma CA action, o sistema pode assumir quatro estados diferentes, a saber: normal (estado `E1`), excepcional (estado `E2`), *abort* (estado `E0`) e *failure* (estado `E3`).

2.3.4 Classificação dos Mecanismos de Tratamento de Exceções em Sistemas Distribuídos

Miller [25] categoriza os mecanismos de tratamento de exceções em sistemas distribuídos em três gerações.

A primeira geração é baseada no modelo de exceção remota[9]. Nesse modelo, um processo pode lançar uma exceção para outro, denominada exceção remota. A exceção pode ser lançada tanto em um modelo síncrono, como em uma invocação de método remota (*Remote Method Invocation* - RMI), quanto assíncrono, em que a detecção de uma exceção causa uma interrupção na execução da atividade de um processo.

O modelo da primeira geração tem como principais limitações o fato de não permitir coordenação entre as exceções remotas, e não lidar com múltiplas exceções lançadas concorrentemente. Note que a denominação de exceção remota possui a mesma semântica das exceções globais ou externas, isto é, são exceções que devem ser tratadas externamente ao processo que as sinaliza.

O mecanismo de tratadores da segunda geração, denominado modelo de exceções concorrentes[10], tem como principal característica o fato de lidar com múltiplas exceções lançadas concorrentemente, possivelmente ao mesmo tempo. Tal modelo é baseado em um mecanismo de resolução de exceções e na estruturação do programa baseado num modelo de conversação, como aquele definido pelo modelo de *CA actions*.

O principal problema encontrado nos mecanismos de tratamento de exceções da segunda geração é que a regra que guia o fluxo das exceções globais é fixa, isto é, para cada exceção global lançada por um processo participante, a mesma exceção é entregue aos demais participantes. Isso torna o modelo pouco flexível, no sentido de que cada processo

[9]Do ingês, *remote exception model*
[10]Do inglês, *concurrent exception model*

participante deve possuir um tratador para cada tipo de exceção que possivelmente pode ser lançada por cada outro processo participante.

O modelo da terceira geração, denominado modelo de tratamento de exceções globais[11], tenta suprir as deficiências dos modelos anteriores. Em tal modelo, existe uma entidade global que orquestra o tratamento de exceções guiando cada processo envolvido através do lançamento de uma exceção adequada em cada um. As exceções lançadas em cada processo podem diferir entre si. A entidade global utiliza regras de recuperação previamente definidas para determinar qual exceção deve ser lançada em cada participante, ocasionando assim, a invocação do tratador de exceção adequado. Uma abordagem para o mecanismo de tratamento de exceções da terceira geração é o modelo *Guardian*.

2.4 Modelo Guardian

O modelo *Guardian* [23–26] pode ser visto como um modelo para garantir tolerância a falhas em ambientes distribuídos assíncronos, que lida com o principal problema envolvendo tratamento de exceções nesse tipo de ambiente: garantir a invocação de tratadores de exceções semanticamente corretos em todos os processos participantes que necessitam participar da recuperação. De acordo com a classificação de Miller [25], tal modelo é enquadrado como de terceira geração de mecanismos de tratamento de exceções.

O modelo é baseado em um modelo de computação assíncrona, tratadores de exceções globais, separação entre tratamento de exceções globais e tratamento local a um determinado processo, e em um modelo de falhas estendido.

Os principais elementos do modelo *Guardian* são:

1. O conceito de contexto excepcional;

2. Uma entidade global denominada *Guardian Group* (referida também apenas como *guardian*) com um *Guardian Member* associado para cada processo participante;

3. Um conjunto de *Guardian Primitives* utilizadas na comunicação entre os participantes e os *guardian members*, e entre os *guardian members* e o *guardian group*.

A Figura 2.10 ilustra os principais elementos do modelo *Guardian*. Note que a comunicação entre os *guardian members* se dá através das *guardian primitives* oferecidas pelo *guardian group*. Assim, pode-se enxergar o *guardian* como um serviço em que cada *guardian primitive* é vista como uma operação deste serviço.

[11] Do inglês, *global exception handling model*

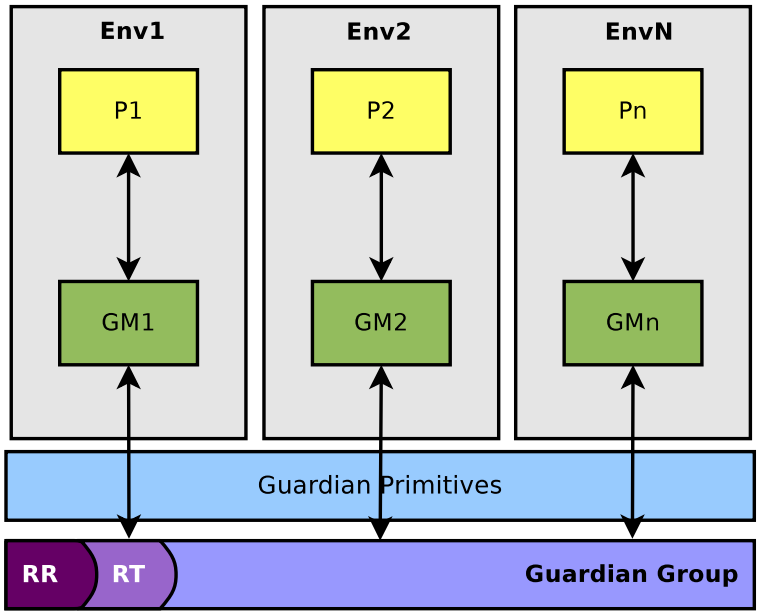

Figura 2.10: Visão conceitual do modelo *Guardian*. Env_i representa o ambiente de execução, P_i um processo participante e GM_i um guardian member. RR e RT representam, respectivamente, as Regras de Recuperação e a Árvore de Resolução de Exceções associadas ao *Guardian Group*. (Adaptado de [26])

2.4.1 Contexto Excepcional

O contexto excepcional representa um bloco de execução ou região de um programa, a qual está associado um bloco de recuperação (como um bloco `try-catch` em Java). Assumem significados diferentes baseado na aplicação, ou seja, são específicos de uma aplicação, e podem ser aninhados. Além disso, um contexto possui um nome simbólico definido pelo usuário, que pode ser concatenado, em tempo de execução, com outros nomes de contextos.

Há três tipos de contextos excepcionais: *signaling context*, *raising context*, e o *target context*. O primeiro é o contexto em que o processo está quando uma exceção é lançada (em Java, por exemplo, utilizando a cláusula `throw`). O segundo representa o contexto em que o processo está quando uma exceção é lançada nele (em Java, por exemplo, uma divisão por zero, o que acarreta no disparo da exceção *ArithmeticException*). Por fim, o terceiro é o contexto em que uma exceção deve ser tratada.

Um processo que lança uma exceção tem o seu *signaling* e *raising context* iguais. Em um sistema que faz uso dos contextos, todo processo possui um *raising context*, porém o *signaling context* só está presente naqueles que lançam explicitamente uma exceção.

A existência dos contextos permite que tratadores adequados sejam invocados em um determinado processo. Quando uma exceção é lançada, um *target context* é especificado

no objeto excepcional, possibilitando que o *raising context* seja diferente do *target context*. Deste modo, permite-se a atribuição de um significado a uma exceção, de forma dinâmica, baseando-se no fluxo atual do programa.

No modelo *Guardian*, há um contexto reservado de mais alto nível denominado `Init`. Tal contexto representa o início da execução do processo, antes da ativação de qualquer outro contexto. A presença do `Init` permite que este seja usado como um *target context* padrão para o tratamento de exceções que não foram tratadas em contextos de níveis inferiores.

2.4.2 Modelo de Tratamento Exceções Globais

Como mostrado na Figura 2.10, a comunicação entre elementos que compoem o modelo *Guardian* se dá da seguinte forma: cada processo participante se comunica com o seu respectivo *guardian member* e cada *guardian member* se comunica com o *guardian group*. A comunicação entre esses pares é dada, em ambos os casos, através das *guardian primitives*. Já que os *guardian members* são representações lógicas do *guardian group*, a comunicação através das *guardian primitives* é válida.

O *guardian* dá suporte ao tratamento de exceções globais da seguinte forma:

1. Utilizando o método `enableContext`, cada participante define um contexto e as exceções que aquele contexto está apto a tratar. Cada *guardian member*, associado ao seu respectivo participante guarda uma lista desses contextos e das exceções relacionadas a cada contexto. Um contexto está ativo a partir do momento em que `enableContext` é invocado, e se torna inativo com a invocação do método `disableContext`.

2. Quando um processo participante precisa lançar uma exceção global (também referenciada como exceção externa), isto é, uma exceção que deve ser tratada de forma coordenada por um grupo de processos participantes, faz-se o uso do método `gthrow`. A chamada de tal método causa o bloqueio do participante até que uma exceção seja lançada nele. Exceção essa que é lançada pelo próprio método `gthrow`. A exceção global é comunicada ao *guardian member* respectivo e este, por sua vez, a repassa para os demais *guardian members* através do *guardian group*. Se houver exceções lançadas simultaneamente por diferentes participantes, cada *guardian member* receberá o conjunto dessas exceções.

3. Todos os processos que participarão do tratamento de exceções coordenado devem ser suspensos sincronamente. Quando um *guardian member* recebe uma exceção do *guardian group*, ele checa se o participante associado está suspenso e, em caso neg-

ativo, ele suspende-o. Isso garante que o *guardian group* conheça todas as exceções concorrentes antes de iniciar a recuperação cooperativa.

4. Caso o participante suporte interrupções, o *guardian member* associado interrompe o participante e o tratador associado àquela interrupção invoca o método `checkExceptionStatus`. O método checa se há algum exceção que deve ser entregue ao participante. Caso positivo, o método bloqueia até que a exceção seja lançada no participante; caso contrário, o método retorna. Para aqueles participantes que não dão suporte a interrupções, o método `checkExceptionStatus` deve ser invocado periodicamente.

5. Uma vez que todos os participantes estão suspensos, o *guardian* invoca as regras de recuperação, específicas da aplicação, estabelecidas previamente. O *guardian* provê para as regras a exceção lançada, bem como o contexto atual de todos os participantes envolvidos. Assim que a regra adequada para a exceção lançada é encontrada, determina-se qual exceção deve ser lançada em cada participante, bem como o *target context* em que esta deve ser tratada. Note que não necessariamente uma mesma exceção deve ser lançada em todos os participantes. Dependendo das regras de recuperação definidas, diferentes tipos de exceções podem ser lançadas com diferentes *target contexts* para diferentes participantes. As exceções determinadas são repassadas para os *guardian members* e entregues aos respectivos participantes.

6. Assim que a exceção é entregue aos participantes, é esperado que cada tratador invoque o método `propagate` a fim de determinar se aquela exceção dever ser tratada naquele contexto ou em um contexto diferente. Um resultando booleano indica se a exceção deve ser tratada em tal tratador, ou deve ser propagada para um contexto de nível superior.

Como explicado nos itens 3 a 5, antes do *guardian* fazer uso das regras de recuperação, todos os processos devem se encontrar em um estado suspenso, garantindo ao *guardian* o conhecimento de todas as exceções lançadas. Caso haja mais de uma exceção lançada, assim que o sistema atinge um ponto de suspensão (isto é, todos os participantes envolvidos estão em um estado suspenso), diz-se que tais exceções são concorrentes. Exceções concorrentes devem ser resolvidas através de um mecanismo de resolução de exceções antes de serem tratadas.

O modelo *Guardian* utiliza o mecanismo de árvore de resolução de exceções para lidar com exceções concorrentes. É possível definir várias árvores de resolução associadas a diferentes níveis de prioridade de exceções. Assim, exceções concorrentes são ditas

independentes se possuem diferentes níveis de prioridade, e, portanto, são resolvidas com diferentes árvores de resolução.

Para cada conjunto de exceções concorrentes pertencente ao mesmo nível, utiliza-se a respectiva árvore de resolução para obter uma *resolved exception*. As *resolved exceptions* são, então, entregues às regras de recuperação.

Na implementação de Miller [25, 26], o *guardian* é implementado como uma máquina de estado replicada em cada *guardian member* pertencente ao grupo. Quando o sistema atinge um ponto de suspensão, cada *guardian member* possui o mesmo conjunto ordenado de exceções concorrentes para ser resolvido. Isso é garantido já que o modelo é baseado em um modelo de broadcast atômico First-In-First-Out (FIFO) confiável. Cada *guardian member* executa, então, a resolução das exceções localmente e entrega uma ou mais *resolved exceptions* para o seu respectivo processo participante. Todas as exceções que devem ser entregues ao participante são enfileiradas e entregues sequencialmente.

O modelo ainda assume que o método `checkExceptionStatus` é executado periodicamente pelo participante, ou então que interrupções são permitidas quando um contexto é ativado. Isso é requerido para que o processo participante seja estruturado de tal forma que todas as exceções pendentes sejam tratadas antes da volta à execução do código normal. Além disso, o modelo não permite que um participante lance novas exceções através do método `gthrow` enquanto houver exceções pendentes. Caso o *guardian* identifique uma situação como esta, o processo é finalizado com uma `TerminateException`.

2.4.3 Guardian Primitives

Como mencionado anteriormente, a comunicação entre os processos participantes e o *guardian group* se dá da seguinte forma: cada processo participante se comunica com seu respectivo *guardian member* e cada *guardian member* se comunica com o *guardian group*. Tal comunicação é feita através do uso de um conjunto de métodos estabelecidos no modelo, denominados *guardian primitives*.

Os principais métodos que compõem as *guardian primitives* são:

1. `void enableContext(Context c, ExceptionList el);`
2. `void removeContext();`
3. `void gthrow(GlobalException ex, ParticipantList pl);`
4. `Boolean propagate();`
5. `void checkExceptionStatus();`

Os dois primeiros métodos são utilizados para controle dos contextos. O método `enableContext` define um contexto `c` que possui tratadores para cada exceção presente na lista de exceções `el`. Assim, quando uma exceção, que foi previamente definida na lista de exceções, é lançada no contexto `c`, o tratador associado àquela exceção é invocado. Note que o método não define os tratadores, ele apenas informa ao *guardian* as exceções que podem ser tratadas em um determinado contexto. Assim, o *guardian* guarda uma lista de contextos para cada participante (organizada em formato Last-In-First-Out (LIFO)), em que cada elemento é determinado pelo par contexto e sua lista de exceção associada.

O método `removeContext`, por sua vez, remove o último contexto adicionado na lista de contextos, tornando-o inativo.

Os métodos `gthrow` e `propagate` são utilizados para controlar o fluxo das exceções globais. O primeiro é utilizado por um processo participante para lançar uma exceção global `ex` para um conjunto de participantes especificados em `pl`. Caso a lista de participantes não seja informada, o *guardian* assume que todos os participantes associados a ele devem ser notificados da exceção lançada. O uso de `pl` permite que um subconjunto dos participantes seja envolvido no processo de recuperação de erro coordenado, tornando o modelo mais flexível. O modelo permite ainda o uso de uma expressão regular para representar um conjunto de participantes.

A invocação do método `gthrow` por um participante causa a suspensão de todos os participantes listados em `pl`, e termina com o lançamento de uma exceção no processo invocador.

No modelo *Guardian*, toda exceção global é do tipo `GlobalException` e carrega consigo a informação do processo que lançou a exceção, o contexto em que essa foi lançada (*signaling context*), bem como o contexto em que tal exceção deve ser tratada (*target context*). O modelo define algumas exceções globais, denominadas *membership exceptions*, tais como: `JoinException`, `LeaveException`, `SuspendException` e `UnhandledException`.

Uma `JoinException` é lançada pelo *guardian* toda vez que um novo participante se associa ao grupo. Semelhantemente, uma `LeaveException` é lançada toda vez que um participante deixa o grupo. Já a `SuspendException` é utilizada internamente pelo *guardian* para implementar os pontos de suspensão e suspender um participante toda vez que `checkExceptionStatus` é invocado.

Toda vez que uma exceção não pode ser tratada nos contextos internos, ela atinge o contexto `Init` (contexto de mais alto nível) definido pelo *guardian*. Pode-se associar um tratador genérico, que captura qualquer tipo de exceção, a tal contexto. Nesse caso, o tratador pode lançar uma `UnhandledException` para o *guardian* indicando que as exceções anteriormente lançadas não puderam ser tratadas. Assim, o *guardian* pode tentar reparar o dano com alguma ação de recuperação, como reiniciar o processo falho, por exemplo.

O uso de `propagate` por um processo participante tem como finalidade determinar se uma dada exceção deve ser tratada no contexto corrente em que o método foi invocado, ou então propagada para um contexto de nível mais alto (em Java, usando `throw` por exemplo). O método retorna `false` para o primeiro caso, e `true` para o segundo. O retorno é obtido através da comparação do contexto corrente com o *target context* especificado para a exceção lançada. Tal método é necessário já que o *guardian* não tem controle sobre o fluxo de propagação de uma exceção lançada por ele em um participante.

Por fim, o método `checkExceptionStatus` permite a um participante checar a existência de exceções globais pendentes, isto é, que devem ser tratadas. Quando invocado, `checkExceptionStatus` checa se há alguma exceção na lista de exceções pendentes. Caso essa esteja vazia, o método simplesmente retorna. Caso haja uma exceção do tipo `SuspendException`, o método bloqueia até que uma nova exceção seja adicionada à lista. Quando isso ocorre, tal exceção é lançada no participante.

O modelo ainda prevê o suporte a noção de interrupções com os métodos `enableInterrupts` e `disableInterrupts`. A ativação de interrupções por um participante permite que o seu respectivo *guardian member* o interrompa e entregue uma exceção pendente.

2.4.4 Regras de Recuperação

As regras de recuperação são regras específicas da aplicação utilizadas toda vez que uma exceção global é sinalizada para o *guardian*. Tais regras, que são replicadas em cada *guardian member*, tomam como entrada uma exceção e a lista de contextos de cada participante. O mecanismo de regras gera, então, uma exceção para cada participante, que deve ser tratada em um contexto específico denotado em *target context*.

Um participante é identificado através de um identificador. Este é expresso pelos nomes dos contextos, presentes em sua lista de contexto, separados por um caractere delimitador tal como "/" ou ".", ao invés de um nome simbólico e fixo. Isso permite que um grupo de participantes seja identificado com o uso de expressões regulares. Por exemplo, seja L1 = {C1, C2, C3, C4} a lista de contextos de um participante A, e L2 = {C1, C2, C3} a lista de contextos de um participante B, tem-se como identificador de A a *string* C1/C2/C3/C4 e de B C1/C2/C3. Assim, o uso de uma expressão regular do tipo */C4 retorna todos os participantes cujo contexto corrente é C4. No exemplo em questão, o participante A é obtido. Já uma expressão regular C1/C2/*, retorna todos os participantes cujo os dois primeiros contextos na lista de contextos são C1 e C2, respectivamente. No exemplo dado, isso resulta no participante A e B. O uso de expressões regulares confere maior flexibilidade na manipulação dos participantes se comparado ao uso de nomes simbólicos fixos.

Há dois tipos de regras de recuperação, uma para *single exception*, isto é, quando não

há exceções concorrentes, e outra para o caso de exceções serem lançadas concorrentemente por diferentes participantes. No primeiro caso, a regra definida para a exceção lançada constrói, com base na lista de contexto de cada participante e nos predicados estabelecidos, uma lista com as exceções que devem ser lançadas em cada participante. A Figura 2.11 apresenta a estrutura geral de uma regra para *single exception*.

```
1  Entrada: Excecao global "ex"
2  Saida: OEL (Output Exception List)
3
4  Para cada expressao regular PE especificada nas regras faca {
5    Seja PL a lista de participantes que satisfaz PE;
6    Para cada selecao especificada pelo predicado S faca {
7      Seja PS um subconjunto de PL que satisfaca S;
8      Para cada participante p em PS faca {
9        Ep = determinaExcecaoEContexto();
10       OEL.inserir(p, Ep(Cp));
11     }
12   }
13 }
```

Figura 2.11: Estrutura geral de uma regra de recuperação para *single exception*. P_E representa uma expressão regular utilizada para selecionar um conjunto de participantes. P_L representa a lista de participantes selecionados utilizando P_E. S representa um predicado utilizado para selecionar um subconjunto de P_L. P_S representa o subconjunto selecionado utilizando S. E_p representa a exceção que deve ser lançada no participante p determinada com base em alguma heurística específica da aplicação. C_p representa o *target context* em que E_p deverá ser tratada. OEL contém as exceções a serem lançadas em cada participante. (Adaptado de [26])

Uma vez que a regra é executada e a OEL é formada, o *guardian* dispara cada exceção da lista nos participantes correspondentes.

