Ações Atômicas Coordenadas na Plataforma Java EE

Este exemplar corresponde à redação final da Dissertação devidamente corrigida e defendida por Peterson Peixoto dos Santos e aprovada pela Banca Examinadora.

Campinas, 11 de abril de 2010.

Profa. Dra. Cecília Mary Fischer Rubira
(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

Santos, Peterson Peixoto dos

Sa59a Ações atômicas coordenadas na plataforma Java EE/ Peterson Peixoto dos Santos-- Campinas, [S.P. : s.n.], 2010.

Orientador : Cecília Mary Fischer Rubira

Dissertação (mestrado) - Universidade Estadual de Campinas, Instituto de Computação.

1.Tolerância a falhas. 2.Tratamento de exceção. 3. Ações atomicas coordenadas. 4. Confiabilidade de software. 5. Desenvolvimento baseado em componentes. 6. Programação orientada a aspectos. I. Rubira, Cecília Mary Fischer. II. Universidade Estadual de Campinas. Instituto de Computação. III. Título.

Título em inglês: Coordinated atomic actions in Java EE platform

Palavras-chaves em inglês (Keywords): 1.Fault tolerance. 2. Exception handling.
3. Coordinated atomic actions. 4. Software dependability. 5. Component-based development.
6. Aspect-oriented programming.

Área de concentração: Engenharia de Software

Titulação: Mestre em Ciência da Computação

Banca examinadora: Profa. Dra. Cecília Mary Fischer Rubira (IC-UNICAMP)
Prof. Dr. Delano Medeiros Beder (EACH-USP)
Profa. Dra. Islene Calciolari Garcia (IC-UNICAMP)

Data da defesa: 23/02/2010

Programa de Pós-Graduação: Mestrado em Ciência da Computação

TERMO DE APROVAÇÃO

Dissertação Defendida e Aprovada em 23 de fevereiro de 2010, pela Banca examinadora composta pelos Professores Doutores:

Prof. Dr. Delano Medeiros Beder
EACH / USP

Profª. Drª. Islene Calciolari Garcia
IC / UNICAMP

Profª. Drª. Cecília Mary Fischer Rubira
IC / UNICAMP

Instituto de Computação
Universidade Estadual de Campinas

Ações Atômicas Coordenadas na Plataforma Java EE

Peterson Peixoto dos Santos

Abril de 2010

Banca Examinadora:

- Profa. Dra. Cecília Mary Fischer Rubira (Orientadora)
- Prof. Dr. Delano Medeiros Beder (EACH-USP)
- Profa. Dra. Islene Calciolari Garcia(IC-UNICAMP)
- Profa. Dra. Eliane Martins (IC-UNICAMP) - suplente
- Prof. Dr. Ivan Luiz Marques Ricarte (FEEC-UNICAMP) - suplente

Resumo

À medida que os sistemas de software evoluem, precisam garantir requisitos funcionais e de qualidade cada vez mais complexos e com maior rigor de qualidade. Nos últimos anos, várias abordagens e ferramentas têm sido propostas para guiar o processo de desenvolvimento de software visando atingir altos níveis de qualidade. O Desenvolvimento Baseado em Componentes (DBC) é uma das técnicas mais bem aceitas tanto na indústria quanto no meio acadêmico e se propõe a compor sistemas de software a partir de componentes reutilizáveis já prontos e, se possível, de uma relativamente pequena quantidade de linhas de código específicas para a aplicação. Existem diversas plataformas para DBC, das quais Java Enterprise Edition (Java EE) é uma das mais populares.

Por outro lado, tolerância a falhas é uma das abordagens mais empregadas para construir sistemas que consigam prover seus serviços especificados mesmo na presença de diferentes tipos de falhas de forma a atingir os níveis desejados de confiabilidade. O conceito de Ação Atômica Coordenada (CA Action) foi proposto para prover tolerância a falhas em sistemas concorrentes orientados a objetos, integrando os conceitos complementares de conversação (concorrência cooperativa) e transação atômica (concorrência competitiva) e estabelecendo uma semântica para tratamento de exceções concorrentes (exceções lançadas simultaneamente por *threads* concorrentes) além de dar suporte ao uso conjunto de recuperação de erro por avanço e por retrocesso.

A proposta deste trabalho é acrescentar à plataforma de desenvolvimento baseado em componentes *Java Enterprise Edition* (Java EE) alguns mecanismos de tolerância a falhas propostos pelo conceito de CA Action. A implementação da solução proposta foi baseada em Java, programação orientada a aspectos e no conceito de comunicação assíncrona implementada pelos componentes *message-driven beans* da plataforma *Java Enterprise Edition*. A solução foi avaliada através da construção de 2 estudos de caso: (i) uma aplicação JBoss baseada em *message-driven beans* e (ii) um sistema real de faturamento de energia elétrica. Desta forma, procuramos demonstrar a factibilidade de proporcionar mecanismos simples para adaptações permitindo que aplicações desta plataforma possam usufruir de mais benefícios de tolerância a falhas sem grandes modificações em seu código fonte já previamente implementado e implantado.

Abstract

As software systems evolve, they should provide stronger functional and quality requirements. In the last years, many different approaches and tools have been proposed to guide software development process aiming to achieve higher quality levels. Component-Based Development (CBD) is one of the most accepted techniques in the academy as well as in the industry and proposes to build software systems from pre-existing reusable components and, if possible, a relative low quantity of application specific glue code. There are many CBD platforms and Java Enterprise Edition (Java EE) is one of the most popular.

Fault tolerance is one of the most adopted means to build up systems that are capable of providing their intended service, even if only partially, when faults occur, so as the desired reliability levels be achieved. The Coordinated Atomic Action (CA Action) concept was proposed to provide fault tolerance in concurrent object-oriented software systems and to integrate two complementary concepts, conversations (cooperative concurrency) and transactions (competitive concurrency). It establishes a semantic for concurrent exception handling and also supports the combined use of forward and backward error recovery.

This work proposes to extend the component-based development platform Java Enterprise Edition (Java EE) with some of the fault tolerance means proposed by CA Action's concept by incorporating a concurrent exception handling mechanism to the platform. The proposed solution implementation was based on Java, aspect oriented programming and on the asynchronous communication concept implemented by Java EE message-driven bean components. The solution was assessed by two case studies: (i) a JBoss application based on message-driven beans and (ii) a real billing system for electric power companies by which we try to demonstrate the feasibility of providing simple means for adapting Java Enterprise Edition applications in a way that they could appropriate more fault tolerance benefits without big changes in their previously implemented and deployed source code.

Agradecimentos

Chegar até aqui foi uma longa (longa mesmo) caminhada e a falha mente humana não vai permitir me lembrar todos os nomes e situações que com certeza mereciam ser mencionados, no entanto, não posso deixar de agradecer a todos que contribuíram para a conclusão deste trabalho. Eu gostaria de agradecer...

Primeiramente a Deus, o Criador e Mantenedor do universo, a quem devo tudo o que tenho e o que sou e a quem aguardo ansiosamente. *"Eis que venho sem demora" - Ap 22:12.*

À minha esposa Iara que me acompanhou durante todo o tempo, abrindo mão de muita coisa, me apoiando e dando forças sempre, que me ensinou muito em vários aspectos da vida, especialmente sobre relacionamentos, assunto em que eu era totalmente leigo. Aquela que ficou ao meu lado quando cabeludo ou quando careca, quando estive feliz ou triste, enquanto fui gordo e enquanto fui magro, antes e depois que eu conheci a Deus e, mesmo assim, nesses 15 anos desde que nos conhecemos, tem me amado como sou, mas sempre me ajudando a melhorar. "Querida, amo você!"

Aos meus filhos Caio e Camila, dois diamantes maravilhosos que Deus me deu, a quem amo tanto, e por isso procuro ensinar e lapidar, mas que me ensinam e me lapidam muito mais do que podem hoje compreender. Que mudaram minha maneira de viver, minha maneira de pensar e minha maneira de dormir :-). Que Deus os ilumine sempre!

Aos meus pais, Jorge e Dolores, pelo apoio de sempre e pela confiança que sempre depositaram em mim, a quem devo tanto e nunca vou conseguir agradecer por tudo que fizeram e fazem por mim, mas que Deus tem me dado a oportunidade de "redescobri-los". Obrigado por saber que posso contar sempre com vocês!

Às minhas irmãs, Ludmila e Iochabel, por todas as brigas e disputas na infância e adolescência e por todo o apoio e carinho que atualmente têm dado a mim, a Iara, ao Caio e a Camila.

À minha família como um todo, os de perto e os de longe, tios, primos, sobrinhos, avós, cunhados, etc. Aos meus sogros S. Wilson e D. Antonia que durante um bom tempo (e de certa forma até hoje) foram meus segundos pais, pois os visitava com mais frequência do que aos meus pais "biológicos"! Muito obrigado! A todos os parentes que

ajudaram a recuperar o que restou da maior enchente de São Luiz do Paraitinga e aos que podaram as árvores e nos ajudaram a reorganizar nossa "nova" casa em Engenheiro Coelho. Aos parentes queridos que algum dia me acolheram em suas casas. Aos que sempre têm uma palavra de carinho, incentivo e elogio, e também pelos muitos e bons momentos de descontração. Aos que ainda me amam mesmo depois de ficar anos sem vê-los. Obrigado, obrigado, obrigado!

À minha orientadora Cecília Rubira pela sinceridade, apoio, orientação e discussões. Por acreditar neste trabalho e pelas muitas cobranças que fizeram melhorar tanto este trabalho quanto a mim mesmo.

Ao especial Sindo Vasquez que desde a época da graduação sempre deu um jeitinho para ajudar a mim e a tantos outros nos mais diversos assuntos, fossem do CPqD, da Unicamp ou mesmo das situações diárias da vida! (Vou ter que comprar uma concessionária para pagar todos os carros :-)).

Ao Léo Gonçalves pela amizade, pelas muitas discussões sobre família, sobre Deus e sobre o mestrado enquanto "viajávamos" no Branca de Neve; e por tudo o que fez na concepção e evolução do JACAAF.

Ao CPqD, representado nas pessoas da Marilza Higa e da Marcia Harue que me deram condições de cumprir a carga horária das disciplinas obrigatórias, terminar a dissertação e pela disponibilização dos recursos (físicos e humanos) para a preparação e execução dos estudos de caso.

Aos amigos do CPqD que deram contribuições essenciais para que eu conseguisse definir o JACAAF, compilar (que difícil!!!) e executar os estudos de caso: o grande amigo e padrinho Boss, Juliana Leme (e a melhor equipe de assemblers da DSB de 2010 do universo!!!), Domingos, Pedro, Adilson, Ambrosi, Carlão, Paulo Paro, Jonga, Nelson, Alarcon e tantos outros que não vou conseguir me lembrar agora...alguns talvez nem saibam o quanto ajudaram :-). À equipe do GRT por entender minhas ausências.

Ao pequeno grupo Vaso Novo pelo carinho, grande apoio durante este último ano e pelas muitas orações.

A todos os "caroneiros" do UNASP, em especial à família Mubarac, com os quais pude dividir muitas vezes as "viagens" de ida e de volta para Campinas e, muito mais do que as viagens, pudemos dividir também nossas provações, sonhos, problemas e alegrias! Obrigado por tornarem esse trajeto diário menos cansativo e mais feliz!

Aos amigos da Unicamp, os da graduação, os que conheci no laboratório de sistemas distribuídos (LSD) e os que conheci durante as disciplinas da pós. Todos que de alguma forma contribuíram para que pudesse chegar até aqui, em especial: Fernando Castor, Patrick Brito, Eduardo Machado, Leonardo Tizzei, Carlos Eduardo, Douglas, Marcelo, Ivo, Amanda, Leonel, Sheila, Rachel, Silvia, Lara, Guarido *and so on...*:-)

Aos professores do IC, em especial a Eliane Martins, Catto, Anido, Tomasz, Ariadne,

Heloísa, Claudia Bauzer e Cecília Rubira.

Aos professores e componentes da banca Delano e Islene, por aceitarem o convite mesmo dentro do prazo tão apertado.

A todos os funcionários do IC, em especial: Adê e Daniel pelas muitas dúvidas e problemas resolvidos.

Aos amigos do peito, aqueles que não preciso mencionar, pois estarão sempre presentes em meu coração e sabem muito bem disso!

Obrigado a todos os que aqui não estão nomeados mas que de perto ou de longe, por pouco ou por muito tempo, direta ou indiretamente tiveram sua participação na elaboração deste trabalho. Muito obrigado, mesmo!

"Tudo quanto te vier à mão para fazer, faze-o conforme as tuas forças, porque na sepultura, para onde tu vais, não há obra nem projeto, nem conhecimento, nem sabedoria alguma." - Eclesiastes 9:10

"Eis que vem com as nuvens, e todo o olho o verá." - Apocalipse 1:7

"E Deus limpará de seus olhos toda a lágrima; e não haverá mais morte, nem pranto, nem clamor, nem dor; porque já as primeiras coisas são passadas." - Apocalipse 21:4

Sumário

Lista de Figuras

Capítulo 1

Introdução

Atualmente os sistemas de software precisam garantir requisitos funcionais e não-funcionais cada vez mais complexos e com maior rigor de qualidade [16]. Além disso, muitos desses sistemas devem possuir a capacidade de se comunicar através de redes locais e da internet e serem implantados em localizações geograficamente dispersas e com partes fracamente acopladas. Nos últimos anos, várias abordagens e ferramentas têm sido propostas para guiar o processo de desenvolvimento de software visando atingir altos níveis de qualidade. O Desenvolvimento Baseado em Componentes (DBC) é uma delas. O DBC ou engenharia de software baseada em componentes (ESBC) é um ramo da disciplina de engenharia de software com ênfase na decomposição dos sistemas em componentes funcionais ou lógicos com interfaces para comunicação bem definidas [68]. É uma das técnicas mais bem aceitas tanto na indústria quanto no meio acadêmico e visa prover facilidades para o desenvolvimento de sistemas de grande complexidade, além da reutilização de software. Existem diversas plataformas para DBC, das quais *Java Enterprise Edition* (Java EE) [36] é uma das mais populares.

Em sistemas distribuídos de grande escala, falhas são frequentes por diversas razões. O simples fato de possuírem maior complexidade resulta potencialmente em um maior número de falhas de projeto. Outro aspecto importante é o fato de ser distribuído, que por si já traz algumas implicações: (i) falhas parciais do sistema, isto é, podem ocorrer falhas em alguns componentes e ainda assim o sistema deve continuar a prover seus serviços, mesmo que de forma parcial [1]; (ii) a simples presença de concorrência, pois sistemas concorrentes são notoriamente difíceis de se construir; (iii) heterogeneidade dos ambientes de distribuição; (iv) dependência da estabilidade dos canais de comunicação.

Para que os níveis desejados de confiabilidade sejam atingidos, mecanismos para detectar e tratar erros devem ser inseridos nas fases do desenvolvimento do software anteriores à implementação [55]. Tolerância a falhas[1] é uma abordagem muito empregada para

[1]Do inglês: fault tolerance.

construir sistemas que consigam prover seus serviços especificados mesmo na presença de diferentes tipos de falhas. A tolerância a falhas é constituída de quatro fases que proveem meios gerais para que se possam implementar maneiras de prevenir que falhas levem o sistema a apresentar um defeito. Estas quatro fases são: (i) detecção de erros, (ii) confinamento e avaliação de danos, (iii) recuperação de erros e (iv) tratamento de falhas e continuação do serviço.

Em sistemas distribuídos de grande escala e baseados em componentes, tolerância a falhas torna-se fundamental, especialmente levando-se em conta os aspectos de distribuição, concorrência competitiva e cooperativa. Em sistemas competitivos, os componentes participantes compartilham recursos em comum sem nenhum tipo de coordenação pré-estabelecida, de tal forma que o processamento ocorre sem que nenhuma parte conheça a outra e sem coordenação explícita entre elas. Sistemas cooperativos, por sua vez, são projetados como um conjunto de processos cooperando explicitamente para atingir um objetivo comum.

Para proporcionar um maior grau de confiabilidade a estes tipos de sistemas e implementar conceitos de tolerância a falhas, um mecanismo bem conhecido é o de tratamento de exceções. O tratamento de exceções[2] [29] é um mecanismo bastante utilizado para incorporar tolerância a falhas em sistemas de software e tem por objetivo estruturar a atividade excepcional de um componente de forma a facilitar a detecção, sinalização e tratamento de erros.

Outro conceito importante dentro do universo de tolerância a falhas é o de **ação atômica**[3]. Adotando a definição de atividade como sendo uma sequência de transições do estado externo de um componente, podemos descrever o conceito de ação atômica da seguinte forma: "A atividade de um grupo de componentes constitui uma ação atômica se não existe nenhuma interação entre o grupo e o resto do sistema durante aquela atividade" [1]. Evidentemente, não pode existir nenhuma interação entre o componente e o resto do sistema durante uma transição de estado. Essa ausência de interações durante a transição de estado pode ser tomada como critério para atomicidade. Portanto, para os demais componentes do sistema, todas as atividades dentro de uma ação atômica apresentam-se como uma ação indivisível.

Outro importante conceito desenvolvido para prover tolerância a falhas em sistemas distribuídos e concorrentes é o de **Ação Atômica Coordenada**[4] (CA Action). O conceito de CA Action foi proposto por Xu et al. [59] para prover tolerância a falhas em sistemas concorrentes orientados a objetos, integrando os conceitos complementares de conversação [53] (concorrência cooperativa) e transação atômica[28] (concorrência com-

[2]Do inglês: exception handling.
[3]Do inglês: atomic action.
[4]Do inglês: coordinated atomic action.

petitiva) e estabelecendo uma semântica para tratamento de exceções concorrentes, além de dar suporte ao uso conjunto de recuperação de erro por avanço e por retrocesso.

Em uma **conversação**[5] [53], cada processo participante deve salvar seu estado antes de iniciar sua participação. Enquanto participa de uma conversação, os processos podem se comunicar somente com os outros processos participantes desta conversação. Esta restrição na comunicação limita a propagação de erros e elimina a possibilidade do efeito dominó [53]. Ao fim da conversação cada participante realiza um teste de aceitação para avaliar se o processamento ocorreu conforme esperado. Se algum processo não passa em seu teste de aceitação, então uma exceção é levantada e cada processo deve tentar se recuperar do erro. O estado original de cada processo, salvo antes de entrar na conversação, é restaurado e cada processo executa um algoritmo alternativo. Diferentes processos podem entrar em diferentes momentos, sendo necessário apenas o estabelecimento de um ponto de recuperação para cada processo, porém, devem sair obrigatoriamente ao mesmo tempo, implicando que o teste de aceitação em cada processo foi satisfeito.

Para integrar a computação competitiva são utilizadas **transações atômicas**: "uma única ação lógica, composta por uma sequência de operações básicas em dados ou objetos compartilhados de maneira protegida, isto é, provendo as propriedades ACID[6] para todas as operações desta transação" [28]. As quatro propriedades ACID são:

- **Atomicidade** - refere-se a garantia de que ou todas as operações de uma transação são efetivadas ou nenhuma delas o é;
- **Consistência** - significa que o estado final do sistema (um banco de dados, por exemplo), após a transação, esteja íntegro, assim como o era antes da transação ser iniciada, isto é, suas regras de negócio não foram desrespeitadas por alguma operação mal sucedida;
- **Isolamento** - a habilidade de fazer com que as operações de uma determinada transação estejam isoladas das demais transações, isto é, nenhuma alteração realizada por uma transação poderá ser percebida por outras transações até que aquela termine;
- **Durabilidade** - garantir que após as operações da transação terem sido executadas com sucesso, elas serão persistidas de forma que mesmo que o sistema apresente um defeito após a confirmação da transação, suas alterações poderão ser consultadas e percebidas futuramente.

[5]Do inglês: conversation.
[6]Do inglês: Atomicity, Consistency, Isolation and Durability.

1.1 Problema

Um problema recorrente no desenvolvimento de aplicações de larga escala baseadas em componentes é tornar essas aplicações capazes de funcionar mesmo em caso de manifestação de falhas. A plataforma *Java Enterprise Edition* utiliza o mecanismo de tratamento de exceções de Java [38], isto é, tratamento sequencial de exceções e um modelo de transações sem o conceito de transações aninhadas[7], para se recuperar de erros. Esses mecanismos ajudam o desenvolvedor a focar seus esforços nas regras de negócio, mas deixam em aberto algumas questões relativas à tolerância a falhas em sistemas concorrentes tais como: (i) de que maneira lidar com erros correlacionados ? (ii) como garantir que um erro não se propague para as partes consistentes do sistema ? (iii) o que fazer no caso de vários componentes sinalizarem exceções simultaneamente, isto é, concorrentemente ? A especificação *Java Enterprise Edition* não fornece respostas satisfatórias a essas perguntas.

A recuperação de erros em sistemas concorrentes e distribuídos é sabidamente complicada devido a uma série de fatores, tais como o alto custo para se obter um consenso, ausência de uma visão global do estado do sistema, múltiplos erros concorrentes e dificuldades para garantir isolamento do erro. Esses tipos de sistemas requerem mecanismos especiais de recuperação de erros que atendam às suas principais características. Como não é possível desenvolver um mecanismo de recuperação de erros genérico que seja aplicável a todos os tipos de sistemas concorrentes e distribuídos, foram desenvolvidas técnicas para apoiar a recuperação de erros em sistemas deste tipo. Transações distribuídas [28] e ações atômicas [12] são dois exemplos de técnicas bem conhecidas para estruturação de sistemas distribuídos tolerantes a falhas competitivos e cooperativos respectivamente.

1.2 Trabalhos relacionados

1.2.1 DRIP e CAA-DRIP

Zorzo e Stroud propuseram o *framework* DRIP[8] [71] que estende o conceito de Multiparty Interaction [21, 37] e implementa o conceito de Dependable Multiparty Interaction (DMI). Uma DMI é uma computação competitiva e cooperativa em sistemas distribuídos orientados a objetos, com tratamento de exceções concorrentes e sincronização na finalização da computação, utilizando a linguagem Java. Uma vez que as semânticas de uma DMI a respeito de tratamento de exceções são menos restritivas do que as utilizadas em CA Actions, os autores utilizaram o DRIP para implementar CA Actions.

[7] Do inglês: nested transactions.

[8] Dependable Remote Interacting Processes.

Capozucca et al. propõem uma evolução do *framework* DRIP chamada CAA-DRIP. Com este novo *framework*, a proposta é permitir que programadores implementem somente as semânticas de uma DMI que se aplicam CA Action com a mesma terminologia e conceitos nos níveis de projeto e implementação. Uma melhoria no modelo do *framework* anterior também permitiu que um menor número de instâncias de objetos, comparado ao DRIP, fossem necessárias para a implementação de uma mesma ação atômica [16]. Por ser uma implementação mais específica para CA Actions do que o DRIP, seus conceitos estão mais bem acoplados neste *framework*, evitando que o programador tenha que se preocupar com semânticas específicas e que não eram garantidas pelo *framework* original.

Romanovsky, Periorellis e Zorzo descrevem em seu trabalho o uso de CA Actions para integrar aplicações web com tolerância a falhas [57], utilizando o *framework* DRIP para implementar a maior parte das ações atômicas da aplicação. Sua principal proposta é demonstrar a validade do uso de técnicas de estruturação para prover tolerância a falhas, no caso CA Actions, para integrar aplicações web complexas. Não propõem, no entanto, nenhuma técnica ou *framework* novo. A abordagem de nosso trabalho diverge desta, principalmente pelo fato de trabalharmos com processos dentro de um servidor de aplicações da plataforma *Java Enterprise Edition*, mas também pelo uso do *framework* DRIP que não faz parte de nosso trabalho.

Os *frameworks* DRIP e CAA-DRIP diferem da abordagem proposta em nosso trabalho principalmente pelo fato de que nos concentramos especificamente em sistemas baseados em componentes dentro da plataforma *Java Enterprise Edition* o que não é o caso dos *frameworks* acima descritos. O uso de programação orientada a aspectos utilizado em nosso trabalho também é um grande diferencial, pois não foi utilizado nos trabalhos citados nesta seção.

1.2.2 PKUAS

Feng et al. [24] descrevem uma abordagem de tratamento de exceções utilizando recursos do *middleware* PKUAS (PeKing University Application Server), um servidor de aplicações desenvolvido pela Universidade de Pequim e que implementa a especificação *Java Enterprise Edition*. Resumidamente, propõem uma metodologia para identificar e descrever em XML as exceções de determinados componentes, e, a partir de um modelo de tratamento de exceções, delegar ao servidor de aplicações a tarefa de identificar e "pré-tratar" as exceções, buscando evitar que estas se propaguem até a aplicação. Não estão explícitos, no entanto, os mecanismos utilizados pelo PKUAS para interceptar e tratar essas exceções. Além disso, o trabalho assume a premissa de que existe um modelo de arquitetura da aplicação que contém a descrição dos componentes (interfaces providas e requeridas), a partir do qual o modelo de tratamento de exceções e todo o trabalho se

desenvolvem. Nosso trabalho utilizará o JBoss, um servidor de aplicações da plataforma *Java Enterprise Edition*, devido ao seu extenso uso dentro e fora da indústria.

