Técnicas de Estruturação de Software no Desenvolvimento de Sistemas-de-Sistemas Confiáveis

Ricardo de Mendonça da Silva

Dissertação de Mestrado

Instituto de Computação
Universidade Estadual de Campinas

Técnicas de Estruturação de Software no Desenvolvimento de Sistemas-de-Sistemas Confiáveis

Ricardo de Mendonça da Silva[1]

Dezembro de 2003

Banca Examinadora:

- Profª. Drª. Cecília Mary Fischer Rubira (Orientadora)
- Prof. Dr. Avelino Francisco Zorzo
  Faculdade de Informática - PUCRS
- Profª. Drª. Eliane Martins
  Instituto de Computação - UNICAMP
- Profª. Drª. Ariadne Maria Brito Rizzoni Carvalho (Suplente)
  Instituto de Computação - UNICAMP

[1] Apoio financeiro da FAPESP (processo 01/06064-3) e do CNPq

FICHA CATALOGRÁFICA ELABORADA PELA BIBLIOTECA DO IMECC DA UNICAMP

Silva, Ricardo de Mendonça

Si38t Técnicas de estruturação de software no desenvolvimento de sistemas-de-sistemas confiáveis / Ricardo de Mendonça Silva -- Campinas, [S.P. :s.n.], 2003.

Orientador : Cecília Mary Fischer Rubira.

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

1.Engenharia de software. 2. Tolerância a falha. 3. Software – Arquitetura. 4. Software – Desenvolvimento. I. Rubira, Cecília Mary Fischer. II. Universidade Estadual de Campinas. Instituto de Computação. III. Título.

Técnicas de Estruturação de Software no Desenvolvimento de Sistemas-de-Sistemas Confiáveis

Este exemplar corresponde à redação final da Dissertação devidamente corrigida e defendida por Ricardo de Mendonça da Silva e aprovada pela banca examinadora.

Campinas, 16 de Dezembro de 2003.

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.

Resumo

Num mercado cada vez mais competitivo, as empresas de software são constantemente desafiadas a desenvolverem sistemas de software confiáveis e extensíveis em prazos cada vez mais curtos. Tais sistemas devem ser capazes de acompanhar as freqüentes evoluções ocasionadas pela volatilidade dos requisitos de negócios. Por outro lado, o fato de vidas e de investimentos financeiros dependerem desses sistemas de software exige que eles sejam realmente confiáveis.

O desenvolvimento de sistema de software baseados em componentes aliado à elaboração de uma arquitetura de software vem se apresentando como uma alternativa bastante atraente para atingir tais objetivos. Dentro desta área, um tópico de pesquisa emergente é o desenvolvimento de técnicas de tolerância a falhas que são aptas a lidar com as características de Sistemas-de-Sistemas (SdSs). SdSs são sistemas complexos construídos a partir de sistemas-componentes autônomos e heterogêneos que provêem seus serviços individualmente mesmo se o SdSs não estiver apto no momento a prover os seus.

Neste trabalho, apresentamos uma abordagem arquitetural para a composição de sistemas-componentes baseados em *contratos de composição* e um esquema de tratamento de exceções, baseado no conceito de ação atômica coordenada, que considera a execução concorrente dos componentes arquiteturais. O uso de contratos de composição permite um alto grau de desacoplamento entre os sistemas-componentes, além de coordená-los durante a realização de uma ação. Tudo isto, aliado ao emprego de um esquema de tratamento de exceções, permite utilizar a arquitetura de software desenvolvida nesta dissertação durante o desenvolvimento de SdSs confiáveis, como comprovado pelos estudos de casos realizados.

Abstract

In a rapidly changing and competitive market, the industry has constantly been faced with the challenge of developing reliable and extensible software systems in shorter period of times. These systems should be able to follow the frequent evolutions due to the business requirements volatility. Furthermore, these systems are also required to be reliable, since their failure may endanger human lives or cause great financial losses.

Component-based software development associated to the software architecture elaboration has been presented as an alternative widely adopted to achieve these objectives. In this area, an emergent research topic is the creation of fault tolerance techniques that can deal with the characteristics of the System-of-Systems (SoSs). SoSs are complex systems built from autonomous and heterogeneous component-systems that provide their services individually even if the SoSs are not able to provide theirs.

In this work, we present an architectural approach to compose component-systems based on the *composition contracts* and an exception handling based on *coordinated atomic action* (CA Action) scheme that considers the concurrent execution of architectural components. The composition contracts allow a high level decoupling between the component-systems, and coordinate them during an action. These features, added to the use of an exception handling scheme, allows the use of the software architecture developed in this work for the development of SoSs, such as proved by the implemented case studies.

Agradecimentos

Depois de tantas dificuldades e desafios enfrentados durante a realização deste trabalho, fica difícil resumir a satisfação de ter chegado ao fim de mais uma longa jornada. No entanto, não posso deixar de lembrar de todos aqueles que direta ou indiretamente contribuíram para que eu alcançasse mais esta conquista.

Gostaria de agradecer

- primeiramente a Deus, por sempre ter me acompanhado em toda minha vida, nunca me faltando bênçãos quando estas mais foram necessárias;
- à minha querida esposa, Eveline, por todo carinho, paciência e incentivo despendido durante todos os momentos em que estive realizando este mestrado;
- à professora Cecília Rubira, pela experiência e conhecimentos transmitidos, além de toda paciência e tempo investidos para me incentivar e orientar nos rumos desta pesquisa;
- a Paulo Astério, por todo o apoio oferecido durante o tempo em que trabalhamos juntos;
- a Fernando Castor, pelas idéias que trocamos durante as várias discussões que tivemos;
- a todos os amigos que conheci aqui na Unicamp, pela convivência agradável que me proporcionaram;
- aos professores que compõem a banca examinadora, pelo cuidado e atenção que tiveram ao ler, corrigir e comentar esta dissertação;
- à Fapesp e ao CNPq pelo apoio financeiro;
- e finalmente aos meus pais, Aldemir e Dilma, que durante anos se dedicaram e batalharam para que eu alcançasse meus objetivos. Dedico-lhes esta obra como reconhecimento da participação deles no que hoje eu sou.

Conteúdo

Resumo vi
Abstract vii
Agradecimentos viii
Conteúdo ix
Lista de Figuras xii
1 Introdução 1
1.1 Problemas 2
1.2 Objetivos 3
1.3 Solução Proposta 4
1.4 Trabalhos Relacionados 5
1.5 Organização deste documento 7
2 Fundamentos de Tolerância a Falhas e Sistemas-de-Sistemas 9
2.1 Conceitos de Tolerância a Falhas 9
2.1.1 Sistema Computacional 10
2.1.2 Definições: Falha, Erro e Defeito 11
2.1.3 Componente Tolerante a Falhas Ideal 12
2.1.4 Etapas de Implantação de Tolerância a Falhas 13
2.2 Sistemas-de-Sistemas 14
2.2.1 Incompatibilidades entre Sistemas-Componentes 15
2.3 Considerações Finais 19
3 Princípios de Estruturação de Software 21
3.1 Desenvolvimento Baseado em Componentes 21
3.2 Arquitetura de Software 22
3.2.1 Estilo Arquitetural C2 23
3.3 Framework Orientado a Objetos 24
3.4 Contratos de Coordenação 25
3.5 Considerações Finais 27

4 Ações Atômicas Coordenadas 29
4.1 Tipos de Concorrência 30
4.2 Propriedades de Ações Atômicas Coordenadas 30
4.3 Aninhamento de Ações Atômicas Coordenadas 31
4.4 Parametrização, Ligação e Instanciação de uma Ação Atômica Coordenada 32
4.5 Ações Atômicas Coordenadas e Sistemas-de-Sistemas 33
4.6 Considerações Finais 34
5 Arquitetura FT-SC3 37
5.1 Descrição Geral da Arquitetura FT-SC3 38
5.2 O Componente Contrato de Composição 40
5.3 O Componente Interceptador 42
5.4 O Componente Tratador de Exceções 44
5.5 Considerações Finais 46
6 Ferramentas de Apoio para Arquitetura FT-SC3 49
6.1 Framework FT-SC3 49
6.1.1 Tipos de Mensagens 50
6.1.2 Protocolo dos Componentes 53
6.1.3 Protocolo dos Conectores 55
6.1.4 Como utilizar o Framework FT-SC3 56
6.2 Um Ambiente para Modelagem da Arquitetura FT-SC3 57
6.3 Considerações Finais 59
7 Estudos de Casos 61
7.1 Estudo de Caso 1 – Sistema Bancário Simples 62
7.2 Estudo de Caso 2 – Sistema Bancário Evolutivo 64
7.2.1 Cenário 1: Um Participante e nenhum Contrato 64
7.2.2 Cenário 2: Dois Participantes e um Contrato 65
7.2.3 Cenário 3: Três Participantes e dois Contratos 68
7.3 Estudo de Caso 3 – Sistema Varejista 71
7.3.1 Motivação e Solução 73
7.3.2 Coordenação entre Gerenciador do Sistema e PDVs 74

7.3.3 Coordenação dentro dos PDVs 76
7.4 Considerações Finais 78
8 Conclusões e Trabalhos Futuros 81
8.1 Análise Global dos Resultados 81
8.1.1 Benefícios 81
8.1.2 Limitações 82
8.2 Contribuições 83
8.3 Trabalhos Futuros 85
8.4 Conclusões 86
Bibliografia 89