A Figura 2.12 apresenta um exemplo de regra de recuperação para uma *single exception*. A regra se aplica a um sistema composto por dois participantes: `Hotel` e `Flight`. `Init/Main/HotelExecution` e `Init/Main/FlightExecution` representam, respectivamente, os identificadores com todos os contextos de cada participante. Considere `Main` como o contexto em que algumas configurações são efetuadas, e `HotelExecution` e `FlighExecution` como os contextos em que os participantes realmente executam suas funções.

Seja `S1` o estado de execução do sistema em que os dois participantes se encontram em seus respectivos contextos `Execution`. Nesse estado, uma eventual falha na execução do participante `Hotel` leva ao lançamento de `HotelServiceFailureException` para o *guardian* e, por conseguinte, à execução da regra de recuperação definida na Figura 2.12.

Tal regra faz com que uma `HotelServiceFailureException` seja lançada no contexto `Init` do participante `Hotel`, e que a mesma exceção seja lançada no participante `Flight`, porém com um *target context* diferente. A definição do contexto de tratamento permite que a exceção seja tratada no tratador mais adequado em cada participante.

Um possível ação de recuperação para o tratador associado ao contexto `Init` do participante `Hotel` seria lançar uma exceção do tipo `UnhandledException` para o *guardian*, informando que um erro interno inesperado ocorreu e não pode ser tratado. Deste modo, tal participante pode ser reinicializado, por exemplo. Já o tratador associado ao contexto `Main` do participante `Flight`, sabendo que houve uma falha na execução do Hotel, poderia interromper sua execução, desfazer as ações realizadas, e reinicializar o processo. Tais ações variam, contudo, de acordo com as especificidades da aplicação.

```
1 Caso a excecao lancada seja igual a HotelServiceFailureException entao {
2   Seja p = PS = PL os elementos que satisfazem PE = '*/HotelExecution'
3   OEL.insert(p, HotelServiceFailureException('Init'));
4   Seja p = PS = PL os elementos que satisfazem PE = '*/FlightExecution'
5   OEL.insert(p, HotelServiceFailureException('*/Main'));
6 }
```

Figura 2.12: Exemplo de uma regra de recuperação para *single exception*.

As regras de recuperação para exceções concorrentes tomam como entrada o conjunto de exceções que foram lançadas pelos participantes concorrentemente. A partir disso, um mecanismo de resolução de exceções é executado, como por exemplo substituir o conjunto de exceções concorrentes por uma única outra exceção, remover uma ou mais exceções da lista, ou processar, sequencialmente, as exceções em uma dada ordem. As exceções resultantes desse processo são então processadas utilizando as regras de recuperação para *single exception*, originando assim uma lista de exceções que devem ser lançadas em cada participante envolvido.

Como mencionado na Seção 2.4.2, é possível associar níveis de prioridade para as exceções globais. Quando várias exceções são lançadas concorrentemente pelos participantes, o *guardian* organiza-as em um vetor de nível de prioridade. Cada posição do vetor possui um conjunto de exceções concorrentes para um determinado nível. Diferentes níveis significam independência entre as exceções, e portanto uma árvore de resolução de exceções deve ser definida para cada nível.

Após a construção do vetor de prioridade, aplica-se o mecanismo de resolução de exceções baseado em árvores de resolução para cada conjunto de exceções concorrentes, começando com o nível de maior prioridade.

A Figura 2.13 apresenta a estrutura geral de uma regra para exceções concorrentes. Tem-se CEL como a lista de exceções concorrentes ordenada por nível de prioridade, e

OSL como a lista de exceções resolvidas para cada conjunto de exceções concorrentes em $CEL[l]$, sendo que l varia de 0 até $N-1$, com N representando o número de níveis de prioridade.

```
1 Entrada: CEL (Concurrent Exception List)
2 Saida: OSL (Output Sequential List)
3
4 Para cada nivel de prioridade l = 0 ate N - 1 faca {
5   Seja ESl o conjunto de excecoes concorrentes no nivel l;
6   Seja Er a excecao determinada por meio de resolucaoExcecao(ESl);
7   OSL.insert(Er);
8 }
```

Figura 2.13: Estrutura geral de uma regra de recuperação para exceções concorrentes. ES_l representa o conjunto de exceções concorrentes no nível l presentes em $CEL[l]$. E_r representa a exceção resolvida por meio do mecanismo de árvore de resolução de exceções. (Adaptado de [26])

Retomando o exemplo do sistema composto pelos participantes `Hotel` e `Flight`, suponha que quando ambos estão em seus contextos `Execution`, isto é, `Hotel` se encontra em `Init/Main/HotelExecution` e `Flight` em `Init/Main/FlightExecution`, uma falha de execução ocorra em cada um dos processos. Deste modo, `Hotel` dispara uma `HotelServiceFailureException` para o *guardian*, enquanto que o participante `Flight` dispara uma `FlightServiceFailureException`. Suponha que ambas exceções pertençam ao mesmo nível de prioridade. Ao identificar a ocorrência das exceções concorrentes, o *guardian* procura, na árvore de resolução de exceções de nível correspondente ao das exceções concorrentes lançadas, o menor ancestral comum entre o conjunto de exceções concorrentes.

Analisando a árvore de resolução de exceções apresentada na Figura 2.14 tem-se `TerminationException` como a exceção resolvida. O *guardian* usa, então, tal exceção como entrada para as regras de recuperação para *single exceptions*, e a partir disso, obtém a lista de exceções que devem ser lançadas em cada participante envolvido no tratamento coordenado.

2.5 Considerações Finais

Neste capítulo foram apresentados alguns conceitos chaves para o entendimento da solução proposta: Arquitetura Orientada a Serviços, Arquitetura de Componentes de Serviços, e mecanismos de tolerância a falhas.

A respeito de SOA foi traçada uma visão geral, e uma análise das principais vantagens e desvantagens adquiridas com o uso desse tipo de padrão arquitetural.

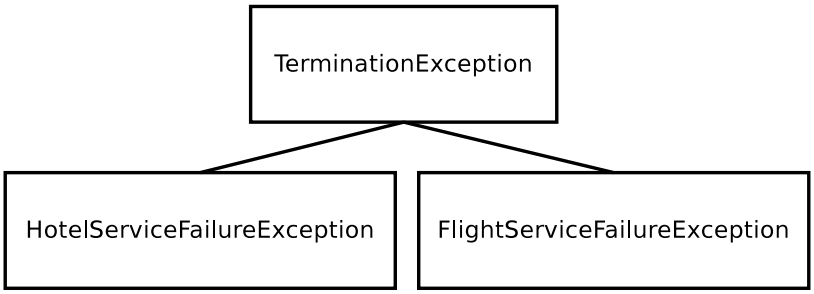

Figura 2.14: Exemplo de árvore de resolução de exceções.

Os conceitos apresentados sobre Arquitetura de Componentes de Serviços mostraram a flexibilidade e facilidade na composição de serviços, garantidos às aplicações que seguem tal arquitetura. A separação da lógica de negócio dos detalhes de comunicação, bem como as várias automatizações realizadas pelo ambiente de execução do SCA, podem ser vistas como grandes vantagens dessa arquitetura.

Apresentou-se também alguns conceitos envolvendo tolerância a falhas, que se resumem em não permitir que erros, originados de falhas, ocasionem um defeito no sistema. Nesse contexto, o uso de mecanismos de tratamento de exceções visa oferecer ferramentas necessárias para estruturar atividades de tolerância a falhas. Contudo, quando se trata de sistemas distribuídos assíncronos, é necessário uma maior sofisticação dos mecanismos de tratamento de exceções a fim de satisfazer necessidade requeridas nesse tipo de ambiente, tais como: o fato de que exceções podem ser lançadas assincronamente; e múltiplos tipos de exceções podem ser lançados por diferentes processos ao mesmo tempo, necessitando da presença de um coordenador.

Foram apresentados três gerações de mecanismos de tratamento de exceções para sistema assíncronos: modelo de exceção remota; modelo de exceções concorrentes; e modelo de tratamento de exceções globais.

O primeiro modelo não permite a comunicação e coordenação entre os tratadores, além de não lidar com exceções concorrentes.

O modelo da segunda geração evita a necessidade de comunicação e coordenação entre os tratadores estruturando a aplicação em um estilo baseado em conversações. Destaque-se o modelo de ações atômicas coordenadas que utiliza características dos modelos de conversação e transação. Além disso, tal modelo oferece mecanismos de resolução para o caso de múltiplas exceções concorrentes serem lançadas, como árvores de resolução de exceções.

O terceiro modelo baseia-se em uma entidade global que coordena o tratamento de exceções distribuído entre os vários processos participantes. O modelo utiliza regras de recuperação específicas da aplicação para determinar qual exceção deve ser lançada em

cada participante, bem como, em qual contexto de execução ela deve ser tratada. Assim, guia-se o tratamento da exceção para o tratador mais apropriado, que pode diferir de um participante para outro.

Apresentou-se o modelo *Guardian* como uma implementação do modelo da terceira geração de tratamento de exceções em sistemas assíncronos. Este se diferencia do modelo de ações atômicas coordenadas em dois aspectos fundamentais: primeiro, quando uma exceção é lançada para outro processo, o *guardian* pode mapeá-la para uma outra diferente, bem como determinar diferentes contextos de tratamento em cada participante. Assim, guia-se o tratamento invocando o tratador mais adequado. No modelo de ações atômicas coordenadas, por sua vez, para cada exceção lançada por um participante, a mesma exceção é repassada para os demais para ser tratada no mesmo contexto de execução. Segundo, quando várias exceções concorrentes são resolvidas em uma nova exceção, o *guardian* pode lançar diferentes exceções com diferentes contextos de tratamento para cada participante, ao passo que no modelo de ações atômicas coordenadas, a exceção resolvida é repassada para todos os participantes com o mesmo contexto de tratamento.

Considerando G como um grupo de participantes associados ao *guardian group*, destaca-se que um participante P, com um contexto de execução $C1$, onde $P \in G$, pode lançar uma exceção $E1$ pra um subgrupo G', onde $G' \subseteq G$, sendo que $E1$ pode ser mapeada, com base nas regras de recuperação, para diferentes exceções $E2, E3, E4, ..., En$, com diferentes contextos de tratamento $C2, C3, C4, ..., Cn$, que serão lançadas em cada participante P', onde $P' \in G'$.

Capítulo 3

Suporte para Tratamento de Exceções Coordenadas no Apache Tuscany SCA

Este capítulo apresenta a forma como o modelo *Guardian* foi implementado no *framework Apache Tuscany SCA*. Num primeiro momento, serão mostrados alguns detalhes relacionados ao Tuscany (seção 3.1), principalmente com relação ao seu ambiente de execução (seção 3.1.1) e sua arquitetura de invocação (seção 3.1.2). Num segundo momento, será apresentado, de fato, como os conceitos chave do modelo *Guardian* foram implementados no *Apache Tuscany SCA* (seção 3.2). Apresenta-se os detalhes para os conceitos do *guardian group* (seção 3.2.1), *guardian primitives* (seção 3.2.2), contextos excepcionais (seção 3.2.3), regras de recuperação (seção 3.2.4), árvore de resolução de exceções (seção 3.2.5) e *guardian members* (seção 3.2.6). A seção 3.3 traz algumas considerações finais sobre o capítulo.

3.1 Apache Tuscany SCA

Apache Tuscany[1] é um projeto *open source* da *Apache Software Foundation*, disponível sob a licensa *Apache2*[2], que fornece uma implementação para uma Arquitetura de Componentes de Serviços (*Service Component Architecture* - SCA), um Objeto de Dados de Serviço (*Service Data Object* - SDO) e um Serviço de Acesso a Dados (*Data Access Service* - DAS). SDO e DAS são tecnologias autônomas e não são pré-requisitos para o uso da SCA.

[1] http://tuscany.apache.org

[2] http://www.apache.org/licenses/LICENSE-2.0.html

SCA permite a composição de aplicações baseadas em serviços de negócios de uma forma neutra com relação a aspectos tecnológicos, isto é, serviços podem ser desenvolvidos e conectados utilizando diferentes tecnologias, tanto para a implementação da lógica de negócio, quanto para resolver aspectos de comunicação. Contudo, a construção do *framework* SCA, em si, é feita com base em uma implementação nativa (atualmente suporta-se Java e C++).

Este trabalho foi desenvolvido com uso do Tuscany SCA Java v1.x, versão baseada nas especificações v1.0 da OSOA.

3.1.1 Ambiente de Execução do Tuscany Java SCA

Como apresentado por Laws [18], o ambiente de execução do Tuscany SCA possui uma arquitetura modular e plugável, permitindo que os usuários escolham funcionalidades de interesse dentre um conjunto de opções. Além disso, tal estrutura torna mais fácil o desenvolvimento de extensões, bem como a integração com outras tecnologias. A Figura 3.1 apresenta a estrutura principal do ambiente de execução do Tuscany SCA.

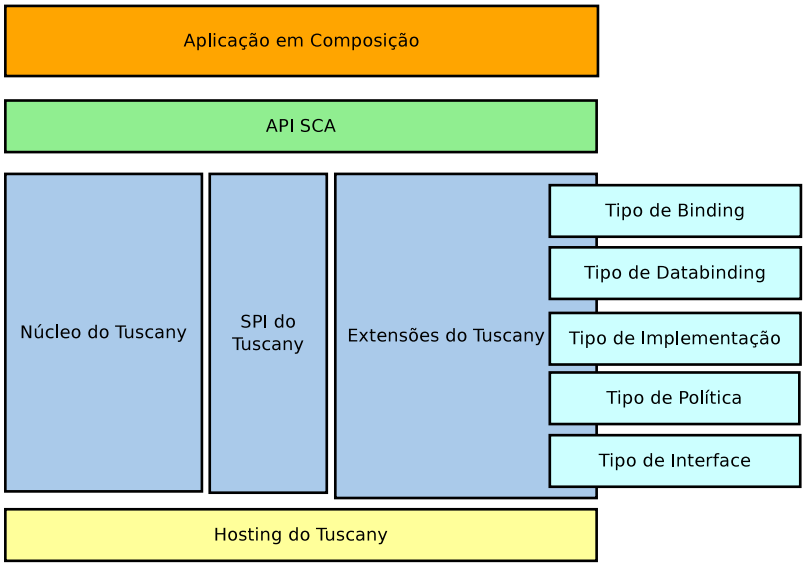

Figura 3.1: Estrutura principal do ambiente de execução do Tuscany SCA. (Adaptado de [18])

- **Aplicação em Composição**: representa a aplicação de negócio que está sendo construída com o uso do Tuscany SCA. A aplicação é descrita através de um modelo XML de montagem (SCDL), definido pela especificação do SCA, onde componentes são descritos, conectados e referenciam artefatos de implementação, tais como uma classe Java, um processo BPEL, ou um script Ruby.

- **API do SCA**: delimita a fronteira entre as aplicações baseadas em composição, escritas com o Tuscany SCA, e o resto do ambiente de execução. É através desta que os componentes definidos no bloco de Aplicação em Composição se comunicam com o ambiente de execução.

- **Núcleo do Tuscany**: oferece suporte para a construção de componentes e serviços, estruturação de componentes em composições, e o gerenciamento das aplicações resultantes.

- **Extensões do Tuscany**: o ambiente de execução do Tuscany SCA foi projetado para ser extensível ao ponto de englobar tanto tecnologias existentes quanto aquelas em desenvolvimento. Os principais pontos de extensão são: *binding*, *databinding*, implementação, política e interface.

  Os *bindings* dão suporte para diferentes tipos de protocolos de comunicação tais como SOAP/HTTP, JSON-RPC, e RMI, utilizados para estabelecer a troca de informação entre componentes, sejam eles pertencentes a uma mesma composição ou não.

  Databindings dão suporte para diferentes formatos de dados que são usados para troca de informações entre serviços, tais como SDO, JAXB e Axiom. O núcleo do Tuscany oferece um *framework* de *databinding* que permite que serviços se comuniquem de maneira transparente, mesmo usando diferentes formatos de dados. Dessa forma, o desenvolvedor fica livre da tarefa de definir explicitamente conversões entre formatos de dados.

  Uma implementação prove suporte para a concretização da lógica de negócio de um componente em alguma tecnologia específica, tal como Java, BPEL, linguagens *script* e o *framework* Spring. O Tuscany SCA permite a troca de informações entre diferentes componentes, utilizando diferentes implementações, sem a necessidade da escrita de um código de adaptação por parte do desenvolvedor. A forma como a comunicação se dá fica a cargo dos *bindings*.

  As extensões de política separam as restrições ou exigências que podem ser aplicadas em um componente, ou na interação entre componentes, do desenvolvimento da lógica de negócio dos serviços. Deste modo, conquista-se flexibilidade no ajuste da estrutura para suportar políticas como autorização, autenticação, confidencialidade e integridade, sem impacto no código da lógica de negócio.

  As extensões de interface permitem que as interfaces de serviços sejam descritas nas mais variadas tecnologias, tais como interface Java e WSDL.

- **SPI do Tuscany**: a *Service Provider Interface* (SPI) do Tuscany permite que novas tecnologias em desenvolvimento sejam agregadas ao Tuscany SCA em forma de extensões. Desse modo, novos *bindings*, *databindings*, tipos de implementação, políticas e interfaces podem ser criados e adicionados ao projeto.

- ***Hosting* do Tuscany**: um conjunto de *hostings* permite que o desenvolvedor escolha, dentre um conjunto de opções, qual a melhor forma de rodar a aplicação feita com uso de composições do Tuscany SCA. Uma variedade de plataformas de *hosting* são suportadas no Tuscany SCA, incluindo Tomcat, Jetty e Geronimo.

Dentre os blocos apresentados do ambiente de execução do Tuscany SCA, aqueles que mais se destacam para o entendimento da solução proposta, e que portanto serão mais detalhados, são os tipos de política e implementação.

Políticas

O termo políticas refere-se a um conjunto de restrições ou exigências que pode ser aplicado a um componente ou à interação entre componentes [7]. Um exemplo de política seria exigir que a troca de mensagens entre componentes seja feita de forma criptografada. Outro exemplo está em exigir a autenticação de um usuário no acesso a um serviço.

A especificação do SCA define dois tipos de políticas: políticas de interação e políticas de implementação [7]. A primeira se aplica na comunicação entre componentes, influenciando na forma como a troca de informação entre serviços e referências ocorre. Políticas de interação associadas a serviços dizem respeito às obrigações que uma referência para tal serviço deve cumprir, ao passo que a forma como uma referência é capaz de interagir com os serviços associados a ela é definida nas políticas de interação associadas às referências. Já as políticas de implementação, descrevem as restrições de execução de um componente dentro do ambiente de execução.

O *framework* de políticas do SCA é composto de dois elementos principais: *intents* e conjunto de políticas[3] [7].

As *intents* são usadas para descrever requisitos de políticas de forma abstrata, tanto de um componente quanto na interação entre componentes via comunicação serviço/ referência. Desse modo, permite-se, a descrição das políticas requeridas no sistema em um nível de abstração que não envolve os detalhes de configuração utilizados em tempo de *deployment*.

Um conjunto de políticas define uma ou mais políticas expressas em alguma forma concreta, como por exemplo em *WS-Policy*. Cada conjunto de políticas está associado a um tipo de *binding* ou implementação. Assim, com base em informações de configuração

[3]Do inglês, *policy set*

associadas a um componente ou composição, utiliza-se o conjunto de política para aplicar uma política específica em um componente ou em um *binding* de um serviço ou referência.