1.2.3 Vecellio e Sanders

O trabalho de Vecellio e Sanders [70] mostra ser possível alterar o servidor de aplicações JBoss [34] para que ele monitore e intercepte as chamadas de métodos das aplicações nele implantadas sem alterar o seu código, de forma que, pela verificação de pré e pós-condições e, invariantes, seja possível antever situações críticas ou de risco. Por sua proposta seria possível, por exemplo, verificar se todas as pré-condições para operações críticas estão sendo respeitadas, antes de realmente executá-las ou então, a cada determinado intervalo de tempo, determinar se um conjunto de componentes está em um estado consistente ou ainda garantir que, na manifestação de erros, os clientes receberão notificações pré-determinadas. Como em sua solução o próprio servidor de aplicações é alterado, caso seja necessário um *upgrade* do servidor de aplicações essas alterações precisarão ser revistas. Apesar de trabalhar com o servidor de aplicações JBoss, a proposta apresentada no artigo não aborda o tratamento de exceções concorrentes e também não utiliza programação orientada a aspectos que fazem parte central do escopo de nosso trabalho.

1.2.4 Container-Managed Exception Handling

Simons [65], fortemente influenciado pelo trabalho de Vecellio e Sanders [70], utilizou os mesmos mecanismos do servidor de aplicações JBoss para prover às aplicações da plataforma *Java Enterprise Edition* condições para interceptar as chamadas de métodos, seus retornos e o lançamento de exceções e, desta forma, permitir acrescentar tratamento de exceções e validações do tipo pré e pós-condições à plataforma sem alterar o código das aplicações previamente implantado. Em seu trabalho, Simons define o *framework Container-Managed Exception Handling* (CMEH). No entanto, muitas das facilidades providas pelo CMEH podem ser implementadas utilizando-se programação orientada a aspectos através da linguagem AspectJ. Uma das principais razões pelas quais Simons afirma não ter utilizado a linguagem AspectJ, embora a tenha utilizado no início de seu trabalho, é o fato de que para se combinar o comportamento do aspecto com uma aplicação pronta era necessário ter acesso ao código fonte da aplicação para que o processo de combinação[9] fosse feito em tempo de compilação da aplicação, o que realmente não faz muito sentido quando se trata de componentes COTS[10] para os quais normalmente não se tem acesso ao código fonte para recompilação. Nas versões mais atuais do AspectJ, no entanto, esse problema já foi resolvido e a combinação pode ser feita a partir

[9] Do inglês: weaving.
[10] Do inglês: common-off-the-shelf.

do *bytecode* resultante da compilação normal da aplicação. Outro ponto questionado por Simons quanto ao uso do AspectJ, refere-se ao fato de um aspecto não poder lançar uma exceção que seja subclasse da classe `Exception` e que não esteja previamente declarada no método da classe que está sendo monitorada. Isso, em sua opinião, dificulta o processo de tradução de exceções[11], uma das facilidades providas pelo CMEH. Para esta segunda objeção, R. Laddad argumenta em [40] que se essa restrição não existisse, poderia levar a um grande impacto em potencial no sistema como um todo. Suponhamos, por exemplo, que um aspecto pudesse lançar uma exceção verificada[12] do tipo `SQLException` para a qual o método monitorado não estava preparado. O mecanismo de tratamento de exceções da linguagem Java não tem como tratá-la. Há duas formas de se resolver essa questão: (i) acrescentar ao método monitorado um bloco de tratamento para esta exceção, alterando seu código fonte, ou (ii) acrescentar o tipo de exceção `SQLException` à lista de exceções declaradas pelo método monitorado. Se a segunda opção for adotada, os métodos da aplicação que invocam este método monitorado passam a ter o mesmo problema e precisam decidir por uma das duas opções acima e assim sucessivamente. Laddad argumenta ainda que essa "limitação" do AspectJ é semelhante à restrição do mecanismo de tratamento de exceções da linguagem Java de não permitir que métodos sobrescritos por uma classe derivada declarem que vão lançar algum novo tipo de exceção verificada [40]. Nosso trabalho difere-se do trabalho proposto por Simons primariamente pela maneira de implementar as soluções, pois o *framework* JACAAF proposto em nosso trabalho utiliza programação orientada a aspectos enquanto o CMEH utiliza *interceptors*, um conceito interno do servidor de aplicações da plataforma *Java Enterprise Edition*. Além disso, embora o CMEH possa tratar múltiplas exceções lançadas simultaneamente não há como relacioná-las num único processamento lógico ou em uma ação atômica coordenada, o que é feito pelo *framework* JACAAF. Pode-se dizer, portanto, que o CMEH não implementa o tratamento de exceções concorrentes de uma maneira coordenada.

1.2.5 Beder

Beder [5] propôs uma arquitetura de software genérica para o desenvolvimento de sistemas concorrentes confiáveis baseado em reflexão computacional e construiu um *framework* orientado a objetos na linguagem Java que provê uma infraestrutura genérica para o desenvolvimento de sistemas concorrentes confiáveis. Seu trabalho difere-se do nosso em vários aspectos, mas principalmente pela sua maior abrangência, pois em seu trabalho, Beder propõe uma arquitetura, um conjunto de padrões de projeto e um *framework* utilizando estes padrões de projeto voltados para elementos da arquitetura.

[11] Do inglês: exception translation.
[12] Do inglês: checked exception.

1.3 Solução

A proposta deste trabalho é acrescentar à plataforma de desenvolvimento baseado em componentes *Java Enterprise Edition* os seguintes mecanismos de tolerância a falhas propostos pelo conceito de CA Actions: criação do conceito de uma ação atômica coordenada para uma computação cooperativa, tratamento de exceções concorrentes e permitir atomicidade para objetos não-atômicos. A implementação da solução proposta foi baseada na linguagem Java, componentização e programação orientada a aspectos. Foi projetada para ser utilizada principalmente em computações com comunicação assíncrona e foi concretizada na criação do *framework* JACAAF (Java CA Action Framework).

A escolha do conceito de CA Action baseou-se no fato de que a funcionalidade escolhida como o principal estudo de caso deste trabalho, a geração de arquivos para prestação de contas do sistema de faturamento de energia elétrica, possui características que demandam uma solução tolerante a falhas, orientada a objetos e com tratamento de exceções concorrentes para uma computação cooperativa, o que se encaixa perfeitamente no conceito de CA Action. Além de já existir um considerável volume de pesquisa experimental tanto para desenvolver esquemas de CA Action em Java como também na aplicação de CA Action para desenvolver estudos de casos reais.

A aplicação alvo do estudo de caso 2, possui uma solução *ad hoc* para o controle de concorrência, pois originalmente a plataforma *Java Enterprise Edition* não fornece suporte neste sentido. Como exemplo podemos citar os componentes *message-driven beans* que existem para realizar computação assíncrona em lote na plataforma *Java Enterprise Edition*, mas que não possuem um controle de concorrência que permita relacioná-los num único processamento lógico e nem realizar um tratamento de exceções concorrentes, o que leva o desenvolvedor a buscar soluções alternativas e até mesmo deixar de usar esses componentes em determinados casos, como foi o caso do sistema de faturamento de energia elétrica utilizada no estudo de caso 2.

A solução foi avaliada através da construção de 2 estudos de caso: (i) uma aplicação JBoss baseada em *message-driven beans* e (ii) um sistema real de faturamento de energia elétrica. A solução provê, portanto, mecanismos de simples adaptações que vão permitir que as aplicações da plataforma *Java Enterprise Edition* possam usufruir de mecanismos mais sofisticados de tolerância a falhas com a facilidade de utilizar anotações[13] para marcar classes e métodos onde se deseja acrescentar os mecanismos descritos acima e sem a necessidade de grandes modificações em seu código fonte previamente implementado e implantado[14].

[13] Do inglês: annotations.
[14] Do inglês: deployed.

1.4 Contribuições

As contribuições deste trabalho são:

- Projeto e implementação do *framework* JACAAF (Java CA Action Framework) utilizando a linguagem Java e a linguagem AspectJ de programação orientada a aspectos, compatível com a especificação da plataforma *Java Enterprise Edition*, que permite acrescentar conceitos de tolerância a falhas às aplicações desenvolvidas para esta plataforma, apenas com o uso de anotações e pequenas alterações no código fonte. Com isso, permitindo a evolução de aplicações pré-existentes no sentido de dar mais dependabilidade e também servindo de base para a implementação de novas aplicações da plataforma.
- Projeto, implementação e avaliação do uso do *framework* desenvolvido, aplicado a uma aplicação JBoss baseada em *message-driven beans* e a um sistema comercial de faturamento de alta complexidade e missão crítica atualmente em produção há dois anos.
- Execução de estudo de caso num contexto industrial, utilizando uma aplicação real e de missão crítica atualmente em produção.

1.5 Organização

Este documento está organizado em 5 capítulos, da seguinte forma:

- Capítulo 2 - Fundamentos de tolerância a falhas e modularização: descreve os principais fundamentos teóricos nos quais este trabalho foi embasado, subdividido em "Falha, Erro e Defeito", "Tratamento de Exceções", "Arquitetura de Software", "Desenvolvimento Baseado em Componentes", "A plataforma Java EE e seus componentes" e "Programação Orientada a Aspectos".
- Capítulo 3 - Solução proposta: *framework* JACAAF: descreve a estrutura e o funcionamento do *framework* JACAAF e como pode ser incorporado a uma aplicação.
- Capítulo 4 - Estudos de caso: descreve os 2 estudos de caso realizados. O primeiro utilizando uma aplicação JBoss baseada em *message-driven beans* e o segundo utilizando um sistema de faturamento de energia elétrica comercial e de missão crítica. Analisa descritiva e quantitativamente os resultados obtidos.
- Capítulo 5 - Conclusões e Trabalhos Futuros: possui as conclusões deste trabalho, cita as limitações da solução e propõe os trabalhos futuros e evoluções para o *framework* JACAAF.

- Apêndice A - Código fonte do *framework* JACAAF.

Capítulo 2

Fundamentos de tolerância a falhas e modularização

Desde o momento em que o computador se tornou uma ferramenta indispensável para a sociedade, podemos observar um interessante paradoxo: os benefícios trazidos pelos sistemas computacionais são tantos quantos os prejuízos que estes nos proporcionam quando deixam de funcionar ou quando funcionam incorretamente [2].

Sistemas de hardware normalmente incluem mecanismos para detecção e correção de falhas. Alguns exemplos desses mecanismos são o uso de códigos de redundância cíclica (CRC) e de replicação de dispositivos (como em discos rígidos RAID). Para sistemas de software que envolvem risco para vidas humanas (como aplicações médicas) ou risco de grandes perdas financeiras (como na pesquisa espacial), chamadas críticas, é essencial garantir que o serviço pelo qual a aplicação é responsável continuará sendo provido, mesmo que parcialmente, em situações nas quais falhas ocorram. Mesmo para sistemas que não são de missão crítica, diversas razões motivam o uso de mecanismos que os tornem capazes de prover serviços na presença de falhas [2]. Definimos um sistema como tolerante a falhas se este é capaz de prover o serviço de acordo com sua especificação, mesmo na presença de falhas [1].

2.1 Falha, Erro e Defeito

Embora os termos "falha", "erro" e "defeito" sejam usados no dia-a-dia como sinônimos, na terminologia de tolerância a falhas, possuem significados distintos e bem definidos. Um defeito ocorre quando um sistema deixa de prover um determinado serviço ou o faz em desacordo com a sua especificação. Defeitos são observáveis externamente ao sistema. Um defeito é causado por um erro, isto é, uma inconsistência no estado interno do sistema. O surgimento de erros pode ter como consequência a ocorrência de defeitos. Uma falha

corresponde a um evento ou sequência de eventos que propicia o surgimento de um erro num componente ou no sistema como um todo. Um erro é a manifestação de uma falha no sistema. É interessante observarmos que uma falha pode resultar em um, mais de um ou até mesmo em nenhum erro. Esse fato, aliado à impossibilidade de se conhecer todas as causas de uma falha, dificultam o processo de sua detecção. Outro fator que dificulta a detecção de uma falha é o fato de que várias falhas distintas podem ocasionar um mesmo erro. Por esse motivo, a identificação do erro não resulta naturalmente na identificação da falha que o causou. Desta forma, faz-se necessário uma análise rigorosa do contexto de manifestação dos erros para que seja possível inferir suas causas, para em seguida, tratá-las.

De acordo com Anderson e Lee [1], a tolerância a falhas se divide em quatro fases que devem formar a base do projeto e implementação de um sistema tolerante a falhas. Essas fases são (i) detecção de erros, (ii) confinamento e avaliação de danos, (iii) recuperação de erros e (iv) tratamento de falhas e continuação de serviço.

2.1.1 Detecção de erros

Para que um sistema seja capaz de tolerar uma falha, seus efeitos precisam primeiro ser detectados. Uma falha se manifesta em um sistema através de um erro, ou seja, uma inconsistência em seu estado interno. Embora falhas não possam ser detectadas diretamente [1], erros podem. Consequentemente todo sistema tolerante a falhas deve ser capaz de detectar erros que possam vir a se manifestar em seu estado. O sucesso de qualquer estratégia de tolerância a falhas depende grandemente da efetividade das técnicas de detecção de erros. Um exemplo de técnica bastante popular para detectar erros é o uso de invariantes; predicados que devem ser verdade durante toda a execução do sistema. Se, em algum momento da execução do sistema, uma dessas invariantes não for satisfeita, um erro ocorreu.

2.1.2 Confinamento e avaliação de danos

Desde o acionamento de uma falha, passando pela detecção do erro, até o início de uma ação para tratá-lo, existe um período de tempo crítico durante o qual, através das interações entre os componentes, este erro pode ser espalhado para outras partes do sistema. Portanto, após a detecção do erro e antes de qualquer ação corretiva, é necessário avaliar a extensão dos danos causados ao sistema, pois é possível que já tenha afetado uma parte bem maior do estado do sistema do que se supõe *a priori*. Sistemas tolerantes a falhas

devem, portanto, ser desenvolvidos de modo que, quando ocorram, os erros possam ser confinados a regiões bem definidas, facilitando a avaliação dos danos.

Erros se propagam através da comunicação entre os componentes do sistema. Logo, a determinação da extensão dos danos causados por erros depende do exame do fluxo de informação entre diferentes componentes. Este exame pode revelar as fronteiras que confinam o erro.

A determinação das fronteiras de um erro pode ser facilitada pelo uso de mecanismos que organizam as atividades (computação e comunicação) do sistema em fluxos bem definidos. Neste contexto, o conceito de ação atômica torna-se importante. O suporte a ações atômicas deve permitir que um processo, entre, execute operações sob o controle do mecanismo de ação atômica e então deixe a ação atômica. Uma ação não será atômica se, antes de seu término, ocorrer um fluxo de informação de ou para processos externos àquela ação.

A Figura 2.1 ilustra dois processos que participam de uma ação atômica (as ações atômicas são representadas pelas linhas curvas). Se o processo P2 detectou um erro no instante t, então, uma estratégia para a avaliação dos danos deveria supor que todas suas atividades desde t3 são suspeitas e que as mudanças em seu estado deveriam ser verificadas (ou abandonadas). Levando em conta que a atividade de P2 desde t3 tem sido atômica, o erro não poderia ter sido propagado para P1. Analogamente, se P1 detectou um erro no instante t, então todas suas atividades desde t1 deveriam ser consideradas suspeitas. Devido a interações entre os processos P1 e P2, todas as atividades de P2 desde t2 também deveriam ser consideradas suspeitas e, portanto, deveriam ser verificadas (ou abandonadas).

2.1.3 Recuperação de erros

Quando um erro é detectado, é preciso modificar o estado do sistema para um estado consistente e bem-definido para que o sistema possa continuar sua execução normal. Sem esta fase, a ocorrência de defeitos é muito provável, por isso, a recuperação de erros é um dos aspectos mais importantes na tolerância a falhas e uma das áreas onde há mais trabalhos disponíveis na literatura. Técnicas para recuperação de erros se dividem em duas grandes categorias que diferem na abrangência e na maneira como corrigem o estado do sistema. A distinção entre essas duas categorias origina-se da possibilidade de projetistas do sistema preverem alguns possíveis danos causados por falhas que podem ocorrer durante a execução do sistema. Para o caso de danos previstos, é possível implementar uma estratégia de recuperação de erros que consista em fazer mudanças específicas e limitadas no estado do sistema, de forma a avançá-lo a um novo e consistente estado. Assim, a recuperação de erros ocorre através do avanço de seu estado. Este mecanismo

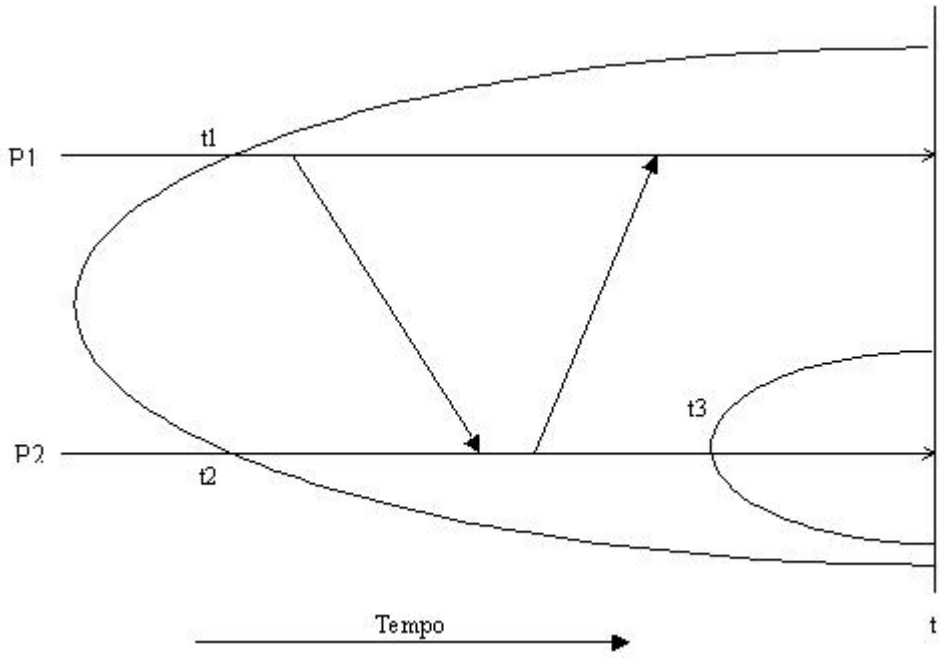

Figura 2.1: Ação Atômica

é denominado recuperação de erros por avanço[1]. Para o caso de danos não previstos ao estado do sistema, a única solução viável é substituir o estado do sistema como um todo. Este mecanismo restaura o sistema ao estado anterior à manifestação da falha. Como essa restauração de estado pode levar o sistema a um estado que já tenha existido no passado e, portanto, tenta simular o retrocesso do tempo, esta categoria de recuperação é denominada recuperação de erros por retrocesso[2] [1].

Técnicas de recuperação de erros por retrocesso restauram o estado do sistema para algum estado conhecido previamente, na esperança de que este seja consistente. Essas técnicas possuem as seguintes características [1]: (i) independência de avaliação de danos; (ii) capacidade de prover recuperação de falhas arbitrárias; (iii) abrangência, isto é, ser aplicável a todos os sistemas; e (iv) devem ser facilmente providas como um mecanismo. Mecanismos de pontos de recuperação[3] são um exemplo típico de técnica de recuperação por retrocesso. Seu funcionamento mais simples é garantir que um estado consistente e completo do sistema (*recovery point*) seja armazenado e, a partir de então, este estado salvo possa ser restaurado quando necessário. Transações atômicas e conversação, já descritos na seção 1, também são exemplos bastante conhecidos para prover recuperação de erros por retrocesso.

[1]Do inglês: forward error recovery.
[2]Do inglês: backward error recovery.
[3]Do inglês: recovery points.

Técnicas de recuperação de erros por avanço, por sua vez, partem do princípio de que, quando um erro é detectado, o sistema deve ser levado para um novo estado garantidamente consistente, através de alterações limitadas em seu estado. Recuperação de erros por avanço permite que políticas de recuperação eficientes e altamente especializadas sejam implementadas. Entretanto, essas técnicas requerem uma quantidade maior de informação contextual para que possam ser empregadas. A recuperação de erros por avanço é [1]: (i) dependente da avaliação de danos; (ii) inapropriada para recuperar o sistema de falhas não-antecipadas; (iii) projetada especificamente para um sistema particular; e (iv) impossível de se implementar como um mecanismo genérico. Recuperação de erros por avanço normalmente é implementada em sistemas de software através de tratamento de exceções.

2.1.4 Tratamento de falhas e continuação de serviço

Após a detecção do erro, seu confinamento e remoção, o sistema, teoricamente, está pronto para prover seus serviços novamente. Porém, em algumas situações, levar o sistema para um estado consistente não é o suficiente para garantir que este funcionará de forma correta. Por exemplo, se um erro ocorre em decorrência de uma falha de projeto, corrigir o erro não resolve o problema, pois a falha que o produziu é permanente e, portanto, esse erro voltará sistematicamente a acontecer. Neste caso, a única maneira de garantir que o sistema funcionará corretamente é remover a própria falha. Um problema no tratamento de falhas é que a detecção de um erro não necessariamente serve para identificar a falha que o originou.

Conforme descrito na seção 2.1.2, um erro pode ter um efeito não-local sobre o estado do sistema e relacioná-lo com a ocorrência de uma falha pode ser muito difícil. Partindo do pressuposto de que a falha foi identificada corretamente, técnicas para tratamento de falhas normalmente se baseiam na ideia de substituir o componente problemático por uma variante correta.

2.2 Tratamento de exceções

O comportamento normal de um sistema é responsável por oferecer os serviços especificados em seus requisitos. Porém, existem circunstâncias que impedem a execução adequada desses serviços. Como se espera que essas circunstâncias ocorram raramente, o sinal que indica a ocorrência de um erro é tradicionalmente chamado de exceção [15]. Apesar de se esperar uma ocorrência rara, isto não é o que ocorre na prática. Devido à grande influência do meio externo nos sistemas computacionais, as situações excepcionais são bastante frequentes [15].

A semântica de tolerância a falhas nos diz que quando um erro é detectado, é necessário sinalizar sua ocorrência para que o sistema, caso inclua mecanismos de tolerância a falhas, seja capaz de corrigi-lo. Portanto, para que o sistema possa continuar a se comportar de acordo com sua especificação, mediante a ocorrência ou lançamento de uma exceção, é necessário tratá-la. A parte do sistema que implementa seus mecanismos de recuperação de erros, responsável por torná-lo tolerante a falhas, é chamada de comportamento excepcional ou anormal.

Tratamento de exceções [29] é um mecanismo conhecido para a implementação de tolerância a falhas em sistemas de software. Os Mecanismos de Tratamento de Exceções (MTE) permitem que os desenvolvedores definam o comportamento excepcional dos sistemas, que consiste tanto das atividades de detecção e lançamento das exceções, quanto dos seus tratadores correspondentes. Diversas linguagens de programação populares dão suporte a tratamento de exceções, através da definição e implementação de sistemas de tratamento exceções.

Quando um erro é detectado e uma exceção é gerada ou lançada, o MTE da linguagem de programação ativa automaticamente o tratador mais próximo que se adeque ao tipo da exceção lançada. Dessa forma, se a mesma exceção puder ser lançada em diferentes partes do programa, é possível que diferentes tratadores sejam executados. Por esse motivo, o local de lançamento da exceção também é conhecido como Contexto de Tratamento da Exceção (CTE). Um CTE é uma região de um programa onde exceções de um mesmo tipo são tratadas de maneira uniforme. Cada CTE possui um conjunto de tratadores associados a ele e cada tratador, por sua vez, está associado a um tipo de exceção em particular. Uma exceção lançada dentro de um CTE pode ser capturada por um dos seus tratadores. Quando a exceção é tratada, o sistema pode retomar o seu comportamento normal de execução. Caso não seja encontrado nenhum tratador para a exceção, esta é sinalizada para o CTE de nível imediatamente superior. Exemplos de CTE em linguagens orientadas a objetos são comandos, blocos de código, métodos, classes e objetos.

2.2.1 Tratamento de exceções em sistemas concorrentes cooperativos

Em muitos sistemas não-concorrentes, vigora a suposição de que erros são independentes. Baseado nesta premissa, é possível tratar cada exceção como se fosse a única em determinado instante de tempo. A maioria dos sistemas de software existentes e a maioria das linguagens de programação que implementam tratamento de exceções (C++, Java, Ada, Eiffel, etc.) aderem a esse modelo de funcionamento conhecido como sequencial. Em sistemas concorrentes cooperativos, no entanto, não é seguro supor que os erros identificados são independentes. Em tais sistemas, múltiplas unidades de computação (processos,

threads, componentes, etc.) concorrentes se comunicam assincronamente e cooperam com o fim de atingir um objetivo comum [12]. Em sistemas concorrentes cooperativos, se múltiplas exceções são lançadas simultaneamente, é necessário tratá-las de maneira cooperativa, já que podem ser decorrentes da mesma falha. Nestes casos, o tratamento de exceções deve ser cooperativo e envolver as unidades de computação participantes da mesma forma como é feito durante o comportamento normal do sistema.