Lista de Figuras

Figura 1. Esboço da solução arquitetural para integração de sistemas-componentes. 5
Figura 2. Sistema Computacional do ponto de vista externo 10
Figura 3. Falha, erro e defeito 11
Figura 4. Componente ideal tolerante a falhas 13
Figura 5. Elementos básicos do estilo arquitetural C2 23
Figura 6. Ação Atômica Coordenada Aninhada 32
Figura 7. Visão geral da Arquitetura FT-SC3 39
Figura 8. Estrutura básica de um conector de composição 40
Figura 9. Definição do componente contrato de composição. 42
Figura 10. Estrutura interna de um interceptador e de um contrato de composição 45
Figura 11. Definição de um componente tratador de exceções 46
Figura 12. Diagrama de classes das mensagens de solicitações do Framework FT-SC3 51
Figura 13. Diagrama de classes das mensagens de notificações do Framework FT-SC3 ... 52
Figura 14. Arquitetura do sistema bancário do Estudo de Caso 1 62
Figura 15. Diagrama de Seqüência do Estudo de Caso 1. 63
Figura 16. Sistema Bancário com um Participante e nenhum Contrato 65
Figura 17. Sistema Bancário com dois Participantes e um único Contrato 66
Figura 18. Diagrama de Seqüência do Cenário 2 67
Figura 19. Diagrama de Seqüência para um Tratamento de Exceções no Cenário 2. 68
Figura 20. Sistema Bancário com três Participantes e dois Contratos 69
Figura 21. Diagrama de Seqüência para o Cenário 3 70
Figura 22. Esboço do Sistema Varejista 72
Figura 23. Coordenação entre o Gerenciador do Sistema e os PDVs 74
Figura 24. Diagrama de Seqüência entre Gerenciador do Sistema e PDVs 75
Figura 25. Coordenação dentro dos PDVs 76
Figura 26. Diagrama de seqüência entre os sub-componentes do PDV 77

Capítulo 1

Introdução

O desenvolvimento de sistemas computacionais modernos baseados em componentes é dificultado por dois principais fatores: (i) a complexidade inerente ao integrar sistemas de diferentes áreas e (ii) a necessidade de prover serviços com alto grau de confiabilidade [19]. Além disso, as exigências impostas pelo mercado requererem que os sistemas de software venham a se adequar rapidamente às modificações que constantemente ocorrem nas suas regras de negócio [3, 1], tornando as propriedades de reuso e manutenibilidade como características fundamentais para o sucesso destes sistemas [12, 56]. Além disso, muitos destes sistemas de software modernos estão se tornando críticos no tocante a segurança, devido à quantidade de vidas e/ou investimentos financeiros que são perdidos caso eles venham a apresentar defeitos [34].

Sistemas baseados em componentes se apresentam como uma solução promissora para resolver tais problemas, visto que eles foram criados com o objetivo de [19]: (i) apoiar o desenvolvimento de sistemas através da composição de componentes; (ii) apoiar o desenvolvimento de componentes como entidades reutilizáveis; e (iii) facilitar a manutenção e atualização através de substituição e modificação de componentes.

Sistemas-de-Sistemas (SdSs) são sistemas baseados em componentes que integram vários sistemas autônomos, heterogêneos já existentes, capazes de prover seus serviços independentemente do SdSs estar apto a prover os seus. Tais sistemas são denominados de sistemas-componentes visto que eles constituem os componentes que compõem o SdSs. O contexto deste trabalho considera que os sistemas-componentes são tratados como caixas-pretas [16] e definidos através de suas interfaces. Desenvolvedores de SdSs lidam com novos problemas que são distintos dos encontrados em sistemas baseados em componentes tradicionais, tais como o desconhecimento e a quantidade de possíveis incompatibilidades entre os sistemas-componentes envolvidos. Com isto surge a necessidade de pesquisas que permitam gerenciar o ciclo de desenvolvimento de tais sistemas e, principalmente, a integração entre os sistemas-componentes.

Em geral, nenhuma suposição pode ser feita sobre o projeto interno e implementação de um sistema-componente. Por exemplo, quando se integra um *Web Service* [35] a um sistema de software, sua implementação real é dinamicamente conectada em tempo de execução e a única informação disponível é a especificação da interface pública do serviço oferecido. Assim, não há garantias sobre os atributos de qualidade da implementação real, tais como sua corretude, disponibilidade e confiabilidade. Para alcançar tais propriedades de qualidade, este trabalho focou principalmente no desenvolvimento de uma arquitetura de software [47] utilizando o estilo arquitetural C2 [59] que permitisse a composição de sistemas-componentes concorrentes através do uso de contratos de composição. Um contrato de composição estende o conceito de contratos de coordenação [2] para incluir um esquema de tratamento de exceções [18] baseado em uma ação atômica coordenada [66, 67].

Um contrato de coordenação é uma conexão que pode ser estabelecida entre um grupo de objetos através do qual regras e restrições são impostas quando estes colaboram para a realização de um novo serviço. Tais regras e restrições determinam as formas específicas de interações e/ou adaptações para os novos requisitos [2]. Uma ação atômica coordenada [66, 67] é um mecanismo para estruturação de sistemas concorrentes tolerantes a falhas que unifica a noção de recuperação de erro por avanço e retrocesso. Em uma ação atômica coordenada a recuperação de erro por avanço é apoiada por meio de um de tratamento de exceções que leva em consideração a possibilidade de múltiplas exceções serem lançadas concorrentemente pelos objetos participantes. Ações atômicas coordenadas ainda integram concorrência competitiva e cooperativa dentro de um arcabouço conceitual que facilita a construção de sistemas confiáveis.

1.1 Problemas

Nos dias atuais, a reutilização de componentes previamente desenvolvidos é uma questão chave no desenvolvimento de software, visto que usualmente as empresas, por questões econômicas e de tempo, desejam criar novas funcionalidades mais complexas e confiáveis a partir das funcionalidades básicas já existentes nos sistemas que elas já possuem e utilizam, e que nem sempre são sistemas confiáveis.

Com o intuito de buscar uma forma mais organizada para composição de sistemas, surgem as pesquisas relacionadas a SdSs, que visam encontrar soluções arquiteturais que auxiliem no projeto de tais sistemas. Durante a criação de SdSs, questões como a independência dos sistemas-componentes e sua natureza evolutiva são pontos críticos a serem estudados [42], tendo em vista que o desenvolvimento de um SdSs não se resume na criação das novas funcionalidades a partir dos sistemas-componentes, mas numa estruturação que permita criar sistemas capazes de se adaptarem para atender aos novos requisitos que surgem devido às modificações das regras de negócio das empresas, além de manter a independência dos sistemas-componentes.

Este trabalho tem por objetivo estudar uma abordagem arquitetural que permita a criação de SdSs confiáveis. Uma abordagem apropriada para a construção de SdSs confiáveis consiste em empregar técnicas de estruturação de software, com propriedades de tolerância a falhas associadas, que permitem-nos definir regiões de confinamento e recuperação de erros. Muitos SdSs são sistemas concorrentes e distribuídos, onde o tratamento de exceções é mais difícil de ser realizado, visto que os sistemas-componentes operaram concorrentemente, muitas vezes competindo por recursos e cooperando para realizar uma atividade em conjunto. Neste contexto, erros podem ser propagados através da comunicação inter-componentes.

O desenvolvimento de uma arquitetura de software que considere todos os pontos críticos na construção de SdSs confiáveis apresenta-se como uma forma organizada para atingir as qualidades necessárias para o sistema. A partir deste preceito, nos limitamos ao estudo de uma arquitetura de software que permitisse a construção de SdSs confiáveis, com uma séria preocupação na extensibilidade, reuso e manutenibilidade.

1.2 Objetivos

Os principais objetivos deste trabalho são:

- Definir uma solução arquitetural com propriedades de tolerância a falhas que seja apta a lidar com as características de SdSs;

- Definir um esquema de ações atômicas coordenadas e tratamento de exceções adequado para SdSs, utilizando tecnologias de desenvolvimento baseado em componentes existentes; e
- Definir um conjunto de diretrizes que guie a construção de SdSs confiáveis, passando pela sua arquitetura e projeto detalhado.

1.3 Solução Proposta

SdSs estão sujeitos a diferentes tipos de falhas, e a forma mais adequada de lidar com tais falhas é aceitar que qualquer sistema de software pode apresentar falhas e que devemos empregar mecanismos de tolerância a falhas para tolerá-las durante a execução do sistema.

O desenvolvimento de uma arquitetura de software tem objetivo de projetar a estrutura global de um sistema de grande porte, de forma a atender aos requisitos atribuídos ao sistema, e prover suporte ao reuso [7]. Com isto em mente, foi desenvolvida uma arquitetura de software, denominada Arquitetura FT-SC3 (**F**ault-**T**olerant **S**ystem **C**omposition using **C**oordination **C**ontracts), que guia a composição de sistemas-componentes autônomos, heterogêneos e muitas vezes não-confiáveis, tais como componentes COTS[2] [43], *Web Services* [35] e sistemas legados, com o objetivo principal de prover serviços confiáveis, de forma transparente, aos clientes desta aplicação, sem perder de vista as restrições impostas pelo uso de sistemas-componentes.

Durante o desenvolvimento da Arquitetura FT-SC3 foi utilizado o estilo arquitetural C2. Uma arquitetura no estilo C2 é composta por componentes que são interligados através de conectores explícitos. Componentes se comunicam através de mensagens assíncronas. Escolhemos o estilo C2 porque experimentos anteriores com este estilo arquitetural mostraram que ele é adequado para a composição de componentes autônomos [43]. Nossa arquitetura é composta por três camadas: (i) a camada computacional, formada por sistemas-componentes que executam operações básicas; (ii) a camada de coordenação, que coordena os sistemas-componentes na criação de serviços mais complexos e impõe as novas regras de negócio sobre os serviços da camada computacional [63, 64]; e (iii) a

[2] COTS - Commercial Off-The-Shelf

camada de aplicação onde os clientes do novo sistema composto podem solicitar os serviços oferecidos pela camada de coordenação.