Uma vez que as *intents* dão assistência ao *deployer* na escolha dos *bindings* apropriados e das políticas concretas que satisfazem os requisitos abstratos definidos pela mesma, seu uso permite o *binding* tardio[4] de serviços e referências para um *binding* específico do SCA. O uso do modelo de *binding* tardio promove o reuso de componentes. Assim, um componente pode ser empregado em diferentes contextos de aplicação, sendo que cada contexto pode utilizar um *binding* e uma política concreta diferente. Associar um conjunto de políticas e definir *bindings* durante a fase de escrita do componente limita a sua capacidade de reuso em um novo contexto.

A descrição de uma *intent* é feita no Tuscany SCA através da linguagem XML. O *pseudo-schema* de uma *intent* é mostrado na Figura 3.2, sendo que:

- ***@name***: define o nome da *intent*;
- ***@constrains*** (opcional): especifica qual elemento do SCA a *intent* está designada a configurar (SCA *binding* ou uma implementação);
- ***@requires*** (opcional): especifica o conjunto de todas as *intents* requeridas pela *intent* definida. Tais *intents* devem ser satisfeitas para que a nova *intent* seja aplicada.

```
1 <intent name="NCName" constrains="listOfQNames"? requires="listOfQNames"? >
2   <description>
3     <!-- description of the intent -->
4   </description>
5 </intent>
```

Figura 3.2: *Pseudo-schema* de uma *intent*. ([7])

A Figura 3.3 apresenta um exemplo de definição de uma *intent* para autenticação. Um serviço marcado com esse tipo de *intent* requer que suas mensagens sejam autenticadas, sem se preocupar, no entanto, com a forma como isso é alcançado. Utilizando o modelo de *binding* tardio, quando o *binding* é escolhido para o serviço, por exemplo SOAP sobre HTTP, em tempo de *deployment*, o *deployer* aplica uma política concreta apropriada que satisfaça a política abstrata definida pela *intent*, tal como o uso de *WS-Security*.

Um exemplo de um conjunto de políticas para autenticação com uso do *WS-Security* é mostrado na Figura 3.4. Note que a linha 2 define um conjunto de políticas com nome `wsAuthenticationPolicy` que proverá políticas de autenticação sobre um *binding* que

[4]Do inglês, *late binding*

```
1  <!-- Policy Intents Defined by the SCA Runtime -->
2  <intent name="authentication" constrains="sca:binding tuscany:binding">
3    <description>
4      Specifying this intent on references requires necessary authentication
           information to be sent along with outgoing messages. Specifying this
            intent on service requires incoming messages to be authenticated
5    </description>
6  </intent>
```

Figura 3.3: Exemplo de *intent* descrevendo uma política de autenticação.

faz uso de Web Services (`binding.ws`). As linhas 3-12 trazem os dados de configuração necessários para a autenticação que concretizam a política, em resumo: o nome de um usuário e o acesso a seu *password* através de uma classe de *callback*.

```
1  <!-- WS Security POLICY SETS -->
2  <sca:policySet name="wsAuthenticationPolicy" provides="authentication"
       appliesTo="sca:binding.ws">
3    <tuscany:wsConfigParam>
4      <parameter name="InflowSecurity">
5        <action>
6          <items>UsernameToken</items>
7          <passwordCallbackClass>
8            helloworld.ServerPWCBHandler
9          </passwordCallbackClass>
10       </action>
11     </parameter>
12   </tuscany:wsConfigParam>
13 </sca:policySet>
```

Figura 3.4: Exemplo de um conjunto de política descrevendo uma política de autenticação com uso do *WS-Security*.

Implementação

Define-se como implementação de um componente a representação concreta da lógica de negócio em alguma tecnologia [6]. Por tecnologia entende-se não apenas linguagens de programação, tais como Java e C++, mas também um *framework* ou um ambiente de execução. Cada implementação da especificação do SCA é livre para definir suporte para variados tipos de implementação. Alguns exemplos de implementações de componentes definidas no *Apache Tuscany* são:

- ***implementation.java***: Suporte para componentes SCA implementados com classes Java;

- ***implementation.script***: Suporte para componentes SCA implementados com linguagens de *scripts*;
- ***implementation.spring***: Suporte para componentes implementados com uso do *framework Spring*;
- ***implementation.ejb***: Suporte para componentes implementados utilizando *Enterprise JavaBeans* (EJB);
- ***implementation.bpel***: Suporte para componentes implementados em BPEL (*Business Process Execution Language*) [2];
- ***implementation.osgi***: Suporte para OSGi;
- ***implementation.xquery***: Suporte para componentes implementados em XQuery;
- ***implementation.web***: Suporte para aplicações Web JEE;
- ***implementation.composite***: Suporte para o uso de composições como componentes;

Os aspectos configuráveis de um implementação são denominados tipo do componente[5] [6]. Serviços, referências e propriedades são aspectos configuráveis de uma implementação. Tais aspectos podem ser previamente estabelecidos, bem como terem valores associados às suas características. A Tabela 3.1 reúne as características associadas ao *component type* de uma implementação. Nota-se que uma implementação pode definir uma interface, *binding*, URI, *intents* e conjuntos de política, para serviços e referências. As propriedades podem ter um tipo e um valor associado a este. Por fim, uma implementação pode definir *intents* e conjuntos de política para ela própria.

A maioria das características definidas por uma implementação para serviços, referências e propriedades podem ser sobrescritas por um componente que usa e configura uma dada implementação. No entanto, algumas características não podem ser sobrescritas, como as *intents*. Outras só podem ser modificadas em certas circunstâncias, como o caso das interfaces.

Em tempo de execução, a forma da instância de uma implementação depende da tecnologia usada (uma instância de uma implementação Java é diferente de uma que utiliza BPEL, por exemplo). A lógica de negócio da instância é derivada da implementação em que ela foi baseada, ao passo que os valores para os aspectos configuráveis são derivados do componente que configura tal implementação.

[5]Do inglês, *component type*

	Interface	Binding	URI	Intents	Conjunto de Políticas	Tipo e Valor Padrão
Serviços	•	•	•	•	•	
Referências	•	•	•	•	•	
Propriedades						•
Implementação				•	•	

Tabela 3.1: Características associadas aos aspectos configuráveis de uma implementação. As linhas contemplam os aspectos configuráveis, enquanto as colunas reúnem as características associadas a cada um.

A Figura 3.5 mostra o relacionamento entre implementação, instância da implementação, componente e *component type*. Note que as instâncias da implementação possuem o seu *component type* configurado com base nos valores definidos em um componente. O lado esquerdo da figura mostra os elementos definidos pelo usuário (nível da aplicação em composição), enquanto o lado direito apresenta os elementos definidos pelo Tuscany SCA (nível do núcleo do Tuscany).

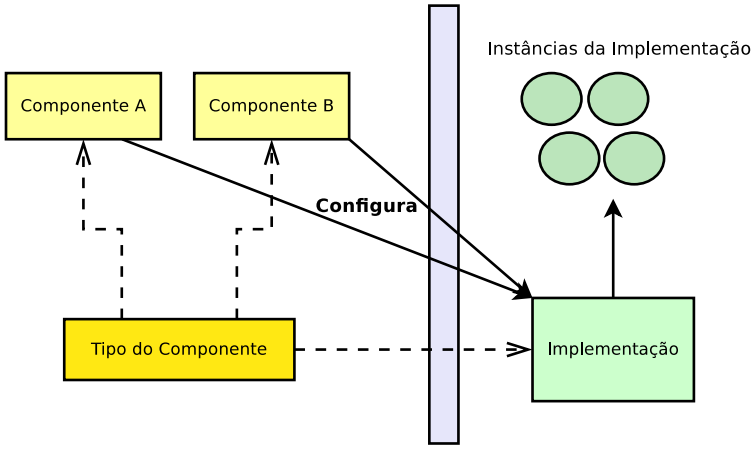

Figura 3.5: Relacionamento entre implementação, instância da implementação, componente e *component type*. (Adaptado de [6])

O *component type* é definido no Tuscany SCA com o auxílio de um arquivo XML, ou então, dependendo do tipo de implementação, determinado por introspecção de código, como com o uso de anotações em Java. A Figura 3.6 mostra o *pseudo-schema* para a definição de um *component type*. Note a existência de um elemento para a definição de cada aspecto configurável: *service*, *reference*, *property* e *implementation*, representando serviço, referência, propriedade e implementação, respectivamente.

```
1 <componentType xmlns="http://www.osoa.org/xmlns/sca/1.0" constrainingType="
    QName"? >
2   <service .../>*
3   <reference .../>*
4   <property .../>*
5   <implementation .../>?
6 </componentType>
```

Figura 3.6: *Pseudo-schema* de um tipo do componente. ([6])

Um exemplo de definição de um *component type* é mostrado na Figura 3.7. O componente que configura tal *component type* deve definir um serviço que tenha uma interface WSDL que referencia "`http://sample/calculator#wsdl.interface(Calculator)`" (linha 3). De modo semelhante, o componente deve definir uma referência com uma interface WSDL que referencia "`http://sample/calculator#wsdl.interface(Divide)`" (linha 7).

```
1 <componentType xmlns="http://www.osoa.org/xmlns/sca/1.0">
2   <service name="CalculatorService">
3     <interface.wsdl interface="http://sample/calculator#wsdl.interface(
        Calculator)"/>
4   </service>
5
6   <reference name="AddService">
7     <interface.wsdl interface="http://sample/calculator#wsdl.interface(
        Divide)"/>
8   </reference>
9 </componentType>
```

Figura 3.7: Exemplo de um *component type*.

3.1.2 Arquitetura de Invocação do Tuscany

Como explicado por Becker [5], ao rodar uma aplicação SCA, a primeira etapa adotada pelo *runtime* do Tuscany consiste em ler os arquivos de configuração de composição SCDL. Os vários arquivos SCDL são inspecionados, e os métodos `factory` ajudam a instanciar os vários objetos em memória que concretizam os componentes especificados. A próxima etapa instancia as ligações[6] que conectam os componentes. Nessa fase, as ligações entre serviços e referências são instanciadas em tempo de execução, levando em consideração os *bindings* mencionados no arquivo de composição.

Um *runtime wire* é uma coleção de cadeias de invocação[7], sendo que há uma cadeia

[6]Do inglês, *runtime wires*
[7]Do inglês, *invocation chains*

de invocação para cada método da interface de negócio. A fim de guiar cada chamada de método para a cadeia de invocação correta, é criado o chamado tratador[8], que funciona como um distribuidor de mensagens. Cada cadeia de invocação é constituída de um conjunto de invocadores[9] e interceptadores[10].

Invocadores provem a lógica de invocação para as tecnologias de implementação e os protocolos de comunicação abstraídos nos *bindings*. Por exemplo, há um `JavaImplementationInvoker` invocado quando se tem componentes utilizando o tipo de implementação `implementation.java`. De forma semelhante, há um `BPELInvoker` invocado quando se tem componentes utilizando o tipo de implementação `implementation.bpel`. Os interceptadores, por sua vez, são um tipo especial de invocador dedicados a prover funcionalidade adicionais, tais como transformação de dados, segurança, autenticação e controle de transação.

Para uma referência de um componente, um *runtime wire* é criado para representar a invocação de saída sobre o *binding* selecionado. Já para um serviço, um *runtime wire* é criado para representar a invocação de entrada com relação ao tipo de implementação.

Além das cadeias de invocação, cadeias de *callback*[11] podem ser adicionadas aos *runtime wires* para promover uma invocação via *callback* a um componente.

A Figura 3.8 representa visualmente a criação dos *runtime wires* em uma comunicação entre a referência e o serviço de dois componentes.

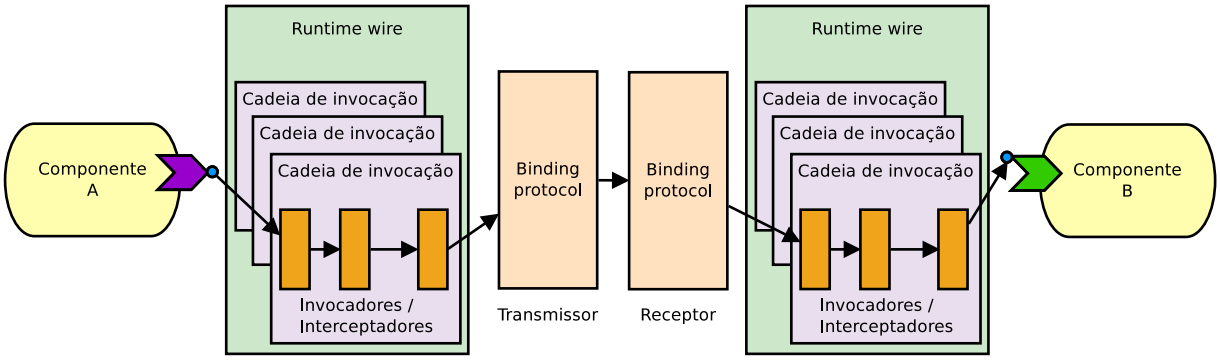

Figura 3.8: Criação dos *runtime wires* na comunicação referência/serviço entre dois componentes. (Adaptado de [5])

Quando o componente `A` faz uso do serviço do componente `B` através de sua referência, o *runtime* do Tuscany cria um *runtime wire* para o serviço e outro para a referência dos componentes referidos. Um *handler* é utilizado pelo *runtime* para descobrir qual

[8]Do inglês, *handler*
[9]Do inglês, *invokers*
[10]Do inglês, *interceptors*
[11]Do inglês, *callback chains*

cadeia de invocação está associada ao método invocado pela referência do componente `A`. Uma vez descoberta, os dados utilizados por parâmetro no método são passados para os invocadores e interceptores, que criam a mensagem e a repassam para o protocolo de comunicação abstraído no *binding* especificado. A parte transmissora envia a mensagem pelo protocolo para a parte receptora, que ao recebê-la responsabiliza-se por encaminhá-la até a cadeia de invocação correta. A mensagem, passada através dos invocadores e interceptadores, é encaminhada para o método correspondente do serviço do componente `B`. O método então é invocado, e caso haja algum valor de retorno, este é repassado ao componente `A`, pelo mesmo caminho, porém em sentido reverso.

3.2 Guardian-SCA: Modelo Guardian Aplicado ao Apache Tuscany SCA

Como apresentado na seção 2.4, o modelo *Guardian* é um modelo de tolerância a falhas aplicado em sistemas distribuídos, que prove solução para os principais problemas envolvendo tratamento de exceções nesse tipo de ambiente, isto é, exceções podem ser lançadas de forma assíncrona, a comunicação e coordenação entre os tratadores de exceções pode ser necessária, e múltiplas exceções podem ser lançadas ao mesmo tempo por diferentes processos participantes. Todos esses problemas convergem para um problema comum: garantir a invocação dos tratadores de exceções semanticamente corretos em todos os processos que participam da recuperação. A solução prevista no modelo conta com a utilização de alguns elementos chaves, a saber: *guardian group*, *guardian primitives*, contextos excepcionais, regras de recuperação, árvore de resolução de exceções e *guardian members*.

As próximas seções detalham como cada um dos elementos do modelo *Guardian* foi implementado no *Apache Tuscany SCA*, culminando no modelo chamado de *Guardian-SCA*.

3.2.1 Guardian Group

Através do uso da SPI do Tuscany, a representação do *guardian group* foi implementada como um novo tipo de implementação: `implementation.guardian`.

A Figura 3.9 apresenta o *pseudo-schema* do novo tipo de implementação definido. Note que a configuração da implementação é feita através do elemento `guardianProperties` (linha 3). Os atributos `@recovery_rules` (linha 4) e `@resolution_trees` (linha 5) permitem a especificação das regras de recuperação e árvores de resolução de exceções, respectivamente. Ambos atributos são especificados através de um arquivo XML.

```
1 <tuscany:implementation.guardian>
2
3   <tuscany:guardianProperties
4     recovery_rules="pathToRecoveryRulesFile"
5     resolution_trees="pathToResolutionTreesFile"/>
6
7 </tuscany:implementation.guardian>
```

Figura 3.9: *Pseudo-schema* para o tipo de implementação *implementation.guardian*.

O *component type* do `implementation.guardian` define `GuardianGroup` como a interface de serviço de um componente que utilize tal implementação. A interface `GuardianGroup` estende a interface `GuardianPrimitives`, que representa a abstração das *guardian primitives*. Desse modo, o *guardian group* é utilizado como um componente, sendo que toda comunicação com este é feita através das *guardian primitives*, definidas como sua interface padrão.

3.2.2 Guardian Primitives

As *guardian primitives* foram implementadas como uma interface de serviço. A interface `GuardianPrimitives`, apresentada na Figura 3.10, traz as operações definidas no modelo *Guardian* que são utilizadas na comunicação entre Participantes $\leftrightarrow$ *Guardian Members* e *Guardian Members* $\leftrightarrow$ *Guardian Group*.

```
1 public interface GuardianPrimitives {
2
3   public void enableContext(Context context);
4   public Context removeContext();
5   public void gthrow(GlobalExceptionInterface ex,
6                      List<String> participantList);
7   public boolean propagate(GlobalExceptionInterface ex);
8   public void checkExceptionStatus() throws GlobalException;
9 }
```

Figura 3.10: *Guardian Primitives* representadas como uma interface de serviço.

Como apresentado na seção 2.4.3, os dois primeiros métodos (linha 3 e 4) da Figura 3.10 são dedicados à gerência dos contextos. As linhas 5 a 7 mostram os métodos encarregados do controle do lançamento de exceções globais. Por fim, a linha 8 traz o método responsável por checar a existência de exceções globais pendentes.

3.2.3 Contextos Excepcionais e Exceções Globais

Os contextos excepcionais, ativados e desativados por meio das *guardian primitives* `enableContext()` e `removeContext()`, são representados através da classe `Context`. Um objeto `Context` abstrai um contexto excepcional e é definido por um nome e uma lista de exceções globais que podem ser tratadas naquele contexto.

Uma exceção global é definida pela classe `GlobalException`. Visto que as exceções globais são do tipo *checked exception*, tal classe estende `java.lang.Exception`. Além de herdar todas as características de `Exception`, uma `GlobalException` agrega um *signaling context*, um *target context* e o participante que lançou a exceção. Todas as exceções globais, que são coordenadas pelo *guardian*, devem estender `GlobalException`.

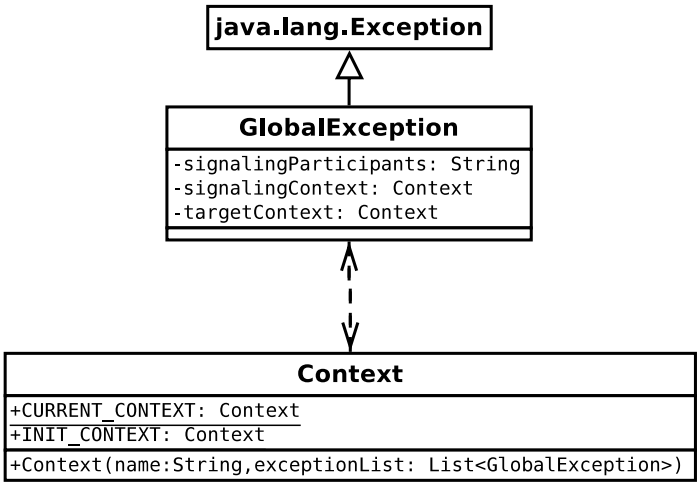

Figura 3.11: Dependências entre `Context`, `GlobalException` e `java.lang.Exception`.

A Figura 3.11 mostra um diagrama de classe resumido as dependências entre `Context`, `GlobalException` e `java.lang.Exception`. Em `GlobalException`, que estende `java.lang.Exception`, nota-se os principais atributos utilizados pelo *guardian* no controle e gerência das exceções globais. `Context` define um construtor que toma como parâmetro um nome e uma lista de exceções globais que podem ser tratadas em tal contexto. Além disso, percebe-se a definição de contextos padrão, que são utilizados internamente pelo *guardian*: `CURRENT_CONTEXT`, representando o contexto corrente de execução de um participante, e `INIT_CONTEXT`, representando o contexto de mais alto nível existente antes da ativação de contextos definidos pelo desenvolvedor.