Quando ocorre o lançamento de exceções concorrentes, é necessário identificar qual tratador deve ser executado em cada um dos participantes. Para isso, devido à relação estreita entre o tratador e o tipo da exceção, é necessário identificar um tipo de exceção que seja equivalente às exceções concorrentes lançadas. Para isso, é possível utilizar-se de uma técnica conhecida como grafo (ou árvore) de resolução [12], que estrutura as exceções de maneira hierárquica. Havendo o lançamento concorrente de exceções, o tipo da exceção equivalente nesse grafo é o ancestral mais próximo, que seja comum a todas as exceções lançadas. Sendo assim, as exceções de um grafo podem ser classificadas em dois tipos: (i) exceções simples, que são representadas pelas folhas do grafo; e (ii) exceções compostas, que se referem aos demais nós da hierarquia. A Figura 2.2 mostra um exemplo de um grafo de resolução de exceções estruturado em forma de árvore. Neste exemplo, se as exceções `Exceção2` e `Exceção3` forem lançadas simultaneamente, serão acionados os tratadores associados à exceção `Exceção1`, pois esta é o ancestral mais próximo e comum às exceções lançadas.

Por tudo o que foi dito e também devido às complicações inerentes à programação de sistemas concorrentes, mecanismos de tratamento de exceções para sistemas concorrentes cooperativos são sensivelmente diferentes de mecanismos para sistemas sequenciais.

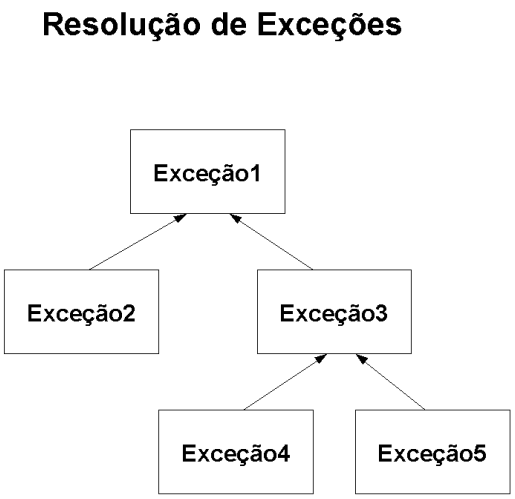

Figura 2.2: Árvore de resolução de exceções

2.2.2 Componente tolerante a falhas ideal

O conceito de componente tolerante a falhas ideal (CTFI) [1] define um *framework* conceitual para estruturar tratamento de exceções em sistemas de software. Um CTFI é um componente no qual as partes responsáveis pelo comportamento normal e pelo comportamento excepcional estão separadas e bem-definidas dentro da sua estrutura interna, permitindo que erros sejam detectados e tratados com maior facilidade. O objetivo da abordagem de CTFI é proporcionar uma forma de estruturar sistemas que minimize o impacto dos mecanismos de tolerância a falhas em sua complexidade global.

A Figura 2.3 apresenta a estrutura interna de um componente tolerante a falhas ideal e os tipos de mensagem que ele troca com outros componentes na arquitetura. Ao receber uma requisição de serviço, um CTFI fornece uma resposta normal se a requisição é processada com sucesso. Se a requisição de serviço não é válida, é sinalizada uma exceção de interface. Se o serviço está disponível, mas ocorre uma falha durante o processamento da requisição e o CTFI não é capaz de tratá-la internamente, é levantada uma exceção de defeito.

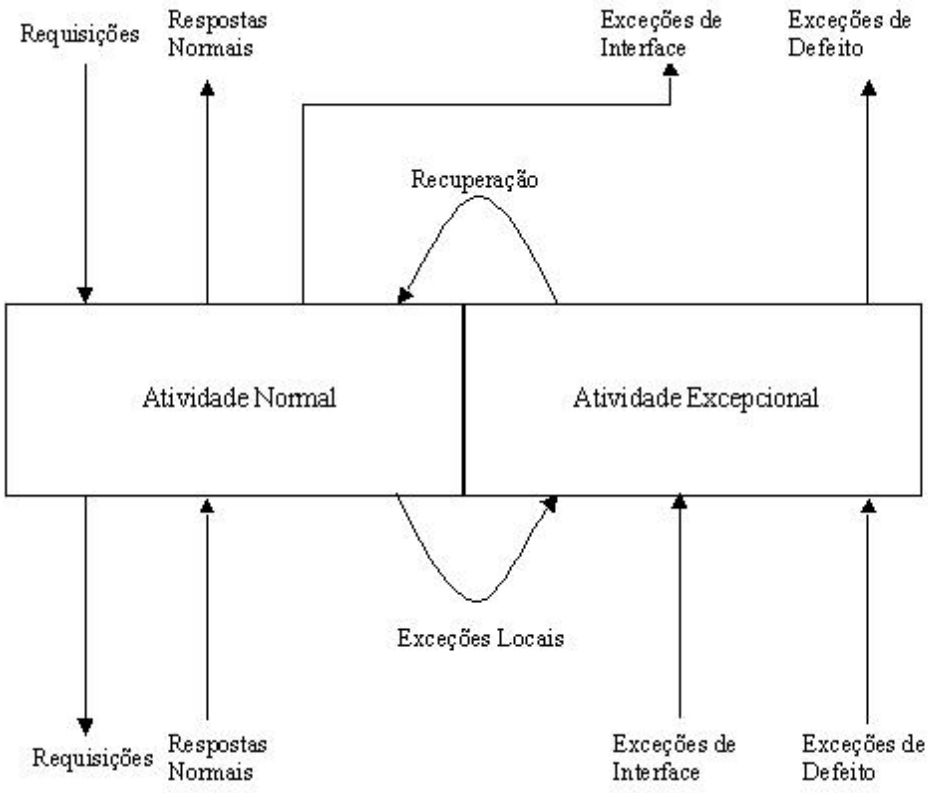

Figura 2.3: Componente Tolerante a Falhas Ideal

2.2.3 Ações Atômicas Coordenadas (CA Actions)

O conceito de CA Action foi proposto para prover tolerância a falhas de hardware e de software num sistema concorrente orientado a objetos, integrando os conceitos complementares de conversação (concorrência cooperativa) e transação atômica (concorrência competitiva) e, estabelecendo uma semântica para tratamento de exceções concorrentes, além de dar suporte ao uso conjunto de recuperação de erro por avanço e retrocesso [59]. Para a computação cooperativa, CA Action usa como base o conceito de conversação descrito na seção 1 e que pode ser definido como "uma técnica de tolerância a falhas para estruturar sistemas concorrentes a realizar recuperação de erros de maneira coordenada produzindo resultados de forma dependente e cooperativa" [53]. Cada participante pode se comunicar exclusivamente com outros participantes da mesma conversação e ao final do processamento todos os participantes fazem um teste de aceitação. Se pelo menos um desses testes falhar, uma exceção é levantada e todos devem tentar se recuperar. O estado original (antes do início da conversação) de cada participante é recuperado e utiliza-se um algoritmo alternativo, quando disponível, numa nova tentativa.

Para integrar a computação competitiva CA Action utiliza transações atômicas: "uma única ação lógica, composta por uma sequência de operações básicas em dados ou objetos compartilhados de maneira protegida, isto é, provendo as propriedades ACID[4] para todas as operações desta transação" [28]. As quatro propriedades ACID são: atomicidade, consistência, isolamento e durabilidade, conforme descritas na seção 1.

A Figura 2.4 mostra um exemplo do funcionamento geral de uma CA Action. Cada CA Action é criada como uma unidade com múltiplas entradas, com papéis[5] sendo ativados por participantes os quais cooperam e trocam informações através de objetos internos. A cooperação também ocorre nos casos em que exceções são lançadas: neste momento, os tratadores de exceções apropriados são executados para se recuperar do estado com erro, avançando (recuperação de erro por avanço) ou retrocedendo (recuperação de erro por retrocesso) seu estado. Quando uma CA Action se inicia, uma transação, que é finalizada juntamente com a CA Action, é iniciada nos objetos externos. A CA Action pode terminar normalmente (saída normal) ou excepcionalmente (saídas: *exceptional*, *abort* ou *failure*). Quando uma CA Action termina excepcionalmente, uma exceção é sinalizado ao contexto que a invocou, que pode recursivamente corresponder a uma outra CA Action (aninhamento)[6]. É interessante perceber que uma CA Action pode se degenerar numa transação, caso somente um papel esteja envolvido, ou ainda numa conversação, quando nenhum objeto externo está sendo acessado [56]. Todo objeto externo acessado a partir de uma CA Action deve ser capaz de ser restaurado para o seu último estado consistente

[4]Do inglês: Atomicity, Consistency, Isolation and Durability.
[5]Do inglês: role.
[6]Do inglês: nesting.

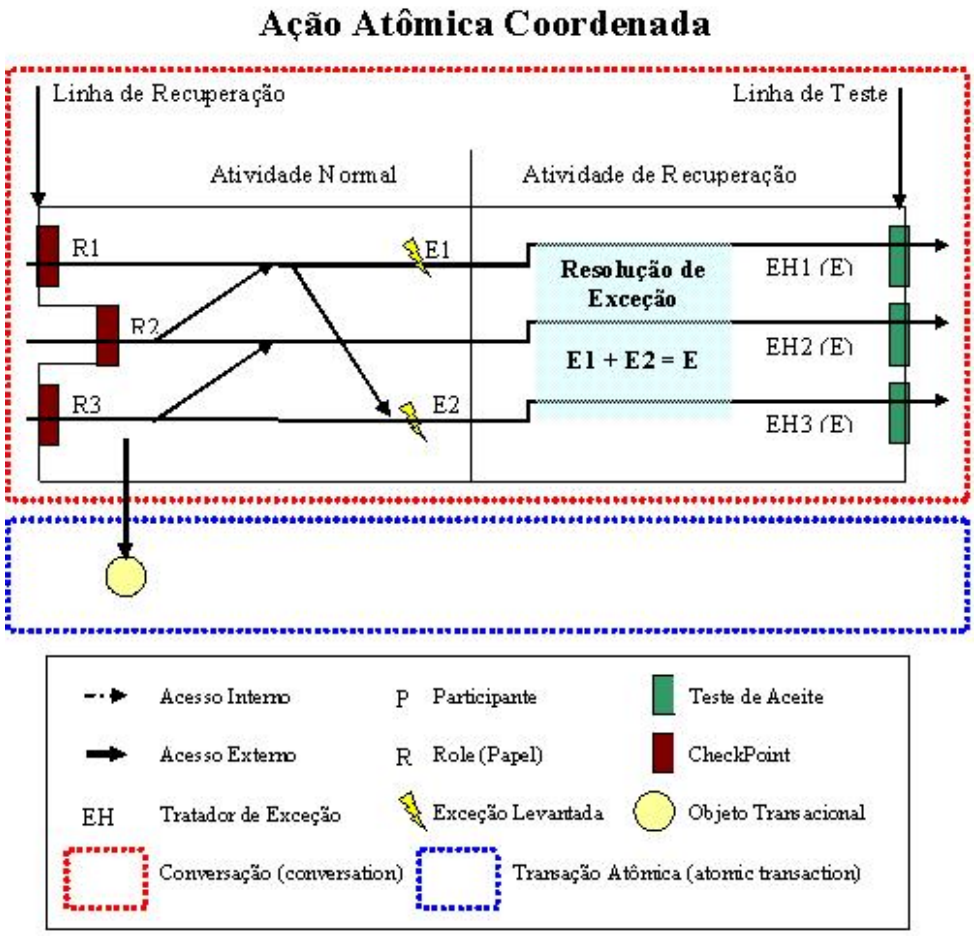

Figura 2.4: CA Action

(caso a recuperação de erro por retrocesso seja disparada) e prover seu próprio mecanismo de recuperação de erros [59].

A semântica de tratamento de exceções de CA Action especifica que na ocorrência de uma exceção, o mecanismo de recuperação de erros por avanço deve ser iniciado primeiro. Neste ponto, a CA Action ainda pode terminar normalmente, caso o mecanismo de recuperação consiga completar o serviço solicitado ou terminar excepcionalmente (saída do tipo: *exceptional*), caso o serviço consiga ser parcialmente provido. Em todo caso, se uma exceção é levantada durante a recuperação de erro por avanço, o mecanismo de recuperação de erro por retrocesso é disparado. A principal função do mecanismo de recuperação de erro por retrocesso é recuperar o estado dos participantes e de todos os objetos externos para seus respectivos últimos estados consistentes. Esse processo é conhecido por *rollback*. Se a recuperação de erro por retrocesso é bem sucedida, a CA Action termina com uma saída do tipo *abort*, indicando que o serviço requerido não pôde ser

provido, mas que o estado original (antes da chamada da CA Action) foi recuperado. Se por alguma razão, a recuperação de erro por retrocesso não puder ser completada, a CA Action apresentou um defeito e, portanto, uma saída do tipo *failure* é retornada indicando que o sistema está num estado não definido.

Na Figura 2.4 pode-se ver o tratamento de uma exceção de forma coordenada durante a execução de uma CA Action. Neste exemplo, os participantes 1, 2 e 3 (P1, P2 e P3) ativam seus respectivos papéis (R1, R2 e R3), guardam uma referência a seus respectivos estados (*checkpoint* ou *recovery line*) antes de qualquer troca de mensagem e iniciam uma CA Action. Durante o processamento, R1 levanta a exceção E1 e R3 a exceção E2[7]. Neste instante, o mecanismo de tratamento de exceção da CA Action decide através de uma árvore ou um grafo de exceção qual é a exceção "resultante" e informa aos papéis, que por sua vez, passam o controle para seus tratadores de exceção (EH1, EH2 e EH3, respectivamente) para realizarem a recuperação de erro por avanço. A recuperação é bem sucedida e os três papéis terminam seus testes de aceitação com sucesso, descartando a referência a seus estados iniciais. A CA Action termina com sucesso.

Suponhamos agora que ocorresse outra exceção durante a recuperação de erro por avanço. Nesta situação, o mecanismo de recuperação de erro por retrocesso seria ativado para retornar o sistema a um estado consistente. No pior caso, poderia ocorrer ainda uma terceira exceção durante a fase de recuperação de erro por retrocesso, impedindo o sistema de se recuperar, o que levaria o sistema para um estado inconsistente ou indefinido.

2.3 Arquitetura de Software

Assim como acontece com o desenvolvimento baseado em componentes, não existe uma definição universalmente aceita para arquitetura de software [4, 52]. Há um conjunto de características, porém, que é mencionado na grande maioria das definições existentes na literatura. Essas características são sumarizadas pela definição adotada por Clements e Northrop [10]:
"Arquitetura de software é a estrutura dos componentes de um sistema, suas inter-relações e, os princípios e guias que nortearam seu projeto e evolução através do tempo".

A arquitetura de software, através de um alto nível de abstração, define o sistema em termos de seus componentes arquiteturais, que representam unidades abstratas do sistema responsáveis por alguma computação, a interação entre essas entidades, que são materializadas explicitamente através dos conectores e, os atributos e funcionalidades de cada um [66]. A maneira como os componentes de um sistema ficam dispostos é conhecida como

[7]De acordo com o conceito de CA Action, as exceções E1 e E2 levantadas podem ser de diferentes tipos, porém, nesta versão do *framework* proposto como solução, implementamos apenas um único tipo de exceção.

configuração. Essa visão estrutural do sistema em um alto nível de abstração proporciona benefícios importantes, que são imprescindíveis para o desenvolvimento de sistemas de software complexos. Os principais benefícios são: (i) a organização do sistema como uma composição de componentes lógicos; (ii) a antecipação da definição das estruturas de controle globais; (iii) a definição da forma de comunicação e composição dos elementos do projeto; e (iv) o auxílio na definição das funcionalidades de cada componente projetado. Além disso, uma propriedade arquitetural representa uma decisão de projeto relacionada a algum requisito não-funcional do sistema, que quantifica determinados aspectos do seu comportamento, como confiabilidade, reusabilidade e modificabilidade [9, 66]. Essas decisões são tomadas no início do projeto do sistema e são, portanto, difíceis de se mudar em etapas posteriores do desenvolvimento.

A presença de uma determinada propriedade arquitetural pode ser obtida através da utilização de estilos arquiteturais que possam garantir a preservação dessa propriedade durante o desenvolvimento do sistema [63, 45]. Um estilo arquitetural caracteriza uma família de sistemas que são relacionados pelo compartilhamento de suas propriedades estruturais e semânticas. Esses estilos definem inclusive restrições de comunicação entre os componentes do sistema. Alguns exemplos de estilos são: o baseado em camadas, cliente-servidor e *pipes and filters* [61].

Propriedades arquiteturais são derivadas dos requisitos do sistema e influenciam, direcionam e restringem todas as fases do seu ciclo de vida. Sendo assim, a arquitetura de software é um artefato essencial nos processos de desenvolvimento de softwares modernos, sendo útil em todas as suas fases [18]. A importância da arquitetura fica ainda mais clara no contexto do desenvolvimento baseados em componentes. Isso acontece, uma vez que na composição de sistemas, os componentes precisam interagir entre si para oferecer as funcionalidades desejadas. Além disso, devido à diferença de abstração entre a arquitetura e a implementação de um sistema, um processo de desenvolvimento baseado em componentes deve apresentar uma distinção clara entre os conceitos de componente arquitetural, que é abstrato e não é necessariamente instanciável e, componente de implementação, que representa a materialização de uma especificação em alguma tecnologia específica e deve necessariamente ser instanciável. É interessante lembrar que a arquitetura de sistemas de software muito complexos normalmente envolve diversos estilos arquiteturais.

2.4 Desenvolvimento baseado em componentes (DBC)

A ideia de se construir sistemas a partir de partes pré-fabricadas é antiga na indústria de uma forma geral. Há décadas que carros, computadores e até mesmo casas são construídos

dessa maneira. No entanto, o conceito de produção de software a partir de componentes pré-fabricados é relativamente recente. Foi mencionado pela primeira vez por Mcllroy em 1969 [43]. No entanto, o interesse pelo desenvolvimento baseado em componentes (DBC) veio a se intensificar quase três décadas depois. Em 1996 foi realizado o *First International Workshop on Component-Oriented Programming* (WCOP'96), a partir do qual o interesse pelas técnicas baseadas em componentes passou a crescer e a se consolidar, seja sob o título de desenvolvimento baseado em componentes, ou de engenharia de software baseada em componentes (ESBC). Alguns autores afirmam que a reutilização de componentes de software tem potencial para reduzir drasticamente os custos e o tempo necessários para construir uma aplicação [46]. Há diversas definições para ESBC e DBC, das quais adotaremos as seguintes:

- "Em DBC, a tarefa dos desenvolvedores é compor um (grande) sistema de software a partir de (um grande número de) componentes reutilizáveis e, se possível, uma relativamente pequena quantidade de linhas de código específicas para a aplicação" [60];
- "A engenharia de software baseada em componentes trata da rápida composição de sistemas a partir de componentes onde: (i) componentes e *frameworks* tem propriedades certificadas; e (ii) essas propriedades certificadas formam a base para se prever as propriedades do sistema construído a partir destes componentes" [3].

Szyperski afirma que um componente, no contexto do desenvolvimento baseado em componentes, deve [69]: (i) presumir que faz parte de uma arquitetura; (ii) apresentar funcionalidades através de interfaces providas; (iii) explicitar suas dependências paramétricas através de interfaces requeridas; (iv) ter dependências estáticas; (v) ser baseado em alguma plataforma de componentes específica; (vi) requerer outros componentes; e (vii) requerer contexto por-instância. O autor afirma ainda que, para tornar possível a composição de componentes, todas as dependências e suposições importantes da implementação de um componente precisam ser capturadas. Essa é uma afirmação importante, já que vai contra a tendência dos modelos de componentes atuais de enfatizar apenas as funcionalidades que componentes proveem.

Apesar das diferenças entre as definições, há um consenso de que no DBC a ênfase está na integração entre componentes e não apenas em sua construção.

Para tornar viável a integração entre componentes desenvolvidos independentemente, foram criados os modelos (ou plataformas) de componentes. Um modelo de componentes especifica padrões que devem ser seguido pelos desenvolvedores que o adotam. A conformidade com um modelo de componentes é uma das características que diferencia componentes de outras formas de modularização [3]. Para auxiliar no desenvolvimento baseado em modelos de componentes, foram desenvolvidos diversos *frameworks* de componentes,

implementações de serviços que dão suporte a um determinado modelo de componentes e garantem que suas restrições não são violadas. Atualmente, os modelos de componentes mais populares são Enterprise JavaBeans, da Sun [67], DCOM [47], .NET da Microsoft [48], e o CORBA Component Model, da OMG [49].

2.5 A plataforma Java EE e seus componentes

2.5.1 O servidor de aplicações Java EE

A plataforma *Java Enterprise Edition* (Java EE)[36], anteriormente conhecida como J2EE, provê uma abordagem baseada em componentes para projetar, desenvolver, compor e implantar aplicações de grande porte e oferece um modelo de aplicação distribuído e multicamadas, com reutilização de componentes, segurança e controle transacional baseado na linguagem Java [38]. Seus componentes são chamados de *Enterprise Java Beans* (EJB).

Um servidor de aplicações[8] Java EE é um software que disponibiliza um ambiente para a instalação e execução de determinadas aplicações. Os servidores de aplicações também são conhecidos como software de *middleware*. O objetivo do servidor de aplicações Java EE é disponibilizar uma plataforma que permite ao desenvolvedor abstrair-se de algumas das complexidades de um sistema computacional complexo, permitindo que se preocupe com as regras de negócio e não com questões de infraestrutura da aplicação. O servidor de aplicações provê essa infraestrutura comum às aplicações de grande porte como segurança, distribuição, transação, disponibilidade, balanceamento de carga e tratamento de exceções. Além do ambiente para a execução dos componentes *Enterprise Java Beans*, o servidor de aplicações também permite a execução de objetos mais simples da linguagem Java, conhecidos como POJO. POJO é um acrônimo para *Plain Old Java Object* e é usado para enfatizar que um determinado objeto é um objeto comum (ou normal) da linguagem Java e não um objeto diferenciado. É usado para explicitar que o objeto Java em questão não segue nenhum dos principais modelos de objetos Java, nem a convenções ou *frameworks* da linguagem, especialmente que não se trata de um *Enterprise Java Bean* [50]. O servidor de aplicações JBoss é uma implementação *open source* da especificação da plataforma *Java Enterprise Edition* e foi utilizada para validar a solução proposta neste trabalho.

A Figura 2.5 nos permite perceber como um servidor de aplicações Java EE está dividido funcionalmente em camadas.

A camada cliente é a interface entre o usuário e o sistema e pode ser composta desde um simples *browser* que se comunica com o servidor através do protocolo HTTP, até aplicações inteiras que se comunicam com o *container* via chamada remota de método (RMI). O seu

[8]Do inglês: application server.

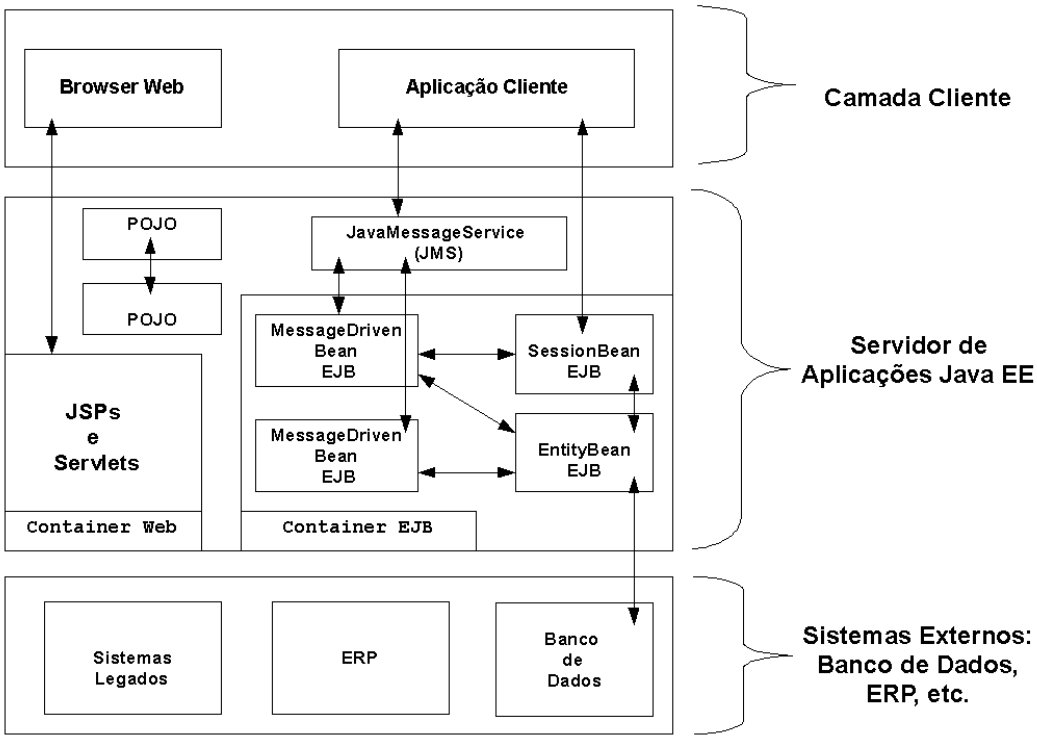

Figura 2.5: O servidor de aplicações Java EE

conteúdo é criado a partir de *Servlets*, que são componentes que dinamicamente processam requisições e respostas gerando dados HTML e XML para a camada de apresentação. A tecnologia *Java Server Pages* (JSP) é utilizada para produzir aplicações que acessem banco de dados, manipulem arquivos no formato texto, capturem informações a partir de formulários web e captem informações sobre o visitante e sobre o servidor de aplicações Java EE. Dentro do *container* Web, os JSPs funcionam de forma muito semelhante aos *Servlets* também gerando HTML.