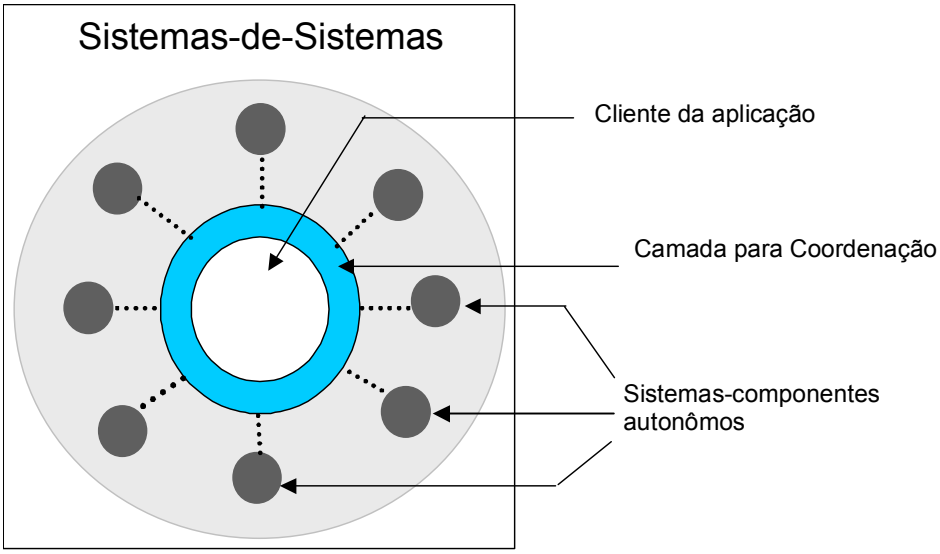

Figura 1. Esboço da solução arquitetural para integração de sistemas-componentes.

A Figura 1 mostra um esboço geral da arquitetura de software proposta neste trabalho, onde vários sistemas-componentes podem se conectar através de uma camada para coordenação, capaz de coordena-los a fim que eles cooperem na realização de serviços mais complexos, além de impor novas propriedades aos serviços oferecidos. Estes novos serviços são disponíveis para os clientes da aplicação de forma transparente.

Baseado no conceito de ações atômicas coordenadas, a Arquitetura FT-SC3 possui um esquema de tratamento de exceções capaz de tratar várias exceções lançadas concorrentemente pelos sistemas-componentes.

1.4 Trabalhos Relacionados

Este trabalho foi inspirado inicialmente pela idéias apresentadas por Beder et al [9], para a construção de SdSs [42] confiáveis utilizando uma abordagem arquitetural. Os autores desse trabalho apresentam uma arquitetura de software baseada no conceito de componente tolerante a falhas ideal [38] para construções de sistemas de software

complexos integrados com ações atômicas coordenadas. Por fim, eles fizeram algumas sugestões para aplicar as idéias no contexto de SdSs confiáveis.

Em [58], Tartanoglu apresenta o conceito de *Web Service Composition Action* (WSCA), que também utiliza ações atômicas coordenadas para realizar a composição de *Web Services* na criação de um sistema confiável. A principal diferença entre o trabalho descrito neste documento e o conceito de WCSA é que a abordagem aqui adotada utiliza o conceito de contratos de coordenação, permitindo melhor adaptabilidade, além de utilizar um projeto arquitetural para ser aplicado a classes de sistemas baseados em serviços gerais, e não apenas a *Web Services*.

Semelhante ao nosso trabalho, Pires [48] propõe uma solução arquitetural que realiza composição de *Web Services* de forma confiável, usando um estilo arquitetural em camadas, entretanto, na sua solução somente é possível realizar recuperação de erro por retrocesso, sem considerar tratamento de exceções concorrentes.

Zorzo e Stroud [68] descrevem um *framework* para implementação de *Dependable Multiparty Interactions* (DMI). Em cada DMI os participantes se unem para produzir um estado conjunto temporário intermediário, usam este estado para executar alguma atividade e então saem da interação para continuarem suas atividades normais. Um DMI é construído através do encadeamento de interações não confiáveis, onde cada interação implementa os tratadores para as exceções levantadas na interação anterior na cadeia. A principal diferença entre o trabalho descrito neste documento para o supracitado está na utilização de contratos de coordenação para auxiliar na integração de sistemas-componentes. Além disso, os participantes dentro de uma DMI implementam suas próprias medidas para se recuperarem de erros. Essa abordagem é razoável para sistemas orientados a objetos, mas pode ser forte demais quando se trata de sistemas baseados em componentes.

O trabalho de Lemos em [39] descreve um estilo arquitetural em que conectores são considerados entidades de primeira-classe que implementam a descrição do comportamento colaborativo entre componentes. Estes conectores especiais são chamados pelo autor de conectores de cooperação. Esse trabalho é complementar ao nosso, visto que o conector de composição da nossa arquitetura pode ser visto como um conector de cooperação (Seção 5.2) descrito com maiores detalhes.

O conceito de um componente C2 ideal (iC2C) apresentado por Guerra et al [31] define uma estrutura para incorporar tolerância a falhas dentro de um sistema baseado em componentes no nível arquitetural. Esse trabalho foi posteriormente refinado por Castor et al [13], que definiu um sistema de tratamento de exceções a nível arquitetural baseado no conceito de iC2C. Ambas abordagens tratam do problema de criação de sistemas confiáveis baseados em componentes utilizando o emprego de estratégias no nível arquitetural. Entretanto, nenhum deles cuida de tratamento de exceções concorrentes, nem se preocupa com a composição de componentes heterogêneos.

1.5 Organização deste documento

Este documento está organizado da seguinte forma:

- **Capítulo 2** apresenta alguns conceitos fundamentais sobre Tolerância a Falhas, definindo o que é um componente tolerante a falha ideal e a diferença entre falha, erro e defeito. Além disso, é descrito o conceito de Sistemas-de-Sistemas e os principais problemas encontrados na integração de sistemas-componentes;
- **Capítulo 3** descreve alguns princípios sobre estruturação de software utilizados no decorrer de todo este trabalho. Neste capítulo, descrevemos os conceitos de desenvolvimento de software baseado em componente, arquitetura de software, estilo arquitetural C2 e contratos de coordenação;
- **Capítulo 4** apresenta em detalhes as principais propriedades e características de ação atômica coordenada, além fazer um breve comentário das vantagens em utilizar ações atômicas coordenadas para construir Sistemas-de-Sistemas;
- **Capítulo 5** apresenta a Arquitetura FT-SC3 que guia a composição de sistemas-componentes durante o desenvolvimento de um sistema tolerante a falhas, utilizando contratos de composição;
- **Capítulo 6** descreve as ferramentas de apoio que foram criadas para dar suporte ao desenvolvimento de sistemas baseado na Arquitetura FT-SC3. Neste capítulo detalhamos o Framework FT-SC3 desenvolvido, onde são apresentados detalhes de implementação desta arquitetura de software. Além disso, descrevemos um

ambiente para modelagem desenvolvido para a criação de sistemas que utilizam a Arquitetura FT-SC3;

- **Capítulo 7** apresenta três estudos de casos desenvolvidos para mostrar a viabilidade do uso da Arquitetura FT-SC3 no desenvolvimento de SdSs, permitindo identificar suas vantagens e limitações; e
- **Capítulo 8** faz uma análise global dos resultados obtidos, sugere alguns trabalhos que podem ser realizados para aprimorar a abordagem aqui apresentada e faz uma síntese das conclusões obtidas a partir deste trabalho.

Capítulo 2

Fundamentos de Tolerância a Falhas e Sistemas-de-Sistemas

Este capítulo define os conceitos de tolerância a falhas e Sistemas-de-Sistemas utilizados neste trabalho. Na Seção 2.1, apresentamos os conceitos e técnicas referentes à implantação de tolerância a falhas em sistemas computacionais. Tais técnicas são fundamentais para que se consiga desenvolver sistemas confiáveis. Durante este trabalho adotamos as terminologias de tolerância a falhas proposta por Lee e Anderson em [38]. Na Seção 2.2, são descritos os conceitos e problemas encontrados durante a construção de Sistemas-de-Sistemas, levantando as principais características que devem ser contempladas em uma arquitetura de software que se proponha a ser utilizada na construção deste tipo de sistema. E finalmente, a Seção 2.3 faz um resumo de principais conceitos apresentados neste capítulo.

2.1 Conceitos de Tolerância a Falhas

Com a penetração dos computadores em todos os ramos da sociedade moderna, os sistemas computacionais tornaram-se uma ferramenta indispensável, visto os inúmeros benefícios alcançados com seu uso. Tais benefícios tornaram tão imprescindíveis para a sociedade, que os prejuízos proporcionados quando os sistemas computacionais deixam de funcionar ou funcionam incorretamente são tão grandes ou maiores quanto foram os benefícios trazidos com o uso dos computadores [6]. Daí o surgimento do termo *Dependable Systems*[3].

[3] Dependable Systems – Sistemas Confiáveis

Sistemas confiáveis [4] são sistemas que não irão desviar do comportamento esperado pelo projetista diante de falhas de projeto, físicas ou interação com os usuários, ou ainda permitir que vírus ou ataques maliciosos afetem seus serviços essenciais. *Sistemas disponíveis*[5] são sistemas que irão se manter disponíveis para os usuários pela maior parte do tempo. Pode-se aumentar a disponibilidade do sistema, quando se melhora a confiabilidade do sistema ou quando se reduz o tempo em que o sistema permanece indisponível. Tolerância a falhas é a melhor forma de manter a *confiabilidade* e *disponibilidade* destes sistemas, pois muitas vezes é impossível evitar que falhas aconteçam, como por exemplo, falhas decorrentes da deterioração de hardware ou falhas de projeto [38, 24].

2.1.1 Sistema Computacional

Para o entendimento de sistemas tolerantes a falhas é necessário definir o conceito de sistema computacional. Do ponto de vista do seu comportamento externo, um sistema computacional, ou simplesmente sistema, pode ser definido como, qualquer mecanismo que mantém um padrão de comportamento através de uma interface entre o mecanismo e seu ambiente [38]. Logo, qualquer sistema-componente deve ter bem definido quais os serviços providos e requeridos por ele, especificando para os demais sistemas que constituem o ambiente o que é realizado e de quais serviços ele depende. A interface é o ponto de interação de um sistema com o ambiente.