3.2.4 Regras de Recuperação

As regras de recuperação são fornecidas ao `implementation.guardian` via o atributo `@recovery_rules` do elemento `tuscany:guardianProperties`. Tal atributo recebe o

caminho para um arquivo XML que contém a definição das regras. O *pseudo-schema* para definição das regras de recuperação é apresentado na Figura 3.12.

```
1  <recovery_rules>
2    <rule name="name" signaled_exception="className">
3
4      <participant match="regularExpression">
5        <throw_exception class="className"
6            target_context="contextName"
7            min_participant_joined="number"?
8            max_participant_joined="number"?>
9          <affected_participants>keyword</affected_participants>?
10       </throw_exception>
11     </participant>*
12
13   </rule>*
14 <recovery_rules>
```

Figura 3.12: *Pseudo-schema* para a definição das regras de recuperação.

O *pseudo-schema* foi criado com base na análise de exemplos de regras de recuperação definidas por Miller em seus trabalhos [23, 25, 26]. Em tais trabalhos, as regras de recuperação são definidas pelo desenvolvedor da aplicação com o uso de estruturas condicionais `if-else`. Essa abordagem permite que o desenvolvedor utilize, na definição das regras, todos os recursos oferecidos pela linguagem de programação em que o modelo foi implementando. Contudo, aumenta-se a possibilidade da existência de falhas ocasionadas por erros na lógica de programação, além de exigir um maior conhecimento do desenvolvedor de como as regras devem ser implementadas.

A definição de uma linguagem XML para descrição das regras de recuperação permite que o desenvolvedor concentre-se na definição das ações a serem tomadas, sem a necessidade de saber como essas regras vão ser implementadas. O código gerado por reflexão computacional evita a existência de possíveis erros de lógica de programação. Além disso, a existência de um arquivo para descrição das regras confere um maior dinamismo à aplicação, já que não há a necessidade de recompilação de código. Assim, as regras de recuperação podem ser facilmente modificadas sem que isso impacte no código que controla a sua execução.

Conforme apresentado na Figura 3.12, uma ou mais regras são definidas com o uso do elemento `rule` (linha 2). Para cada `rule` especifica-se um nome (via atributo `@name`) e a exceção para a qual aquela regra se aplica (via atributo `@signaled_exception`). Cada elemento `rule` pode selecionar um ou mais conjunto de participantes para atribuir uma nova exceção. Isso se dá com o uso do elemento `participant` (linha 4).

Um `participant` seleciona um conjunto de participantes através do uso de uma ex-

pressão regular atribuída ao atributo `@match`. Já que um participante é identificado pela lista dos nomes de seus contextos, a expressão regular é formada por nomes de contextos (aqui representado por `<Context>`) separados pelo caractere `.`. O caractere `*` pode ser usado como um coringa, significando que não importa em qual contexto o participante se encontra.

Como mostrado na Figura 3.13, algumas expressões válidas são: `*`, `<Context>`, `*.<Context>`, `<Context>.*`, `*.<Context>.*`. `*` seleciona todos os participantes; `<Context>` e `*.<Context>` selecionam apenas os participantes cujo contexto atual é `<Context>`; `<Context>.*` seleciona os participantes cujo primeiro contexto é `<Context>`; `*.<Context>.*` seleciona todos os participantes que tenham o contexto `<Context>` ativado em algum momento, sendo que este não é nem o primeiro nem o último contexto a ser ativado.

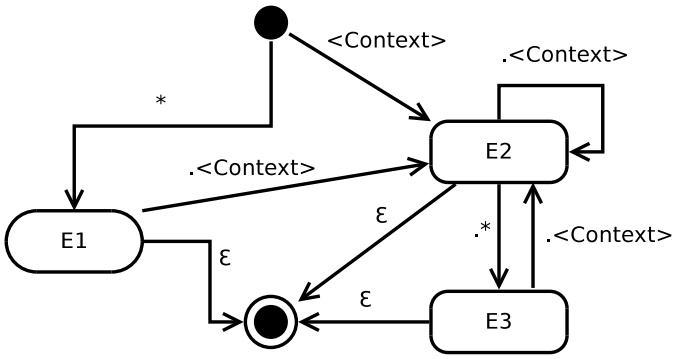

Figura 3.13: Máquina de estado para representação das expressões regulares válidas. Tais expressões são utilizadas pelas regras de recuperação na seleção de um conjunto de participantes.

A fim de identificar um conjunto de participantes, uma palavra-chave pode ser atribuída ao atributo `@match`, ao invés de uma expressão regular. Com base no exemplos de aplicações definidos por Miller, extraiu-se `SIGNALER` como uma palavra-chave que representa o participante que lançou a exceção definida no atributo `@signaled_exception` do elemento `rule`.

Definiu-se ainda o caractere `||` para expressar o operador lógico `OU`. Seja `<EXP>` uma expressão regular validada segundo a máquina de estados representada na Figura 3.13, ou então uma palavra-chave, é possível a utilização de expressões do tipo $\texttt{<EXP}_1\texttt{>}$ `||` $\texttt{<EXP}_2\texttt{>}$ `|| ... ||` $\texttt{<EXP}_n\texttt{>}$ como valor do atributo `@match`.

Uma vez que os participantes foram selecionados, utiliza-se o elemento `throw_exception` (linha 5) para definir a exceção a ser lançada naquele conjunto. A exceção é informada através do atributo `@class`, ao passo que o contexto em que tal exceção deve ser lançada para tratamento é informado no atributo `@target_context` (linha 6). Os atributos opcionais `@min_participants_joined` (linha 7) e `@max_participants_joined` (linha 8) são

utilizados para especificar, respectivamente, o número mínimo e máximo necessário de processos participantes associados ao *guardian group*, para que a exceção indicada no atributo `@class` do elemento `throw_exception` seja lançada. Em uma aplicação de réplica de servidores, por exemplo, pode-se utilizar tais atributos para controlar o número máximo de falhas em réplicas. Suponha uma arquitetura com um servidor e cinco réplicas. Atribuir o valor 3 em `@max_participants_joined` impõe que, no mínimo o servidor e mais três réplicas devem estar ativas, ou seja, no máximo há falha em duas réplicas. A falha de uma terceira réplica ocasiona a validação da condição (a quantidade de participantes passa de 4 para 3), e por consequência leva ao lançamento de uma exceção, que pode ser disparada para finalizar o servidor, ou então acionar o funcionamento de novas réplicas.

O elemento opcional `affected_participants` (linha 9) é utilizado para filtrar, dentre o conjunto de participantes selecionados pela expressão regular fornecida em `@match`, quais participantes realmente terão a nova exceção lançada neles. O modelo para definição das regra suporta o uso das palavras-chave `FIRST` e `LAST`. `FIRST` seleciona o primeiro participante da lista de participantes, ao passo que `LAST` seleciona o último. A ordem dos participantes é determinada pela ordem em que o *guardian* recebe as requisições para associação.

3.2.5 Árvore de Resolução de Exceções

Assim como as regras de recuperação, a árvore de resolução de exceções é fornecida ao `implementation.guardian` através de um arquivo XML. O atributo `@resolution_trees`, pertencente ao elemento `tuscany:guardianProperties`, guarda o caminho para tal arquivo. A Figura 3.14 apresenta o *pseudo-schema* para a definição da árvore de resolução de exceção.

```
1 <resolution_trees>
2   <resolution_tree exception_level="level">
3     <exception class="className" />*
4   </resolution_tree>*
5 </resolution_trees>
```

Figura 3.14: *Pseudo-schema* para a definição das árvores de resolução de exceções.

Como previsto no modelo *Guardian*, uma ou mais árvores de resolução de exceções podem ser definidas e associadas para cada nível de exceção. A definição de uma árvore para um nível específico é realizada através do elemento `resolution_tree` (linha 2), onde o atributo `@exception_level` determina o nível das exceções para a qual aquela árvore se aplica.

A hierarquia das exceções é construída através do aninhamento do elemento `exception` (linha 3), sendo que o nome da classe de cada exceção envolvida na hierarquia é informada no atributo `@class`.

Algoritmo para Busca do Menor Ancestral Comum

Seja $\mathtt{E}_1$, $\mathtt{E}_2$, ..., $\mathtt{E}_n$ as exceções lançadas concorrentemente por diferentes participantes, o mecanismo de árvore de resolução de exceções promove a resolução através da busca do menor ancestral comum entre as `N` exceções.

O problema de busca do menor ancestral comum[12] em uma árvore foi solucionado com a implementação do algoritmo proposto por Bender e Farach [8, 21]. Tal algoritmo foi escolhido por oferecer uma desempenho em tempo linear no pré-processamento da árvore e responder `lca(i,j)` em tempo constante, onde `lca(i,j)` é a função que computa o menor ancestral comum entre os nodos `i` e `j`.

A Figura 3.15 resume o funcionamento do algoritmo de Bender e Farach.

$$\boxed{lca(i,j) = E[RMQ_L(R[i], R[j])}$$

Figura 3.15: Equação que expressa o funcionamento do algoritmo de Bender e Farach para computação do menor ancestral comum em uma árvore.

Seja `T` uma árvore com `n` nodos. Seja `E` um passeio euleriano da árvore obtido pela listagem dos nodos visitados em uma busca em profundidade[13] de `T` começando pelo nodo raiz. Seja `L` um vetor de níveis, onde `L[i]` contém a profundidade da árvore para o nodo `E[i]`. Ambos os vetores `E` e `L` contém $2n-1$ elementos e podem ser construídos em tempo linear através do algoritmo de busca em profundidade. Seja `R` um vetor de tamanho `n`, tal que `R[i]` contém o índice para a primeira ocorrência do nodo `i` em `E`. Seja $\mathtt{RMQ}_L\mathtt{(i,j)}$[14] a função que denota a posição da ocorrência do menor elemento no vetor `L` entre os índices `i` e `j` (inclusivo). Assim, para os nodos `i` e `j`, o menor ancestral comum deles é o nodo com menor profundidade na árvore que é visitado entre a ocorrência de `i` e a ocorrência de `j` em um passeio euleriano.

Vejamos um exemplo do funcionamento do algoritmo. Seja `T` a árvore apresentada na Figura 3.16 cuja raiz é `N0` e `N1` a `N6` são nodos intermediários.

Os primeiros passos do algoritmo envolvem uma etapa de pré-processamento em que os vetores `E`, `L` e `R` são computados.

Executando um passeio euleriano a partir de `N0` em `T`, utilizando uma busca em profundidade, obtém-se o vetor `E`:

[12] Do inglês, *Lowest Common Ancestor - LCA*
[13] Do inglês, *Depth First Search*
[14] Abreviação para *Range Minimum Query*

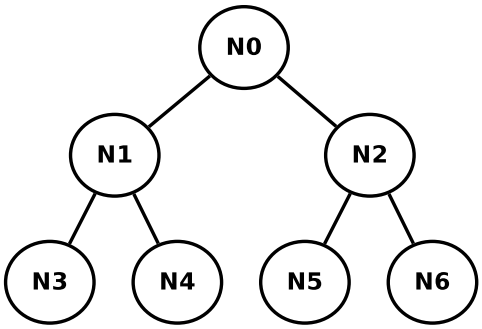

Figura 3.16: Árvore de resolução de exceções, sendo `N0` a raiz e `N1` a `N6` nodos intermediários.

$$\boxed{E = \{N0, N1, N3, N1, N4, N1, N0, N2, N5, N2, N6, N2, N0\}}$$

O vetor L é obtido através da computação da profundidade de cada nodo da árvore presente em `E`:

$$\boxed{L = \{0, 1, 2, 1, 2, 1, 0, 1, 2, 1, 2, 1, 0\}}$$

`L[0]=0` já que o nodo `N0`, presente em `E[0]`, está no nível 0 da árvore. De forma semelhante, L[8]=2 já que o nodo `N5`, presente em `E[8]`, está no nível 2 da árvore. Uma vez que `L` referencia cada elemento de `E`, ambos os vetores possuem o mesmo tamanho igual a $2n - 1$. Para o exemplo em questão tem-se $2(7) - 1 = 13$ elementos.

A computação de R resulta num vetor de tamanho igual a `n`, sendo que cada posição `R[i]` contém o índice da primeira ocorrência do nodo `i` em `E`. Considerando os índices 0 a 6 representando os nodos `N0` a `N6`, respectivamente, tem-se:

$$\boxed{R = \{0, 1, 7, 2, 4, 8, 10\}}$$

Assim, `R[0]` $= 0$ já que a primeira ocorrência de `N0` em `E` é na posição 0. Semelhantemente `R[5]` $= 8$ já que a primeira ocorrência de `N5` é na posição 8. O mesmo se aplica para as demais posições de `R`.

Após o pré-processamento, basta aplicar a função apresentada na Figura 3.15 para a obtenção do menor ancestral comum entre dois nodos. Por exemplo, suponha que se deseja o menor ancestral comum entre os nodos `N3` e `N5`, logo tem-se:

$$\boxed{\begin{aligned} lca(N3, N5) &= E[RMQ_L(R[3], R[5]) \\ &= E[RMQ_L(2, 8)] \\ &= E[6] \\ &= N0 \end{aligned}}$$

Quando se tem mais de um exceção lançada concorrentemente, o algoritmo é aplicado para um par de exceções, e o resultado é aplicado para a próxima exceção da lista, ou seja, sejam $E_1, E_2, ..., E_N$ as exceções lançadas concorrentemente, a exceção resolvida E_R é calculada por $lca(E_1, E_2, ..., E_N) = lca(E_N, lca(..., (lca(E_4, lca(E_3, lca(E_1, E_2))))))$.

Retornando ao exemplo da árvore apresentada na Figura 3.16, suponha que deseja-se calcular o menor ancestral comum entre `N2`, `N3` e `N5`. Assim, tem-se:

$$\begin{aligned} lca(N2, N3, N5) &= lca(N2, lca(N3, N5)) \\ &= lca(N2, N0) \\ &= E[RMQ_L(R[2], R[0]) \\ &= E[RMQ_L(7, 0)] \\ &= E[0] \\ &= N0 \end{aligned}$$

Note que, não importando a ordem em que fossem realizadas as operações de `lca` o resultado seria o mesmo.

3.2.6 Guardian Members

Os *guardian members* foram implementados no *Apache Tuscany SCA* como um novo tipo de política, mais especificamente, uma nova *intent*: `guardianExceptionHandling`. Assim como na definição do `implementation.guardian`, utilizou-se a SPI do Tuscany para a definição desse novo tipo de política. A Figura 3.17 apresenta a definição da *intent* `guardianExceptionHandling`. Note que o atributo `@constrains`, na linha 5, especifica que tal *intent* se aplica às referências de um componente java.

A aplicação de tal *intent* cria um interceptador nas cadeias de invocação associadas aos métodos invocados pela referência em que a *intent* foi aplicada. Tal interceptador é responsável por invocar o *guardian member* correspondente ao participante.

A Figura 3.18 demonstra a estrutura utilizada na implementação dos *guardian members*. Note que ao invocar um método de uma referência marcada com a *intent* `guardian-ExceptionHandling` (abreviada na figura para "`guardianEH`"), um interceptador `Guardian-ExceptionHandlingInterceptor` é criado na cadeia de invocação daquele método. Cada interceptador utiliza a *factory* `GuardianMemberFactoryImpl` para obter a instância concreta do *guardian member* correspondente do seu participante (`GuardianMemberImpl`). A utilização da *factory* permite o controle e manutenção dos *guardian members*, de forma que apenas um *guardian member* seja criado para cada componente participante presente na composição.

```
1  <definitions xmlns="http://www.osoa.org/xmlns/sca/1.0"
2    targetNamespace="http://tuscany.apache.org/xmlns/sca/1.0"
3    xmlns:sca="http://www.osoa.org/xmlns/sca/1.0">
4
5    <intent name="guardianExceptionHandling" constrains="sca:implementation.
         java/sca:reference">
6      <description>
7        All messages to the guardian group implementation will be intercepted
             by the guardian member.
8      </description>
9    </intent>
10 </definitions>
```

Figura 3.17: Definição da *intent* `guardianExceptionHandling` para abstração dos *guardian members*.

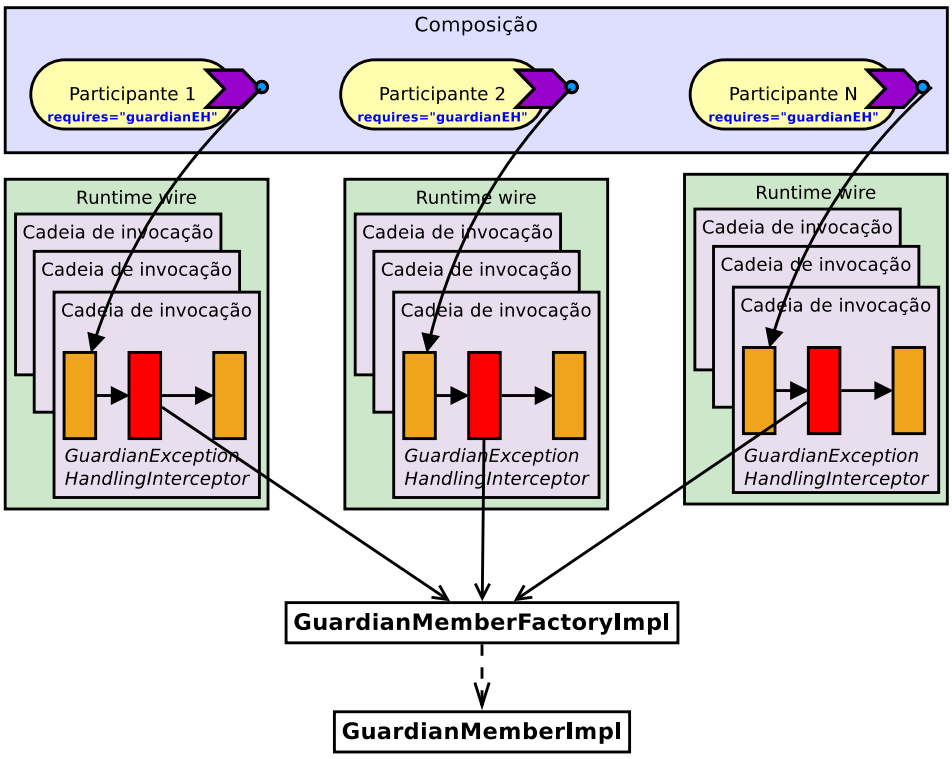

Figura 3.18: Representação interna do *guardian member* nas cadeias de invocação do Tuscany SCA.

Pensando no *guardian member* como um componente de serviço, teríamos sua implementação definida por `GuardianMemberImpl`, com uma interface de serviço definida por `GuardianMember` (interface estendida de `GuardianPrimitives`), e uma referência para o componente que define o *guardian group*. Contudo, se fosse implementado de tal forma,

haveria uma necessidade de declaração explícita dos componentes *guardian members* no arquivo de configuração SCDL. Já que a comunicação Participantes ↔ *Guardian Members*, e *Guardian Members* ↔ *Guardian Group* se dá através das `GuardianPrimitives`, pode-se utilizar o recurso de políticas para abstrair a existência dos *guardian members*, simplificando a comunicação para Participantes ↔ *Guardian Group*.

3.3 Considerações Finais

Este capítulo foi dividido em duas partes. A primeira dedica-se a apresentar maiores informações sobre o *framework* Tuscany SCA, mostrando detalhes do seu ambiente de execução e de sua arquitetura de invocação. A segunda parte mostra como os conceitos chaves do modelo *Guardian* foram implementados sobre essa estrutura. Em resumo, tem-se a definição de um novo tipo de implementação (`implementation.guardian`), uma nova *intent* (`guardianExceptionHandling`) e duas linguagens XML para descrição das regras de recuperação e árvores de resolução de exceções, respectivamente. A solução foi alcançada de forma evolutiva, e com base em discussões realizadas com a comunidade *Apache Tuscany SCA*.