No servidor de aplicações Java EE propriamente dito, estão localizados os *containers* Web e EJB. O *container* Web é responsável por prover o conteúdo de apresentação (Servlets e JSPs), ou seja, a interface gráfica que o usuário irá realmente ver e com a qual irá interagir através de seu *browser*. Já o *container* EJB é o ambiente de execução dos componentes da plataforma Java EE, sendo responsável por prover os serviços da plataforma e gerenciar o ciclo de vida de seus componentes; normalmente nesses componentes estão concentradas as regras de negócio da aplicação.

Muitos serviços são providos pelo servidor de aplicações. O serviço JMS (Java Message Service) é um deles e disponibiliza facilidades para uso de mensagens. Através da implementação do JMS, o servidor de aplicações JBoss pode prover serviço de mensagens assíncronas. A implementação do JBoss para o JMS é chamada JBossMQ. Esse serviço

é provido através de um conjunto de canais ou "tópicos"[9]. Produtores e consumidores dessas mensagens se registram no tópico JMS e as mensagens são enviadas assincronamente através desses tópicos. Os consumidores podem se registrar a qualquer momento e em qualquer quantidade e qualquer um dos produtores registrados pode enviar mensagens para o tópico. O produtor não tem conhecimento e não precisa se preocupar se há um ou vários consumidores para sua mensagem e os consumidores, por sua vez, não têm de se preocupar com o estado do produtor. Uma mensagem JMS pode ser uma simples *string* ou conter um objeto complexo. O JBoss traz em sua instalação padrão um conjunto de tópicos pré-definidos, mas outros podem ser acrescentados conforme a necessidade.

Finalmente, a última camada representa a interação do servidor com sistemas de informação externos, como banco de dados, aplicações legadas, sistemas de ERP, etc.

Enterprise JavaBeans (EJBs) são os componentes da plataforma *Java Enterprise Edition*. São executados dentro de um *container*, isto é, um ambiente de execução dentro do servidor de aplicações que provê de forma transparente para o desenvolvedor, alguns dos requisitos não-funcionais da arquitetura de software de uma aplicação [34]. Existem três diferentes tipos de EJBs [33]:

- *Session Bean*: representa uma sessão dentro do *container* e executa processamentos síncronos. Tal como uma sessão, é exclusiva de um cliente e, portanto, não é compartilhada. Logo que o cliente finaliza sua sessão, o *session bean* é "liberado" e outro cliente pode utilizá-lo sem referência alguma ao processamento ou cliente anterior. Um exemplo clássico de aplicação para esse tipo de EJB é o carrinho de compras em sites de *e-commerce*, onde o cliente retira um carrinho para seu uso exclusivo, coloca nele seus produtos, finaliza a compra e depois o devolve.

- *Entity Bean*: representa uma entidade de negócio persistida em algum meio de armazenamento. É persistente, isto é, a entidade representada pelo *entity bean* e seus atributos (estado) são persistentes mesmo depois da aplicação ter sido desligada. Pode ser mapeada como um registro em uma tabela de banco de dados relacional e pode posteriormente ser recuperada mesmo que o servidor de banco de dados esteja inativo num determinado momento.

- *Message-Driven Bean* (MDB): processa mensagens de forma assíncrona. Seu funcionamento básico consiste em receber uma mensagem, fazer o processamento de acordo com a regra de negócio implementada e com o conteúdo da mensagem recebida e notificar o serviço de mensagens do servidor (Java Message Service ou simplesmente JMS) sobre a recepção e processamento de cada mensagem. É bastante utilizado para processamentos assíncronos em lote.

[9]Do inglês: topics.

2.5.2 Message-Driven Beans (MDB)

Os *message-driven beans* surgem com o propósito de ser uma alternativa dentro do modelo de componentes de EJB para duas limitações que os outros 2 tipos de EJB não resolvem. A primeira delas se refere ao fato de que um cliente dos demais EJBs precisam necessariamente aguardar o retorno de sua chamada, pois só respondem de forma síncrona. A segunda limitação está relacionada ao fato de que o cliente deve conhecer a interface para conseguir invocar algum serviço (método) do EJB; uma mudança nestas interfaces implicam necessariamente numa mudança nos clientes. Os *message-driven beans*, portanto, surgem para que o cliente possa fazer uma comunicação assíncrona através de mensagens, de forma que não precisem conhecer a interface do componente que proverá o serviço e nem aguardar o retorno de sua chamada.

A diferença mais visível entre os *message-driven beans* e os demais tipos de EJB é que as aplicações clientes não acessam os *message-driven beans* através de interfaces. Diferentemente de um *session* e de um *entity bean* que são compostos por interfaces e uma classe (*bean*), um *message-driven bean* possui apenas uma classe (*bean*) e é acessado somente através do serviço de mensagens; o cliente, portanto, se comunica com o serviço de mensagens e este com o *message-driven bean*. Sob alguns aspectos o *message-driven bean* funciona como um *session bean* do tipo *stateless*: (i) suas instâncias não guardam nenhuma referência ou estado de um cliente específico; (ii) todas as suas instâncias são equivalentes, permitindo que o *container* EJB atribua uma mensagem a qualquer instância daquele *message-driven bean* disponível; (iii) o *container* EJB pode fazer um *pool* dessas instâncias para permitir o processamento concorrente de um grande volume de mensagens; (iv) uma única instância pode processar mensagens provenientes de diferentes clientes.

Quando uma mensagem é recebida, o *container* EJB invoca o método `onMessage()` do *message-driven bean* para processar a mensagem. Este método normalmente lê e processa a mensagem de acordo com a lógica de negócio da aplicação. Desta forma, não é possível invocar nenhum método do *message-driven bean* de forma direta e, uma vez iniciado o processamento da mensagem a instância do *message-driven bean* fica "incomunicável", apesar do fato que a instância ainda pode invocar outros métodos utilitários ou ainda utilizar serviços de um *entity* ou *session bean*.

2.6 Programação Orientada a Aspectos (AOP)

A programação orientada a aspectos (AOP) [39] foi proposta na segunda metade dos anos 90 como uma maneira de modularizar interesses transversais[10], ou seja, requisitos periféricos que afetam o sistema como um todo e cuja implementação está espalhada em

[10]Do inglês: crosscutting concerns.

diferentes módulos do sistema, diluída no código responsável pelos requisitos funcionais. Exemplos de tais interesses são *logging*, autenticação e segurança.

A programação orientada a aspectos baseia-se na ideia de que sistemas computacionais são melhor programados se especificarmos separadamente os seus vários interesses e alguma descrição de seus relacionamentos, e se usarmos os mecanismos providos por este tipo de programação para combiná-los em um programa. Um elemento importante da programação orientada a aspectos é a dicotomia entre linguagem base, ou de componentes, e linguagem de aspectos [8]. Essa dicotomia se baseia nos seguintes princípios: (i) sistemas são decompostos em aspectos e componentes; (ii) aspectos modularizam interesses transversais; (iii) componentes modularizam os requisitos ou interesses não-transversais; e (iv) aspectos devem ser explicitamente representados a parte de componentes e outros aspectos. Alguns exemplos de linguagens orientadas a aspectos são AspectJ [40] e Eos [58]. A primeira usa Java como linguagem base, enquanto a segunda usa C#. Programas nas linguagens base e de aspectos são combinados através de um processo semelhante à compilação chamado *weaving*. A ferramenta responsável por esta tarefa é chamada de *weaver*.

2.6.1 Propriedades fundamentais

As duas propriedades fundamentais que tornam a programação orientada a aspectos útil e diferenciam esse paradigma de seus predecessores são: quantificação e transparência [23]. Quantificação é a capacidade de escrever comandos unitários que são capazes de afetar diversas partes de um programa. A quantificação torna possível expressar comandos do tipo "nos programas P, sempre que a condição C for verdadeira, execute a ação A". Transparência, no contexto da programação orientada a aspectos, significa que programas escritos na linguagem base não precisam ser preparados especificamente para serem combinados com aspectos, ou seja, podem ser escritos como se os aspectos simplesmente não existissem. Alguns autores afirmam que transparência é um termo muito forte e que há benefícios em organizar programas na linguagem base de tal maneira que combiná-los com os aspectos seja mais fácil [31]. Por isso, muitos autores preferem usar o termo não-invasão[11], ao invés de transparência. É importante enfatizar que, independentemente do termo empregado, é fundamental a ideia de que programas na linguagem base não devem fazer referência explícita a aspectos.

Diversos trabalhos existentes na literatura mostram que a programação orientada a aspectos é útil para modularizar vários requisitos comuns no desenvolvimento de sistemas de software tais como distribuição [13, 64], persistência [40, 64], autenticação [40], tratamento de exceções [6, 42] e implementação de padrões de projeto [17, 30, 32]. O impacto

[11] Do inglês: non-invasiveness.

da programação orientada a aspectos tanto na academia quanto na indústria nos últimos anos é tão grande que a revista *Technology Review* de janeiro de 2001 incluiu a programação orientada a aspectos na lista das "dez tecnologias emergentes que mudarão o mundo" [54].

2.6.2 AspectJ

AspectJ foi a primeira implementação prática de programação orientada a aspectos desenvolvida no fim da década de 90 por membros da mesma equipe que criou o termo programação orientada a aspectos[12] e foi implementada na linguagem Java [41]. Atualmente faz parte do projeto *open source* `eclipse.org` [20].

Um aspecto é a unidade central do AspectJ da mesma forma como uma classe é a unidade central da linguagem Java. Cada aspecto define uma função específica que pode afetar várias partes de um sistema, como distribuição, por exemplo. Aspectos podem alterar a estrutura estática de um sistema adicionando membros (atributos, métodos e construtores) a uma classe, alterando a hierarquia do sistema, ou até mesmo convertendo o lançamento de uma exceção verificada[13] por uma não-verificada (exceção de *runtime*). Esta característica de alterar a estrutura estática de um programa é chamada *static cross-cutting*.

Além de afetar a estrutura estática, um aspecto também pode afetar a estrutura dinâmica de um programa. Isto é possível através da interceptação de pontos no fluxo de execução, chamados *join points*, e da adição de comportamento antes ou depois dos mesmos, ou ainda através da obtenção de total controle sobre o ponto de execução de forma que se desejado, um método invocado não seja nem mesmo executado. Exemplos de *join points* são: invocação e execução de métodos, inicialização de objetos, execução de construtores, tratamento de exceções, acesso e atribuição a atributos, entre outros. Também é possível definir um *join point* como resultado da composição de vários *join points*.

Pointcut é uma construção que seleciona *join points* e coleta informações do seu contexto de execução.

Advices definem a computação a ser realizada quando o fluxo de execução atinge os *join points* selecionados pelos *pointcuts*. Existem diferentes tipos de advices no AspectJ:

- *before*: Executa quando o *join point* é alcançado, mas imediatamente antes da sua computação
- *after returning*: Executa após a computação com sucesso do *join point*

[12] Do inglês: aspect oriented programming.
[13] Do inglês: checked exception.

- *after throwing*: Executa após a computação sem sucesso do *join point*, ou seja, após o lançamento de uma exceção
- *after*: Executa após a computação do *join point* independente da situação
- *around*: Executa quando o *join point* é alcançado e pode tomar a decisão de executar ou não o *join point* ou ainda executá-lo com outros parâmetros

Simplificadamente, o funcionamento mais comum de um aspecto inclui a definição de *pointcuts*, os quais selecionam *join points* e executam os *advices*. Um código que exemplifica alguns dos conceitos acima pode ser o seguinte:

```
1 public aspect ExampleAspect {
2     before() : execution(void Account.credit(float)) {
3         System.out.println("About to perform credit operation");
4     }
5     declare parents: Account implements BankingEntity;
6 }
```

No trecho de código acima, o aspecto `ExampleAspect` é criado e terá os seguintes efeitos sobre o sistema base:

- nas linhas 2 e 3 define-se que toda vez que o método `void credit()` com parâmetro do tipo `float`, da classe `Account` for executado, antes de sua computação propriamente dita, a mensagem texttt"About to perform credit operation" será escrita na saída padrão.
- na linha 5 a classe `Account` passa a implementar a interface `BankingEntity`.

2.7 Resumo

Este capítulo apresentou os fundamentos teóricos para o entendimento do problema e da solução a ser apresentada neste trabalho. Em sua primeira seção 2.1 definiu conceitualmente os termos falha, erro e defeito, além de descrever as fases da tolerância a falhas. Na seção 2.2 foi descrito o funcionamento do tratamento de exceções como um mecanismo de tolerância a falhas em software, o conceito de componente tolerante a falhas ideal e os conceitos e mecanismos que compõem o conceito de CA Action. Na seções 2.3 e 2.4, descreveu de maneira sucinta o que é arquitetura de software e definiu o desenvolvimento baseado em componentes, respectivamente, para que na seção 2.5 fosse explicado o conceito de plataforma de componentes e pudesse ser detalhado um servidor de aplicações para a plataforma *Java Enterprise Edition*. Os conceitos de programação orientada a aspectos e o detalhamento da linguagem AspectJ foram mostrados na seção 2.6. Desta forma o capítulo coloca o embasamento teórico para entendimento deste trabalho.

Capítulo 3

Solução proposta: *framework* JACAAF

Este capítulo descreve o *framework* JACAAF (Java CA Action *Framework*), composto por um conjunto de classes e interfaces Java e de aspectos implementados na linguagem AspectJ, com o intuito de permitir, com poucas intervenções no código fonte, que aplicações *Java Enterprise Edition* possam usufruir das seguintes facilidades: (i) criação do conceito de uma ação atômica coordenada para uma computação cooperativa, (ii) uso de tratamento de exceções concorrentes e (iii) suporte para atomicidade voltado para objetos não-atômicos.

A Figura 3.1 mostra os três pontos onde *framework* JACAAF atua no servidor de aplicações Java EE:

- **Ponto 1**: o *framework* JACAAF pode atuar dentro do *container* EJB, monitorando lançamento de exceções e chamadas de métodos entre os *enterprise java beans*(EJBs), além de poder ser invocado pelos *enterprise java beans* da aplicação.

- **Ponto 2**: o *framework* pode monitorar o lançamento de exceções e chamadas de métodos de classes Java comuns (ou POJO)[1], isto é, classes que não são componentes da plataforma *Java Enterprise Edition*. Estas classes se encontram dentro do servidor de aplicações da plataforma *Java Enterprise Edition*, mas estão localizadas fora do *container* EJB.

- **Ponto 3**: o *framework* JACAAF ainda pode atuar monitorando lançamento de exceções e interceptando as chamadas de métodos entre os POJOs e os *enterprise java beans*.

[1]Do inglês: plain old java object.

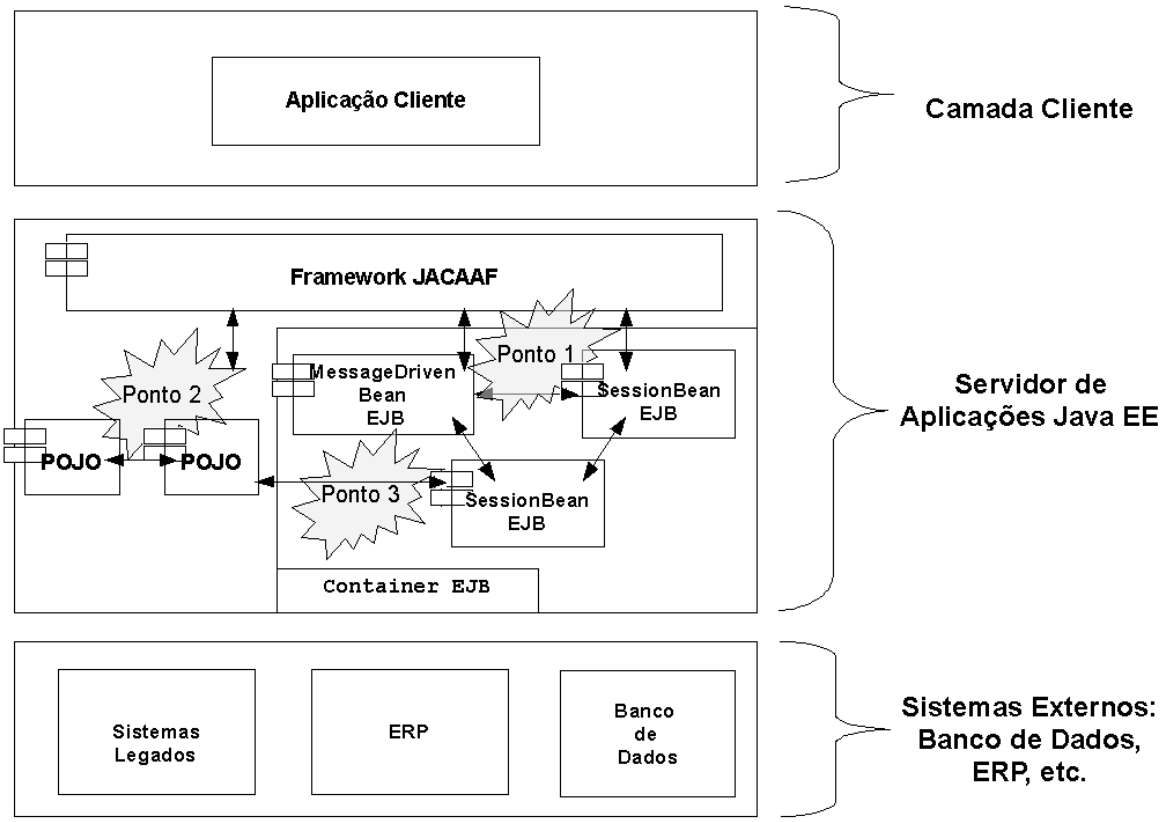

Figura 3.1: O *framework* JACAAF e o servidor de aplicações Java EE

Em seu objetivo mais amplo, o *framework* JACAAF fornece condições para estruturar aplicações implantadas no servidor de aplicações da plataforma *Java Enterprise Edition* para uma CA Action.

3.1 Arquitetura do *framework* JACAAF

A Figura 3.2 mostra os componentes arquiteturais do *framework* JACAAF e a relação existente entre eles. Arquiteturalmente o *framework* JACAAF está dividido em quatro componentes que interagem entre si e interceptam eventos ocorridos dentro do servidor de aplicações. A função de cada um dos quatro componentes é a seguinte:

- **CAAction** é o componente responsável pelo controle do processamento como um todo e que mantém uma referência para cada participante e os objetos atômicos de cada participante (quando existirem). É através dele que os participantes solicitam a entrada em uma CA Action.

- **AtomicObject** é o componente responsável por permitir que objetos passem a ser atômicos. É utilizado tanto pelos participantes quanto pelo componente `CAACtion`.

- **ExceptionHandling** é o componente responsável pela definição, captura, resolução e tratamento das exceções concorrentes ou não. É através dele que o componente `CAAction` conhece o estado de erro dos participantes. É este componente que após o processo de resolução lança as exceções a serem tratadas pelos participantes. Não é acessado pelos participantes diretamente.

- **Checking** é o componente responsável pelo comportamento de verificação do estado da CA Action como um todo e informa aos participantes se devem ou não continuar seu processamento individual.

A Figura 3.3 nos permite visualizar a atuação do *framework* JACAAF dentro do servidor de aplicações Java EE, além de mostrar sua relação com componentes internos e com a arquitetura de uma aplicação típica da plataforma, com suas camadas e visão cliente-servidor.

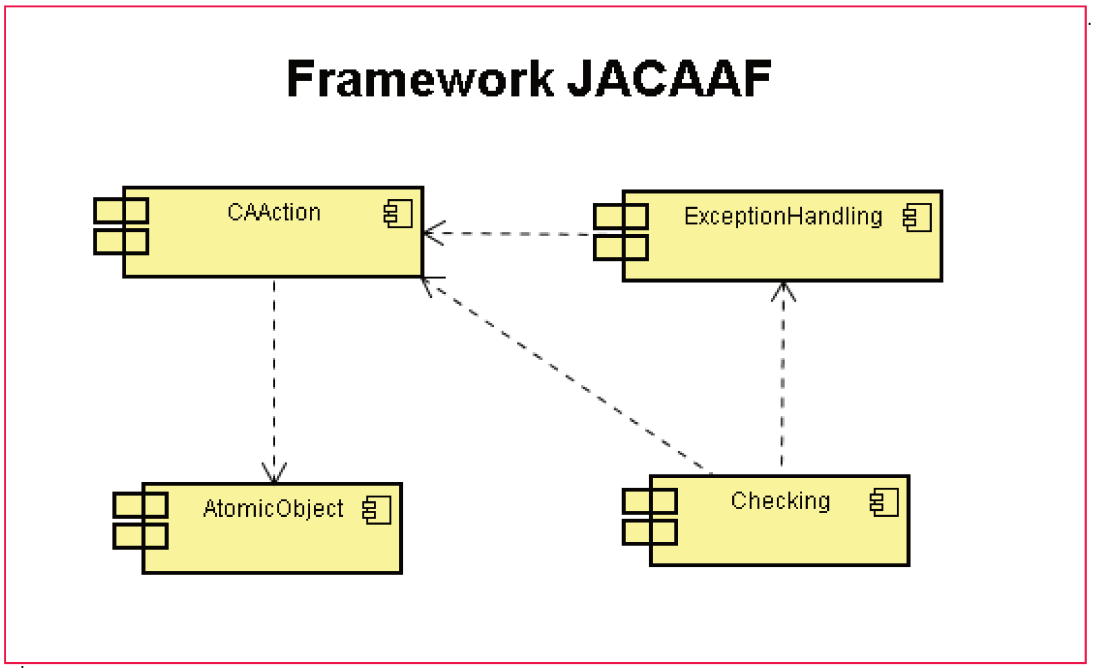

Figura 3.2: Arquitetura de componentes do framework JACAAF em UML

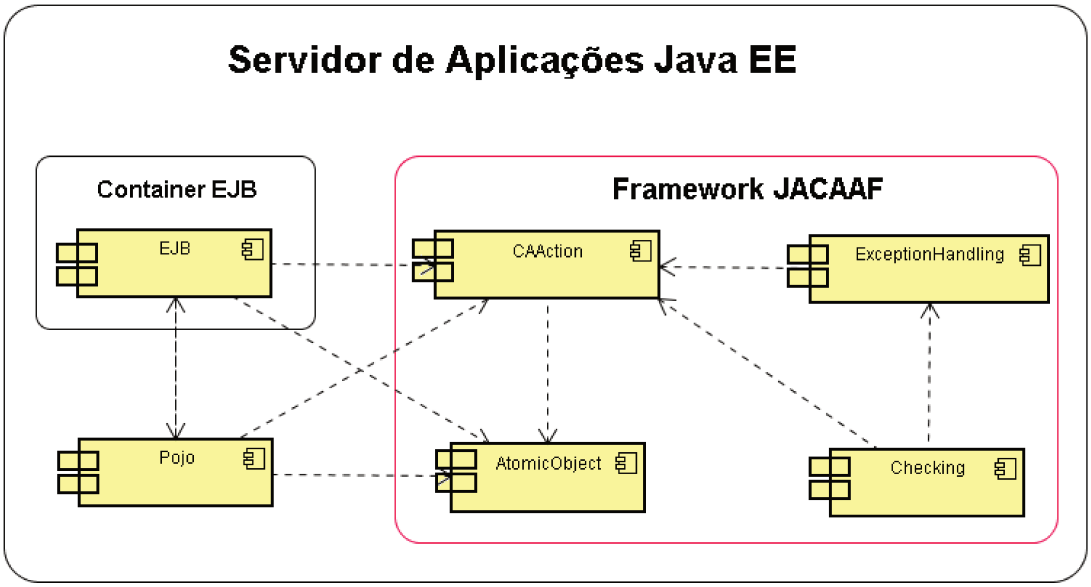

Figura 3.3: Arquitetura de componentes do framework JACAAF e o servidor de aplicações Java EE em UML

3.2 Projeto detalhado

Do ponto de vista do projeto detalhado e da visão de desenvolvimento da arquitetura de software, o *framework* JACAAF está dividido em 2 pacotes Java: `caaction` e `aspects`. O pacote `caaction` concentra as classes que monitoram a CA Action e no pacote `aspects` estão os aspectos, as anotações e as exceções do *framework* para seu acoplamento a uma aplicação específica implantada no servidor de aplicações Java EE.

3.2.1 O pacote caaction

Este pacote implementa os componentes arquiteturais `CAAction` e `AtomicObject`. É neste pacote que estão localizadas as classes que controlam a execução da CA Action e os objetos atômicos externos a ela. É importante lembrar que para o uso de objetos externos dentro do contexto de uma CA Action, eles devem proporcionar seus próprios mecanismos de atomicidade [59]. O *framework* JACAAF permite que objetos comuns possam ter essa característica de atomicidade de uma maneira bem simples. Para que um objeto passe a ser "atômico" deve estar marcado como `@Controlled` e ter seus métodos *setters* marcados como `@Transactional`. Após as marcações, é necessário dizer ao *framework* JACAAF quais mudanças de estado do objeto devem ser salvos. Para isso o *framework* JACAAF utiliza o padrão de projeto[2] `memento` utilizado para permitir que um objeto possa ser restaurado a um estado anterior [44].