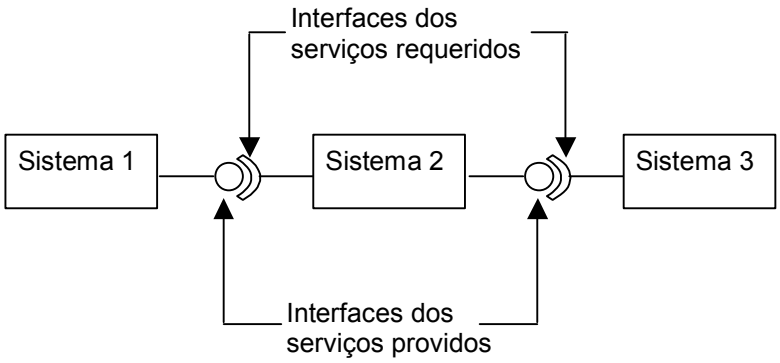

Figura 2. Sistema Computacional do ponto de vista externo

[4] Em inglês: reliable systems.
[5] Em inglês: available systems.

A Figura 2 exibe três sistemas computacionais do ponto de vista externo. O Sistema 2 possui uma interface definindo os serviços que ele requer, neste exemplo esta interface interage com o Sistema 1 responsável por prover tais serviços. Por outro lado, o Sistema 2 possui uma interface que define os serviços providos, que neste caso são utilizados pelo Sistema 3.

Um sistema computacional também pode ser definido em termos de sua estrutura interna, onde um sistema pode ser visto como um conjunto de componentes que interagem sobre o controle de um projeto [38]. O componente de um sistema é um outro sistema com comportamento e estrutura interna próprios, que por sua vez também pode ser formado por sub-componentes. O projeto de um sistema também é um componente que tem a responsabilidade de controlar os demais componentes.

2.1.2 Definições: Falha, Erro e Defeito

As *falhas* são causadas por fenômenos naturais de origem interna ou externa e ações humanas acidentais ou intencionadas. Quando ocorrem falhas no sistema, sua manifestação dá origem aos *erros* que subseqüentemente resultam na origem de *defeitos* no sistema (Figura 3). Um defeito é definido como um desvio do comportamento especificado para o sistema.

Um sistema se encontra em um estado errôneo quando ocorre um erro que leva o sistema a um estado não previsto na sua especificação, neste momento ações corretivas podem ser realizadas com a finalidade de levar o sistema para um estado válido evitando a ocorrência de um defeito.

Figura 3. Falha, erro e defeito

Para ilustrar, considere que durante a programação foi cometido um erro de digitação do programador, trocando um sinal de adição por um de subtração durante a realização de um cálculo. Com este tipo de **falha** o sistema pode cometer um **erro** de

cálculo e acabar provocando um **defeito** no sistema, que vende um produto com o preço errado. Note que uma falha pode provocar nenhum, um ou mais erros, visto que o cálculo pode nunca ser acessado durante a execução do sistema. Um erro não necessariamente conduz a um defeito, visto que o sistema poderia realizar outra forma de cálculo que vendesse o produto com preço correto.

Se um sistema encontra-se num estado errôneo, então um defeito no projeto ou em um de seus componentes ocorreu. Desta forma, o erro em um dos componentes ou no projeto pode levar todo o sistema a ficar num estado errôneo.

2.1.3 Componente Tolerante a Falhas Ideal

Como já discutido, um sistema computacional é definido como um conjunto de componentes que interagem sob o controle de um projeto provendo, assim, novos serviços. Um componente pode, recursivamente, ser formado por sub-componentes. Componentes atendem a requisições de serviços e enviam mensagens de solicitação de serviços a serem atendidas por outros componentes. Por este ângulo, podemos ver um sistema-computacional como uma hierarquia de componentes.

Cada componente pode produzir dois tipos de respostas para uma determinada requisição, respostas normais ou excepcionais (anormais). Respostas normais são produzidas quando o componente realiza o serviço requisitado de forma satisfatória, atendendo as especificações do projeto. Já as respostas excepcionais ocorrem quando o serviço requisitado não é realizado a contento, desviando do comportamento especificado. As respostas excepcionais de um componente são denominadas *exceções*.

Como conseqüência do particionamento das respostas em duas categorias, a atividade de um componente também pode ser particionada em atividade normal e atividade excepcional. A atividade normal de um componente implementa os serviços como especificado para o componente. A atividade excepcional implementa as medidas de tolerância as falhas que causaram a manifestação das exceções quando ocorreu um desvio no comportamento previsto para o componente.

Esta separação entre atividade normal e excepcional leva-nos à definição do componente ideal tolerante a falhas apresentado na Figura 4.

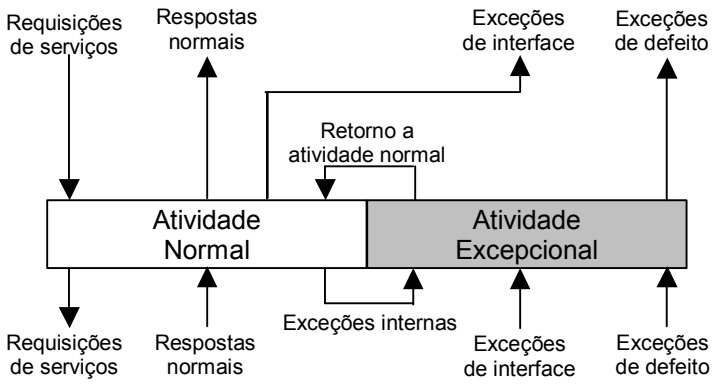

Figura 4. Componente ideal tolerante a falhas

Exceções podem ser classificadas como: (i) exceções internas; (ii) exceções de defeito; e (iii) exceções de interface. As exceções internas ocorrem quando o componente detecta uma situação inesperada durante a atividade normal. Depois que um componente levanta uma exceção interna, ele tenta tratar esta exceção através de mecanismos de tolerância a falhas e retorna para sua atividade normal logo em seguida. Caso não seja possível a recuperação da exceção interna, o componente lança uma exceção de defeito indicando que não foi possível realizar o serviço solicitado. Exceções de interface são lançadas por um componente quando ele recebe solicitações de serviços que não estão especificados para que ele realize. Normalmente, este tipo de exceção é provocado por uma falha no projeto do sistema.

2.1.4 Etapas de Implantação de Tolerância a Falhas

O estudo de sistemas tolerantes a falhas tem o objetivo de estudar formas de prevenir erros e falhas que possam ocasionar um defeito no sistema. O mecanismo dos sistemas para tratamento de falhas consiste de quatro fases [38]: (i) detecção de erro, (ii) confinamento e avaliação do dano; (iii) recuperação de erros; e (iv) tratamento de falhas e continuidade do serviço.

1. **Detecção de erros**: Tem por objetivo identificar os erros do sistema e levantar exceções que sinalizem a existência deles;
2. **Confinamento e avaliação do dano**: Quando um erro é detectado, algum tempo já se passou desde da manifestação da falha. Este tempo para detectar pode levar

muitos outros componentes a serem influenciados pelo resultado incorreto que pode ter sido propagado, ocasionando novos erros nos demais componentes. Desta forma, antes de iniciar a fase de recuperação de erros, é necessário avaliar a extensão do dano causado ao sistema, definindo precisamente as fronteiras do dano;

3. **Recuperação de erros**: Após a identificação do erro e definição de extensão do dano causado ao sistema, é necessário recuperar o estado do sistema, tornando a colocá-lo em um estado livre de erros. A recuperação do erro é responsabilidade do tratador (*handler*) da exceção sinalizada; e
4. **Tratamento de falhas e continuidade do serviço**: Embora o sistema tenha retornado para um estado livre de erros, ainda é necessária uma fase no tratamento de falhas que permita que o sistema volte a sua atividade normal e continue funcionando. Além disso, deve-se garantir que a mesma falha não venha a provocar o mesmo erro no sistema, fazendo-o voltar a um estado errôneo.

2.2 Sistemas-de-Sistemas

O desenvolvimento de sistemas consistindo de sistemas existentes é uma área de pesquisa emergente [22, 42]. Sistemas-de-Sistemas (SdSs)[6] são sistemas complexos construídos a partir de sistemas-componentes já existentes, autônomos, heterogêneos e com objetivos distintos, que devem prover seus serviços individuais mesmo se o SdSs não estiver apto no momento para prover os seus.

Um SdSs se distingue de um sistema baseado em componentes tradicional pelas seguintes características:

- **independência dos sistemas-componentes**, onde cada sistema-componente deve ser capaz de continuar operando independentemente, mesmo quando ele estiver integrado ao SdSs;

[6] Em inglês: Systems of Systems (SoSs)

- **natureza evolutiva do sistema**, onde o desenvolvimento e existência do SdSs são evolutivos, com novas funções e requisitos sendo inseridos, removidos e alterados à medida em que as regras de negócios do sistema vão sendo alteradas;
- **distribuição geográfica dos sistemas-componentes**, que limita as interações entre os sistemas-componentes, obrigando que os componentes troquem apenas informações básicas, evitando que grande quantidade de dados fique trafegando pela rede; e
- **comportamento emergente**, em que as funcionalidades e propósitos dos SdSs não existem em nenhum sistema-componente, mas surgem das interações colaborativas entre os sistemas-componentes.

Durante a realização da arquitetura proposta neste trabalho, foram consideradas todas estas características particulares dos SdSs.

A independência e o tamanho dos sistemas-componentes exigem que na construção de SdSs seja dada especial atenção no projeto das interfaces, visto que os sistemas-componentes são desenvolvidos independentemente dos SdSs. As funcionalidades do SdSs devem surgir somente da interação entre os componentes. Já a arquitetura dos SdSs deve ser uma estrutura que reflita a especificação do padrão de comunicação.