Capítulo 4

Estudo de Caso: Exemplo de Utilização do modelo Guardian-SCA - Servidor com N Réplicas

Este capítulo apresenta uma aplicação distribuída, implementada através do Tuscany SCA, que utiliza o modelo *Guardian-SCA* para prover um mecanismo de tolerância a falhas de réplicas em um servidor. A estrutura da aplicação é apresentada na Figura 4.1. Nessa aplicação, um servidor recebe requisições de um cliente (`1.requisição`), as processa, e devolve o resultado (`2.resposta`). A cada requisição, as réplicas são informadas do novo estado do servidor (`2x.envia atualização`), para que possam manter seu estado consistente (`3x.aplica atualização`). Em caso de falha no servidor, uma nova réplica assume tal papel. A falha de uma réplica leva à remoção desta da lista de réplicas do servidor, evitando o envio de atualizações desnecessárias. Essa aplicação foi uma adaptação do exemplo "*Primary-Backup System with N-Backups*" apresentado por Miller em seu trabalho [26].

O presente capítulo está organizado da seguinte forma: a seção 4.1 mostra como estruturar a aplicação utilizando o modelo *Guardian-SCA*; a seção 4.2 apresenta a utilização do modelo de programação do *Guardian-SCA*, via invocação das *guardian primitives*, dentro dos componentes participantes; posteriormente, nas seções 4.3 e 4.4, apresentam-se as regras de recuperação e a árvore de resolução de exceções associadas à aplicação; por fim, na seção 4.5, mostra-se os principais eventos ocorridos na execução da aplicação em situações com e sem a presença de erros. A seção 4.6 traz algumas considerações finais sobre o capítulo.

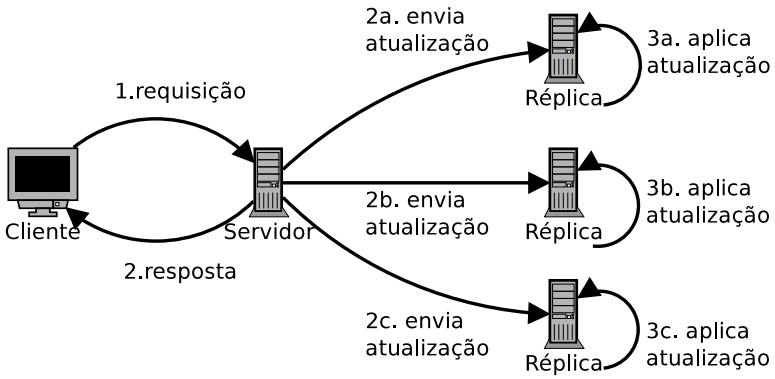

Figura 4.1: Estrutura da aplicação de servidor com N réplicas.

4.1 Estrutura da Aplicação

Uma vez que cada réplica deve manter o mesmo estado do servidor, toda alteração feita neste deve ser informada àquelas. Do mesmo modo, a ocorrência de uma falha, seja no servidor ou em alguma das réplicas, deve ser informada para os demais participantes. No caso de falha no servidor, as réplicas devem ser informadas, de forma que alguma delas possa assumir o papel de servidor. Para o caso de falha em alguma das réplicas, o servidor deve ser informado para que assim não envie mais mensagens de atualização para tal réplica.

Pelo fato do servidor e das réplicas rodarem paralelamente e se comunicarem através da troca de mensagens assíncronas, duas ou mais falhas podem ocorrer na mesma janela de tempo, caracterizando uma situação de exceções concorrentes.

A utilização do modelo *Guardian* facilita a construção de tal aplicação, tanto para o caso em que há ocorrência de falhas isoladas, quanto na existência de exceções concorrentes. A Figura 4.2 apresenta, por meio do modelo de componentes de serviço do Tuscany SCA, a estrutura da aplicação encapsulada na composição `ServerBackupComposition`.

Cada componente `Participant` representa um nodo que pode ser configurado para trabalhar tanto como um servidor quanto como uma réplica. Note que cada um desses componentes apresenta uma interface de serviço, utilizada na comunicação entre o servidor e suas réplicas (omitiu-se a interface de serviço consumida pelo cliente na comunicação com o servidor), e duas referências. Uma referência possibilita que cada participante tenha conhecimento um do outro e possam trocar informações (omitiu-se as ligações entre os participantes para efeito de simplificação visual). A outra, marcada com um `*`, é utilizada na comunicação com o componente `GuardianComponent`. O `*` na figura representa que tal referência foi marcada com a *intent* `guardianExceptionHandling`.

A descrição da Figura 4.2, por meio da *Service Component Definition Language*, é apresentada na Figura 4.3.

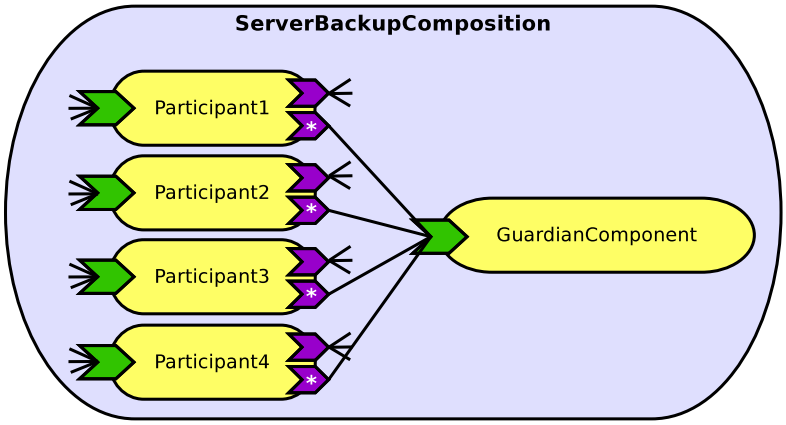

Figura 4.2: Estrutura da aplicação de servidor com N réplicas utilizando o Tuscany SCA com a extensão do modelo *Guardian*.

As linhas 5-16 trazem a definição do componente `Participant1`. Tal componente foi implementado através de uma implementação Java (`implementation.java`), abstraída na classe `NodeImpl` (linha 6). A interface de serviço trata-se de uma interface java `Node`, implementada por `NodeImpl` (linha 8-10). O componente conta ainda com uma referência `nodes`, que aponta para cada um dos outros componentes participantes (linhas 12-13), e outra denominada `guardian` que aponta para o componente que abstrai o *guardian group* (linhas 14-15). Note que esta última está marcada com `requires=‘‘tuscany:guardian-ExceptionHandling’’`, o que levará à criação de um *guardian member* para tal participante, durante a fase de elaboração das cadeias de invocação.

As linhas 18-29 apresentam a definição, de forma semelhante, do componente `Participant2`. Repare que a principal diferença se dá na definição da referência `nodes`, modificada para referenciar os componentes `Participant1`, `Participant3` e `Participant4` (linhas 25-26).

Os componentes `Participant3` e `Participant4`, por serem definidos de forma semelhante, tiveram suas descrições omitidas (linhas 31-33).

As linhas 35-42 apresentam a definição do componente `GuardianComponent`, que abstrai o *guardian group*. Por utilizar uma implementação do tipo `implementation.guardian` (linha36), é necessário a definição do caminho para os arquivos quem contém as regras de recuperação e as árvores de resolução de exceções, através dos atributos `@recovery_rules` (linha 39) e `@resolution_trees` (linha40), respectivamente. Esses arquivos serão detalhados posteriormente.

Note que, apesar dos participantes possuírem uma ligação direta para o componente que representa o *guardian group*, isso não altera o modelo de comunicação proposto por Miller, onde participantes se comunicam com *guardian members*, e estes, por sua vez, se comunicam o *guardian group*. Diz-se isso porque a *intent* requerida na referência

```
1  <composite xmlns="http://www.osoa.org/xmlns/sca/1.0"
2    xmlns:tuscany="http://tuscany.apache.org/xmlns/sca/1.0"
3    name="ServerBackupComposition">
4
5    <component name="Participant1">
6      <implementation.java class="NodeImpl"/>
7
8      <service name="NodeInterface">
9        <interface.java interface="Node"/>
10     </service>
11
12     <reference name="nodes"
13       target="Participant2 Participant3 Participant4"/>
14     <reference name="guardian" target="GuardianComponent"
15       requires="tuscany:guardianExceptionHandling"/>
16   </component>
17
18   <component name="Participant2">
19     <implementation.java class="NodeImpl"/>
20
21     <service name="NodeInterface">
22       <interface.java interface="Node"/>
23     </service>
24
25     <reference name="nodes"
26       target="Participant1 Participant3 Participant4"/>
27     <reference name="guardian" target="GuardianComponent"
28       requires="tuscany:guardianExceptionHandling"/>
29   </component>
30
31   <component name="Participant3"> ... </component>
32
33   <component name="Participant4"> ... </component>
34
35   <component name="GuardianComponent">
36     <tuscany:implementation.guardian>
37
38     <tuscany:guardianProperties
39       recovery_rules="recoveryRules.xml"
40       resolution_trees="resolutionTrees.xml"/>
41     </tuscany:implementation.guardian>
42   </component>
43 </composite>
```

Figura 4.3: Descrição da aplicação de servidor com N réplicas via SCDL.

dos componentes `Participant` para o componente `GuardianComponent` leva à criação implícita de cada *guardian member*.

```
1 public interface Node {
2   @OneWay
3   public void execute();
4
5   public void sendUpdate();
6
7   public void applyUpdate();
8 }
```

Figura 4.4: `Node.java`: Interface de serviço dos componentes `Participant`.

A interface de serviço `Node` é apresentada na Figura 4.4. Observa-se a existência de três métodos: `execute()`, `sendUpdate()` e `applyUpdate()`. O primeiro é usado pra iniciar a execução do componente participante. Note a existência da *annotation* `@OneWay`, definida pelo Tuscany, indicando que a execução desse método será assíncrona. O segundo método é usado pelo servidor para enviar atualizações às réplicas. Por fim, o método `applyUpdate()` é utilizado pelas réplicas para aplicar as atualizações recebidas do servidor.

Para efeito de simplificação, nenhum *binding* foi especificado, o que encarrega o *runtime* do Tuscany de definir o *binding* mais apropriado. Rodar a aplicação localmente, ou de forma distribuída (especificando *bindings*), não acarreta em grandes impactos na elaboração dos componentes, visto que a principal diferença está no protocolo de comunicação escolhido. Assim, apenas o arquivo de configuração SCDL deve ser alterado.

4.2 Utilização do Modelo Guardian-SCA

Como pode ser notado na Figura 4.3, todos os participantes são instâncias de uma mesma classe: `NodeImpl`. A Figura 4.5 apresenta a estrutura geral da utilização do *guardian* na classe, via invocação das *guardian primitives*.

Primeiramente, dentro de um determinado escopo, ativa-se um contexto através da *guardian primitive* `enableContext()` (linha 4). O contexto permanece ativo até que a *guardian primitive* `disableContext()` seja invocada (linha 15), ou então um contexto aninhado seja ativado. Entre a invocação de `enableContext()` e `disableContext()` tem-se a definição do corpo do contexto. Como cada contexto deve possuir, associado a este, tratadores capazes de tratar as exceções definidas no momento de sua criação, inicia-se um bloco `try-catch`[1] (linha 6). Após a ativação do contexto e início do bloco de tratamento, é necessário checar se há ou não exceções globais pendentes a serem tratadas.

[1] A utilização do termo "bloco `try-catch`" refere-se tanto a blocos em que há somente um `try` seguido de um ou mais `catch`, quanto àqueles que fazem uso da cláusula `finally`.

```
1  //Escopo
2  {
3    //Ativa um contexto
4    guardian.enableContext(c);
5
6    try{
7      //Checa por excecoes pendentes
8      guardian.checkExceptionStatus();
9
10     //Codigo especifico da aplicacao
11     . . .
12
13   }catch (Exception e) { . . . }
14   finally {
15     guardian.removeContext();
16   }
17 }
```

Figura 4.5: Estrutura geral da utilização do modelo *Guardian* na classe `NodeImpl`.

Isso é feito através da invocação da *guardian primitive* `checkExceptionStatus()` (linha 8). Posteriormente, tem-se o código específico da aplicação (linha 10). As linhas 13-15 apresentam as cláusulas `catch` e `finally`, utilizadas no tratamento das possíveis exceções a serem lançadas em tal contexto, sejam elas locais ou globais.

Note que a invocação das *guardian primitives* é feita através da referência mantida para o *guardian group*, denominada `guardian`. A aplicação invoca o *guardian* como se fosse uma chamada de procedimento de um objeto local, deixando a cargo do Tuscany, a função de efetuar as devidas transformações para que a comunicação se realize sobre o *binding* especificado.

Optou-se por associar o escopo da estrutura apresentada na Figura 4.5 ao escopo de um método, acarretando na criação de um método em `NodeImpl` para cada contexto utilizado. A Figura 4.6 apresenta, de forma mais detalhada, a estrutura da classe `NodeImpl`. As linhas 4-6 mostram a definição das variáveis que representam os contextos: `mainContext`, `primaryContext` e `backupContext`. O primeiro representa um contexto neutro de configuração que cada nodo ativa antes de assumir o papel de servidor ou réplica. O segundo, representa o contexto do servidor, ao passo que o terceiro é utilizado como contexto das réplicas. A variável `role` (linha 7) é utilizada para definir o papel de cada nodo, como será mostrado posteriormente. As linhas 9-13 definem as referências utilizadas pelos componentes `Participant`. Utiliza-se a *annotation* `@Reference`, definida pelo Tuscany SCA, onde os atributos `name` e `required` indicam, respectivamente, o nome utilizado para denotar tal referência no arquivo SCDL, e sua obrigatoriedade. As linhas 9-10 trazem a definição da referência `nodes`, enquanto a referência `guardian` é mostrada nas linhas 12-

13. Ambas são obrigadas a aparecer de forma explicita no arquivo SCDL (`required = true`). Nas linhas 15-26, tem-se a instanciação dos contextos no construtor da classe. Note que cada objeto é construído com base em uma *String*, representando o nome simbólico daquele contexto, e uma lista de exceções globais que podem ser tratadas no mesmo. As exceções globais definidas para cada contexto, bem como a situação excepcional que representam, podem ser visualizadas na Tabela 4.1.

```
1  @Scope("COMPOSITE")
2  public class NodeImpl implements Node {
3
4    private Context mainContext;
5    private Context primaryContext;
6    private Context backupContext;
7    private int role;
8
9    @Reference(name = "nodes", required = true)
10   public List<Node> nodeList;
11
12   @Reference(name = "guardian", required = true)
13   public GuardianGroup guardianGroup;
14
15   public NodeImpl() {
16     mainContext = new Context("MAIN");
17     mainContext.addException(PrimaryFailedException.class);
18     mainContext.addException(PrimaryExistsException.class);
19
20     primaryContext = new Context("PRIMARY");
21     primaryContext.addException(BackupFailedException.class);
22     primaryContext.addException(BackupJoinedException.class);
23
24     backupContext = new Context("BACKUP", null);
25   }
26
27   @OneWay
28   public void execute() {...}
29
30   private void primaryService() {...}
31
32   private void backupService() {...}
33 }
```

Figura 4.6: Estrutura mais detalhada da classe `NodeImpl`.

As linhas 27-32 apresentam os três métodos utilizados para se aplicar a estrutura presente na Figura 4.5, onde um contexto é ativado, e as atividades específicas da aplicação para aquela região são definidas dentre de um bloco de tratamento. Os métodos `execute()` (linha28), `primaryService()` (linha30) e `backupService()` (linha 32) aplicam-se, respec-

Exceção	Contexto	Significado
PrimaryFailedException	MAIN, PRIMARY	Erro interno ocorrido no servidor primário
PrimaryExistsException	MAIN	Indica a existência de um servidor primário ativo
BackupFailedException	MAIN, PRIMARY, BACKUP	Erro interno ocorrido em uma réplica
BackupJoinedException	PRIMARY	Indica que uma nova réplica foi ativada

Tabela 4.1: Exceções globais associadas a cada contexto e as situações excepcionais abstraídas por essas.

tivamente, para os contextos `MAIN`, `PRIMARY` e `BACKUP`.

A descrição do método `execute()` é mostrada na Figura 4.7. Seguindo a estrutura da Figura 4.5, o método começa ativando o contexto `MAIN` na linha 3. Uma vez ativado, inicia-se a definição do bloco `try-catch` (linhas 6-24), porém, com a diferença de que este se encontra dentro de um laço `while(true)` (linha 5). Utiliza-se tal artifício já que os nodos, sejam eles na condição de servidor ou réplica, executam constantemente, seja esperando por requisições dos clientes, no caso do servidor, ou por atualizações, no caso das réplicas. A linha 7 checa a existência de exceções globais pendentes para o contexto `MAIN`. Caso não haja exceções a serem disparadas, o método continua sua execução, que consiste em configurar o nodo como servidor (`primaryService()`) ou réplica (`backupService()`), dependendo do valor associado a variável `role` (linhas 9-13). O comportamento excepcional será analisado posteriormente, mas note que uma cláusula `catch` foi associada para cada exceção global que o contexto pode tratar, permitindo a definição de uma ação específica a ser tomada dependendo da exceção lançada. As ações consistem em: pré-configurar o participante como uma réplica, atribuindo o valor `BACKUP` à variável `role`, mediante a ocorrência de uma `PrimaryExistsException` (linhas 15-16); propagar uma exceção `PrimaryFailedException` para o contexto superior ao contexto `MAIN`, caso a *guardian primitive* `propagate()` retorne `true`, ou pré-configurar o participante como um servidor, atribuindo o valor `PRIMARY` a variável `role` (linhas 18-20); ou, na ocorrência de uma `BackupFailedException`, propagar tal exceção para o contexto superior, se necessário (linhas 22-23).

O método `primaryService()` é apresentado na Figura 4.8. A mesma estrutura utilizada na construção do método `execute()` é aplicada na construção de `primaryService()`, sendo que as principais diferenças se dão no contexto que é ativado, no código da aplicação específico e nos tratadores definidos. O contexto ativado em `primaryService()` é aquele

```
1  @OneWay
2  public void execute() {
3    guardian.enableContext(mainContext);
4
5    while (true) {
6      try {
7        guardian.checkExceptionStatus();
8
9        //Configurando como servidor...
10       if (role == PRIMARY) primaryService();
11
12       //Configurando como replica...
13       else backupService();
14
15     } catch (PrimaryExistsException ex) {
16       role = BACKUP;
17
18     } catch (PrimaryFailedException ex) {
19       if (guardian.propagate(ex)) throw ex;
20       role = PRIMARY;
21
22     } catch (BackupFailedException ex) {
23       if (guardian.propagate(ex)) throw ex;
24     }
25   }
26 }
```

Figura 4.7: Descrição do método `execute()`.

referente ao servidor, ou seja, `PRIMARY` (linha 6). O código da aplicação para esse contexto consiste em processar as requisições do cliente (abstraída nas linhas 11-12), enviar as atualizações para as réplicas ativas (linhas 14-15) e, por fim, responder à requisição do cliente (abstraída nas linhas 17-18).

Os tratadores especificados nesse contexto aplicam-se às exceções `PrimaryServiceFailureException`, internamente no servidor, que não pode ser resolvido. Neste caso, o tratador comunica os demais participantes do erro ocorrido, invocando a *guardian primitive* `gthrow()` com uma `PrimaryFailedException` como parâmetro. Essa sim, por se tratar de uma exceção global, é informada ao *guardian* (linhas 20-21). As duas outras exceções são lançadas pelo *guardian*, via a *guardian primitive* `checkExceptionStatus()`, já que são exceções globais. Ambos os tratadores atualizam a lista de réplicas do servidor, sendo que o primeiro remove uma réplica falha, enquanto que o segundo informa que novas réplicas estão disponíveis (linhas 23-27).

A Figura 4.9 mostra o método `backupService()`. Uma vez que esse representa o estado de funcionamento das réplicas, tem-se, na linha 4, a ativação do contexto `BACKUP`.

```
1  private void primaryService() {
2    boolean backupAvailable = isThereBackupAvailable();
3
4    while (true) {
5
6      guardian.enableContext(primaryContext);
7
8      try {
9        guardian.checkExceptionStatus();
10
11       //Processa a requisicao do cliente...
12       //. . .
13
14       //Envia atualizacoes para as replicas
15       if (backupAvailable) updateBackups();
16
17       //Envia a resposta para o cliente...
18       //. . .
19
20     } catch (PrimaryServiceFailureException ex) {
21       guardian.gthrow(new PrimaryFailedException(), null);
22
23     } catch (BackupFailedException ex) {
24       backupAvailable = isThereBackupAvailable();
25
26     } catch (BackupJoinedException ex) {
27       backupAvailable = true;
28
29     } finally {
30       guardian.removeContext();
31     }
32   }
33 }
```

Figura 4.8: Descrição do método `primaryService()`.