Vejamos um exemplo de uso. Consideremos a classe `SimpleObject` já adaptada para o uso do *framework* JACAAF conforme o código a seguir:

[2]Do inglês: design pattern.

```
1  @Controlled // Principal anotação do framework que possibilita
2              // atomicidade
3  public class SimpleObject {
4    protected String x = "";
5    protected long y = 0;
6
7    public SimpleObject() {}
8
9    public SimpleObject(String x, long y) {
10     this.x = x;
11     this.y = y;
12   }
13
14   /**
15     * Return the X coordinate
16     */
17   public String getX(){
18     return x;
19   }
20
21   /**
22     * Set the X coordinate
23     */
24   @Transactional // Anotação que diz ao framework quais métodos
25                  // devem ser considerados na alteração do
26                  // estado de um objeto atômico
27   public void setX(String newX) {
28     x = newX;
29   }
30
31   public void setY(long y) {
32     this.y = y;
33   }
34
35   public long getY() {
36     return y;
37   }
38 }
```

No código acima, a linha 1 mostra a marcação `@Controlled` usada para permitir a atomicidade. As linhas 3, 4 e 5 declaram a classe `SimpleObject` e seus atributos `x` e `y`. Os construtores da classe são declarados nas linhas 7 a 12. Das linhas 17 a 37 são declarados os `getters` e `setters` dos atributos. Na linha 24 encontramos a marcação `@Transactional` que permite ao aspecto de atomicidade monitorar as alterações no atributo `x`.

Consideremos o seguinte trecho de código para tratar um objeto da classe

`SimpleObject`:

```
1   // Cria uma referência para salvar os objetos "'atômicos"
2   // para aquele participante
3   RoleManager.getInstance().begin();
4
5   SimpleObject obj = new SimpleObject();
6   RoleMemento memento = new RoleMemento();
7   // Coloca o memento como um listener nas alterações do objeto "obj"
8   obj.addPropertyChangeListener(memento);
9   obj.setX("2!");
10
11  // Salva no repositório o estado atual x="2!"
12  RoleManager.getInstance().save();
13
14  // Atualiza o objeto com x="1!"
15  obj.setX("1!");
16
17  // Faz rollback no repositório e retorna o objeto obj
18  // para o estado anterior x="2!"
19  RoleManager.getInstance().rollback();
20  memento.restoreState();
```

A linha 3 mostra a obtenção da instância da classe `RoleManager` e de um identificador único para o participante. A linha 5 mostra a criação de uma instância da classe `SimpleObject`. A linha 6 apresenta a obtenção de uma instância da classe `RoleMemento` que representa o repositório de objetos atômicos de um participante. Na linha 8 fazemos a associação do repositório com o objeto atômico a ser observado. Na linha 9 alteramos o estado do objeto `obj` inserindo a *string* ``2!'' em seu atributo `x` e, para salvar este estado, vemos na linha 12 a chamada ao método `save()`. Na linha 15 alteramos novamente o estado do objeto `obj`, desta vez inserindo a *string* ``1!'' em seu atributo `x`, mas desta vez o estado deve ser retornado à situação anterior, o que é mostrado nas linhas 19 e 20. Um objeto, portanto, pode ter alguns de seus estados salvos para que posteriormente possam ser restaurados conforme necessário.

A Figura 3.4 mostra o diagrama de classes do pacote `caaction`. As duas principais classes que coordenam a CA Action e seus participantes são `CAActionMonitor` e `RoleManager`. A classe `CAActionMonitor` é a responsável pela CA Action como um todo e possui uma referência interna aos repositórios de estados salvos de todos os participantes da ação, além da referência para o status de todos os participantes, se estão com erro ou não. A classe `RoleManager`, por sua vez, é responsável pelo estado cada participante e por manter um repositório (`memento`) dos objetos externos utilizados. A classe `RoleManager` é também responsável pela comunicação com a classe `CAActionMonitor` para informar o estado de um participante específico e atualizar repositório de objetos externos deste

participante.

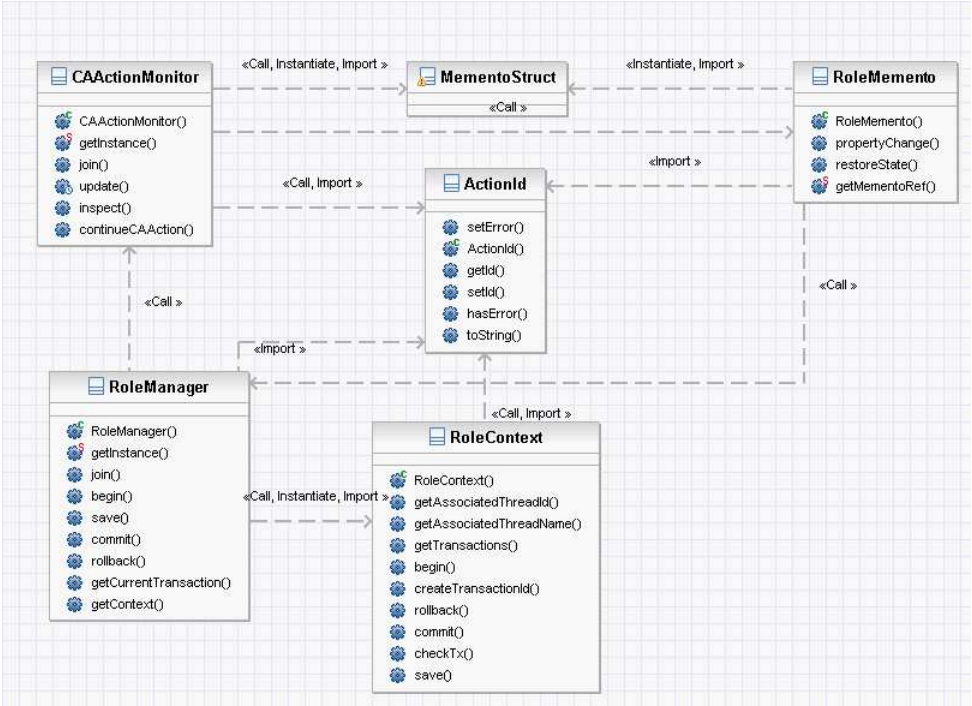

Figura 3.4: O pacote caaction em UML

3.2.2 O pacote aspects

O pacote `aspects` é composto por classes Java e pelos aspectos propriamente ditos e implementa os componentes arquiteturais *ExceptionHandling* e *Checking*. É neste pacote que estão definidas as anotações[3] `@Controlled`, `@Transactional` e `@Check` utilizadas para estender a aplicação alvo com as funcionalidades necessárias para o uso do *framework* JACAAF. A Figura 3.5 mostra o diagrama de classes deste pacote e sua relação com o pacote `caaction`.

A anotação `@Controlled` permite que um objeto possa ser observado para que as alterações causadas por métodos marcados com a anotação `@Transactional` sejam salvas num repositório em memória para futura utilização especialmente no caso de *rollback*.

A anotação `@Check` marca os métodos que antes de serem executados verificam na classe `CAActionMonitor` se a ação coordenada ainda é válida, isto é, se nenhum participante lançou alguma exceção. Caso algum participante tenha lançado uma exceção, o aspecto correspondente a esta verificação lança uma exceção do tipo `RollBackException`.

[3]Do inglês: annotations.

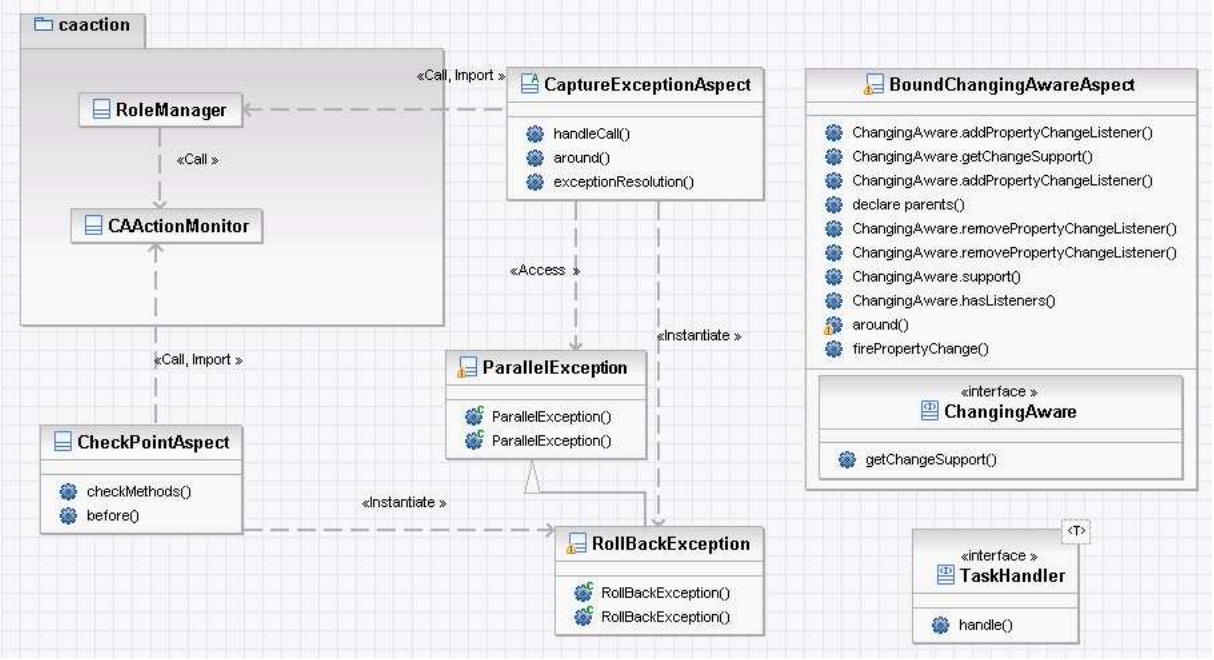

Figura 3.5: As classes do pacote aspects e sua relação com as principais classes do pacote caaction em UML

No pacote `aspects` também é definida a interface `TaskHandler` que permite o uso do aspecto *CaptureExceptionAspect* que trata de exceções e que será descrito adiante juntamente com os demais aspectos.

No pacote `aspects` existem 3 aspectos:

- `BoundChangingAwareAspect`: define o comportamento atômico dos objetos marcados como @Controlled. Monitora as alterações de estado realizadas a partir dos métodos *setters* marcados como `@Transactional` e salva esses estados de maneira a permitir sua recuperação posterior.

- `CheckPointAspect`: antes da execução dos métodos marcados como `@Check` verifica na classe `CAActionMonitor` se a ação coordenada ainda deve continuar seu processamento. Em caso negativo lança uma exceção do tipo `RollBackException`.

- `CaptureExceptionAspect`: Captura as exceções lançadas em execuções das classes marcadas como `@Controlled`.

As duas exceções do *framework* também estão definidas no pacote `aspects`. A exceção `ParallelException` representa a exceção raiz do *framework* da qual todas as demais devem ser derivadas. A exceção *RollBackException* é a única exceção realmente instanciada

nesta versão do *framework* e representa a impossibilidade de recuperação e que, portanto, seus tratadores em cada participante devem fazer o *rollback* das ações realizadas.

3.3 O funcionamento do *framework* JACAAF

Para exemplificar o uso e o funcionamento do *framework* JACAAF, vamos utilizar o estudo de caso 1 realizado neste trabalho sobre uma aplicação exemplo do JBoss, um servidor de aplicações Java EE.

A Figura 3.6 mostra a CA Action alvo deste processamento onde um conjunto de mensagens será processado pelos *message-driven beans* descritos na classe `TextMDB` com o intuito de realizar um processamento concorrente. Esse processamento será tratado como uma CA Action.

Neste exemplo, realizamos o mapeamento dos conceitos de CA Action mostrados na Figura 2.4 da seção 2.2.3 da seguinte forma:

- Os participantes desta CA Action são as instâncias da classe `TextMDB` aqui chamadas de `TextMDB1` e `TextMDB2`.
- A exceção lançada pelo participante `TextMDB1` é do tipo `RollBackException` que é também a exceção resultante do processo de resolução de exceção, pois neste exemplo nosso grafo de resolução se resume a um único tipo de exceção.
- Os tratadores acionados são, portanto, os tratadores da exceção do tipo `RollBackException`.

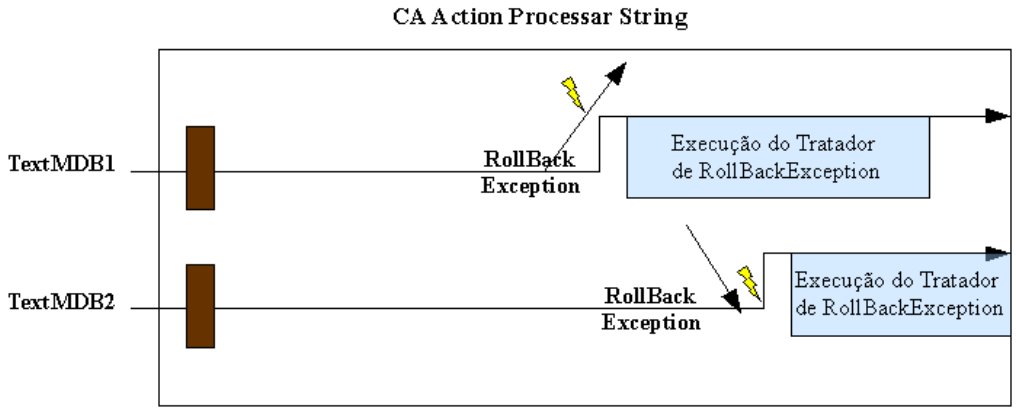

Figura 3.6: CA Action Processar String

3.3.1 Cenário de sucesso

A Figura 3.7 descreve de forma simplificada as chamadas realizadas durante a execução da aplicação utilizando o *framework* JACAAF, mostrando o que acontece quando uma mensagem é entregue a uma instância do *message-driven bean* `TextMDB`. A primeira atividade que ocorre quando uma mensagem é recebida é a execução do método `onMessage()`. Pode-se verificar pelo diagrama que o objeto `TextMDB` possui a marcação `@Controlled` e, portanto pode ter métodos que verifiquem se a CA Action deve continuar. O método `handle()` da classe `TextMDB` será observado pelo aspecto `CaptureExceptionAspect` para capturar exceções.

Após a recepção da mensagem o *message-driven bean* faz uma chamada ao método `handle()`, porém, como este método está marcado como `@Check`, isso faz com que o aspecto `CheckPointAspect` seja acionado.

```
1 \\ Este pointcut intercepta todas as chamadas de
2 \\ métodos @Check nas classes @Controlled.
3 pointcut checkMethods(): call(@Check * *(..)) && within(@Controlled *);
```

Este trecho de código do aspecto `CheckPointAspect` nos mostra na linha 3 o código responsável por definir quais métodos serão interceptados: todas as chamadas de métodos marcados como `@Check` que estejam definidos em uma classe marcada como `@Controlled`.

O aspecto `CheckPointAspect` faz uma chamada à instância da classe `CAActionMonitor` para verificar se a CA Action deve continuar. Caso a resposta seja positiva, o método `handle()` da classe `TextMDB` é então realmente executado.

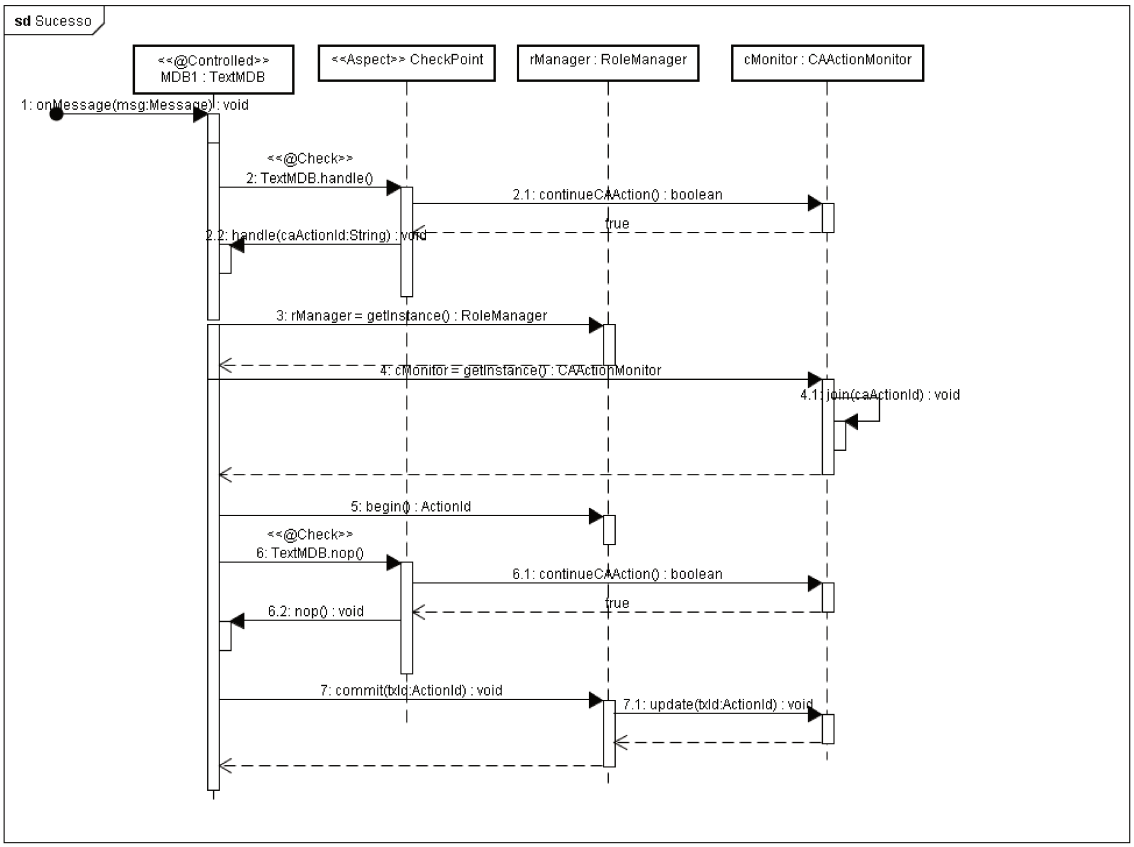

Figura 3.7: Diagrama de sequência representando um cenário de sucesso em UML

O método `handle()` executa faz seu registro como participante da CA Action. Em seguida executa o método `nop()`, que por estar marcado como `@Check`, aciona o aspecto `CheckPointAspect`. A verificação no objeto da classe `CAActionMonitor` é feita novamente e, por fim, o processamento é finalizado, atualizando o status do participante na classe `RoleManager` que por sua vez atualiza o status no objeto da classe `CAActionMonitor`.

3.3.2 Cenário excepcional

O cenário que acabamos de descrever na seção 3.3.1 é o cenário de sucesso no qual nenhuma exceção ocorreu. Mas o que aconteceria se em uma das verificações fosse informado que um dos participantes lançou uma exceção ? O comportamento para este cenário implementado nesta versão do *framework* JACAAF foi o seguinte: caso um dos participantes tenha ficado marcado com erro na CA Action devido a uma exceção, a CA Action como um todo deve ser abortada. Este novo cenário está descrito na Figura 3.8.

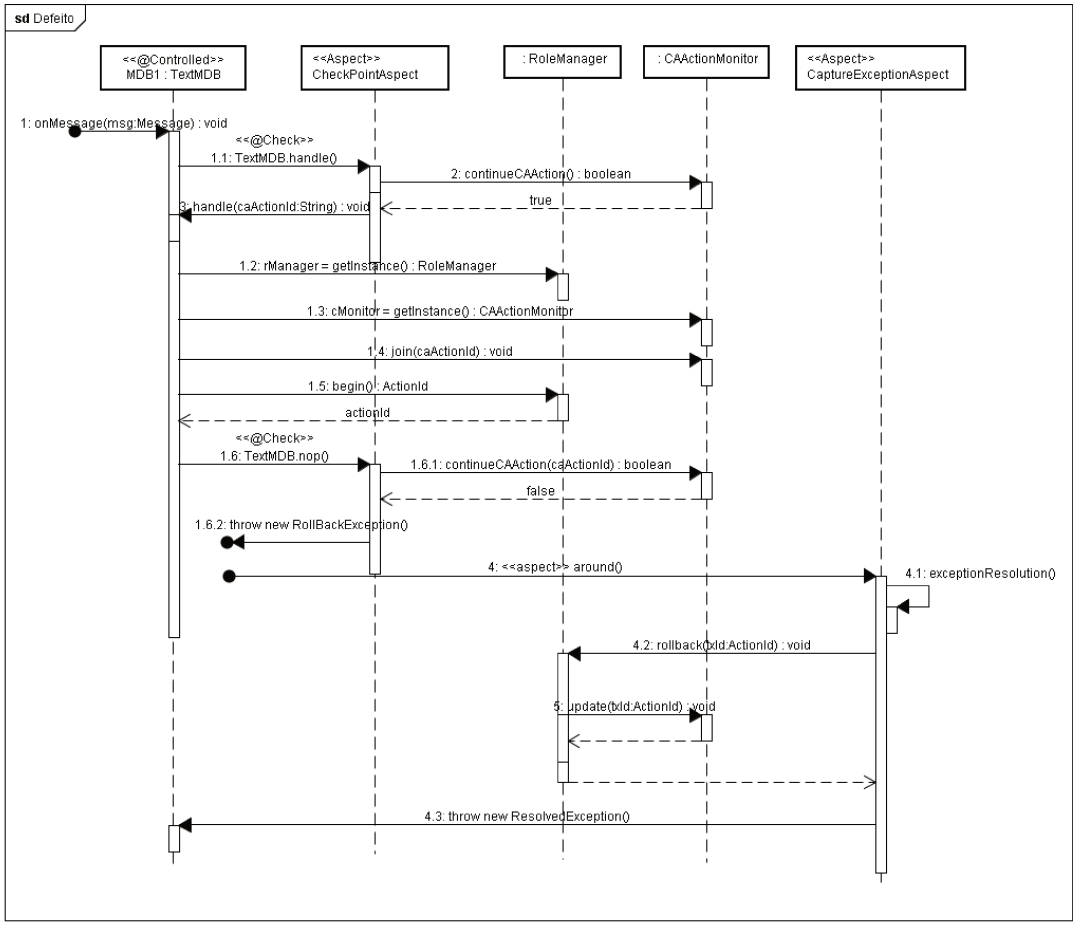

Figura 3.8: Diagrama de sequência representando um cenário excepcional em UML

O processamento se inicia com a recepção da mensagem e a execução do método `onMessage()` do *message-driven bean*. O método `handle()` é invocado e o aspecto de verificação é acionado. A verificação ocorre normalmente e o processamento prossegue com a obtenção das instâncias das classes `RoleManager` e `CAActionMonitor` e o registro como participante na CA Action através de uma chamada de método `join()` do objeto da classe `CAActionMonitor`. Na chamada do método `nop()` ocorre novamente o acionamento do aspecto `CheckPointAspect`. Até este ponto o processamento foi idêntico ao cenário de sucesso descrito na seção anterior (3.3.1). No entanto, uma resposta negativa é obtida ao realizar a verificação no objeto da classe `CAActionMonitor`, o que significa que uma exceção ocorreu em algum dos participantes. O aspecto `CheckPointAspect`, que realiza a verificação, lança uma exceção para o objeto da classe `TextMDB`, mas essa exceção é interceptada pelo aspecto `CaptureExceptionAspect` que inicia o procedimento de resolução de exceções implementado conforme descrito no início desta seção. A exceção resolvida é

então lançada para o objeto da classe `TextMDB` para que o tratador correspondente seja acionado.

3.4 O *framework* JACAAF e sua execução no servidor de aplicações Java EE

Para utilizar o *framework* JACAAF são necessários os seguintes passos:

- incluir as cláusulas *import* referentes às classes que definem as anotações descritas na seção 3.2.2 de acordo com a necessidade. Por exemplo, para que se possa usar o aspecto `CheckPointAspect` em uma classe, é necessário marcar esta classe como `@Controlled` e marcar um de seus métodos como `@Check`. Para tanto é obrigatório fazer o *import* das classes `Check` e `Controlled` que possuem as definições dessas anotações.

- adicionar as marcações às classes e métodos em questão.

- para utilizar o aspecto `CaptureExceptionAspect` é necessário que a classe marcada como `@Controlled` implemente a interface `TaskHandler` através da implementação do método `handle()`. Dentro do código deste método é necessário fazer as invocações dos método que inicializam a CA Action na classe `CAActionMonitor`. Vejamos um código de exemplo:

```
1  /*
2   * Método da interface TaskHandler, é, na verdade, o conteúdo
3   * do método onMessage do MDB.
4   * @see org.jacaaf.aspects.TaskHandler#handle(java.lang.Object)
5   */
6  @Check
7  public void handle(String caActionId) throws Throwable {
8    // Obtem instância do RoleManager
9    RoleManager rManager = RoleManager.getInstance();
10
11   // Cria um ID para o participante da CA Action
12   ActionId tId = rManager.begin();
13
14   // Cria uma referência para a CA Action no CAActionMonitor
15   CAActionMonitor.getInstance().join(caActionId);
```

  A linha 6 possui a marcação necessária para o monitoramento das exceções. A linha 9 obtém uma instância da classe `RoleManager` e a linha 12 obtém um identificador único para o participante. A linha 15 informa à classe `CAActionMonitor` que este participante passará a fazer parte de uma determinada CA Action.