A seguir (Seção 2.2.1) serão descritos alguns problemas encontrados na integração de sistemas-componentes.

2.2.1 Incompatibilidades entre Sistemas-Componentes

Para a criação de SdSs vários problemas distintos dos encontrados nos sistemas tradicionais devem ser solucionados [21]. Embora o enfoque deste trabalho tenha sido principalmente na arquitetura de SdSs, não se atendo aos problemas descritos nesta seção, é importante que o arquiteto de software que irá desenvolver o SdSs tenha em mente todos os problemas a serem enfrentados durante a realização do projeto do sistema.

Alguns problemas que podem ser encontrados durante a integração de sistemas-componentes no desenvolvimento de SdSs são descritos a seguir.

Incompatibilidade Física

A transmissão de bits entre os sistemas-componentes utiliza características mecânicas e elétricas e de codificação que podem variar entre cada sub-sistema;

Incompatibilidade Sintática

Erros sintáticos são causados por estruturas de informações incompatíveis entre os sistemas-componentes. Tal tipo de incompatibilidade pode impedir que informações enviadas por um sistema-componente deixem de ser interpretadas ou sejam interpretadas erroneamente pelo sistema-componente receptor. Isto inclui questões como formatos de dados (número de bits que representam um valor inteiro) e ordem de bits e bytes (bit mais significativo);

Incompatibilidade de Temporização

A partir do momento que um dos sistemas-componentes necessita de propriedades temporais bem específicas que sigam um determinado protocolo, o fator tempo passa a ser vital. Este tipo de situação é bem comum quando um dos sistemas-componentes funciona em tempo real. Isto obriga que todos os serviços oferecidos pelos demais sistemas-componentes obedeçam ao protocolo que rege o funcionamento deste sistema. Já em sistemas-componentes que não funcionam em tempo real, a temporização dos serviços oferecidos deverá respeitar os limites de *timeouts*.

Controle de Concorrência, Consistência de Dados, Distribuição e Tolerância a Falhas

Um SdSs é um sistema heterogêneo em vários aspectos inclusive no que diz respeito a tolerância a falhas, pois os seus sistemas-componentes são desenvolvidos separadamente utilizando diferentes (ou nenhuma) abordagens na recuperação de erros. No entanto, o SdSs deve ser apto a lidar com situações onde diversos sistemas-componentes manifestam comportamentos considerados anormais (falhos) pelo o SdSs, com o objetivo de atender aos requisitos do SdSs.

Durante o desenvolvimento de SdSs pode-se utilizar, entre os sistemas-componentes, relacionamentos independentes, competitivos e/ou cooperativos (Seção 4.1) com vários sistemas-componentes realizando serviços coletivamente a fim de realizarem uma determinada tarefa. A escolha das técnicas de tolerância a falhas e de uma

estruturação de software a serem empregadas dependem essencialmente da identificação destes relacionamentos. Em cenários mais realistas, SdSs são aplicações muito sofisticadas que lidam com problemas complexos em controle de concorrência, integridade e consistência de dados, paralelismo, distribuição e tolerância a falhas.

Um exemplo disto ocorre na integração de diversos sistemas de venda de várias lojas de um shopping center, onde se deseja que todas as lojas dêem um desconto único durante uma promoção relâmpago com início e término pré-determinados. Se no instante de início da promoção, uma das lojas não estiver apta a realizar a promoção, ou uma das lojas finalizar a promoção antes do previsto, vários aspectos deverão ser tratados, tais como: (i) **concorrência**, a fim de evitar que duas promoções distintas sejam realizadas sobre o mesmo produto num dado instante; (ii) **consistência de dados**, para que uma loja não apresente descontos diferenciados, quando se deseja uma promoção simultânea em todas as lojas; (iii) **distribuição**, para que o início e o fim da promoção entre as lojas sejam sincronizados; e (iv) **tolerância a falhas**, que deverá recuperar os sistemas de todas as lojas caso uma das lojas apresente erro e saia prematuramente da promoção.

Incompatibilidade na Representação de Dados

Este problema atinge a comunicação entre os sistemas-componentes que utilizam estruturas de dados distintas. Enquanto que num sistema-componente orientado a objetos um produto é representado como um objeto, em outro sistema-componente o mesmo produto pode ser representado simplesmente como uma única tabela no banco de dados. Esta desconformidade obrigará que haja um tratamento [62] para que os sistemas-componentes consigam compreender a representação dos dados quando houver uma troca de informação.

Incompatibilidade de Padrões de Interação

As interfaces podem ser classificadas de acordo com o tipo de interação que elas permitem, podendo ser: (i) *information-pull;* ou (ii) *information-push*. Denominam-se de *information-pull* todas interfaces as quais a transferência de dado é iniciada a partir do pedido do receptor. Os mecanismos mais comuns são: (i) variáveis compartilhadas; (ii) leitura em arquivos; (iii) *stream socket*; (iv) *polling*; e (v) busca em banco de dados. Denominam-se de *information-push* todas interfaces as quais a transferência de dados e

controle é iniciada a partir do pedido do emissor. Os mecanismos mais comuns são: (i) chamada de procedimento remoto (RPC[7]); (ii) mensagem de evento; (iii) escrita em arquivo; (iv) *message socket*; (v) atualização em banco de dados; (vi) interrupções; e (vii) exceções. Sendo as duas últimas apenas transferência de controle.

Na criação de um SdSs, os sistemas-componentes podem possuir interfaces distintas, sendo necessário o emprego de uma técnica de adaptação de componentes [62] adequado, para tratar tais problemas.

Incompatibilidade no Controle de Fluxo

Existem dois tipos básicos de controle de fluxo: (i) controle de fluxo explícito; e (ii) controle de fluxo implícito. No controle de fluxo explícito, o receptor após receber uma mensagem com sucesso envia uma mensagem de reconhecimento ao emissor. Já no controle de fluxo implícito, receptor nunca envia um reconhecimento da mensagem, visto que tanto o receptor quanto o emissor da mensagem sempre confiam no canal de comunicação.

Incompatibilidades na integração entre sistemas que utilizem controles de fluxo é inevitável quando se integram diversos sistemas-componentes, visto que um sistema que utiliza um controle de fluxo explícito poderá ficar a espera de uma mensagem de reconhecimento, quando na realidade o outro sistema utiliza um controle de fluxo implícito e não enviar a mensagem de reconhecimento necessária ao outro.

Incompatibilidade na Resolução de Nomes

Resolução de nomes implica em associar um identificador a uma entidade (objetos, estruturas, tabelas) num determinado contexto. Ao associar sistemas-componentes num SdSs, que não foram projetados para funcionarem juntos, tais questões devem ser verificadas: (i) compartilhamento e movimento de entidades entre sistemas que possuem projetos de identificadores diferentes; e (ii) consistência de objetos replicados com identificadores distintos.

Enquanto que num sistema-componente a representação de um produto é identificada pelo seu código de barras, em outro o processo de identificação pode utilizar o

[7] RPC: Remote procedure call

número de referência do fabricante. Isto obrigará a realização de uma tradução dos identificadores sempre que uma mensagem enviada entre os sistemas-componentes referenciar um determinado produto do outro sistema-componente.

Incompatibilidade Semântica

Em sistemas-componentes construídos separadamente, pode-se encontrar problemas devido a diferentes significados associados a um único nome. Por exemplo, enquanto que em um sistema de uma loja assume-se que a informação sobre tipo de produto diz respeito à seção e subseção em que o produto se enquadra, no sistema de outra loja o tipo de produto apenas indica se o produto é perecível, ou não.

Incompatibilidade Pragmática

O problema de pragmática ocorre quando o mesmo nome é usado para diferentes propósitos por diferentes sistemas-componentes. Um exemplo disto está no preço de venda de um produto, onde tanto no sistema de loja, quanto no sistema do fornecedor possuem a mesma semântica, embora sabemos que o preço de venda do fornecedor não é o mesmo de venda da loja.

2.3 Considerações Finais

Este capítulo foi dividido em duas partes. A primeira abordou sobre alguns conceitos fundamentais de tolerância a falhas e a segunda sobre o conceito de Sistemas-de-Sistemas (SdSs). Todos estes conceitos foram utilizados no restante desta dissertação.

Dentro do que foi descrito sobre tolerância a falhas, iniciamos com a definição de *sistema computacional*, que pode ser visto como um mecanismo que mantém um padrão de comportamento através de uma interface, ou como um conjunto de componentes que interagem sobre o controle de um projeto. Nesta segunda definição, um componente de um sistema é outro sistema que pode ser formado por sub-componentes e assim sucessivamente.

Neste trabalho adotamos a terminologia proposta por Lee e Anderson [29], que define o que é um componente tolerante a falhas e a diferença entre falha, erro e defeito. Por fim, foram definidas as quatro fases do mecanismo para tratamento de falhas, que são

descritas como: (i) detecção de erro; (ii) confinamento e avaliação do dano; (iii) recuperação de erros; e (iv) tratamento de falhas e continuidade do serviço.

Na subseção relativa a SdSs, definimos as características principais que devem e foram consideradas na abordagem arquitetural proposta nesta dissertação, são elas: independência dos sistemas-componentes, natureza evolutiva do sistema, distribuição geográfica dos sistemas-componentes e comportamento emergente.

Por fim, foi apresentado uma lista com os principais problemas de incompatibilidade entre os sistemas-componentes. Apesar desta dissertação ter seu enfoque principal na arquitetura de SdSs, não se atendo especificamente aos problemas de incompatibilidades, acreditamos que seja importante que o arquiteto de software tenha conhecimento sobre estes problemas durante a elaboração do SdSs.