A atividade específica para tal contexto consiste em aplicar as atualizações recebidas do servidor, pelo método `sendUpdate()`, via o método `applyUpdate()` (linhas 9-10). Note também a existência de um tratador associado a exceção `ApplyUpdateFailureException`. Por se tratar de uma exceção local, que representa a ocorrência de um erro interno na réplica, a mesma é repassada para os demais participantes via a exceção global `BackupFailedException`. Utiliza-se, para isso, a *guardian primitive* `gthrow()`, como mostrado nas linhas 12-13.

Analisando a estrutura da classe `NodeImpl`, nota-se que os métodos `primaryService()` e `backupService()`, que configuram, respectivamente, o nodo para atuar como servidor e réplica, são invocados internamente ao método `execute()`. Já que cada um desses

método está associado a ativação de um contexto, isso cria três possíveis estados para os participantes: `MAIN`, `MAIN.PRIMARY` e `MAIN.BACKUP`. O primeiro ocorre quando o participante executa `execute()`, e ainda não se configurou para atuar como servidor ou réplica. Atuando como servidor implica na invocação de `primaryService()`, e portanto o segundo estado de contexto é atingido. Por outro lado, a invocação de `backupService()` implica na configuração do nodo como réplica, e tem-se assim o terceiro estado de contexto. O aninhamento dos métodos leva ao aninhamento dos contextos.

```
1  private void backupService() {
2    while (true) {
3
4      guardian.enableContext(backupContext);
5
6      try {
7        guardian.checkExceptionStatus();
8
9        //Aplica as atualizacoes recebidas do servidor
10       applyUpdate();
11
12     } catch (ApplyUpdateFailureException ex) {
13       guardian.gthrow(new BackupFailedException(), null);
14
15     } finally {
16       guardian.removeContext();
17     }
18   }
19 }
```

Figura 4.9: Descrição do método `backupServiceNodeImpl`.

Até aqui, descreveu-se como a aplicação foi organizada com a utilização das `guardian primitives`. Porém, para o entendimento do seu funcionamento, é necessário, antes, entender como as regras de recuperação e as árvores de resolução foram definidas.

4.3 Definição das Regras de Recuperação

Quando um participante invoca a *guardian primitive* `gthrow()` para lançar uma exceção global para um conjunto de participantes, o *guardian* utiliza as regras de recuperação definidas para a aplicação, a fim de descobrir qual exceção deve ser lançada em cada participante presente na lista, bem como o contexto de tratamento (*target context*) de tal exceção.

Um trecho do arquivo de regras de recuperação[2], que mostra a primeira regra, pode

[2]Para efeito de simplificação na leitura do código, abstraiu-se a hierarquia de pacotes em que se

ser visto na Figura 4.10. A regra denominada `Rule1` é acionada quando uma exceção do tipo `JoinException` é lançada, como pode ser visto na linha 3. O *guardian* lança tal exceção toda vez que um participante se associa ao grupo. A execução da regra `Rule1` implica no lançamento de uma `BackupJoinedException`, com *target context* igual a `PRIMARY`, para todos os participantes que possuem `PRIMARY` como o contexto corrente, isto é, o participante que desempenha o papel de servidor (linhas 4-7). Por outro lado, uma `PrimaryExistsException`, com *target context* igual a `MAIN`, é lançada no participante que causou o disparo de *JoinException*, isto é, o `SIGNALER`. No entanto, para que essa última exceção seja lançada, deve haver pelo menos dois participantes ativos, sendo o `SIGNALER` um deles (linhas 9-13).

```
1  <recovery_rules>
2    <!-- Um novo participante associa-se no grupo -->
3    <rule name="Rule1" signaled_exception="pack.JoinException">
4      <participant match="*.PRIMARY">
5        <throw_exception class="pack.BackupJoinedException"
6          target_context="PRIMARY" />
7      </participant>
8
9      <participant match="SIGNALER">
10       <throw_exception class="pack.PrimaryExistsException"
11         target_context="MAIN"
12         min_participant_joined="2" />
13     </participant>
14   </rule>
15   . . .
16 </recovery_rules>
```

Figura 4.10: Descrição da regra `Rule1`.

Para o caso de uma `PrimaryFailedException` ser lançada definiu-se a regra `Rule2`, como mostra a linha 4 da Figura 4.11. Tal exceção significa a ocorrência de um erro interno no participante que desempenha o papel de servidor. As linhas 5-8 definem o lançamento de uma `PrimaryFailedException`, com *target context* igual a `INIT_CONTEXT`, para o participante que desempenha o papel de servidor na aplicação (logo possui seu contexto corrente como `PRIMARY`). De forma similar, a mesma exceção, porém com *target context* igual a `MAIN`, é lançada no primeiro participante da lista de participantes associados ao *guardian*, que possui seu contexto corrente igual a `BACKUP`, ou seja, desempenha o papel de uma réplica (linhas 10-15).

A regra `Rule3`, apresentada na Figura 4.12, é utilizada pelo *guardian* toda vez que uma `BackupFailedException` for sinalizada como uma exceção global (linha 4). Para

encontram as classes para o pacote genérico `pack`. Isso não significa, porém, que todas as classes se encontram no mesmo pacote.

```
1  <recovery_rules>
2    . . .
3    <!-- Erro interno no servidor -->
4    <rule name="Rule2" signaled_exception="pack.PrimaryFailedException">
5      <participant match="*.PRIMARY">
6        <throw_exception class="pack.PrimaryFailedException"
7          target_context="INIT_CONTEXT" />
8      </participant>
9
10     <participant match="*.BACKUP">
11       <throw_exception class="pack.PrimaryFailedException"
12         target_context="MAIN">
13         <affected_participants>FIRST</affected_participants>
14       </throw_exception>
15     </participant>
16   </rule>
17   . . .
18 </recovery_rules>
```

Figura 4.11: Descrição da regra `Rule2`.

tal regra, as linhas 5-8 definem o lançamento de uma `BackupFailedException`, com *target context* igual a `PRIMARY`, no participante que atua como servidor (participante com contexto corrente igual a `PRIMARY`). A mesma exceção é lançada, porém com *target context* igual a `INIT_CONTEXT`, para o participante que disparou a exceção, isto é, o `SIGNALER` (linhas 10-13).

```
1  <recovery_rules>
2    . . .
3    <!-- Erro interno em alguma replica -->
4    <rule name="Rule3" signaled_exception="pack.BackupFailedException">
5      <participant match="*.PRIMARY">
6        <throw_exception class="pack.BackupFailedException"
7          target_context="PRIMARY" />
8      </participant>
9
10     <participant match="SIGNALER">
11       <throw_exception class="pack.BackupFailedException"
12         target_context="INIT_CONTEXT" />
13     </participant>
14   </rule>
15   . . .
16 </recovery_rules>
```

Figura 4.12: Descrição da regra `Rule3`.

A última regra (`Rule4`) definida no arquivo de regras de recuperação, aplica-se quando

uma `PrimaryBackupFailedTogetherException` é lançada (linhas 4-5). Como o nome sugere, tal exceção representa uma falha ocorrida, na mesma janela de tempo, no servidor, e em uma ou mais réplicas, e é lançada pelo *guardian* com base no mecanismo de árvore de resolução de exceções. As linhas 7-10, determinam o lançamento de uma `PrimaryFailedException`, com *target context* igual a `INIT_CONTEXT` nos participantes que possuem `PRIMARY` como contexto corrente. Uma `BackupFailedException`, com *target context* igual a `INIT_CONTEXT`, é lançada no participante que lançou a exceção e que se encontra no contexto igual a `BACKUP` (linhas 12-15). Como `PrimaryBackupFailedTogether-Exception` não é uma exceção primitiva, lançada pelos participantes, mas sim oriunda do mecanismo de resolução baseada em árvore de exceções, o uso de `SIGNALER` assume o significado do participante que lançou uma das exceções concorrentes. Por fim, as linhas 17-22 definem o lançamento de uma `PrimaryFailedException`, com *target context* igual a `MAIN`, no primeiro participante da lista de participantes associados ao *guardian*, que possui o seu contexto corrente igual a `BACKUP` e não lançou uma das exceções concorrentes.

```
1  <recovery_rules>
2    . . .
3    <!-- Erro interno no servidor e replica ao mesmo tempo -->
4    <rule  name="Rule4"
5      signaled_exception="pack.PrimaryBackupFailedTogetherException">
6
7      <participant match="*.PRIMARY">
8        <throw_exception class="pack.PrimaryFailedException"
9          target_context="INIT_CONTEXT" />
10     </participant>
11
12     <participant match="*.BACKUP,SIGNALER">
13       <throw_exception class="pack.BackupFailedException"
14         target_context="INIT_CONTEXT" />
15     </participant>
16
17     <participant match="*.BACKUP,!SIGNALER">
18       <throw_exception class="pack.PrimaryFailedException"
19         target_context="MAIN">
20         <affected_participants>FIRST</affected_participants>
21       </throw_exception>
22     </participant>
23   </rule>
24 </recovery_rules>
```

Figura 4.13: Descrição da regra `Rule4`.

4.4 Definição as Árvores de Resolução

Para a aplicação desenvolvida, as exceções globais (apresentadas na Tabela 4.1) pertencem todas ao mesmo nível de prioridade. Sendo assim, apenas uma árvore de resolução de exceções precisa ser definida para resolver o problema de exceções lançadas concorrentemente.

Uma vez que, apenas `PrimaryFailedException` e `BackupFailedException` são lançadas através da *guardian primitive* `gthrow()`, o único caso possível de exceções concorrente é quando se tem ambas exceções ocorrendo na mesma janela de tempo. Para tal caso, aplica-se o mecanismo de resolução baseado na árvore de resolução de exceções apresentada na Figura 4.14.

```
1 <resolution_trees>
2   <resolution_tree exception_level="1">
3     <exception class="pack.PrimaryBackupFailedTogetherException">
4       <exception class="pack.PrimaryFailedException"/>
5       <exception class="pack.BackupFailedException"/>
6     </exception>
7   </resolution_tree>
8 </resolution_trees>
```

Figura 4.14: Árvore de resolução de exceções para aplicação de servidor com N réplicas.

Note que a exceção resolvida consiste no menor ancestral comum entre as exceções filhas, no caso tem-se `PrimaryBackupFailedTogetherException`. O *guardian* usa a árvore de exceção para descobrir a exceção resolvida, que pode então ser encaminhada às regras de recuperação.

É possível ainda que mais de dois participantes falhem ao mesmo. Para a dada aplicação, isso resulta, ou na falha do servidor e de N outras réplicas, ou então na falha de N réplicas. No primeiro caso, tem-se uma `PrimaryFailedException`, e N `BackupFailedException` entre a exceções lançadas, logo a exceção resolvida será `PrimaryBackupFailedTogetherException`. Já no segundo caso, a árvore de resolução não apresenta uma exceção resolvida, já que todas as exceções lançadas são do mesmo tipo (`BackupFailedException`). Nesse tipo de situação, o *guardian* simplesmente processa as exceções, sequencialmente, com o auxílio das regras de recuperação.

4.5 Execução da Aplicação

Suponha que a ordem dos participantes que se associam ao *guardian group* seja do `Participant1` ao `Participant4`. Note que, como todos os participantes são instâncias de uma

classe, a ordem em que eles se associam não interfere no funcionamento final da aplicação. A execução normal da aplicação ocorre da seguinte maneira:

1. O participante `Participant1` associa-se ao *guardian group* com o contexto `MAIN` ativado; o participante bloqueia em `checkExceptionStatus()`; uma `JoinException` é lançada pelo *guardian*; o *guardian* checa as regras de recuperação; nenhuma exceção é lançada via `checkExceptionStatus()`, o método simplesmente retorna; o participante atinge o contexto de servidor `MAIN.PRIMARY`.

2. O participante `Participant2` associa-se ao *guardian group* com o contexto `MAIN` ativado; o participante bloqueia em `checkExceptionStatus()`; uma `JoinException` é lançada pelo *guardian*; o `guardian` executa a regra de recuperação `Rule1`; uma `BackupJoinedException`, com *target context* igual a `PRIMARY`, é entregue ao participante `Participant1`; uma `PrimaryExistsException`, com *target context* igual a `MAIN`, é entregue ao participante `Participant2`;

3. Quando o participante `Participant1` invoca `checkExceptionStatus()`, de dentro do contexto `PRIMARY`, a exceção `BackupJoinedException` é lançada nele; o participante atualiza sua lista de réplicas e inicia o envio de atualizações.

4. Quando o participante `Participant2` invoca `checkExceptionStatus()` de dentro do contexto `MAIN`, a exceção `PrimaryExistsException` é lançada nele e ele se torna uma réplica.

Para cada novo participante que se associa ao *guardian group*, os passos 2 a 4 se repetem e novas réplicas são ativadas. A partir desse ponto, a aplicação segue seu fluxo de execução normal, onde o servidor recebe requisições de um cliente, as processa, envia atualizações para suas réplicas e responde para o cliente. Caso um erro interno ocorra no servidor, a aplicação procede da seguinte forma:

1. O participante `Participant1` falha e uma `PrimaryFailedException` é lançada para o *guardian*.

2. O *guardian* executa a regra de recuperação `Rule2`.

3. O *guardian* entrega uma `PrimaryFailedException`, com *target context* igual `INIT_CONTEXT`, para o participante `Participant1`.

4. O *guardian* entrega uma `PrimaryFailedException`, com *target context* igual a `MAIN`, para a primeira réplica associada ao *guardian* (neste caso, o participante `Participant2`).

5. Quando o participante `Participant1`, com sua lista de contextos igual a `MAIN.PRIMARY`, invoca `checkExceptionStatus()`, a exceção `PrimaryFailedException` é lançada nele, e propagada até o seu contexto `INIT_CONTEXT`, causando a interrupção desse participante.

6. Quando o participante `Participant2`, com sua lista de contexto igual a `MAIN.BACKUP`, invoca `checkExceptionStatus()`, a exceção `PrimaryFailedException` é lançada nele, e propagada até o contexto `MAIN`, fazendo com que o participante se torne o novo servidor.

Note que o mecanismo de tratamento de exceções coordenado, onde o erro manifestado em um participante interfere nos demais, faz com que uma réplica assuma o papel de um servidor falho. Passos semelhantes são executados na ocorrência de um erro em alguma das réplicas:

1. O participante `Participant2` falha, e uma `BackupFailedException` é lançada para o *guardian*.

2. O *guardian* executa a regra de recuperação `Rule3`.

3. O *guardian* entrega uma `BackupFailedException`, com *target context* igual `PRIMARY`, para o participante `Participant1`.

4. O *guardian* entrega uma `BackupFailedException`, com *target context* igual a `INIT_CONTEXT`, para o participante `Participant2`.

5. Quando o participante `Participant1`, com sua lista de contextos igual a `MAIN.PRIMARY`, invoca `checkExceptionStatus()`, a exceção `BackupFailedException` é lançada nele, causando a remoção daquela réplica da sua lista de réplicas.

6. Quando o participante `Participant2`, com sua lista de contexto igual a `MAIN.BACKUP`, invoca `checkExceptionStatus()`, a exceção `BackupFailedException` é lançada nele, e propagada até o contexto `INIT_CONTEXT`, causando a interrupção desse participante.

A ocorrência de uma falha em uma réplica, simplesmente a elimina da lista de réplicas do servidor. Suponha agora que exceções concorrentes são lançadas pelos participantes `Participant1` e `Participant2`. Nesse caso, a aplicação procede da seguinte forma:

1. O participante `Participant1` falha, e uma `PrimaryFailedException` é lançada para o *guardian*.

2. O participante `Participant2` falha, e uma `BackupFailedException` é lançada para o *guardian*.
3. O *guardian* detecta a ocorrência de exceções na mesma janela de tempo, e invoca o mecanismo de árvore de resolução de exceção. Tem-se `PrimaryBackupFailedTogetherException` como exceção resolvida.
4. O *guardian* invoca a regra de recuperação `Rule4`.
5. O *guardian* entrega uma `PrimaryFailedException`, com *target context* igual `INIT_CONTEXT`, para o participante `Participant1`.
6. O *guardian* entrega uma `BackupFailedException`, com *target context* igual a `INIT_CONTEXT`, para o participante `Participant2`.
7. O *guardian* entrega uma `PrimaryFailedException`, com *target context* igual a `MAIN`, para a primeira réplica associada ao *guardian* que não lançou uma das exceções (neste caso, o participante `Participant3`).
8. Quando o participante `Participant1`, com sua lista de contextos igual a `MAIN.PRIMARY`, invoca `checkExceptionStatus()`, a exceção `PrimaryFailedException` é lançada nele, e propagada até o seu contexto `INIT_CONTEXT`, causando a interrupção desse participante.
9. Quando o participante `Participant2`, com sua lista de contexto igual a `MAIN.BACKUP`, invoca `checkExceptionStatus()`, a exceção `BackupFailedException` é lançada nele, e propagada até o contexto `INIT_CONTEXT`, causando a interrupção desse participante.
10. Quando o participante `Participant3`, com sua lista de contexto igual a `MAIN.BACKUP`, invoca `checkExceptionStatus()`, a exceção `PrimaryFailedException` é lançada nele, e propagada até o contexto `MAIN`, fazendo com que o participante se torne o novo servidor.

Ao final tem-se o participante `Participant3` como novo servidor, que passa a responder requisições dos clientes, e a enviar atualizações para a réplica `Participant4`.

Como pode ser observado, toda exceção, com *target context* igual a `INIT_CONTEXT`, lançada para um participante, ocasionou a interrupção do mesmo. Isso por que, para a dada aplicação, nenhum tratamento específico pode ser adotado para resolver o problema ocorrido internamente, seja em um participante que represente um servidor, ou em uma réplica. No entanto, isso não desqualifica o funcionamento do modelo de tolerância a falhas usado, já que o propósito de garantir que um servidor falho seja substituído por uma réplica foi alcançado.

4.6 Considerações Finais

Este capítulo apresentou como utilizar o modelo *Guardian-SCA*, para prover uma solução de réplicas em um servidor. Primeiramente, mostrou-se como estruturar a aplicação, via o arquivo SCDL, utilizando o modelo de componentes SCA. Os participantes foram implementados através de componentes Java e ligados a um componente que abstrai o *guardian group*. Num segundo momento, foi apresentada uma forma de utilização das *guardian primitives*, através da referência mantida para o componente do *guardian*. A estrutura criada consistia em especificar um corpo de um contexto, delimitado pelas primitivas de ativação e desativação, dentro de um determinado escopo. A construção do corpo envolve a checagem de exceções globais pendentes, seguido de um código específico da aplicação apropriado para aquele contexto. O corpo é definido dentro de um bloco `try-catch`, sendo que cada `catch` é responsável por oferecer as ações de recuperação necessárias para cada exceção global suportada pelo contexto. Mostrou-se ainda a definição das regras de recuperação e da árvore de resolução de exceções para a aplicação em questão. Por fim, todos os elementos foram ligados através da descrição do funcionamento da aplicação em diversas situações possíveis, a saber: estado de ausência de erros; falha no servidor; falha em alguma réplica; e falhas concorrentes.

Capítulo 5

Modelo de Programação Orientada a Aspectos para o Guardian-SCA

Este capítulo apresenta uma extensão do modelo *Guardian*-SCA, implementada com a utilização de aspectos, e criada com a finalidade de ser uma alternativa mais simples e intuitiva de utilização do modelo *guardian*, além de separar o código normal do código de tratamento de exceções da aplicação. Primeiramente, uma breve introdução sobre programação orientada a aspectos e AspectJ é apresentada na seção 5.1. A seção 5.2 mostra a utilização de programação orientada a aspectos como mecanismo para separação do código de tratamento de exceções de uma aplicação do código normal, bem como, quando isso é algo benéfico, ou não, tendo em vista a qualidade do código gerado como um todo. O modelo *Guardian-SCA* com aspectos, em si, é apresentado na seção 5.3. As considerações finais sobre o capítulo são apresentadas na seção 5.4.