- fazer o *weaving* para que os aspectos possam ser combinados com o código original da aplicação gerando uma nova aplicação.
- para que o servidor de aplicações Java EE possa executar os aspectos que agora fazem parte da nova aplicação, é necessário adicionar às bibliotecas do servidor de aplicações as classes de *runtime* da linguagem AspectJ.

3.5 Resumo

Este capítulo mostrou através da apresentação de diagramas de classe e de sequência a principal contribuição deste trabalho, o *framework* JACAAF. Foram descritas sua arquitetura, composição, interação entre seus componentes e como ele interage com a aplicação que o utiliza, além de exibir e explicar pedaços do código fonte do *framework*. Também foram descritos os passos necessárias para que uma aplicação possa utilizá-lo.

Capítulo 4

Estudos de caso

O objetivo dos estudos de caso realizados é demonstrar a aplicabilidade do *framework* JACAAF em uma aplicação real. Porém para os primeiros testes do *framework*, seu aprimoramento e correção de defeitos, utilizamos o primeiro estudo de caso como uma prova de conceito por se tratar de uma aplicação bem simples na qual pode-se controlar o ambiente de execução com mais segurança. Foram realizados 2 estudos de casos:

1. Estudo de caso 1: Aplicação JBoss baseada em *message-driven beans*.
2. Estudo de caso 2: Sistema de faturamento de energia elétrica.

Para ambos os estudos de caso, foi utilizada a infraestrutura disponibilizada pelo Centro de Pesquisa CPqD [11], uma instituição independente, focada na inovação com base nas tecnologias da informação e comunicação (TICs), que desenvolve um amplo programa de pesquisa e desenvolvimento, o maior da América Latina em sua área de atuação, gerando soluções em TICs que são utilizadas em diversos setores: telecomunicações, financeiro, energia elétrica, industrial, corporativo e administração pública.

A máquina utilizada para execução dos estudos de caso foi uma Workstation Dell Power Edge 6650 32 bits, com 4 CPUs Intel Xeon 3Ghz, 8 *cores* e 16Gb de memória RAM executando o sistema operacional Red Hat Enterprise Linux ES release 4.

4.1 Estudo de caso 1: Aplicação JBoss baseada em *message-driven beans*

Neste estudo de caso, o objetivo é realizar uma prova de conceito e validar o funcionamento do *framework* JACAAF nos componentes *enterprise java beans* (EJB) da plataforma *Java Enterprise Edition*. Para tanto foi selecionado o componente do tipo *message-driven bean* (MDB) devido à sua funcionalidade de trabalhar assincronamente e de forma concorrente,

bastante utilizada em sistemas de missão crítica com grandes volumes de processamento em lote.

4.1.1 Descrição da aplicação JBoss baseada em *message-driven beans*

Utilizamos como base para este estudo de caso o exemplo número 2 para *message-driven beans* do site *JBoss Guide* [35]. Para permitir um maior controle sobre a aplicação, foi realizada uma simplificação do exemplo originalmente proposto de forma a facilitar o acompanhamento e variação dos cenários de teste. Após a simplificação no código original, a aplicação passa a funcionar da seguinte maneira: um número configurável de mensagens é enviada para o serviço de mensagens do servidor de aplicações por um cliente externo ao servidor. A função do *message-driven bean* é receber a mensagem e consumi-la. O conteúdo dessa mensagem é uma simples *string*. Depois de ler a *string* o *message-driven bean* executa um *loop* sem nenhuma "operação" específica a não ser consumir CPU por alguns segundos. O tempo de duração deste *loop* varia para cada instância do *message-driven bean*, fazendo com que as mesmas mensagens sejam processadas em diferentes ordens a cada execução, gerando diferentes cenários para observação do comportamento do *framework*. O código abaixo mostra a implementação desse *message-driven bean*:

```
1  public void onMessage(Message msg)
2  {
3        log.info("TextMDB.onMessage, this="+hashCode());
4          String myString = null;
5
6        try {
7              TextMessage tm = (TextMessage) msg;
8              myString = tm.getText();
9          } catch(Throwable t) {
10             log.error("onMessage error reading message", t);
11         }
12
13 // Faz o MDB aguardar alguns segundos para simular
14 // "concorrência" aleatória, pois o tempo depende do hashcode
15         log.info(myStringHashCode + " Sleep a little... " + myString);
16         for (int m = 0; m < myHashCode; m++);
17
18         // chama um método sem operação
19         this.nop();
20
21 }
22
23 private void nop()
```

```
24 {
25         // Faz o MDB aguardar mais alguns segundos para proporcionar
26         // uma maior "concorrência"
27         for (int m = 0; m < myHashCode; m++);
28 }
```

As linhas 7 e 8 fazem o recebimento da mensagem e a transformação de seu conteúdo em uma *string* que é armazenada na variável `myString`.

A linha 16 mostra um *loop* vazio, cujo objetivo é consumir CPU por alguns segundos antes de prosseguir seu processamento.

A linha 19 faz a chamada ao método `nop()` que faz novamente o mesmo *loop* "vazio". A declaração do método e seu conteúdo está descrito nas linhas 23 a 28.

Entendo o conjunto de mensagens a ser processado como um processamento cooperativo, criamos uma CA Action para controlar esse processamento. Para isso incluímos as seguintes alterações no funcionamento do *message-driven bean* descrito anteriormente:

- a *string* que era simplesmente lida, agora é utilizada para dar nome a CA Action a ser criada.
- após aguardar determinado tempo no *loop* "vazio", o *message-driven bean* precisa avaliar se durante o processamento das demais mensagens não ocorreu nenhuma exceção, para somente então poder concluir seu processamento.
- caso haja alguma exceção lançada por alguma das demais mensagens, o processamento deste *message-driven bean* é finalizado com erro.

Para implementar esse comportamento utilizando o *framework* JACAAF, as seguintes alterações foram feitas no código do *message-driven bean*:

```
1  import org.jacaaf.aspects.Check;
2  import org.jacaaf.aspects.Controlled;
3  import org.jacaaf.aspects.RollBackException;
4  import org.jacaaf.aspects.TaskHandler;
5  import org.jacaaf.caaction.ActionId;
6  import org.jacaaf.caaction.RoleManager;
7  import org.jacaaf.caaction.RoleMemento;
8  import org.jacaaf.caaction.CAActionMonitor;
9
10 /**
11  * An MDB that reads the TextMessages it receives
12  * @author  Scott.Stark@jboss.org
13  * @version $Revision: 1.2 $
14  */
15
16 @Controlled
```

```
17 public class TextMDB implements MessageDrivenBean,
18                 MessageListener, TaskHandler<String>
19
20 public void onMessage(Message msg)
21 {
22         //log.info("TextMDB.onMessage, this="+hashCode());
23   String caActionId = null;
24
25   // chamada do método handle() onde agora está a lógica de negócio
26   try {
27          TextMessage tm = (TextMessage) msg;
28          caActionId = tm.getText();
29       } catch(Throwable t) {
30           log.error("onMessage error reading message", t);
31       }
32
33   try {
34         this.handle(caActionId);
35       } catch(Throwable th) {
36           log.error("Exception at onMessage method TextMDB", th);
37           log.info("Essa exception deve ser pega pelo aspecto
38                                         CaptureExceptionAspect!");
39       }
40 }
41
42 /*
43  * Método da interface TaskHandler, é, na verdade, o conteúdo
44  * do onMessage, ou seja, deve possuir a lógica de negócio.
45  *
46  * @see org.jacaaf.aspects.TaskHandler#handle(java.lang.Object)
47 */
48 @Check
49 public void handle(String caActionId) throws Throwable
50 {
51   RoleManager rManager = RoleManager.getInstance();
52   CAActionMonitor.getInstance().join(caActionId);
53   RoleMemento memento = new RoleMemento();
54
55   ActionId actionId = rManager.begin();
56   System.out.println(myStringHashCode + " MyTransactionId: "
57                                                 + actionId);
58
59   // Faz o MDB aguardar alguns segundos para simular
60   // "concorrência" aleatória, pois o tempo depende do hashcode
61   log.info(myStringHashCode + " Sleep a little... " + myString);
62   for (int m = 0; m < myHashCode; m++);
```

```
63
64    // chama um método sem operação
65    this.nop();
66
67    rManager.commit(actionId);
68  }
69
70  @Check
71  private void nop() {
72  ...
```

- Marcação do *message-driven bean* com `@Controlled` (linha 16) e inclusão das cláusulas *import* necessárias (linhas 1 a 8).
- Implementação da interface `TaskHandler` (linha 18), a criação do método `handle()` (linhas 42 a 68) e a migração do código do método `onMessage()` para o método `handle() (linhas 25 a 40)`.
- Inclusão do código de controle da CA Action (linhas 51 a 55).
- Marcação dos métodos `nop()` (linha 70) e `handle()` (linha 48) como `@Check` para que passem a verificar se a CA Action está ativa antes de executarem.

Para aumentar a aleatoriedade do exemplo e poder avaliar o comportamento do sistema em diferentes situações, o trecho de código abaixo foi inserido para que em alguns casos o processamento das mensagens lançassem exceções simulando erros:

```
1  // para hashCodes pares, marca action para rollback.
2  if (myHashCode % 2 == 0) {
3     System.out.println(myStringHashCode + " ROLLBACK!");
4     rManager.rollback(actionId);
5  // Lança uma exceção para os hashCodes() pares.
6     throw new RollBackException();
7  }
```

A linha 2 mostra a ideia usada para a simulação de erro: os *message-driven beans* que possuem `hashcodes` pares devem lançar uma exceção. Como os `hashcodes` variam de instância para instância, isso gera uma aleatoriedade interessante para se observar o comportamento da aplicação como um todo. A linha 4 mostra o *message-driven bean* informando à classe `RoleManager` que fará *rollback* pois "encontrou" um erro. A linha 6 lança a exceção propriamente dita.

O código final do *message-driven bean* `TextMDB` para os cenários de sucesso e excepcional é praticamente o mesmo, exceto pelo fato de que, no cenário de sucesso, as linhas 144 a 151 são comentadas para que não façam o lançamento de exceções. O código completo é descrito a seguir:

```
1  package org.jboss.chap6.ex2;
2
3  import javax.ejb.MessageDrivenBean;
4  import javax.ejb.MessageDrivenContext;
5  import javax.ejb.EJBException;
6  import javax.jms.JMSException;
7  import javax.jms.Message;
8  import javax.jms.MessageConsumer;
9  import javax.jms.MessageListener;
10 import javax.jms.Queue;
11 import javax.jms.QueueConnection;
12 import javax.jms.QueueConnectionFactory;
13 import javax.jms.QueueReceiver;
14 import javax.jms.QueueSender;
15 import javax.jms.QueueSession;
16 import javax.jms.TextMessage;
17 import javax.naming.InitialContext;
18 import javax.naming.NamingException;
19 import org.apache.log4j.Logger;
20 import org.jacaaf.aspects.Check;
21 import org.jacaaf.aspects.Controlled;
22 import org.jacaaf.aspects.RollBackException;
23 import org.jacaaf.aspects.TaskHandler;
24 import org.jacaaf.caaction.ActionId;
25 import org.jacaaf.caaction.RoleManager;
26 import org.jacaaf.caaction.RoleMemento;
27 import org.jacaaf.caaction.CAActionMonitor;
28
29 /**
30  * An MDB that transforms the TextMessages it receives and send the
31  * transformed messages to the Queue found in the incoming message
32  * JMSReplyTo header.
33  *
34  * @author  Scott.Stark@jboss.org
35  * @version $Revision: 1.2 $
36  */
37
38 @Controlled
39 public class TextMDB
40     implements MessageDrivenBean,
41                MessageListener, TaskHandler<String>
42 {
43     private static Logger log = Logger.getLogger(TextMDB.class);
44
45     private MessageDrivenContext ctx = null;
46     private QueueConnection conn;
```

```
47      private QueueSession receivingSession, sendingSession;
48      private Queue commitQueue;
49      private QueueReceiver queueRecv = null;
50      private int myHashCode;
51          private String myStringHashCode;
52
53      public TextMDB()
54      {
55          log.info("TextMDB.constructor, this=" + hashCode());
56          log.debug("constructor.StackTrace",new Throwable("constructor"));
57      }
58
59      public void setMessageDrivenContext(MessageDrivenContext ctx)
60      {
61          this.ctx = ctx;
62          log.info("TextMDB.setMessageDrivenContext, this=" + hashCode());
63      }
64
65      public void ejbCreate()
66      {
67          this.myHashCode = hashCode();
68          this.myStringHashCode = "["+hashCode()+"]";
69          log.info("TextMDB.ejbCreate, this="+this.myStringHashCode);
70          try {
71              setupPTP();
72          } catch(Exception e) {
73              log.error("Failed to init TextMDB", e);
74              throw new EJBException("Failed to init TextMDB", e);
75          }
76      }
77
78      public void ejbRemove()
79      {
80          log.info("TextMDB.ejbRemove, this=" + hashCode());
81          ctx = null;
82          try {
83              if (receivingSession != null) {
84                  receivingSession.close();
85              }
86              if (sendingSession != null) {
87                  sendingSession.close();
88              }
89              if (conn != null) {
90                  conn.close();
91              }
92          } catch(JMSException e) {
```

```
93              log.error("ejbRemove error", e);
94          }
95      }
96
97      public void onMessage(Message msg)
98      {
99        //log.info("TextMDB.onMessage, this="+hashCode());
100         String caActionId = null;
101
102         try {
103             TextMessage tm = (TextMessage) msg;
104             caActionId = tm.getText();
105         } catch(Throwable t) {
106             log.error("onMessage error reading message", t);
107                 throw t;
108         }
109
110         try {
111                 this.handle(caActionId);
112             } catch(Throwable th) {
113             log.error("Exception Occurred at onMessage() TextMDB", th);
114             log.info("But it's all right!");
115             }
116     }
117
118     /*
119      * Método da interface TaskHandler, é, na verdade, o conteúdo
120      * do método onMessage().
121      * (non-Javadoc)
122      *
123      * @see org.jacaaf.aspects.TaskHandler#handle(java.lang.Object)
124      */
125     @Check
126     public void handle(String caActionId) throws Throwable {
127
128         RoleManager rManager = RoleManager.getInstance();
129         CAActionMonitor.getInstance().join(caActionId);
130         RoleMemento memento = new RoleMemento();
131
132         ActionId tId = rManager.begin();
133         System.out.println(myStringHashCode + " MyTransactionId: " + tId);
134
135         // Faz o MDB aguardar alguns segundos para simular
136         // "concorrência" aleatória, pois o tempo depende do hashcode
137         log.info(myStringHashCode + " Sleep a little... " + caActionId);
138         for (int m = 0; m < myHashCode; m++);
```

```
139
140         // chama um método sem operação específica, somente aguarda
141         this.nop();
142
143         // para hashCodes pares, marca a action para rollback.
144         if (myHashCode % 2 == 0) {
145                 System.out.println(myStringHashCode + " ROLLBACK!");
146                 rManager.rollback(tId);
147
148                 // Lança uma exceção para os hashCodes() pares.
149                 throw new RollBackException();
150                         }
151         }
152
153         rManager.commit(tId);
154 }
155
156     @Check
157     private void nop() {
158         // Faz o MDB aguardar mais alguns segundos para proporcionar
159         // uma maior "concorrência"
160         for (int m = 0; m < myHashCode; m++);
161     }
162
163     private void setupPTP() throws JMSException, NamingException
164     {
165         InitialContext iniCtx = new InitialContext();
166         Object tmp = iniCtx.lookup("java:comp/env/jms/QCF");
167         QueueConnectionFactory qcf = (QueueConnectionFactory) tmp;
168         conn = qcf.createQueueConnection();
169         receivingSession = conn.createQueueSession(false,
170                                                    QueueSession.AUTO_ACKNOWLEDGE);
171         sendingSession = conn.createQueueSession(false,
172                                         QueueSession.AUTO_ACKNOWLEDGE);
173
174         // cria a referência para a queue de commit
175         commitQueue = (Queue) ctx.lookup("queue/A");
176
177         queueRecv = receivingSession.createReceiver(commitQueue);
178         conn.start();
179     }
180 }
```

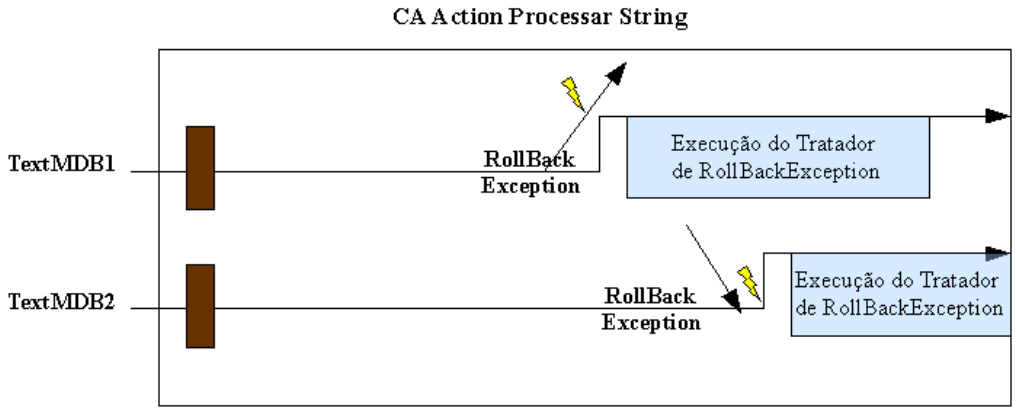

Figura 4.1: TextMDB e CA Action

4.1.2 Descrição da CA Action Processar String

A Figura 4.1 mostra um cenário esquematizado da CA Action Processar String.

Neste cenário, realizamos o mapeamento dos conceitos de CA Action mostrados na Figura 2.4 da seção 2.2.3 da seguinte forma:

- Os participantes desta CA Action são 2 instâncias da classe `TextMDB`, aqui chamadas de `TextMDB1` e `TextMDB2`.
- A exceção lançada pelo participante `TextMDB1` é do tipo `RollBackException` que é também a exceção resultante do processo de resolução de exceção, pois neste exemplo nosso grafo de resolução se resume a um único tipo de exceção.
- Os tratadores acionados são, portanto, os tratadores da exceção do tipo `RollBackException` de cada participante.

Neste cenário o participante `TextMDB1` realiza seu processamento normal e através dos aspectos implementados no *framework* JACAAF são realizadas consultas ao monitor da *CA Action* para verificar se o processamento deve prosseguir. O processamento segue normalmente até que o participante `TextMDB1` lança uma exceção do tipo `RollBackException`. Apesar de não ser um participante do processamento propriamente dito, o *framework* JACAAF captura a exceção lançada e informa o monitor de sua ocorrência. O aspecto `CaptureExceptionAspect`, responsável pelo monitoramento de exceções, inicia o processo de resolução e indica a exceção a ser tratada pelo participante. Nesta versão a resolução foi implementada de forma a simplesmente relançar a exceção

`RollBackException`. O participante `TextMDB1` inicia então o tratamento da exceção lançada. O participante `TextMDB2` ao continuar seu processamento, é informado através do *framework* que deve tratar a exceção `RollBackException` e, imediatamente inicia seu tratamento. Para os participantes `TextMDB1` e `TextMDB2`, a lógica implementada em seus tratadores da exceção `RollBackException` é simplesmente abortar o processamento.

Este cenário excepcional que acabamos de descrever e o cenário de sucesso foram descritos com mais detalhes na seção 3.3 que descreve o funcionamento do *framework* JACAAF.

4.1.3 Preparação do estudo de caso 1

Para a execução deste estudo de caso, foram necessários os seguintes passos:

- download, instalação e configuração da servidor de aplicações JBoss versão 4.0.5 na máquina disponibilizada
- download do pacote de exemplos do site JBoss Guide [35]
- download, instalação e configuração da ferramenta de compilação e montagem (*build*) ANT
- download, instalação e configuração da interface para desenvolvimento de software Eclipse 3.4.0
- download e instalação da extensão do Eclipse (*plug-in*) para utilização do AspectJ
- adequação dos scripts de compilação e montagem do exemplo número 2 dos *message-driven beans*
- compilação, montagem, implantação e execução do exemplo número 2
- simplificação do código conforme listagens descritas na seção 4.1.1
- compilação, montagem, implantação e execução da versão simplificada
- adequação do código para o uso do *framework* JACAAF
- compilação, montagem, implantação da versão alterada
- execução e avaliação dos logs gerados pela aplicação
- correção de defeitos encontrados no *framework* e na aplicação simplificada

- para cada correção realizada na aplicação exemplo era necessário que fossem repetidos os passos de compilação, montagem e implantação da nova versão
- para cada correção realizada no *framework* JACAAF era necessário que o *framework* fosse recompilado e que a aplicação exemplo fosse também recompilada e montada utilizando a nova versão do *framework* e, finalmente, que se repetisse a implantação da nova versão da aplicação

Existem 2 diferentes cenários básicos para este estudo de caso:

- Cenário de sucesso: todas as mensagens enviadas são consumidas e nenhuma exceção é levantada
- Cenário excepcional: algumas das mensagens enviadas não são consumidas corretamente e exceções são levantadas. As mensagens que ainda estão processamento e que não levantaram exceção têm seus tratadores da exceção do tipo `RollBackException` acionados e finalizam com erro.

4.1.4 Execução do estudo de caso 1

O principal objetivo deste estudo de caso era validar se o comportamento proposto para *framework* foi atingido. Esse comportamento consiste em:

- Criar o conceito de ação atômica coordenada correlacionando os participantes num único processamento lógico
- Capturar as exceções lançadas durante o processamento e informar ao monitor da CA Action sobre a ocorrência da exceção
- Disparar o comportamento de verificação junto ao monitor da CA Action na chamada dos métodos marcados como `@Check` e lançar uma exceção quando a CA Action já houver sido informada por outro participante de alguma exceção ocorrida

Como o número de mensagens criadas para serem consumidas é configurável, para avaliar o comportamento e concorrência com diferentes números de *message-driven beans* ao mesmo tempo, foram configurados cenários com 2, 3, 5 e 10 mensagens simultâneas.

4.1.5 Avaliação dos resultados obtidos

Todos os cenários executados foram verificados através dos arquivos de log gerados pela aplicação. As execuções foram repetidas várias vezes e o comportamento observado foi

sempre o esperado, ou seja, nos momentos em que pelo menos um participante sinalizava um erro à classe `CAActionMonitor`, todos os demais participantes passavam a finalizar seu processamento também com erro após o tratamento da exceção do tipo `RollBackException` lançada pelo processo de resolução de exceção. Já nos casos em que nenhum participante sinalizava erro, todas as mensagens eram processadas com sucesso até o fim. Pôde-se confirmar através do log da aplicação que na execução dos métodos marcados como `@Check` ocorreu a verificação do estado da CA Action conforme esperado. Nos cenários excepcionais, durante a verificação do estado da CA Action, pôde-se verificar também o lançamento da exceção do tipo `RollBackException` e também o acionamento do aspecto `CaptureExceptionAspect` responsável por capturar as exceções lançadas.

Desta forma, concluímos que os objetivos esperados neste estudo de caso quanto ao comportamento do *framework* foram atingidos.

4.2 Estudo de caso 2: Sistema de faturamento de energia elétrica

De posse dos resultados do primeiro estudo de caso confirmando o funcionamento do *framework*, foi iniciado este segundo estudo de caso. Neste estudo de caso o principal objetivo foi validar o funcionamento do *framework* JACAAF dentro de uma aplicação real e de missão crítica. Para tanto, foi utilizado o sistema de faturamento de energia elétrica desenvolvido pelo centro de pesquisa CPqD [11] para se verificar haveria alguma perda significativa relativa ao tempo de execução comparando-se as execuções do sistema em sua versão em produção com a execução da versão alterada para incluir os controles e mecanismos presentes no *framework* JACAAF.

4.2.1 Descrição do sistema de faturamento de energia elétrica

Este estudo de caso foi realizado utilizando-se uma das funcionalidades do sistema CPqD Energia Gestão Comercial [22], um sistema de gestão para automatizar os processos e operações de negócio de empresas concessionárias de energia elétrica, atualmente em produção há 2 anos na concessionária de energia elétrica do Estado de Goiás, a Celg [7]. É composto pelos módulos de Relacionamento e Atendimento a Clientes, Faturamento, Arrecadação, Cobrança, Contabilidade e Gestão de Equipamentos de Medição. É um sistema real, de missão crítica e com altos níveis de requisitos não-funcionais de performance, desempenho e disponibilidade. É, atualmente, o maior sistema em número de linhas de códigos já implantado pelo CPqD, com cerca de 8 milhões de linhas de código.

Para o estudo de caso, foi selecionada a funcionalidade de extração de informações para a secretaria da fazenda, conhecida como "Fisco", cujo objetivo é gerar arquivos textos

em formatos pré-definidos para a Secretaria da Fazenda do Estado de Goiás contendo informações sobre as faturas e notas fiscais geradas pelo módulo de Faturamento do CPqD Energia Gestão Comercial. Trata-se de uma prestação de contas mensal e obrigatória. Após a geração dos arquivos, estes devem ser submetidos a um processo de validação utilizando-se um programa validador disponibilizado pela Secretaria da Fazenda, chamado "e-nota fiscal" [62].