Capítulo 3

Princípios de Estruturação de Software

Neste capítulo serão apresentados alguns princípios de estruturação de software para a construção de sistemas flexíveis, reusáveis e robustos. Será definido o que é desenvolvimento de software baseado em componentes (Seção 3.1), visto que toda esta dissertação é baseada neste processo de construção de software.

A Seção 3.2 expõe o conceito de arquitetura de software, onde apresentamos o estilo arquitetural C2 utilizado na Arquitetura FT-SC3 descrita no Capítulo 5. O conceito de *framework* orientado a objetos é apresentado na Seção 3.3.

Na Seção 3.4 apresentamos o conceito de contratos de coordenação, que foi um dos conceitos utilizados para a construção da Arquitetura FT-SC3, com o objetivo de facilitar a evolução do sistema e coordenar as atividades dos componentes do sistema.

E por último, na Seção 3.5, apresentamos algumas considerações finais onde fazemos um breve resumo deste capítulo.

3.1 Desenvolvimento Baseado em Componentes

O desenvolvimento de sistemas baseado na interação de partes pré-existentes de software, com o objetivo de reduzir tempo de desenvolvimento, gerenciar complexidade, melhorar qualidade e distribuir o desenvolvimento de sistemas [12], sempre despertou grande interesse, motivando vários estudos relacionados com modularização de sistemas, estruturação de projetos e mais recentemente com orientação a objetos. O desenvolvimento de software baseado em componentes é um processo de construção de aplicações criadas a partir do reuso de componentes de software, que em muitos casos já foram especificados, construídos e testados [65].

Componentes são unidades independentes de software que encapsulam seu projeto e implementação [65]. Cada componente pode ser desenvolvido, testado e implantado

separadamente, podendo ser combinado a outros componentes, construindo novos componentes de maior porte. Para isto, ele deve possuir uma definição clara das funcionalidades oferecidas e requeridas, que são disponibilizadas através de sua interface. A interface de um componente define as funcionalidades do componente sem revelar os detalhes do projeto e implementação, facilitando a substituição de componentes por outro de especificação equivalente, sem afetar a conexão e a funcionalidade disponível. Isto é possível porque a conexão de componentes ocorre exclusivamente através de sua interface.

3.2 Arquitetura de Software

A arquitetura de software tem o objetivo de guiar todas as etapas do processo de desenvolvimento de sistemas de software complexos e de grandes proporções [7]. Uma arquitetura de software define a estrutura dos componentes de um sistema, seus conectores, e suas configurações, que definem os princípios e diretrizes que guiam o projeto e sua evolução ao longo do tempo [7].

A definição de uma arquitetura de software é fundamental para que se consiga atingir as qualidades necessárias para o sistema. Dentre os atributos de qualidade necessários para criação de SdSs, destacamos:

- **Manutenibilidade**: característica que define a capacidade do sistema de se modificar, seja adicionando ou alterando algum requisito do sistema;
- **Reusabilidade:** característica que define a facilidade de reutilizar componentes já desenvolvidos, objetivando minimizar o esforço na criação ou manutenção de um sistema. Quanto mais genérico e independente da aplicação for um componente, mais fácil será o reuso deste componente em outras aplicações;
- **Tolerância a Falhas**: capacidade do sistema de prevenir erros e falhas que possam ocasionar um defeito no sistema, sendo capaz de se recuperar diante de situações excepcionais; e
- **Testabilidade**: característica que define a facilidade de realizar testes num sistema de software.

3.2.1 Estilo Arquitetural C2

O estudo de arquitetura de software tem revelado certas estruturas recorrentes usadas para guiar a construção de sistemas com problemas específicos [19]. Estas estruturas podem ser identificadas em termos de tipos de componentes, tipos conectores e dos padrões de interação entre componentes e conectores. Muitas destas estruturas foram identificadas e classificadas como um estilo arquitetural. Além de definir os componentes e conectores usados na arquitetura, um estilo arquitetural define as regras e restrições sobre a combinação dos componentes arquiteturais.

O estilo arquitetural C2 é um estilo baseado em componentes visando uma vasta reutilização e flexibilidade na composição de sistemas, enfatizando a ligação fraca entre seus componentes [59]. Este estilo arquitetural é baseado em componentes, conectores e num conjunto de princípios que regem a composição destes. Cada componente e conector possuem duas interfaces, a superior e a inferior, onde os componentes são dispostos em camadas. A interface superior de um componente sempre se conecta a interface inferior de um único conector (Figura 5) e a interface inferior de um componente sempre se conecta à interface superior de outro único conector. Isto é, componentes não podem ser conectados diretamente, apenas através de conectores. Cada interface (superior ou inferior) de um conector pode ser conectada a qualquer quantidade de componentes e conectores. Quando dois conectores se conectam, esta ligação deve ser da interface inferior de uma para a interface superior da outra.

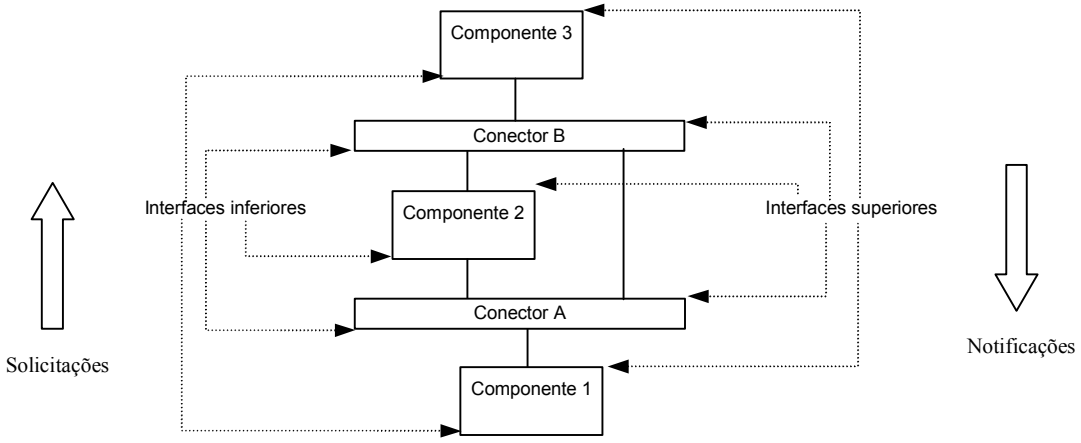

Figura 5. Elementos básicos do estilo arquitetural C2

Os componentes deste estilo possuem um estado interno e são totalmente independentes uns dos outros, permitindo a integração de componentes heterogêneos. Estes componentes se comunicam através de mensagens assíncronas que são roteadas, distribuídas e filtradas pelos conectores.

Existem dois tipos de mensagens no estilo C2: solicitações e notificações. Por convenção, um componente sempre que deseja solicitar algum serviço envia uma mensagem de solicitação através de sua interface superior. O componente, que recebe esta solicitação do conector, atende-a e retorna uma notificação ao conector através de sua interface inferior. O componente que enviou a solicitação ao receber a notificação poderá reagir com a invocação de um de seus serviços internos, visto que sua solicitação já foi atendida com sucesso.

Com a finalidade de apoiar o desenvolvimento de software baseado no estilo C2, foi desenvolvido o Framework C2.FW [43, 60] orientado a objetos. Este *framework* provê uma infraestrutura para construção de aplicações C2. A versão Java [30] do Framework C2.FW compreende um conjunto de classes e interfaces que implementam as abstrações do estilo arquitetural C2, como componentes, conectores, mensagens e conexões. Além da versão em Java o Framework C2.FW também foi implementado em C++, Python e Ada.

Na Seção 6.1 descrevemos uma extensão do Framework C2.FW para apoiar a composição de sistemas-componentes durante a criação de um sistema tolerante a falhas.

3.3 Framework Orientado a Objetos

Um *framework* orientado a objetos (*framework OO*) tem a finalidade de implementar parcialmente uma arquitetura de software, na forma de uma coleção de componentes de software ou classes que podem ser adaptados ou instanciados com o objetivo de criar um sistema de software [62]. O *framework OO*, ou simplesmente *framework*, se diferencia das demais abordagens para reutilização de software por não se restringir ao reuso de trechos de códigos isolados, mas a toda arquitetura de software desenvolvida para um domínio específico. Enquanto as abordagens para reuso de código se limitam à construção de bibliotecas de software, sendo a aplicação responsável por todo

fluxo de controle da aplicação, nos *frameworks* se reutiliza o fluxo de controle já embutido no próprio *framework*, contendo muito da complexidade da interação entre os objetos.

Os *frameworks* permitem reutilizar tanto componentes caixas brancas, quanto componentes caixas pretas. Um componente caixa branca [19] é um componente abstrato construído parcialmente que atua como ponto adaptável, onde o desenvolvedor é responsável por personalizar o componente para se adequar às características específicas da sua aplicação. Para isto, o desenvolvedor utiliza mecanismos orientados a objetos para implementar as interfaces e/ou sobrecarregar os métodos do componente caixa branca. Um componente caixa preta é um componente ou classe desenvolvida que somente pode ser estendida e configurada pelos desenvolvedores, neste caso, o componente caixa preta é mais difícil de ser construído, pois deve disponibilizar todas as alternativas para adaptá-lo à nova aplicação. Entretanto, o reuso de componentes caixa preta torna-se mais simples, visto que o desenvolvedor não necessita de maiores conhecimentos sobre a estrutura interna.

3.4 Contratos de Coordenação

Um contrato de coordenação [1, 4] é uma conexão que pode ser estabelecida entre um grupo de objetos, onde novas regras e restrições são impostas ou sobrepostas no comportamento conjunto destes objetos, criando novas formas específicas de interação e especificação para atender os novos requisitos [2]. Cada contrato de coordenação descreve um conjunto de atividades de coordenação que serão impostas às partes com as quais ele interagirá quando o seu gatilho de ativação for acionado ao serem detectadas determinadas condições no sistema. Um gatilho de ativação de um contrato pode ser uma condição sobre os estados dos objetos participantes da interação, um determinado pedido de serviço, ou um evento ocorrido por um ou mais participantes.