5.1 Programação Orientada a Aspectos

Programação Orientada a Aspectos[1] é uma metodologia utilizada em conjunto com as metodologias de programação orientada a objeto, e programação procedural, que visa incrementá-las com conceitos e construções que permitem modularizar os interesses transversais[2] da aplicação [17]. O interesses centrais são implementados através da metodologia base. Em programação orientada a objetos, por exemplo, as funcionalidades centrais são implementadas através das classes. Os aspectos, por sua vez, são utilizados para encapsular os interesses transversais. Uma vez que unidades diferentes de implementação são utilizadas para a implementação dos interesses centrais e transversais, garante-se uma relação de independência entre eles. Tal independência se dá no sentido de que as unidades

[1]Do inglês, *Aspect Oriented Programing - AOP*
[2]Do inglês, *crosscutting concerns*

de implementação dos interesses centrais não tem ciência da possível intervenção de um aspecto.

Um processo importante em programação orientada a aspectos é a combinação[3] dos interesses centrais com os transversais, como mostrado na Figura 5.1. Em tal figura, o combinador[4] atua na forma de um compilador, tomando como entrada os interesses centrais e transversais (descritos através das regras de combinação[5]), e produzindo como saída o sistema final. Em um sistema baseado em Java, os interesses centrais são descritos em arquivos *.java*, e o sistema final resulta em uma série de arquivos *.class*.

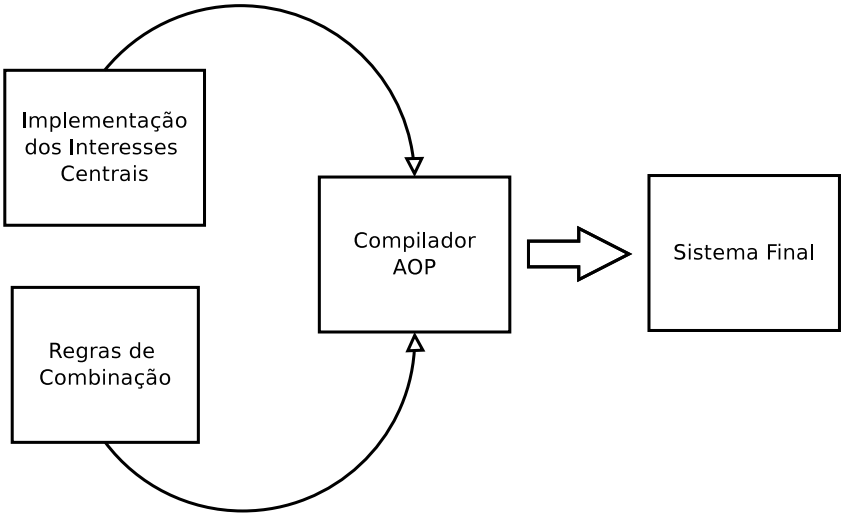

Figura 5.1: Processo de combinação em que o combinador é descrito na forma de um compilador. (Adaptado de [17])

Uma outra abordagem para a combinação poderia ser combinar as regras de combinação com os interesses centrais, e fornecer, então, o código combinado para o compilador da linguagem. Outras abordagens podem ser utilizadas, porém, vale ressaltar que, não importando qual seja escolhida, o combinador não modifica o código original.

Como mencionado, programação orientada a aspectos é uma abordagem, e que, portanto, precisa de uma implementação para ser executada. Uma linguagem de implementação AOP destinada a aplicações Java é a AspectJ [17]. Em tal linguagem, as regras de combinação são descritas através de elementos chaves, tais como pontos de junção[6], pontos de corte[7], e adendos[8].

[3]Do inglês, *weaving*
[4]Do inglês, *weaver*
[5]Do inglês, *weaving rules*
[6]Do ingês, *join points*
[7]Do inglês, *pointcuts*
[8]Do inglês, *advices*

Um ponto de junção é um ponto identificável na execução de um programa, como por exemplo, uma chamada de um método, uma atribuição a um campo membro de um objeto, ou o lançamento de uma exceção. Um ponto de corte, por sua vez, seleciona certos pontos de junção, bem como informações contextuais relativas a tais pontos. A chamada de um método específico, bem como, o objeto alvo em que o método é invocado, e os argumentos utilizados, podem ser capturados através da definição de um ponto de corte, por exemplo.

Os adendos são códigos a serem executados em um ponto de junção selecionado por um ponto de corte. Um adendo pode ser executado antes, depois, ou em torno do ponto de junção, através das primitivas `before`, `after`, e `around`, respectivamente. A primitiva `around` pode modificar o controle de fluxo da aplicação e trocar o código que deveria ser executado no ponto de junção selecionado.

Todos esse elementos, utilizados para estabelecer um novo comportamento durante a execução do programa, são agrupados em unidades de modularização de interesses transversais denominadas aspectos. Estes, contém o código que expressa as regras de combinação utilizadas pelo combinador.

A Figura 5.2 apresenta um exemplo de um aspecto denominado `CheckCreditOpAspect`. A linha 2 traz a definição de um ponto de corte, `creditExecution()`, que intercepta a execução do método `void credit(float)` pertencente a class `Account`. Toda vez que tal ponto de junção é atingido, um adendo é executado depois, imprimindo na tela que o método foi executado (linhas 3-4).

```
1 public aspect CheckCreditOpAspect {
2   pointcut creditExecution() : execution(void Account.credit(float)) {}
3   after() : creditExecution() {
4     System.out.println("Operacao de credito realizada");
5   }
6 }
```

Figura 5.2: Exemplo de um aspecto descrito em AspectJ.

5.2 Tratamento de Exceções e Programação Orientada a Aspectos

Os mecanismos de tratamento de exceções foram desenvolvidos como um meio de melhorar a modularidade e manutenibilidade dos programas que têm necessidade de lidar com situações excepcionais [12]. Como mencionado por Castor [14], idealmente, um mecanismo de tratamento de exceções deve prover meios para que o código de detecção de erro, o de

tratamento de erro, e o código que expressa o comportamento normal do programa, sejam lexicamente separados, de forma que eles possam ser modificados independentemente.

Visualizando o tratamento excepcional como um interesse transversal, deduz-se que uma abordagem AOP pode ser utilizada para atingir a separação do código normal do código excepcional em uma aplicação. No entanto, estudos realizados por Castor [13, 14], utilizando a linguagem AspectJ, mostram que, nem sempre, o uso de aspectos em tal situação acarreta em benefícios para o sistema. Muitas vezes, algumas modificações na estrutura do código podem ser necessárias. Para aplicações em que o código de tratamento de exceções é não-uniforme, fortemente dependente do contexto, ou então muito complexo, a utilização dos aspectos pode não ser possível, ou então: piorar a qualidade geral do sistema; não representar o comportamento original do sistema; ou ainda, exibir um código que apresente "mau cheiro"[9], tais como, campos e métodos criados artificialmente apenas para permitir a intervenção dos aspectos.

5.2.1 Catálogo de Cenários de Tratamento de Exceções

O trabalho desenvolvido por Castor [13, 14] resultou em um catálogo que apresenta cenários de tratamento de exceções, criados a partir de uma classificação proposta, bem como, quando o uso de aspectos para separação do código excepcional do código normal da aplicação é benéfico em cada cenário. A Figura 5.3 apresenta as categorias utilizadas na classificação do código de tratamento de erros.

As Figuras 5.3(a), (b) e (c) referem-se a blocos `try-catch` entrelaçados. Sendo (a) classificado como não entrelaçado, (b) como entrelaçado por sufixo, e (c) entrelaçado por prefixo. Um bloco é dito entrelaçado quando há a execução de algum comando antes, ou depois, do bloco `try-catch`. As figuras (d) e (e) referem-se ao aninhamento de blocos `try-catch`. Em (d) tem-se o que é considerado como um bloco aninhado, enquanto que, em (e), uma situação que não é considerada como aninhamento.

Outra categoria utilizada para classificação dos blocos é a do local do código de lançamento de exceção, apresentada em (f). Denomina-se comando lançador de exceção[10] um código dentro de um bloco `try` que pode lançar exceções que serão tratadas por algum `catch` associado ao mesmo bloco. Diz que um comando lançador de exceção é terminal se ele não é seguido por nenhum outro comando dentro do mesmo bloco `try`. Um código lançador de exceção[11], por sua vez, é o nome dado ao conjunto de todos os comandos lançadores de exceção associados a um mesmo bloco `try`. Considera-se um ETC terminal quando todos os comandos lançadores de exceção que o formam são terminais. Note que em (f) o ETC é classificado como não terminal, pois apesar de `p()` ser um comando

[9] Do inglês *bad smells*
[10] Do inglês, *exception throwing statement*
[11] Do inglês, *Exception-Throwing Code - ETC*

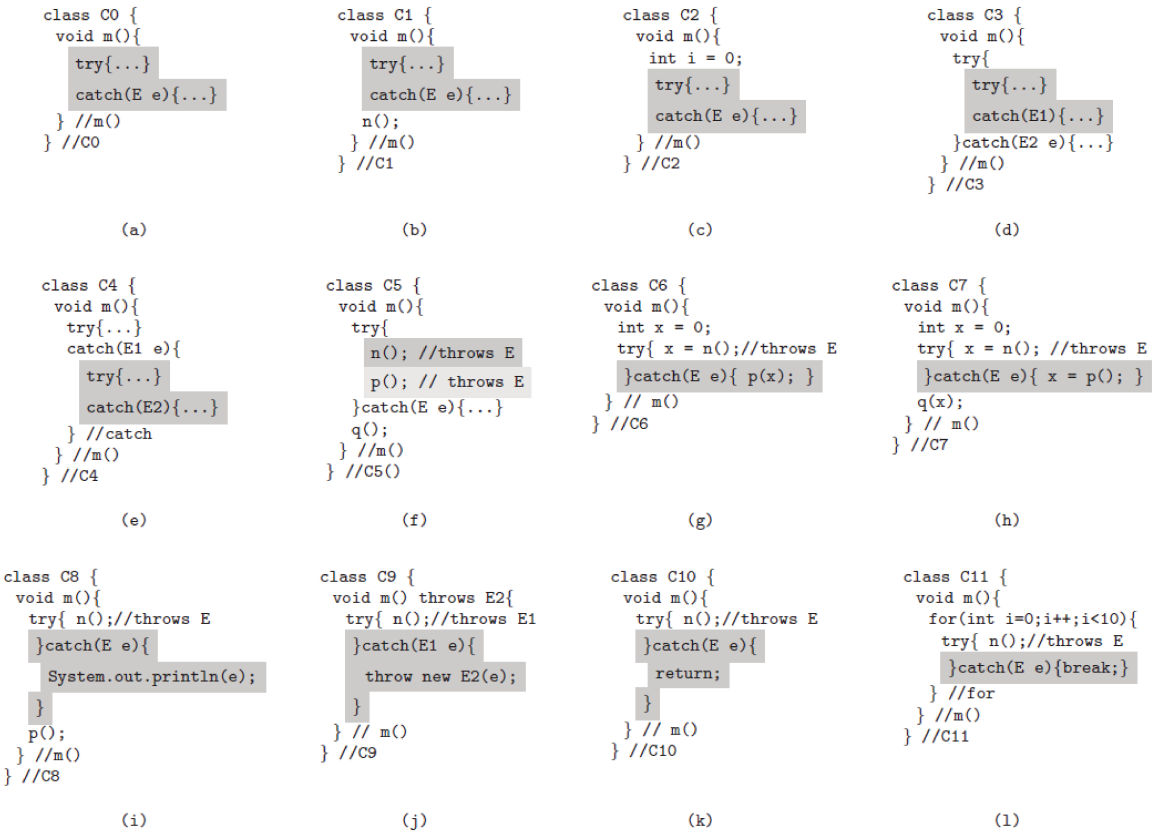

Figura 5.3: Categorias utilizadas para a classificação do código de tratamento de erros. (a) Bloco não entrelaçado. (b) Bloco entrelaçado por sufixo. (c) Bloco entrelaçado por prefixo. (d) Bloco aninhado. (e) Bloco não aninhado. (f) Local do código de lançamento de exceção. (g) Tratador dependente do contexto. (h) Tratador que afeta o contexto. (i) Tratador mascarado. (j) Tratador de propagação. (k) Tratador de retorno. (l) Tratador de iteração de laços. (Adaptado de [14])

lançador de exceção terminal (não há outro comando sendo executado posteriormente), sua existência faz com que `n()` seja considerado um comando lançador de exceção não terminal.

As Figuras 5.3(g) e (h) apresentam blocos categorizados de acordo com a dependência do tratador em variáveis locais. Aquele configura uma situação de um tratador dependente de contexto[12], enquanto que este uma situação de um tratador que afeta o contexto[13]. Por variáveis locais entende-se aquelas definidas no mesmo contexto em que o bloco `try-catch`

[12]Do inglês, *context-dependent handler*
[13]Do inglês, *context-affecting handler*

foi definido. Quando um tratador realiza a leitura de uma variável local, diz-se que ele é dependente do contexto. Porém, se o tratador realiza uma operação de atribuição a alguma variável local, diz-se, então, que ele é um tratador que afeta o contexto. Ambas as situações requerem uma mudança na estrutura do código, o que em geral resulta em um código com "mau cheiro".

Por fim as Figuras 5.3(i), (j), (k) e (l) referem-se ao fluxo de controle após a execução do tratador. Em (i) tem-se um tratador mascarado[14], em que o controle do fluxo passa para o comando que segue, textualmente, ao fim do bloco `try-catch`. Em (j) tem-se um tratador de propagação[15], onde uma nova exceção é lançada de dentro do tratador, alterando o fluxo de execução do programa. O fluxo passa para o tratador mais próximo capaz de capturar tal exceção. Em (k) um tratador de retorno[16] é apresentado. Após o tratador retornar, a execução do sistema volta para o local de onde o método em que o tratador está contido foi invocado. O último tipo de fluxo de controle envolvendo a execução do tratador, mostrado em (l), refere-se ao uso de laços: tratador de iteração de laços[17]. Neste tipo de tratador um laço é declarado envolvendo o bloco `try-catch`, e um comando tal como `break` ou `continue` é executado.

A partir das categorias para classificação dos tratadores, extraiu-se onze cenários, mais comumente encontrados, envolvendo tratamento de exceção. A Tabela 5.1 demonstra os cenários extraídos. Note que para cenários com pontuação entre 0-1, a utilização de aspectos é bem vinda. Pontuação entre 2-3, requer uma análise da situação, podendo ser benéfica ou não. Já para uma pontuação maior que 4, não é recomendado o uso de aspectos. As características marcantes de cada cenário são apresentadas na forma de uma breve descrição na Tabela 5.2.

Considera-se um cenário benéfico à aspectização, se: o código, após a aspectização, não exibir "mau cheiro"; pouca, ou nenhuma alteração na estrutura do código for necessária; se a solução utilizada para extrair os tratadores para aspectos puder ser aplicada para a maioria das instâncias de um cenário; e se a sobrecarga introduzida pelo uso dos aspectos não for muito grande, isto é, para cada bloco `try-catch` extraído para um aspeto, no máximo um novo adendo deve ser criado. A pontuação atribuída a cada cenário foi alocada com base em uma análise dos fatores que dificultam a aspectização do comportamento excepcional, como mostra a Tabela 5.3.

[14]Do inglês, *masking handler*
[15]Do inglês, *propagation handler*
[16]Do inglês, *return handler*
[17]Do inglês, *loop iteration handler*

Cenário	Blocos try-catch entrelaçados		Blocos try-catch aninhados		Local do código de lançamento de exceção		Dependência do tratador em variáveis locais			Fluxo de controle após a execução do tratador				Pontuação	Deve-se extrair?
	sim	não	sim	não	não term.	term.	leitura	escrita	não	masc.	prop.	ret.	laço		
1		•	•	•	•	•			•	•	•	•		0-1	Sim
2	•			•	•	•			•		•	•		0	Sim
3	•		•		•	•			•		•	•		1	Sim
4	•			•		•			•	•				0	Sim
5	•		•			•			•	•				1	Sim
6	•			•	•				•	•				2	Depende
7	•		•		•				•	•				3	Depende
8	•	•		•	•	•	•			•	•	•		2-4	Depende
9	•	•	•		•	•	•			•	•	•		3-5	Não
10	•	•	•	•	•	•		•		•	•	•		3-6	Não
11		•	•	•	•	•	•	•					•	5-9	Não

Tabela 5.1: Cenários de tratamento de exceções de acordo com a classificação proposta. (Adaptado de [14])

Cenário	Descrição
1	Tratador não entrelaçado
2	Tratador não mascarado, entrelaçado
3	Tratador não mascarado, aninhado
4	Tratador entrelaçado, ETC terminal
5	Tratador aninhado, ETC terminal
6	Tratador bloqueador
7	Tratador bloqueador aninhado
8	Tratador dependente de contexto
9	Tratador dependente de contexto, aninhado
10	Tratador que afeta o contexto
11	Tratador de iteração de laços

Tabela 5.2: Descrição dos cenários de tratamento de exceções. (Adaptado de [14])

Pontuação	Descrição	Fatores
1	Dificulta a aspectização de uma forma geral e não limitante	- Blocos try aninhados
2	Em geral, alguma reestruturação do código ou uma sobre-carga de implementação considerável são necessárias. Contudo, o processo envolve técnicas de reestruturação de código bem conhecidas, o que resulta em uma solução genérica que pode, ou não, resultar em um código com "mau cheiro", tais como, criação de campos e método artificiais.	- Combinação de ETC não terminal, tratadores mascarados, e blocos try-catch entrelaçados - Tratadores que fazem leitura de variáveis locais
3	Reestruturação do código é quase sempre necessária e as soluções que se aplicam para cada instância do cenário são geralmente dependente de contexto	- Tratadores que escrevem em variáveis locais - Tratores de iteração de laço

Tabela 5.3: Pontuação para os fatores que dificultam a aspectização do comportamento excepcional de um cenário. (Adaptado de [14])

5.3 Modelo Guardian-SCA com Aspectos

Visando facilitar a utilização do modelo *guardian* implementado no *Apache Tuscany SCA*, uma versão baseada em aspectos foi desenvolvida. Tal versão concentra-se na separação do código excepcional do código normal da aplicação, bem como isenta o programador da necessidade de utilização explícita da maioria das *guardian primitives*.

A partir da análise dos exemplos de utilização do modelo *guardian*, apresentados por Miller em seus trabalhos ([25, 26]), adotou-se a estrutura apresentada na Figura 4.5, como estrutura padrão para construção via aspectos. Tal estrutura assemelha-se a uma estrutura definida por uma conversação. A Figura 5.4 mostra um diagrama de classes com os principais elementos adicionados na extensão `implementation.guardian`. O modelo conta com a definição de duas anotações (`Context` e `CheckPoint`), duas classes abstratas (`HandlerContainer` e `AbstractHandler`), e um aspecto (`AssemblerAspect`), além da classe especializada `DeafaultHandler`.

- **Context**: anotação aplicada a um método, utilizada para especificar um determinado contexto. O nome do contexto é especificado via o atributo `name`, ao passo que a lista das exceções que devem ser tratadas em tal contexto, é especificada via o atributo `exceptions`.

- **Checkpoint**: anotação aplicada a um método, utilizada para especificar se a *guardian primitive* `checkExceptionStatus()` deve ser invocada antes ou depois do corpo principal do método. Para isso, utiliza-se o valor associado ao atributo *default* da anotação: `CheckPointPositions.BEFORE` ou `CheckPointPositions.AFTER`, respectivamente.

- **HandlerContainer**: classe abstrata que especifica uma série de operações para manipulação dos tratadores declarados explicitamente, tais como: `getHandlers()`, `setHandlers()`, `addHandler()`, `removeHandler()`, utilizados para recuperação, substituição, adição, e remoção dos tratadores, respectivamente.