Como resultado de um processamento mensal são gerados volumes contendo 3 diferentes tipos de arquivos: i) Arquivo Mestre, com dados essenciais da nota fiscal, ii) Arquivo Item, com detalhamento dos serviços prestados em cada fatura e, iii) Arquivo Cadastral, com as informações cadastrais do destinatário do documento fiscal.

Durante o processamento são gerados tantos volumes quantos necessários, conforme a seguinte regra descrita nos casos de uso do sistema: "para empresas com quantidade superior a 1 (um) milhão de faturas/mês, os arquivos deverão ser divididos em volumes contendo 1 (um) milhão de faturas. Por exemplo, se determinada empresa emitir 4.513.000 faturas em determinado mês, deverá gerar os arquivos em 5 volumes, com os quatro primeiros contendo informações de 1 milhão de documentos fiscais e o último contendo as informações dos 513.000 documentos fiscais restantes". Essa divisão em volumes pode ser parametrizada no sistema.

O processamento pode apresentar, de maneira geral, três diferentes tipos de problemas, todos relacionados a infraestrutura: (i) defeito no servidor (falta de memória por exemplo), (ii) alguma indisponibilidade (mesmo que temporária) no banco de dados ou ainda (iii) falta de espaço em disco para escrita dos arquivos. Em todos esses casos o processamento deve ser abortado, pois devido à natureza desses problemas, pelo menos um arquivo estará corrompido, inviabilizando o envio dos volumes completos.

4.2.2 Breve histórico da funcionalidade de geração de arquivos

A funcionalidade de geração de arquivos para a secretaria da fazenda, Fisco, passou por evoluções desde a sua concepção. Embora o CPqD Energia Gestão Comercial não seja versionado por funcionalidade, para fins didáticos e em termos de relevância para este trabalho, podemos considerar a existência de três diferentes versões do sistema, sendo duas efetivamente utilizadas pelo cliente e, uma terceira versão que é o resultado deste trabalho.

A primeira versão

Em sua "primeira versão", quando uma das *threads* responsáveis pela geração dos arquivos lançava uma exceção, as demais *threads* participantes não eram informadas do ocorrido, continuando seu processamento desnecessariamente, uma vez que se pelo menos um ar-

quivo não for gerado, todos os volumes devem ser desconsiderados, e o processamento, reiniciado. Caso ocorresse uma exceção em uma das *threads* participantes logo no início do processamento, seria necessário aguardar até que todas as demais *threads* finalizassem, o que poderia levar mais de 1 hora.

A segunda versão

Das dificuldades descritas acima, percebeu-se a necessidade de um melhor gerenciamento sobre as *threads* participantes. Tratando-se de um processamento feito em lote, o caminho natural para se desenvolver esta solução dentro da plataforma *Java Enterprise Edition*, seria utilizar os componentes *Enterprise Java Beans* do tipo *message-driven beans*. Porém, estes componentes não fornecem um mecanismo padrão para o gerenciamento externo de seu ciclo de vida após o início de seu processamento, conforme descrito na seção 2.5.2.

Desta forma, uma solução *ad hoc* foi implementada utilizando-se mecanismos específicos da linguagem Java, mas que ficaram espalhados dentro do código que continha a lógica de negócio. Além disso, com esta solução *ad hoc* (em produção atualmente), caso ocorra alguma exceção que não possa ser tratada por algum dos participantes, a *thread* responsável pelo gerenciamento dos participantes envia uma mensagem para que todas as *threads* sejam finalizadas imediatamente, não provendo portanto, nenhum mecanismo para recuperação e nem mesmo a possibilidade de recuperação cooperativa do processamento.

A terceira versão

Para contornar os problemas da segunda versão acima descritos, foi implementada o que podemos considerar como a terceira versão da funcionalidade. Nesta versão, objeto deste estudo de caso, ao utilizarmos o *framework* JACAAF e suas facilidades, pôde-se criar condições para um gerenciamento sobre os participantes do processamento, uma vez que é possível lançar exceções específicas para que os participantes façam o tratamento e recuperação, além de possibilitar a recuperação de forma cooperativa e, também, permitir que estruturalmente o código de negócio esteja separado do código responsável pelo gerenciamento dos participantes. Embora não realizado no contexto deste trabalho, pudemos perceber que com o uso do *framework* JACAAF, esta funcionalidade pode ser reimplementada utilizando-se os componentes *message-driven bean*.

4.2.3 Descrição da CA Action Gerar Arquivos

Na Figura 4.2, temos o mapeamento do processamento do Fisco, considerando-se a terceira versão descrita em 4.2.2, em uma CA Action conforme descrito na Figura 2.4 da seção

2.2.3 da seguinte forma:

- Os participantes da CA Action Gerar Arquivos são em número de 4 e estão mapeados nas 4 instâncias da classe `FileGenerator` aqui chamadas de `FileGenerator1` a `FileGenerator4`.
- A exceção lançada pelo participante `FileGenerator1` é do tipo `RollBackException` que é também a exceção resultante do processo de resolução de exceção, pois neste cenário, o grafo de resolução de exceções se resume a um único tipo de exceção.
- Os tratadores acionados nos participantes são, portanto, os tratadores deste único tipo de exceção.

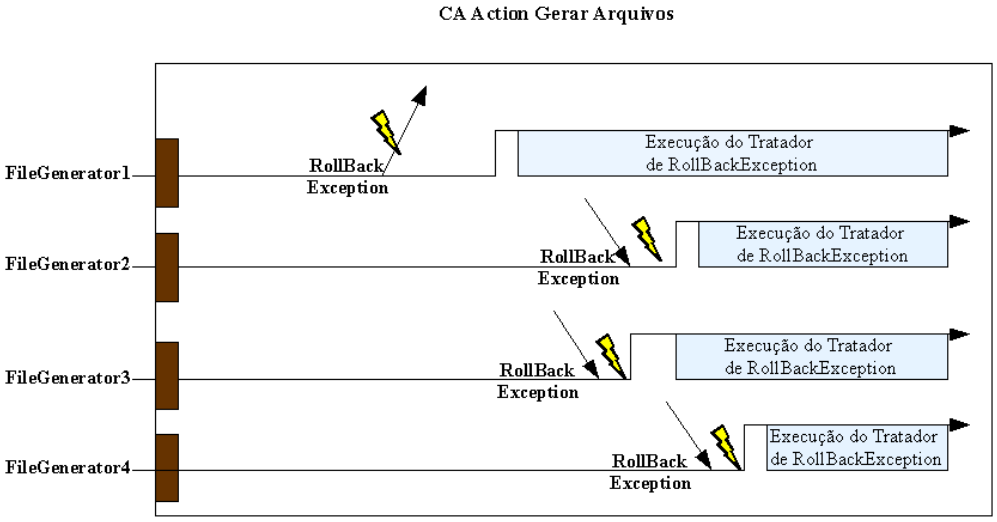

Figura 4.2: CA Action faturamento

Neste cenário, os participantes realizam seu processamento e através das interceptações do *framework* JACAAF são realizadas consultas ao monitor da *CA Action* (não demonstrada na figura pois não é um participante do processamento dos arquivos em si) para verificar se o processamento deve prosseguir. Quando uma resposta positiva é obtida, o processamento dos participantes continua normalmente sem nenhuma alteração. Num determinado momento de seu processamento, o participante `FileGenerator1` lança uma exceção do tipo `RollBackException` que é capturada pelo *framework* JACAAF (também não demonstrado na figura por não participar do processamento dos arquivos propriamente dito) e informada ao monitor da *CA Action*. O aspecto `CaptureExceptionAspect`,

responsável pelo monitoramento de exceções, inicia o processo de resolução e indica a exceção a ser tratada pelo participante (nesta versão do *framework* a resolução foi implementada de forma a simplesmente relançar a exceção do tipo `RollBackException`). O participante `FileGenerator1` inicia então o tratamento da exceção levantada. Os demais participantes `FileGenerator2`, `FileGenerator3` e `FileGenerator4` recebem a exceção do tipo `RollBackException`, obtida através do processo de resolução de exceções, e iniciam seu tratamento. Para todos os participantes, a lógica implementada em seus tratadores da exceção do tipo `RollBackException` é simplesmente finalizar o processamento com erro.

4.2.4 Preparação do estudo de caso 2

Para a execução deste estudo de caso foram necessários os seguintes passos:

- estudo de funcionalidades do sistema CPqD Energia em busca de uma situação favorável ao estudo de caso
- solicitação de disponibilização e atualização de ambiente de execução compatível com a versão atual do sistema CPqD Energia
- download do repositório dos componentes afetados pela funcionalidade escolhida
- download, instalação e configuração da ferramenta de compilação e montagem MAVEN utilizada nos ambientes de desenvolvimento do sistema CPqD Energia
- estudo dos scripts de compilação da aplicação
- configuração da interface para desenvolvimento de software Eclipse 3.4.0 para geração dos componentes do sistema CPqD Energia
- compilação e montagem da aplicação em sua versão atual
- estudo aprofundado da funcionalidade de geração de informações para a secretaria da fazenda para entender seu funcionamento detalhado e permitir que se pudessem ser feitas as alterações necessárias para o uso do *framework*
- alteração do código fonte da aplicação para utilizar o *framework* JACAAF
- evolução dos scripts de compilação e montagem da aplicação para utilizar o processo de combinação dos aspectos com a aplicação alvo
- compilação, montagem e implantação da versão alterada da aplicação

- execução da aplicação nas versões original e alterada
- análise comparativa do conteúdo dos arquivos gerados como resultado da execução da aplicação
- correção de defeitos encontrados no *framework*
- para cada correção realizada no *framework* JACAAF era necessário que o *framework* fosse recompilado e que a aplicação alvo fosse também recompilada e montada utilizando a nova versão do *framework* e, finalmente, que se repetisse a implantação da nova versão da aplicação

4.2.5 Execução do estudo de caso 2

A máquina disponibilizada pelo CPqD para o estudo de caso é de 32 bits e, devido a esta restrição, não é possível alocar memória suficiente para uma execução completa com todas as (mais de 1 milhão de) faturas. Portanto, para a execução do estudo de caso, o universo de faturas utilizadas como entrada de dados foi reduzido de forma a trabalhar com um total de 11.500 faturas.

Na arquitetura atualmente implementada, cada *thread* de processamento gera os 3 tipos de arquivos (volumes) referentes ao mesmo conjunto de dados conforme descrito na seção 4.2.1. De acordo com a divisão dos volumes descrita, são utilizadas tantas *threads* quantos volumes forem necessários para o processamento, permitindo um processamento paralelo. Vamos utilizar essas *threads* como participantes de uma CA Action. Parametrizamos o sistema para que tenha 4 *threads* paralelas (1 por CPU), cada um delas gerando volumes referentes a até 3.000 faturas.

Nas primeiras versões da funcionalidade estudada, surgiu a necessidade de se implementar o tratamento de exceções concorrentes numa forma *ad hoc*, pois muito processamento era desperdiçado quando uma de suas *threads* falhava e, portanto, os arquivos não seriam completamente gerados, mas o processamento das demais *threads* prosseguia até o final. O *framework* JACAAF implementa esse mesmo mecanismo mas de uma forma sistemática e de fácil extensão.

O código fonte do CPqD Energia é registrado e, por motivos legais, não pode ser divulgado neste trabalho, bem como os arquivos resultantes do processamento, pois são provenientes de dados reais de produção de clientes.

Para incluirmos o uso do *framework* JACAAF no sistema de faturamento de energia elétrica as seguintes alterações no código foram necessárias:

- Inclusão das cláusulas *import* necessárias para as marcações e para as interfaces a serem implementadas.

- Marcação como `@Controlled` nas classes participantes da CA Action que executam o código responsável pela geração dos arquivos.
- Marcação de alguns métodos como `@Check`.
- Inclusão do código de registro e criação da CA Action na classe responsável pela geração dos arquivos.

Foram realizadas 20 execuções do processo de geração de arquivos no hardware disponibilizado, sendo que 10 dessas execuções utilizando-se a versão original do sistema atualmente em produção e as outras 10 execuções com o sistema modificado com as alterações para uso do *framework* JACAAF. Foram coletados o tempo de execução de cada uma dessas execuções. Os dados obtidos estão descritos no gráfico 4.3. O maior valor obtido na execução com o sistema original e também com o sistema modificado foram retirados, pois foram valores muito divergentes dos demais, provavelmente gerados por alguma concorrência de uso da máquina. O gráfico a seguir foi gerado, portanto, com um total de 9 execuções para cada versão do sistema.

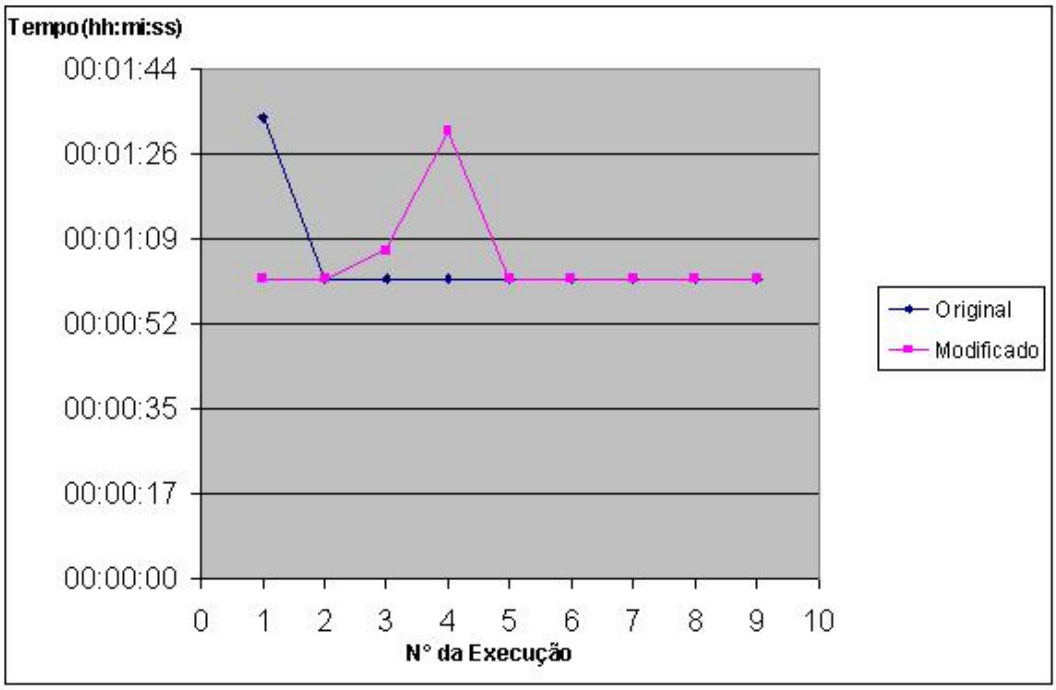

Figura 4.3: Execuções

Durante as primeiras execuções da versão original e da versão modificada, a cada execução os arquivos gerados foram comparados com execuções anteriores para avaliar sua corretude e se as alterações realizadas no código para inclusão do *framework* JACAAF

não injetaram nenhum tipo de falha que pudesse causar alguma degeneração na estrutura ou nos dados dos arquivos gerados. Após algumas execuções as verificações passaram a ser feitas apenas por amostragem.

4.2.6 Avaliação dos resultados obtidos

Como pode ser visto pela Figura 4.3, podemos notar que as execuções realizadas com a versão alterada com o uso do *framework* JACAAF tiveram praticamente o mesmo tempo de execução que as execuções realizadas com a versão original do sistema. Em alguns casos, como a máquina não estava 100% dedicada às execuções do estudo de caso, podemos notar tanto na versão original, quanto na versão alterada, uma pequena variação no tempo das execuções. Entendemos, porém, que são pequenos desvios devido a execução de alguns processos e scripts de controle e administração da máquina durante as execuções do estudo de caso. Desta forma, podemos concluir que para este volume de dados não houve nenhuma latência extra devido aos controles incorporados à aplicação através do uso do *framework*.

Conforme descrito brevemente na seção 1.3, a aplicação alvo deste estudo de caso possui uma solução *ad hoc* para o controle de processamento concorrente, visto que a plataforma *Java Enterprise Edition* não fornece suporte neste sentido. A solução implementada para controle de concorrência no sistema de faturamento de energia elétrica, embora funcional, não está estruturada de maneira sistemática e seu código está localizado dentro de componentes que implementam também regras de negócio da aplicação. Com a utilização do *framework* JACAAF, o tratamento da concorrência de processamento é realizado de uma forma sistemática, componentizada e facilmente extensível, além de seu código estar separado da lógica que implementa as regras de negócio da aplicação.

Como os *message-driven beans* não permitem invocação direta de seus métodos por clientes (somente o serviço de mensagens do servidor de aplicações pode invocá-los), eles ficam incomunicáveis durante o processamento da mensagem, conforme descrito na seção 2.5.2. Desta forma, os desenvolvedores precisam adotar soluções alternativas para que tenham controle de concorrência sobre as instâncias dos *message-driven beans* em processamento. Muitas dessas soluções alternativas passam pela criação *threads* de forma "manual" que por uma consequência natural, deixam de usar os componentes *message-driven beans* providos pela plataforma. Neste sentido, com o controle de concorrência permitido pelo *framework* JACAAF é possível não somente utilizar os *message-driven beans*, mas ainda ter um certo controle sobre sua execução.

4.3 Resumo

Foram apresentados neste capítulo os dois estudos de caso utilizados para avaliar a aplicabilidade do uso do *framework* JACAAF como um *framework* a ser combinado com uma aplicação da plataforma *Java Enterprise Edition* para prover mecanismos de tolerância a falhas de maneira sistemática. Também foi detalhado o funcionamento do *framework* JACAAF em execução e demonstrado parte de seu código fonte.

Capítulo 5

Conclusões e Trabalhos Futuros

Com a crescente demanda por softwares cada vez mais complexos e com a necessidade de diminuição de custos e tempo de desenvolvimento, aliados à exigência de níveis de qualidade cada vez maiores, o uso de DBC e tolerância a falhas no desenvolvimento de sistemas tem se tornado uma constante.

Este trabalho apresentou o *framework* JACAAF *(Java CA Action Framework)*, desenvolvido para permitir que mecanismos de tolerância a falhas, conforme descritos nos conceitos de CA Action, estejam disponíveis para aplicações desenvolvidas para a plataforma *Java Enterprise Edition* (Java EE) e que estas não necessitem ser re-escritas para usufruírem destes mecanismos.

5.1 Conclusões

O trabalho apresentado teve como objetivo unir de forma simples e prática a plataforma *Java Enterprise Edition*, uma das mais notáveis e utilizadas plataformas de DBC, a alguns dos mecanismos previstos por CA Actions, um conceito bastante utilizado para prover tolerância a falhas em sistemas concorrentes e orientados a objetos, utilizando para isso a tecnologia e o paradigma de programação orientada a aspectos (AOP) que também tem sido utilizado de forma crescente nos meios acadêmicos e da indústria, e que permite a evolução de sistemas previamente implantados de forma pouco invasiva. Com os resultados obtidos e verificado nos estudos de caso, entendemos que este objetivo foi alcançado através da implementação do *framework* JACAAF.

5.2 Limitações da solução

O modelo transacional da plataforma *Java Enterprise Edition* normalmente engloba em sua transação desde os componentes da aplicação até o banco de dados ou sistema externo. O *framework* JACAAF, por sua vez, com seu repositório de objetos e seus estados em memória, não permite que o gerenciador de transação do servidor de aplicações atue também em seus objetos, o que faz como que os objetos do repositório façam parte de uma transação "paralela" à transação do servidor de aplicações e, portanto, precisa ser codificada manualmente para que esteja consistente com a transação do servidor de aplicações.

Quando uma aplicação faz uso intenso de banco de dados, o conceito de CA Actions que preconiza aguardar até que o último participante termine seu processamento e seu teste de aceitação para somente então realizar o *commit* de forma a permitir a saída síncrona de todos os participantes da CA Action, pode causar muita contenção de recursos tanto do banco de dados quanto de recursos do próprio servidor de aplicações (transação, *pools*, etc.) tornando inviável, do ponto de vista prático, o uso desse tipo de solução. Devido a isso, não foi implementado no *framework* JACAAF nenhuma política de contenção. Ao invés disso, deixamos para que o próprio desenvolvedor decida (consultando os estados dos objetos salvos e o estado da CA Action como um todo) quando e como fazer o *commit*.

5.3 Trabalhos Futuros

O *framework* JACAAF desenvolvido como resultado deste trabalho, está em sua primeira versão e foi testado somente no contexto dos estudos de casos apresentados no capítulo 4. Ainda existe muito trabalho a ser feito para que o *framework* JACAAF possa se tornar mais robusto, completo e que possa implementar outros aspectos de CA Actions.

Um dos pontos que pode ser facilmente estendido é o de resolução de exceções concorrentes, pois nesta primeira versão, implementamos somente o tratamento de exceções concorrentes de um mesmo tipo. O *framework* pode ser evoluído para que trate exceções concorrente de diferentes tipos, pois embora não tenha sido desenvolvido neste sentido, o ponto para essa extensão está claramente definido. A resolução de exceções através de uma função de resolução ou mesmo uma árvore de resolução [12] pode ser utilizada para resolver esta questão estruturando as exceções de maneira hierárquica.

O desenvolvimento de um estudo de caso utilizando o sistema de faturamento de energia elétrica envolvendo o lançamento de diferentes tipos de exceções concorrentes, também seria de grande valia para aperfeiçoamento e validação do uso do *framework*.

Apêndice A

Código fonte do *framework* JACAAF

BoundChangingAwareAspect.aj

```
1  /*
2   * Copyright (c) 1998-2002 Xerox Corporation,
3   *               2004 Contributors.  All rights reserved.
4   *
5   * Use and copying of this software and preparation of derivative works
6   * based upon this software are permitted.  Any distribution of this
7   * software or derivative works must comply with all applicable United
8   * States exportcontrol laws.
9   *
10  * This software is made available AS IS, and Xerox Corporation makes no
11  * warranty about the software, its performance or its conformity to any
12  * specification.
13  */
14
15 package org.jacaaf.aspects;
16
17 import java.beans.*;
18 import java.io.Serializable;
19 import java.lang.reflect.Method;
20
21 /**
22  * Add bound properties and serialization to Point objects
23  */
24 public aspect BoundChangingAwareAspect {
25
26   public interface ChangingAware extends Serializable {
27     PropertyChangeSupport getChangeSupport();
28   }
```

```
29
30    /*
31     * privately declare a field on Point to hold the property
32     * change support object.  'this' is a reference to a Point object.
33     */
34    private PropertyChangeSupport ChangingAware.support =
35                                        new PropertyChangeSupport(this);
36
37    /*
38     * Declare property change registration methods on Point,
39     * and introduce implementation of the Serializable interface.
40     */
41
42    public void ChangingAware.addPropertyChangeListener(
43                                          PropertyChangeListener listener){
44      support.addPropertyChangeListener(listener);
45    }
46
47    public void ChangingAware.addPropertyChangeListener(
48                                  String propertyName,
49                                  PropertyChangeListener listener){
50      support.addPropertyChangeListener(propertyName, listener);
51    }
52
53    public void ChangingAware.removePropertyChangeListener(
54                                  String propertyName,
55                                  PropertyChangeListener listener){
56      support.removePropertyChangeListener(propertyName,
57                                                          listener);
58    }
59
60    public void ChangingAware.removePropertyChangeListener(
61                                        PropertyChangeListener listener){
62      support.removePropertyChangeListener(listener);
63    }
64
65    public PropertyChangeSupport ChangingAware.getChangeSupport() {
66      return support;
67    }
68
69    public void ChangingAware.hasListeners(String propertyName) {
70      support.hasListeners(propertyName);
71    }
72
73    declare parents : (@Controlled *) implements ChangingAware;
74
```

```
75
76      /*
77       * Send property change event after Y setter completes normally.
78       * Use around advice to keep the old value on the stack.
79       */
80      void around(ChangingAware p): execution(@Transactional void *.*set*(*))
81                                          && within(@Controlled *)
82                                          && target(p) {
83
84        String methodName = thisJoinPoint.getSignature().getName();
85        String propertyName = methodName.substring(3);
86        Method getter = null;
87        try {
88            getter = p.getClass().getMethod("get" + propertyName,
89                                                    new Class[] { } );
90            Object oldValue = getter.invoke(p, null);
91            proceed(p);
92            Object newValue = getter.invoke(p, null);
93            System.out.println("from " + oldValue + " to " + newValue);
94            firePropertyChange(p, propertyName, oldValue, newValue);
95        } catch (Exception e) {
96            e.printStackTrace();
97            throw new RuntimeException("Method get" + propertyName
98                                                    + " not found!");
99        }
100     }
101
102     /*
103      * Utility to fire the property change event.
104      */
105     void firePropertyChange(ChangingAware p, String property,
106                             Object oldval, Object newval) {
107       p.support.firePropertyChange(property, oldval, newval);
108     }
109 }
```