Contratos de coordenação podem ser aplicados sobre objetos simples, com a finalidade de regular ou adaptar o modo como os objetos se comportam, tornando-os aptos para satisfazerem novos requisitos, como por exemplo, criar um participante confiável a partir de um objeto que oferece serviços não confiáveis. Quando o contrato de coordenação é aplicado sobre vários objetos participantes, este passa a se comportar como um agente de sincronização que coordena o modo como os objetos participantes interagem, impondo as

restrições necessárias para obter os requisitos desejáveis para o comportamento conjunto dos objetos participantes. Na descrição de um contrato de coordenação, entende-se por participante um número de interfaces que podem ser instanciadas por implementações concretas quando o contrato de coordenação for ativado por uma determinada configuração.

Na maior parte dos sistemas, a natureza das modificações que ocorrem no domínio de negócio [14] não está no nível do componente que modela as entidades básicas, mas nas regras que regulam as interações entre estas entidades e nas políticas que governam a aplicação das regras negócio [63].

Com a finalidade de alcançar um alto grau de agilidade, onde novos serviços sejam inseridos ou personalizados rapidamente [40], foi realizada uma clara separação dos sistemas em quatro camadas que definam claramente: (i) os componentes, que realizam as atividades básicas do sistema; (ii) os contratos de coordenação, que possuem as regras de negócio do sistema; (iii) as regras que regem as políticas de negócio; e (iv) os contextos onde tais políticas são utilizadas. Esta separação deu origem às quatro camadas descritas por Wermelinger [64]:

- A **camada computacional** provê os tipos de componentes que realizam os serviços básicos do sistema. Cada componente oferece os serviços que alteram ou consultam o estado do componente.
- A **camada de coordenação** provê os tipos de contratos de coordenação que regulam as interações entre os componentes da camada computacional. Cada contrato de coordenação não pode ter conhecimento sobre os demais, não sendo permitido, portanto, um contrato de coordenação invocar explicitamente os serviços oferecidos pelos demais contratos de coordenação;
- A **camada de configuração** oferece operações de reconfiguração sobre as invariantes globais da configuração do sistema. Cada uma destas operações corresponde a um roteiro de reconfiguração que informa como e quais contratos deverão ser utilizados para realizar uma determinada operação. Esta camada também define explicitamente quais invariantes a configuração deve manter durante a evolução do sistema, permitindo, por exemplo, definir os conflitos e dependências entre as regras de negócios implementadas pelos contratos de coordenação; e

- A **camada de contexto** define um conjunto de contextos, cada um para uma determinada forma de interação do sistema. Cada contexto possui um conjunto de serviços de negócio, onde cada um destes serviços é implementado utilizando a camada de configuração, para consultar ou alterar a configuração atual do sistema, além dos componentes e contratos de coordenação existentes respectivamente nas camadas computacional e de coordenação. Contextos impõem restrições sobre as políticas, regras e invariantes declaradas sobre as configurações, contratos de coordenação e componentes, além de refletir as propriedades que são específicas para um contexto de negócio.

Durante este trabalho, focamos principalmente no desenvolvimento das camadas computacional e de coordenação sem previsão de realização de configuração dinâmica, tendo em vista que as questões detalhadas de configuração e contextos fugiam do escopo previsto. Apesar de não termos tratados destas questões, nós acreditamos que novos trabalhos possam ser realizados com esta finalidade (Seção 8.3).

O uso de contrato de coordenação torna os sistemas mais flexíveis, pois permite que novos requisitos reflitam as modificações nas regras de negócio com a simples inclusão, alteração ou remoção de contratos de coordenação. A divisão dos sistemas em quatro camadas permite que os participantes fiquem isolados das regras de negócio do sistema, que na sua grande maioria passam a ser implementadas nos contratos de coordenação. Isto torna os participantes mais simples, facilitando que se encontre e reuse componentes previamente desenvolvidos que atendam os requisitos dos participantes.

Devido às vantagens citadas acima, refletimos na Arquitetura FT-SC3 (Capítulo 5): (i) as camadas de componente e cooperação descrita nesta seção; e (ii) o uso de contratos que permitem coordenar vários componentes.

3.5 Considerações Finais

Neste capítulo introduzimos as técnicas orientadas a objetos em estruturação de sistemas de software flexíveis e reusáveis. Na Seção 3.1, é definido o conceito de desenvolvimento de software baseado em componentes. A Seção 3.2 apresenta a importância da elaboração de uma arquitetura de software que defina a estrutura dos seus

componentes, conectores e configurações, para que se construa sistemas com as qualidades desejáveis. Quando certas estruturas de arquitetura de software são utilizadas recorrentemente para solucionar determinados problemas, estas estruturas depois de identificadas passam a ser denominadas de estilo arquitetural. Dentro deste estudo, apresentamos o estilo arquitetural C2 que tem o objetivo de obter maior reutilização e flexibilidade na composição de sistemas, graças ao desconhecimento dos componentes dos demais, facilitando a integração entre componentes heterogêneos. Este estilo arquitetural é utilizado pela Arquitetura FT-SC3 proposta no Capítulo 5.

Um *framework* orientado a objeto (ver Seção 3.3) implementa partes comuns de uma arquitetura de software, possibilitando que novos sistemas reutilizem componentes caixas brancas e componentes caixas pretas já implementados.

Na Seção 3.4 apresentamos o conceito de contratos de coordenação, que contempla a evolução do sistema além de coordenar as atividades dos componentes que compõem o sistema. A divisão dos sistemas baseados em contratos de coordenação em camadas (camada computacional, camada de coordenação, camada de configuração e camada de contexto) traz uma série de vantagens no tocante a reuso e evolução do sistema. As vantagens obtidas com a divisão do sistema em uma camada de componentes e outra de cooperação fizeram-nos acreditar que seria interessante refletirmos tais camadas na Arquitetura FT-SC3, que também utiliza o conceito de contratos de coordenação.

Capítulo 4

Ações Atômicas Coordenadas

O conceito de ações atômicas coordenadas foi introduzido por Xu et al [66] como um mecanismo para estruturação de sistemas concorrentes tolerantes a falhas, que unifica um esquema para coordenação de atividades concorrentes e apóia a recuperação de erros por avanço e retrocesso entre múltiplos objetos distribuídos em um sistema orientado a objetos.

Ações atômicas coordenadas provêem um arcabouço para tratar com diferentes tipos de concorrência e alcançar tolerância a falhas a partir da extensão e integração de dois conceitos complementares – conversação [50] e transação atômica [36]. Do ponto de vista externo uma ação atômica coordenada se comporta como uma transação atômica, visto que ela pode garantir as propriedades ACID (atomicidade, consistência, isolação e durabilidade) [49].

A recuperação de erros por avanço em uma ação atômica coordenada é apoiada por meio de um mecanismo de tratamento de exceções e um esquema de resolução de exceções que levam em consideração a possibilidade de múltiplas exceções serem lançadas concorrentemente pelos participantes. Ações atômicas coordenadas ainda integram concorrência competitiva e cooperativa, facilitando a construção de sistemas confiáveis complexos.

Na Seção 4.1, definimos os três tipos de concorrências (independente, competitiva e cooperativa) existentes. Nas Seções 4.2 e 4.3, são apresentadas respectivamente as principais propriedades de ações atômicas coordenadas e como é realizado o aninhamento de ações atômicas coordenadas. A Seção 4.4 explica como ocorre a parametrização, ligação (*binding*) e instanciação de uma ação atômica coordenada. A Seção 4.5 mostra porque acreditamos que ações atômicas coordenadas são apropriadas para estruturação de SdSs. E finalmente, na Seção 4.6 é feito um breve resumo, além de expor onde ações atômicas coordenadas foram úteis para nós.

4.1 Tipos de Concorrência

No conceito de ações atômicas coordenadas, um sistema de software é visto como um conjunto de participantes que interagem entre si [67]. Em geral participantes podem utilizar diferentes objetos, os quais podem, ou não, estar sendo executados concorrentemente em diferentes locais de um sistema distribuído. A concorrência entre os participantes pode ser classificada em três categorias distintas, são elas: (i) concorrência independente; (ii) concorrência competitiva; e (iii) concorrência cooperativa. A concorrência é independente se o conjunto de objetos acessados pelos participantes são disjuntos, este caso é o mais trivial de todos visto que os participantes não interagem diretamente entre si. A concorrência competitiva [52] acontece quando participantes concorrentes, projetados para realizar trabalhos distintos, acessam um conjunto comum de objetos de forma não ordenada, ocorrendo uma competição pelos objetos. Por fim, uma concorrência cooperativa acontece quando vários participantes são projetados coletivamente e invocados concorrentemente para realizarem uma tarefa pré-definida. Neste último caso, o acesso ao conjunto de objetos comuns aos participantes é ordenado. Estes três tipos de concorrência podem co-existir em sistemas distribuídos de grande porte.

4.2 Propriedades de Ações Atômicas Coordenadas

Ações atômicas coordenadas oferecem um mecanismo para realizar um grupo de operações atômicas sobre um conjunto de objetos. Estas operações são realizadas cooperativamente por um ou mais *papéis* executados paralelamente dentro de uma ação coordenada. Um papel é uma interface que define o comportamento de um participante quando este estiver fazendo parte de uma ação atômica coordenada. Neste caso pode-se comparar um papel como a personagem assumida por uma pessoa (participante) quando esta estiver fazendo parte de uma peça teatral (ação atômica coordenada).

O primeiro passo para a realização de uma ação atômica coordenada é definir um conjunto de participantes que irão executar os papéis necessários para a realização da ação. Os múltiplos participantes dentro da ação atômica coordenada comunicam-se entre si através de objetos locais, que são puramente internos à ação atômica coordenada e são usados para apoiar concorrência cooperativa.