  Note ainda a presença do método abstrato `getGuardianReference()`. A função de tal método é permitir a recuperação da referência para um componente implementado com `implementation.guardian`. Tal método associa o *container* de tratadores de exceções com o modelo `guardian`, e permite a junção dos elementos via interceptação de aspectos.

- **AbstractHandler**: classe abstrata que representa um tratador genérico para uma `GlobalException`. A ação de recuperação, aplicada à exceção associada ao tratador, deve ser descrita na implementação do método abstrato `execute()`. O método `getExceptionClass()` é utilizado para recuperação da classe da exceção associada

ao tratador. Já o método `getLocalVariables()`, fornece acesso às variáveis locais a um método no qual o tratador é invocado.

- **DefaultHandler**: representa uma implementação padrão e genérica da classe abstrata `AbstractHandler`. Dois construtores são definidos, permitindo a inicialização dos campos `exceptionClass` e `localVariables`. A implementação de `execute` resume-se a um método com corpo vazio. Assim, uma especialização de `DefaultHandler` deve sobrescrever tal método com o código adequado para o tratamento da exceção.

- **AssemblerAspect**: aspecto utilizado na junção do código normal separado e do código excepcional, responsável por criar a estrutura que utiliza os elementos do modelo *guardian* através do adendo `aroundContextAnnotatedMethods`. Tal adendo é executado sobre o ponto de corte `contextAnnotatedMethods`, que intercepta toda chamada de método que contenha as anotações `Context` e `CheckPoint`, e que pertença a uma classe filha de `HandlerContainer`.

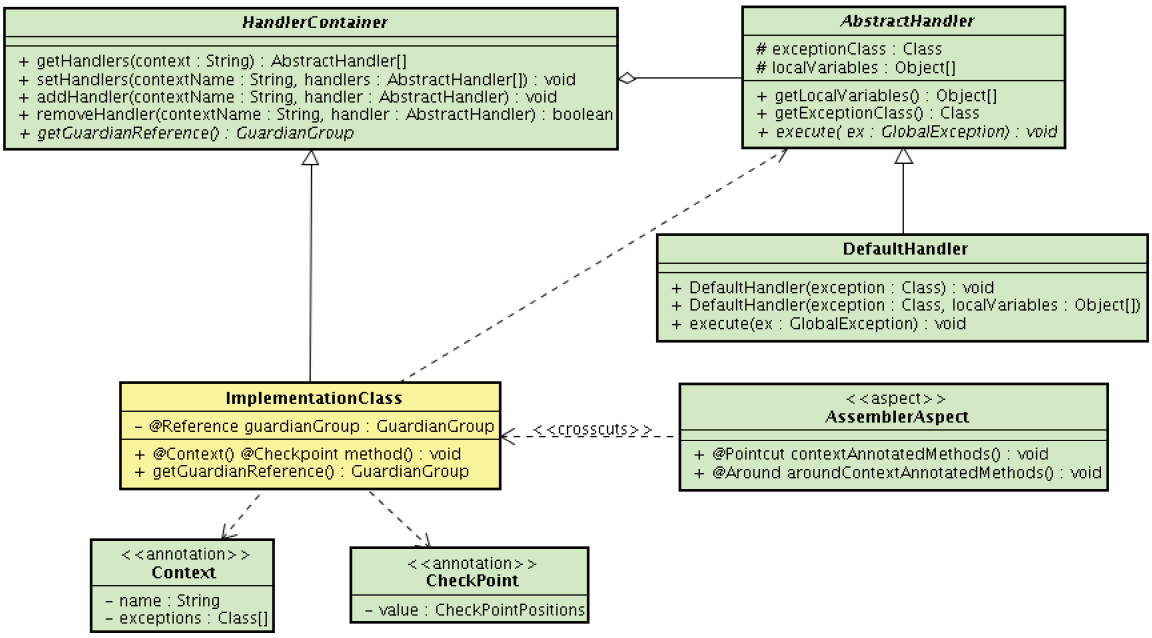

Figura 5.4: Principais elementos adicionados na extensão `implementation.guardian` para suporte a AspectJ e sua interação com uma classe que representa a implementação de um componente.

Em linhas gerais, a utilização do modelo com aspectos se resume em: 1) definir a classe que representa a implementação do componente participante como uma subclasse de `HandlerContainer`; 2) Sobrescrever o método `getGuardianReference()`, de modo

que a referência para o componente implementado com `implementation.guardian` seja obtida como valor de retorno do método; 3) Definir um conjunto de tratadores que sejam subclasses de `AbstractHandler` e armazená-los através dos métodos `addHandler()` ou `setHandlers()`; 4) Anotar cada método que representa um contexto com as anotações `Context` e `CheckPoint`.

Por questões de facilitação na implementação do aspecto, definiu-se que os métodos anotados devem ser públicos e possuir tipo de retorno `void`. O uso de tais anotações restringe o escopo de cada contexto do modelo *guardian* ao escopo do método anotado.

Se, por um lado, a utilização do modelo com aspetos poupa o programador da necessidade de saber como utilizar a maioria das *guardian primitives* (de fato, apenas as primitivas `gthrow()` e `propagate()` poderão ser necessárias na definição das ações de recuperação a serem executadas pelos tratadores), por outro, seu uso limita a utilização do modelo para uma estrutura fixa, o que pode não ser aplicável para todo tipo de aplicação. Além disso, seguindo a classificação proposta por Castor ([13,14]), nem sempre o uso dos aspectos pode resultar em algo benéfico para o código.

Uma vez que, no modelo *guardian*, um bloco `try-catch` é associado para cada contexto, e que, na versão com aspectos, um contexto é restringido ao contexto de um método anotado com as anotações `Context` e `Checkpoint`, tem-se a ausência da situação de blocos `try-catch` entrelaçados e aninhados. Deste modo, os cenários de 2, 3, 4, 5, 6, 7 e 9, da Tabela 5.1, não se aplicam ao modelo desenvolvido. Analisemos então os demais cenários.

O cenário 1 é caracterizado pela ausência de blocos `try-catch` entrelaçados, de operações de leitura ou escrita em variáveis locais, e do uso de laços com o fluxo sendo influenciado por ações executadas de dentro de um tratador. Nesse cenário, as aspectização pode ser feita de forma simples, e portanto, o catálogo recomenda a extração.

Os cenários 8 e 10 são caracterizados pela dependência do tratador em variáveis locais, sendo que, neste, uma operação de escrita é realizada sobre uma variável local, enquanto naquele, realiza-se uma operação de leitura. Ambos os cenários não possuem uma pontuação favorável à utilização de aspectos, uma vez que exigem uma reestruturação do código. Tais mudanças, em geral, culminam em um código com "mau cheiro". Uma solução geral para ambos os casos seria expor a variável local como um campo da classe, permitindo então que ela seja lida e alterada em um tratador separado do código normal da aplicação. Apesar de tal prática não ser recomendada, o modelo implementado com aspectos permite que variáveis locais, expostas como campos, sejam informadas aos tratadores `AbstractHandler`, através dos construtores de **DefaultHandler**.

O último cenário (cenário 11), denominado "tratador de iteração de laços", caracteriza-se pela presença de um laço circundando o bloco `try-catch`, e de comandos `break` ou `continue` no código de tratamento da exceção. O problema é que o laço situa-se na parte do código normal, enquanto um comando que interfere em seu funcionamento situa-se na

parte de tratamento de exceções. Como tais comandos devem estar associados ao laço, para que não haja um erro de compilação, uma reestruturação no código é inevitável, tornando o processo de aspectização prejudicial ao código de uma forma geral.

Diante do exposto, é necessária uma análise de cada situação para determinar se o modelo implementado com aspectos para a separação do código normal do código de tratamento de erros é benéfico ou não. Primeiro, deve-se analisar se a estrutura de utilização do modelo *guardian*, construída via aspecto, aplica-se à situação. Segundo, deve-se analisar se os tratadores se encaixam nos cenários 8, 10, ou 11 do catálogo proposto por Castor, em que o uso de aspectos não é recomendado.

5.4 Considerações Finais

Este capítulo apresentou uma extensão do modelo *Guardian*-SCA, implementada com a utilização de aspectos, que segue uma estrutura lógica baseada em conversações. Tal extensão evita o uso explícito da maioria das *guardian primitives*, simplificando o modelo de programação, além de permitir que o código normal e o código excepcional sejam modificados independentemente, uma vez que estão lexicamente separados. Essa separação confere maior flexibilidade ao modelo, permitindo, por exemplo, que tratadores associados a uma exceção em um certo contexto, sejam modificados em tempo de execução.

Capítulo 6

Conclusões e Trabalhos Futuros

Neste trabalho, foi apresentado o projeto e implementação do modelo *Guardian-SCA*, que consiste na aplicação do modelo *Guardian* em uma Arquitetura de Componentes de Serviços. Dessa forma, a primeira contribuição está em permitir a criação de composições de serviços assíncronos tolerante a falhas em uma arquitetura SCA. Antes deste trabalho, não havia nenhum mecanismo para tratar o comportamento excepcional de composições de serviços assíncronos em uma *framework* SCA.

Com relação à implementação original de Miller [25,26], a versão *Guardian-SCA* possui algumas melhorias. Primeiro, o modelo promove um aumento da alcançabilidade e interoperabilidade, já que diferentes tecnologias SOA podem ser usadas em conjunto em uma arquitetura SCA. A implementação original era puramente baseada em Java. Segundo, uma linguagem baseada em XML foi criada para a definição das regras de recuperação. Na implementação original, as regras de recuperação era definidas pelo desenvolvedor da aplicação através de estruturas condicionais "if-else". Essa abordagem permite que todos os recursos oferecidos pela linguagem de programação em que o modelo foi implementado sejam usados na definição das regras. Entretanto, isto aumenta a possibilidade da ocorrência de erros ocasionados por falhas na lógica de programação, exigindo um maior conhecimento de como tais regras devem ser implementadas. Por outro lado, o uso de uma linguagem XML para descrição das regras de recuperação permite que o foco seja dado à definição do fluxo de exceções globais, evitando a necessidade de saber como tais regras serão implementadas. O código gerado por reflexão computacional previne a existência de possíveis erros na lógica de programação. Além disso, a existência de um arquivo XML para a descrição das regras confere maior dinamismo a aplicação, já que não há necessidade de recompilação de código.

O modelo *Guardian-SCA* define uma forma de construir aplicações baseadas em uma estrutura de conversação. Isto evita o uso explícito da maioria das *guardian primitives*, simplificando o modelo de programação com o modelo *Guardian*. Utiliza-se programação

orientada a aspectos para tal, o que permite também uma separação léxica do comportamento normal e excepcional da aplicação. Aumenta-se assim a flexibilidade do modelo, permitindo, por exemplo, que um tratador associado a uma exceção para ser executado em um contexto específico seja modificado, ou trocado por outro, em tempo de execução.

Por fim, a solução proposta é disponibilizada na forma de código aberto e distribuída sob licensa *Apache2*[1]. Acredita-se assim, que a solução possa ser continuamente aperfeiçoada e utilizada amplamente.

Trabalhos futuros incluem: estender o modelo de árvore de resolução de exceções para diferentes níveis de exceções; implementar algum mecanismo de tolerância a falhas no elemento *guardian group*, evitando assim que este se torne um ponto único de falhas; e, oferecer uma melhor integração do modelo *Guardian* com o *framework Apache Tuscany SCA*, possivelmente estendendo o modelo de definição de componentes e composições XML, incluindo tolerância a falhas neste nível de abstração.

[1] http://www.apache.org/licenses/LICENSE-2.0.html

Referências Bibliográficas

[1] Web services specifications, 2002. http://www.w3.org/2002/ws/.

[2] Alexandre Alves, Assaf Arkin, Sid Askary, Charlton Barreto, Ben Bloch, Francisco Curbera, Mark Ford, Yaron Goland, Alejandro GuÃzar, Neelakantan Kartha, Canyang Kevin Liu, Rania Khalaf, Dieter Konig, Mike Marin, Vinkesh Mehta, Satish Thatte, Danny van der Rijn, Prasad Yendluri, and Alex Yiu. Web services business process execution language version 2.0. Technical report, OASIS, 2007.

[3] Ken Arnold, James Gosling, and David Holmes. *The Java Programming Language*. Pearson, 3rd edition.

[4] Algirdas Avizienis. Toward systematic design of fault-tolerant systems. *Computer*, 30(4):51–58, 1997.

[5] Dan Becker. Service component architecture (SCA) lets you invoke components from different technologies, 2009. Disponível em: http://www.ibm.com/developerworks/opensource/library/os-apache-tuscany-sca/index.html. Último acesso em fevereiro de 2010.

[6] Michael Beisiegel, Vladislav Bezrukhov, Dave Booz, Martin Chapman, Mike Edwards, Anish Karmarkar, Ashok Malhotra, Peter Niblett, Sanjay Patil, and Scott Vorthmann. SCA service component architecture - assembly model specification - extensions for event processing and pub/sub v. 1.0. Technical report, OASIS, 2009. Disponível em: http://www.osoa.org/download/attachments/35/SCA_AssemblyModel_V100.pdf. Último acesso em fevereiro de 2010.

[7] Michael Beisiegel, Dave Booz, Ching-Yun Chao, Mike Edwards, Sabin Ielceanu, Anish Karmarkar, Ashok Malhotra, Eric Newcomer, Sanjay Patil, Michael Rowley, Chris Sharp, and Umit Yalcinalp. Sca policy framework v. 1.0. Technical report, OASIS, 2007. Disponível em: http://www.osoa.org/download/attachments/35/SCA_Policy_Framework_V100.pdf. Último acesso em fevereiro de 2010.

[8] Michael A. Bender and Martín Farach-colton. The LCA problem revisited. In *In Latin American Theoretical Informatics*, pages 88–94, 2000.

[9] A. Capozucca, N. Guelfi, P. Pelliccione, A. Romanovsky, and A. Zorzo. Caa-drip: a framework for implementing coordinated atomic actions. In *ISSRE '06: Proceedings of the 17th International Symposium on Software Reliability Engineering*, pages 385–394, Washington, DC, USA, 2006. IEEE Computer Society.

[10] David Chappell. Introducing sca, 2007. Disponível em: `http://www.davidchappell.com/articles/introducing_sca.pdf`. Último acesso em fevereiro de 2010.

[11] Yuhul Chen and Alexander Romanovsky. Improving the dependability of web services integration. *IT Professional*, 10(3):29–35, 2008.

[12] Flaviu Cristian. A recovery mechanism for modular software. In *ICSE '79: Proceedings of the 4th international conference on Software engineering*, pages 42–50.A, Piscataway, NJ, USA, 1979. IEEE Press.

[13] Fernando Castor Filho, Nélio Cacho, Eduardo Figueiredo, Raquel Maranhão, Alessandro Garcia, and Cecília M. F. Rubira. Exceptions and aspects: the devil is in the details. In *SIGSOFT FSE*, pages 152–162, 2006.

[14] Fernando Castor Filho, Alessandro Garcia, and Cecília M. F. Rubira. Extracting error handling to aspects: A cookbook. In *ICSM*, pages 134–143, 2007.

[15] Anatoliy Gorbenko, Vyacheslav Kharchenko, and Alexander Romanovsky. On composing dependable web services using undependable web components. *Simulation and Process Modelling*, 2(1/2):45–54, 2007.

[16] J. Gray and A. Reute. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann, 1st edition, 1993.

[17] Ramnivas Laddad. *AspectJ in Action: Practical Aspect-Oriented Programming*. Manning Publications, July 2003.

[18] Simon Laws, Mark Combellack, Raymond Feng, Haleh Mahbod, and Simon Nash. *Tuscany in Action*. Manning Publications, 1st edition.

[19] Peter Li, Yuhui Chen, and Alexander Romanovsky. Measuring the dependability of web services for use in e-science experiments. pages 193–205. 2006.

[20] Ben Margolis and Joseph Sharpe. *SOA for the Business Developer - Concepts, BPEL, and SCA*. MC Press, 1st edition, 2007.

[21] Dinesh P. Mehta and Sartaj Sahni. *Handbook Of Data Structures And Applications (Chapman & Hall/CRC Computer and Information Science Series)*. Chapman & Hall/CRC, 2004.

[22] Paulo Merson. SOA - o que e ganha e o que se perde. In *Mundo Java*, number 19, pages 40–44, Rio de Janeiro, RJ, Brazil, 2006.

[23] Robert Miller. *The Guardian Model for Exception Handling in Distributed Systems*. Phd thesis, University of Minnesota, 2002.

[24] Robert Miller and Anand Tripathi. Exception handling in timed asynchronous systems. In Paul Ezhilchelvan and Alexander Romanovsky, editors, *Concurrency in Dependable Systems*, pages 209–227. Kluwer, 2002.

[25] Robert Miller and Anand Tripathi. The guardian model for exception handling in distributed systems. In *SRDS '02: Proceedings of the 21st IEEE Symposium on Reliable Distributed Systems*, page 304, Washington, DC, USA, 2002. IEEE Computer Society.

[26] Robert Miller and Anand Tripathi. The guardian model and primitives for exception handling in distributed systems. *IEEE Trans. Softw. Eng.*, 30(12):1008–1022, 2004.

[27] Michael Papazoglou and Jean Dubray. A survey of web services technology. Technical report, Tibulrg University, 2004. Disponível em: `http://eprints.biblio.unitn.it/archive/00000586/`. Último acesso em maio 2008.

[28] Michael P. Papazoglou. *Web Services: Principles and Technology*. Pearson, Prentice Hall, 2008.

[29] B. Randell. System structure for software fault tolerance. In *Proceedings of the international conference on Reliable software*, pages 437–449, New York, NY, USA, 1975. ACM.

[30] B. Randell, P. Lee, and P. C. Treleaven. Reliability issues in computing system design. *ACM Comput. Surv.*, 10(2):123–165, 1978.

[31] Ricardo Silva, Paulo Guerra, and Cecilia Rubira. Component integration using composition contracts with exception handling. In *3rd Workshop on Exception Handling in Object-Oriented Systems*, pages 1–20, Darmstadt, 2003.

[32] Frederic Souchon, Christophe Dony, Christelle Urtado, and Sylvain Vaultier. A proposition for exception handling in multi-agent systems. In *2nd International Workshop on Software Engineering for Large-Scale Multi-Agent Systems (SELMAS'03)*, Portland, OR, USA, 2003.

[33] Frederic Souchon, Christophe Dony, Christelle Urtado, and Sylvain Vauttier. Improving exception handling in multi-agent systems. In *Software Engineering for Multi-Agent Systems II*, pages 333–337. Springer Berlin-Heidelberg, 2004.

[34] Bjarne Stroustrup. *The C++ Programming Language*. Addison-Wesley Professional, third edition, February 2000.

[35] Clemens Szyperski. *Component Software: Beyond Object-Oriented Programming (2nd Edition)*. Addison-Wesley Professional, 2 edition, 2002.

[36] Ferda Tartanoglu, Valerie Issarny, Alexander Romanovsky, and Nicole Levy. Coordinated forward error recovery for composite web services. In *In 22nd Symposium on Reliable Distributed Systems (SRDS*, pages 167–176, 2003.

[37] Erl Thomas. *SOA Principles of Service Design*. Prentice Hall International, 1st edition, 2007.

[38] Jie Xu, Brian Randell, Alexander Romanovsky, Cecilia M F Rubira, Robert J Stroud, and Zhixue Wu. Fault tolerance in concurrent object-oriented software through coordinated error recovery. In *FTCS '95: Proceedings of the Twenty-Fifth International Symposium on Fault-Tolerant Computing*, page 499, Washington, DC, USA, 1995. IEEE Computer Society.

[39] Jie Xu, Alexander Romanovsky, and Brian Randell. Concurrent exception handling and resolution in distributed object systems. *IEEE Trans. Parallel Distrib. Syst.*, 11(10):1019–1032, 2000.

[40] Diego Zuquim Guimaraes Garcia and Maria Beatriz Felgar de Toledo. A fault tolerant web service architecture. In *LA-WEB '07: Proceedings of the 2007 Latin American Web Conference*, pages 42–49, Washington, DC, USA, 2007. IEEE Computer Society.