CaptureExceptionAspect.aj

```
1  package org.jacaaf.aspects;
2
3  import org.jacaaf.caaction.RoleManager;
4  import org.jacaaf.caaction.RoleMemento;
5  import org.jacaaf.caaction.CAActionMonitor;
6
7  public abstract aspect CaptureExceptionAspect {
8
9    pointcut handleCall() : execution(public void TaskHandler+.handle(..))
10                                          && within(@Controlled *);
11
12   void around() : handleCall() {
13     try {
14           proceed();
15         } catch (Throwable th) {
16
17           // Logica associada as regras de resolucao. Por exemplo, se
18           // for a excecao for filha de ParallelException, faz rollback.
19           if (th instanceof ParallelException) {
20              RuntimeException resolvedException =
21                                  this.exceptionResolution(th);
22              System.err.println("Found a instance of ParallelException
23                                                    exception: " + th);
24
25           // fazer rollback da thread e avisar monitor
26           RoleManager rManager = RoleManager.getInstance();
27           rManager.rollback(rManager.getCurrentTransaction());
28
29           // lancar excecao para tratamento
30           throw resolvedException;
31
32           } else {
33           // Disparar a resolução de exceções ? Tentar de novo ?
34           // Precisa ser implementado
35           System.err.println("Found a \"common\" exception" + th);
36           }
37         }
38   }
39
40   /*
41    * Método que deve conter a lógica de resolução de exceções.
42    */
43   private RuntimeException exceptionResolution(Throwable t){
44                 return new RollBackException();
45   }
```

```
46  }
```

Check.java

```
1  package org.jacaaf.aspects;
2
3  import java.lang.annotation.ElementType;
4  import java.lang.annotation.Retention;
5  import java.lang.annotation.RetentionPolicy;
6  import java.lang.annotation.Target;
7
8  @Retention(RetentionPolicy.RUNTIME)
9  @Target(ElementType.METHOD)
10 public @interface Check { }
```

CheckPointAspect.aj

```
1  package org.jacaaf.aspects;
2
3  import java.io.Serializable;
4  import java.lang.reflect.Method;
5  import org.jacaaf.caaction.CAActionMonitor;
6
7  /**
8   * Add checkpoint behavior to objects.
9   */
10 public aspect CheckPointAspect {
11
12   pointcut checkMethods(): call(@Check * *(..)) && within(@Controlled *);
13
14   before(): checkMethods() {
15     String methodName = thisJoinPoint.getSignature().getName();
16     String threadName = Thread.currentThread().getName();
17
18     System.out.println(threadName + ": checking "+ methodName
19                                                  + "() method call!!");
20     CAActionMonitor cMonitor = CAActionMonitor.getInstance();
21
22     if (! cMonitor.continueCAAction()) {
23       System.out.println(threadName + "At least one thread has error.
24                                                              ABORT!!");
25       throw new RollBackException();
26     }
27   }
28 }
```

Controlled.java

```
1  package org.jacaaf.aspects;
2
3  import java.lang.annotation.Retention;
4  import java.lang.annotation.*;
5
6  @Retention(RetentionPolicy.CLASS)
7  public @interface Controlled {
8
9  }
```

ParallelException.java

```
1  package org.jacaaf.aspects;
2
3  /*
4   * Excecao mae do framework.
5   */
6  public class ParallelException extends RuntimeException {
7
8          public ParallelException(){;}
9
10         public ParallelException(Throwable cause){
11                 super(cause);
12         }
13 }
```

RollBackException.java

```
1  package org.jacaaf.aspects;
2
3  public class RollBackException extends ParallelException {
4
5          public RollBackException(){};
6
7          public RollBackException(Throwable cause){
8                  super(cause);
9          }
10 }
```

TaskHandler.java

```
1  package org.jacaaf.aspects;
2
3  public interface TaskHandler<T> {
4
5          public void handle(T o) throws Throwable;
6  }
```

Transactional.java

```
1  package org.jacaaf.aspects;
2
3  import java.lang.annotation.ElementType;
4  import java.lang.annotation.Retention;
5  import java.lang.annotation.RetentionPolicy;
6  import java.lang.annotation.Target;
7
8  @Retention(RetentionPolicy.RUNTIME)
9  @Target(ElementType.METHOD)
10 public @interface Transactional {
11
12 }
```

ActionId.java

```
1  package org.jacaaf.caaction;
2
3  public class ActionId implements java.io.Serializable {
4
5    private static final long serialVersionUID = -6248648829292599467L;
6    private String id;
7    private boolean error;
8
9    public void setError(boolean error) {
10     this.error = error;
11   }
12
13   public ActionId(String id) {
14     this.id = id;
15   }
16
17   public String getId() {
18     return id;
19   }
20
21   public void setId(String id) {
22     this.id = id;
23   }
24
25   public boolean hasError() {
26     return error;
27   }
28
29   public String toString() {
30     return "" + id;
31   }
```

```
32  }
```

CAActionMonitor.java

```
1  package org.jacaaf.caaction;
2
3  import java.util.ArrayList;
4  import java.util.HashSet;
5  import java.util.Hashtable;
6  import java.util.Iterator;
7  import java.util.List;
8  import java.util.Map;
9  import java.util.Set;
10
11 /**
12  * Acessa todos os mementos de todas as transacoes
13  * @author Leonardo
14  * @author Peterson
15  *
16  */
17 public class CAActionMonitor {
18
19   private static final CAActionMonitor INSTANCE = new CAActionMonitor();
20   private Hashtable<String, Set<MementoStruct>> association;
21   private final ThreadLocal<String> threadMark =
22                                   new ThreadLocal<String>();
23
24   private CAActionMonitor() {
25     association = new Hashtable<String, Set<MementoStruct>>();
26   }
27
28   public static CAActionMonitor getInstance() {
29         return INSTANCE;
30   }
31
32   // Metodo que associa todos as threads em um processamento logico.
33   // Exemplo: join("MyCAACtion")
34   // Todas os participantes da CA Actioin serao associadas entre si
35   // e marcados para "MyCAACtion"
36   public void join(String transactionName) {
37     System.out.println(Thread.currentThread().getName()+" has joined "
38                                                 + transactionName);
39     String mark = threadMark.get();
40     if (mark == null) {
41         threadMark.set(transactionName);
42         if (!association.containsKey(transactionName)) {
43            association.put(transactionName,
44                                   new HashSet<MementoStruct>());
45         }
```

```
46      }
47    }
48
49    public synchronized void update(ActionId txId) {
50      System.out.println("Updating server in thread> " +
51                         Thread.currentThread().getName());
52      String mark = threadMark.get();
53      if (mark == null) {
54         throw new RuntimeException("Calling update on CAActionMonitor
55                                    requires previous join invocation!");
56      } else {
57          if (RoleMemento.getMementoRef().isEmpty()) {
58             MementoStruct mementoStruct = new MementoStruct();
59             mementoStruct.put(txId, new ArrayList<Change>());
60             association.get(mark).add(mementoStruct);
61          }
62          else {
63            association.get(mark).add(RoleMemento.getMementoRef());
64          }
65      }
66    }
67
68    protected Hashtable<String, Set<MementoStruct>> inspect() {
69      return association;
70    }
71
72    public boolean continueCAAction() {
73      String mark = threadMark.get();
74      if (mark != null) {
75         Iterator<MementoStruct> mementoIt = association.get(mark)
76                                                        .iterator();
77         while (mementoIt.hasNext()) {
78            MementoStruct memento = mementoIt.next();
79            Iterator<ActionId> transIterator = memento.keySet()
80                                                      .iterator();
81            while (transIterator.hasNext()) {
82               ActionId transId = transIterator.next();
83               if (transId.hasError())
84                  return false;
85            }
86         }
87      }
88      return true;
89    }
90  }
```

RoleContext.java

```
1  package org.jacaaf.caaction;
2
3  import java.util.Iterator;
4  import java.util.LinkedList;
5  import java.util.UUID;
6
7  public class RoleContext {
8
9    LinkedList<ActionId> transactionList;
10   private long threadId;
11   private String threadName;
12
13   public RoleContext() {
14     threadId = Thread.currentThread().getId();
15     threadName = Thread.currentThread().getName();
16     transactionList = new LinkedList<ActionId>();
17   }
18
19   public long getAssociatedThreadId() {
20     return this.threadId;
21   }
22
23   public String getAssociatedThreadName() {
24     return this.threadName;
25   }
26
27   public LinkedList<ActionId> getTransactions() {
28     return transactionList;
29   }
30
31   public ActionId begin() {
32     LinkedList<ActionId> txs = getTransactions();
33     ActionId newTransaction = createTransactionId();
34     txs.addLast(newTransaction);
35     return txs.getLast();
36   }
37
38   protected ActionId createTransactionId() {
39     String uniqueID = UUID.randomUUID().toString();
40     return new ActionId(uniqueID);
41   }
42
43   public void rollback() {
44     ActionId txId = checkTx();
45     txId.setError(true);
```

```
46    }
47
48    public void commit() {
49      ActionId txId = checkTx();
50      txId.setError(false);
51    }
52
53    protected ActionId checkTx() {
54      Iterator<ActionId> it = getTransactions().descendingIterator();
55      //Checar apenas a 1a transação
56      if (it.hasNext())
57         return it.next();
58
59      throw new RuntimeException("Transaction not active! Call begin()");
60    }
61
62    public ActionId save() {
63      return begin();
64    }
65  }
```

RoleManager.java

```
1  package org.jacaaf.caaction;
2
3  /**
4   * Um RoleManager é uma abstracao para amarrar um contexto
5   * de um participante (RoleContext) com um monitor de CA Action.
6   * Isso permite uma grande separacao de responsabilidades de quem mantem
7   * os objetos transacionais (RoleContext) e quem tem a visao global de
8   * todos os participantes (CAActionMonitor).
9   * Por outro lado, o CAActionMonitor nao deve conhecer RoleManager's
10  * pois podem existir varias implementacoes diferentes coexistindo
11  * no mesmo processamento.
12  * @author Leonardo
13  *
14  */
15 public class RoleManager {
16
17   private static final RoleManager INSTANCE = new RoleManager();
18
19   protected RoleManager() { }
20
21   public static RoleManager getInstance() {
22     return INSTANCE;
23   }
24
25   private static ThreadLocal<RoleContext> transactions;
26
27     static {
28                 transactions = new ThreadLocal<RoleContext>();
29     }
30
31   public void join(ActionId owner, ActionId txId) { }
32
33   public ActionId begin() {
34     // faz update das informacoes
35     return getContext().begin();
36   }
37
38   public ActionId save() {
39     return getContext().begin();
40   }
41
42   public void commit(ActionId txId) {
43     getContext().commit();
44     CAActionMonitor.getInstance().update(txId);
45   }
```

```
46
47    public void rollback(ActionId txId) {
48      getContext().rollback();
49      CAActionMonitor.getInstance().update(txId);
50    }
51
52    public ActionId getCurrentTransaction() {
53      return getContext().checkTx();
54    }
55
56    protected RoleContext getContext() {
57      RoleContext currentContext = transactions.get();
58      if (currentContext == null) {
59         currentContext = new RoleContext();
60         transactions.set(currentContext);
61      }
62      Thread currentThread = Thread.currentThread();
63      long currentId = currentThread.getId();
64      if (currentId != currentContext.getAssociatedThreadId()) {
65         System.err.println("Branching transaction between thread "
66                            + currentThread.getName() + " and "
67                            + currentContext.getAssociatedThreadName());
68      }
69      return currentContext;
70    }
71  }
```

RoleMemento.java

```
1  package org.jacaaf.caaction;
2
3  import java.beans.PropertyChangeEvent;
4  import java.beans.PropertyChangeListener;
5  import java.lang.reflect.Method;
6  import java.util.ArrayList;
7  import java.util.HashMap;
8  import java.util.List;
9  import java.util.Map;
10 import java.util.Set;
11
12 import org.jacaaf.aspects.BoundChangingAwareAspect.ChangingAware;
13
14 public class RoleMemento implements PropertyChangeListener {
15
16   public static ThreadLocal<MementoStruct> MEMENTO_REPOSITORY =
17                                             new ThreadLocal<MementoStruct>();
18
19   public RoleMemento() {
20     MEMENTO_REPOSITORY.set(new MementoStruct());
21   }
22
23   public void propertyChange(PropertyChangeEvent evt) {
24     String propertyName = evt.getPropertyName();
25     Object oldValue = evt.getNewValue();
26     Object source = evt.getSource();
27     System.out.println("Changing value : " + propertyName + ",old: "
28                                                          + oldValue);
29     Change change = new Change(source,propertyName,oldValue);
30
31     ActionId txId = RoleManager.getInstance().getCurrentTransaction();
32     Map<ActionId,List<Change>> map = MEMENTO_REPOSITORY.get();
33     List<Change> found = map.get(txId);
34     if (found == null) {
35        found = new ArrayList<Change>();
36        map.put(txId, found);
37     }
38     found.add(change);
39     return;
40   }
41
42   public void restoreState() throws Exception {
43     ActionId txId = RoleManager.getInstance().getCurrentTransaction();
44     Map<ActionId,List<Change>> map = MEMENTO_REPOSITORY.get();
45     List<Change> found = map.get(txId);
```

```
46      System.out.println("FOUND " + found);
47      if (found != null) {
48         Object[] arr = found.toArray();
49         for(int i = arr.length -1 ; i >= 0 ; i--) {
50            Change g = (Change) arr[i];
51            ChangingAware cw = (ChangingAware)g.source;
52            Method m = cw.getClass().getMethod("set" + g.propertyName,
53                                    new Class[] { g.oldValue.getClass() });
54            m.invoke(g.source, g.oldValue);
55         }
56       }
57    }
58
59    protected static MementoStruct getMementoRef() {
60         return MEMENTO_REPOSITORY.get();
61    }
62  }
63
64  class MementoStruct extends HashMap<ActionId, List<Change>> {}
65
66  class Change {
67    Object source;
68    String propertyName;
69    Object oldValue;
70
71    public Change(Object src, String pName, Object val) {
72      this.source = src;
73      this.propertyName = pName;
74      this.oldValue = val;
75    }
76    public String toString() {
77      return this.propertyName + "," + this.oldValue;
78    }
79  }
```

Referências Bibliográficas

[1] P. A. Lee and T. Anderson. Fault Tolerance: Principles and Practice, Second Edition, Prentice-Hall, 1990.

[2] A. Avizienis. Towards systematic design of fault-tolerant systems. IEEE Computer, April 1997.

[3] F. Bachman, L. Bass, C. Buhman, F. Long, J. Robert, R. Seacord, and K. Wallnau. Volume ii: Technical concepts of component-based software engineering. Technical Report CMU/SEI-2000-TR-008, Software Engineering Institute, Carnegie-Mellon University, April 2000.

[4] L. Bass, P. C. Clements, and R. Kazman. Architecture in Practice. Addison-Wesley, 2nd edition, 2003.

[5] D. M. Beder, C. M. F. Rubira. Tese de doutorado, Instituto de Computação, Unicamp, 2001.

[6] F. Castor Filho, N. Cacho, E. Figueiredo, R. Ferreira, A. Garcia and C. M. F. Rubira. Exceptions and aspects: The devil is in the details. In of the 14thACM SIGSOFT Symposium on Foundations of Software Engineering, November 2006. To appear.

[7] Celg Distribuição. Janeiro 2010. http://celgd.celg.com.br

[8] C. Chavez. Model-Driven Approach for Aspect-Oriented Design. PhD thesis, Pontifícia Universidade Católica do Rio de Janeiro, April 2004.

[9] P. Clements and R. Kazman. Architecture in Practices. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2003.

[10] P. C. Clements and L. Northrop. Software architecture: An executive overview. Technical Report CMU/SEI-96-TR-003, SEI/CMU, February 1996.

[11] Centro de Pesquisa e Desenvolvimento. Janeiro 2010. http://www.cpqd.com.br

[12] R. H. Campbell and B. Randell. Error recovery in asynchronous systems. IEEE Transactions on Software Engineering, SE-12(8):811-826, 1986.

[13] F. Castor Filho and C. M. F. Rubira. Implementing coordinated exception handling for distributed object-oriented systems with AspectJ. In of the VIII Brazilian Symposium on Programming Languages, pages 128-142, May 2004.

[14] F. Castor Filho and C. M. F. Rubira. Tratamento de Exceções no Desenvolvimento de Sistemas Tolerantes a Falhas Baseados em Componentes. PhD thesis, Instituto de Computação, Unicamp, Novembro 2006.

[15] F. Cristian. Exception handling. In T. Anderson, editor, of Resilient Computers, pages 68-97. Blackwell Scientific Publications, 1989.

[16] A. Capozucca, N. Guelfi, P. Pelliccione, A. Romanovsky and Zorzo. CAA-DRIP: a framework for implementing Coordinated Atomic Actions In Proceedings of the 17th International Symposium on Software Reliability Engineering, ISSRE 2006, November 7-10, 2006, Raleigh, North Carolina, pp. 385-394, IEEE Computer Society 2006.

[17] N. Cacho, C. Sant'Anna, E. Figueiredo, A. Garcia, and T. B. C. Lucena. Composing design patterns: A scalability study of aspect-oriented programming. In of 5th ACM Conference on Aspect-Oriented Software Development, pages 109-121, March 2006.

[18] F. Chen, Q. Wang, H. Mei, and F. Yang. An architecture-based approach for component-oriented development. thAnnual International Computer Software and Applications Conference, August 2002.

[19] D. M. Beder, L. E. Buzato. Integração dos Mecanismos de Recuperação de Erros por Avanço e Retrocesso. Tese de mestrado, Instituto de Computação, Unicamp, 1997.

[20] The Eclipse Foundation. Janeiro 2010. http://www.eclipse.org

[21] M. Evangelist, N. Francea, and S. Katz. "Multiparty Interactions for Interprocess Communicationand Synchronization". In IEEE Transactions on Software Engineering,15(11), pp. 1417-1426, November 1989.

[22] CPqD Gestão Comercial (Energia). Janeiro 2010. http://www.cpqd.com.br/solucoes-e-produtos/213-cpqd-gestao-comercial-energia.html

[23] R. Filman and D. Friedman. Aspect-oriented programming is quantification and obliviousness. In of the OOPSLA'2000 Workshop on Advanced Separation of Concerns, October 2000.

[24] Y. Feng, G. Huang, Y. Zhu, H. Mei. "Exception handling in component composition-with the support of middleware". In Proceedings of the 5th international workshop on Softwareengineering and middleware 2005, Lisbon, Portugal September 05 - 06, 2005.

[25] L. Fiege, G. Muhl and F. C. Gartner. A modular approach to building event-based systems. Technical report, TU Darmstadt, 2002.

[26] G. R. M. Ferreira, C. M. F. Rubira. Tratamento de exceções no desenvolvimento de sistemas confiaveis baseados em componentes. Tese de mestrado, Instituto de Compução, Unicamp, 2001.

[27] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. Design Patterns: Elements of Reusable Software Systems. Addison-Wesley, 1995.

[28] J. Gray and A. Reuter. Transaction Processing: Concepts and Techniques, Morgan Kaufmann, 1993.

[29] John B. Goodenough. Exceptional handling: Issues and a proposed notation. , 18(12), 1975.

[30] A. Garcia, C. Sant'Anna, E. Figueiredo, U. Kulesza, C. Lucena, and A. von Staa. Modularizing design patterns with aspects: A quantitative study. In of the 4th ACM Conference on Aspect-Oriented Software Development, pages 3-14, March 2005.

[31] G. Murphy, R. Walker, E. Baniassad, M. Robillard, and M. Kersten. Does aspecto-riented programming work ? of the ACM, 44(10):75-77, October 2001.

[32] J. Hannemann and G. Kiczales. Design pattern implementation in java and AspectJ. In of 17th ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications, pages 161-173, November 2002.

[33] The J2EE 1.4 Tutorial. Setembro 2007. http://java.sun.com/j2ee/1.4/docs/tutorial/doc/index.html

[34] JBoss Community. Agosto 2007. http://jboss.org

[35] The JBoss 4 Application Server Guide. Maio 2009. http://docs.jboss.org/jbossas/jboss4guide/r1/html/index.html

[36] Sun Microsystems. Java technology, Julho 2007. http://java.sun.com/javaee

[37] YJ. Joung and S. A. Smolka. "A Comprehensive study of the Complexity of Multi-party Interaction". In Journalof ACM, 43(l), pp. 75-115, January 1996.

[38] B. Joy, G. Steele, J. Gosling, and G. Bracha. The Java Language Specification, 2nd, Addison Wesley, 2000.

[39] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. V. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. In of the 11th European Conference on Object-Oriented Programming , LNCS 1271, pages 220-242, 1997.

[40] R. Laddad. AspectJ in Action. Manning, 2003.

[41] R. Laddad. AspectJ in Action, 2nd Edition. Manning, 2009.

[42] M. Lippert and C. V. Lopes. A study on exception detection and handling using aspect-oriented programming. In of the 22nd International Conference on Software Engineering, pages 418-427, June 2000.

[43] M. D. McIlroy. Mass-produced software componentes. In P. Naur and B. Randell, editors, Software Engineering, Report on a conference sponsored by the NATO Science Committee, pages 138-155, Garmisch, Germany, October 1969.

[44] Memento Pattern, Wikipedia. Janeiro 2010. http://en.wikipedia.org/wiki/Memento_pattern

[45] R. T. Monroe, A. Kompanek, R. Melton and D. Garlan. Architectural styles, design patterns, and objects. Softw., 14(1):43-52, 1997.

[46] N. Medvidovic, P. Oreizy, and R. N. Taylor. Reuse of of-the-shelf components in c2-style architectures. In Proceedings of the 1997 ACM SIGSOFT Symposium on Software Reusability, May 1997.

[47] Microsoft Corporation. Distributed component object model, 2002. http://www.microsoft.com/com/tech/DCOM.asp.

[48] Microsoft .NET framework. Setembro 2007. http://msdn.microsoft.com/netframework

[49] OMG. The corba component model. Setembro 2007. http://www.omg.org

[50] Plain Old Java Object, Wikipedia. Janeiro 2010. http://en.wikipedia.org/wiki/POJO

[51] P. H. S. Brito, C. M. F. Rubira. Um Método para Modelagem de Exceções em Desenvolvimento Baseado em Componentes. Tese de mestrado, Instituto de Computação, Unicamp, 2005.

[52] D. E. Perry and A. L. Wolf. Foundations for the study of software architecture. Software Engineering Notes , 17(4):40-52, October 1992.

[53] B. Randell. System Structure for Software Fault Tolerance, IEEE Trans. Soft. Eng., vol. SE-1, no.2, pp.220-232, 1975.

[54] A. Regalado et al. 10 emerging technologies that will change the world. Review, pages 97-113, January/February 2001.

[55] C. M. F. Rubira, R. de Lemos, G. Ferreira, and F. Castor Filho. Exception handling in the development of dependable component-based systems. Software - Practice and Experience, 35(5):195-236, March 2005.

[56] A. Romanovsky, B. Gallina and N. Guelfi. Coordinated Atomic Actions for Dependable Distributed Systems: the Current State in Concepts, Semantics and Verification Means. In ISSRE '07: 18th IEEE International Symposium on Software Reliability Engineering, 2007.

[57] A. Romanovsky, P. Periorellis, A. F. Zorzo. "Structuring Integrated Web Applications for Fault Tolerance". In Proceedings of the The Sixth International Symposium on Autonomous Decentralized Systems (ISADS'03) ISADS '03.

[58] H. Rajan and K. Sullivan. Eos: Instance-level aspects for integrated system design. In of Joint 9th European Conference on Software Engineering/11th SIGSOFT Symposium on Foundations of Software Engineering, pages 291-306, Helsinki, Finland, September 2003.

[59] J. Xu, B. Randell, A. Romanovsky, C. M. F. Rubira, R. J. Stroud and Z. Wu. Fault tolerance in concurrent object-oriented software through coordinated error recovery. In *Proceedings of the 25th Symposium on Fault-Tolerant Computing Systems*, pages 499–508, Pasadena, USA, 1995.

[60] R. A. Riemenschneider and V. Stavridou. The role of architecture description languagesin component-based development: The sri perspective. In Proceedings of the 21st International Conference on Software Engineering, 1999.

[61] M. Shaw and P. Clements. A field guide to boxology: Preliminary classification of architectural styles for software systems. In of COMPSAC'96,Washington, DC, USA, August 1996.

[62] Secretaria da Fazenda do Estado de Goiás. Abril 2009. http://www.sefaz.go.gov.br.

[63] M. Shaw and D. Garlan. Architecture: Perspectives on an Emerging Discipline. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1996.

[64] S. Soares, E. Laureano, and P. Borba. Implementing distribution and persistence aspects with AspectJ. In of the 17th ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications, pages 174-190, 2002.

[65] K. Simons and J. Stafford. Container-Managed Exception Handling for the Predictable Assembly of Component-Based Applications, Master's thesis, Dept. of Computer Science, Tufts University, May 2004.

[66] I. Sommerville. Engineering. Addison-Wesley, 6th edition, 2001.

[67] Sun Microsystems. Enterprise Javabeans spec. v2.1 - proposed final draft, 2002. http://java.sun.com/products/ejb/.

[68] Software Componentry, Wikipedia. Agosto 2007. http://en.wikipedia.org/wiki/Software_componentry.

[69] C. Szyperki. Component technology - what, where, and how? In of the 25th International Conference on Software Engineering, pages 684-693. IEEE Computer Society Press, May 2003.

[70] G. Vecellio, W. M. Thomas, and R. Sanders. "Containers for Predictable Behavior of Component-based Software". In Proceedings of the 5th ICSE Workshop on Component-Based Software Engineering: Benchmarks for Predictable Assembly Orlando, Florida, USA May 19-20, 2002.

[71] A. F. Zorzo and R. J. Stroud. A distributed object-oriented framework for dependable multiparty interactions. In OOPSLA '99: Proceedings of the 14th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications, pages 435-446. ACM Press, 1999.