Dentro de uma ação atômica coordenada, os efeitos de qualquer operação que os participantes realizam em objetos externos não são visíveis para outras ações atômicas coordenadas ou participantes de outras ações, até que a ação atômica coordenada termine. Ações atômicas coordenadas estendem o conceito de transação, para permitir não apenas recuperação por avanço, mas também por retrocesso.

Uma ação atômica coordenada pode retornar: (i) um retorno normal, quando a ação foi finalizada com sucesso; (ii) um retorno excepcional (ou degradado), quando a ação foi realizada com sucesso, mas com ressalvas; (iii) uma de aborto, quando a ação não é realizada, mas os participantes são deixados num estado livre dos efeitos da ação; ou (iv) uma exceção de falha, quando a ação não é concluída e os participantes são deixados num estado errôneo.

Durante uma ação atômica coordenada, um ou mais participantes da ação podem levantar uma exceção. Se esta exceção não puder ser tratada localmente pelo participante, então um esquema de resolução de exceções é usado para combinar estas exceções em apenas uma, denominada de exceção resolvida [27, 28]. A exceção resolvida é propagada para todos participantes envolvidos na ação, que invocam um mecanismo de recuperação de erro. Estes ainda podem finalizar a ação atômica coordenada com sucesso usando uma recuperação de erro por avanço. Outra alternativa é a realização de uma recuperação de erro por retrocesso, que desfaz todos os efeitos da ação atômica coordenada, voltando o sistema para o mesmo estado em que ele se encontrava antes do início da ação.

4.3 Aninhamento de Ações Atômicas Coordenadas

Ações atômicas aninhadas podem ser utilizadas recursivamente, como exibido na Figura 6, de forma a permitir a existência de sub-ações dentro de uma outra ação atômica coordenada [49]. Assim como numa transação atômica, um efeito de uma ação atômica coordenada aninhada somente se torna permanente quando a ação externa é finalizada. Entretanto, os efeitos de uma ação atômica coordenada aninhada se tornam visíveis para a ação externa logo após a ação aninhada tenha sido finalizada. Quando uma sub-ação é finalizada, a responsabilidade pela recuperação dos erros fica a cargo da ação que a engloba.

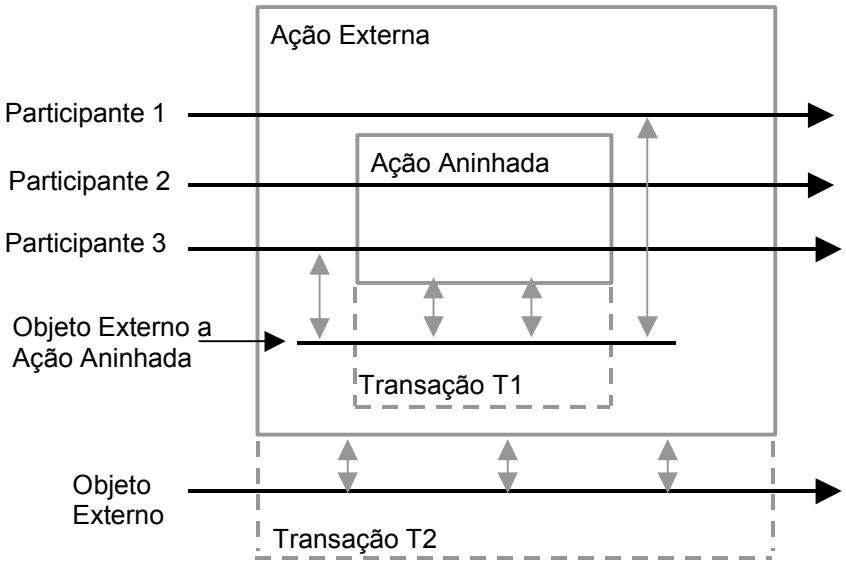

Figura 6. Ação Atômica Coordenada Aninhada

Se uma ação atômica coordenada aninhada interage com objetos que são externos a ela e internos à ação externa, então estas interações devem parecer atômicas com respeito às demais ações atômicas coordenadas aninhadas e a ação atômica coordenada que as engloba. Ou seja, isto significa que ações atômicas coordenadas aninhadas devem abrir uma transação sobre os objetos externos as ações aninhadas, um exemplo disto é a Transação T1 da Figura 6.

4.4 Parametrização, Ligação e Instanciação de uma Ação Atômica Coordenada

Uma instância de uma ação atômica coordenada somente ocorrerá se puder ser encontrado um participante para preencher cada papel especificado para a ação. Em outras palavras, a instância de uma ação atômica coordenada somente será executada quando o número apropriado de participantes, capazes de realizarem todos papéis da ação, estiver fazendo parte desta ação, cada um realizando um papel diferente dentro da mesma ação atômica coordenada.

O mecanismo para chamar uma ação atômica coordenada envolve os seguintes passos:

1. Algum agente é responsável por identificar a necessidade para chamar uma ação atômica coordenada;
2. O agente é responsável por convocar os vários participantes que irão executar os papéis especificados para a ação atômica coordenada; e
3. Os vários participantes confirmam sua participação na ação atômica coordenada, indicando que ele não está participando de outra ação atômica coordenada, estando, portanto, disponível para entrar em uma nova ação.

Existem duas formas de parametrizar uma ação atômica coordenada: (i) quando uma chamada para a ação é instanciada; e (ii) quando um participante confirma que está disponível para atuar em um papel dentro desta chamada. Assim, é razoável, que possa existir parâmetros para a ação atômica coordenada e para cada papel dentro da ação atômica coordenada.

4.5 Ações Atômicas Coordenadas e Sistemas-de-Sistemas

Tomando por base o trabalho de Beder [8], onde foi proposta uma arquitetura de software para o desenvolvimento de sistemas concorrentes confiáveis, foi verificado que ações atômicas coordenadas servem como uma base sólida na estruturação apropriada para SdSs confiável, devido à sua habilidade para [20]:

- Permitir projetar, estruturar e prover SdSs com tolerância a falhas em que sistemas-componentes cooperam e competem;
- Prover suporte para tratamento de exceções, que é vital para sistemas que não são capazes de realizar recuperação por retrocesso; e
- Prover suporte para manter consistência e tolerância a falhas.

Adicionalmente a estas vantagens, ação atômica coordenada permite-nos solucionar problemas de integração de SdSs melhor que os sistemas convencionais de fluxo de trabalho, pois:

- Oferece uma solução mais poderosa para construir atividades de longa duração;

- Pode servir como uma abordagem para construir SdSs de diferentes tipos e em diferentes domínios de aplicação (incluindo sistemas de tempo real [44], Internet, etc);
- Permite integração de SdSs em termos de sistemas-componentes participantes, ao invés de atividades (permitindo estruturar SdSs recursivamente);
- É mais adequada para as tecnologias de criação de sistemas baseados em componentes pré-existentes (tais como CORBA e EJB), uma vez que ela apóia o desenvolvimento em termos de componentes e processos, e não de atividades; e
- Permite que o mesmo paradigma seja usado em todos os níveis do desenvolvimento e integração dos sistemas.

4.6 Considerações Finais

Neste capítulo apresentamos a definição de ações atômicas coordenadas como um mecanismo para estruturação de sistemas concorrentes tolerantes a falhas, capazes de tratar com diferentes tipos de concorrência, além de permitir recuperação de erro por avanço e retrocesso.

Na Seção 4.1 apresentamos os três tipos de concorrência existentes: (i) concorrência independente, quando os participantes não interagem entre si nem utilizam recursos comuns; (ii) concorrência competitiva, quando os participantes competem pela utilização de um recurso comum; e (iii) concorrência cooperativa, quando os participantes, apesar de invocados concorrentemente, realizam uma tarefa coletiva.

A Seção 4.2 descreve as propriedades de uma ação atômica coordenada, em que apresenta o mecanismo que realiza um grupo de operações atômicas sobre um grupo de objetos. Cada participantes desempenha um determinado papel dentro de uma ação atômica coordenada. Para iniciar uma ação coordenada os participantes devem entrar e sair sincronizadamente da transação. Em SdSs esta propriedade de ação atômica coordenada deve ser relaxada, devido à independência dos sistemas-componentes, que na maioria dos casos, não podem ficar a espera do fim de uma ação, não sendo possível sincronizar a saída dos participantes durante uma ação atômica coordenada. Ação atômica coordenada ainda oferece um mecanismo de tratamento de exceções que é capaz de resolver todas as

exceções levantadas pelos participantes, combinando-as e gerando uma *exceção resolvida* que é propagada aos participantes envolvidos na ação. Este mecanismo de tratamento de exceções foi adaptado e utilizado na Arquitetura FT-SC3 (Capítulo 5).

A Seção 4.3 mostra como ações atômicas podem ser aninhadas, o que nos serviu de base para desenvolver uma arquitetura que também permitisse aninhamento de ações.

A Seção 4.4 explica como é realizada a sincronização da entrada dos participantes na ação atômica coordenada, que somente se inicia quando se encontra um participante para cada papel da ação, sincronizando a entrada dos participantes. A parametrização de uma ação atômica coordenada pode ser feita na instanciação de uma ação ou quando cada participante confirma sua entrada na ação.

Por fim, a Seção 4.5 apresenta algumas considerações que nos levaram a crer que o uso de ações atômicas coordenadas seria útil para a construção de uma arquitetura de software para a composição de sistemas-componentes que produza sistemas confiáveis, extensíveis e com alto grau de reuso.

Capítulo 5

Arquitetura FT-SC3

Este capítulo propõe uma arquitetura de software para composição tolerante a falhas de sistemas-componentes, baseada em contratos de coordenação (Seção 3.4) que inclui um esquema de tratamento de exceções (Seção 2.1) baseado numa extensão do conceito de ações atômicas coordenadas (Capítulo 4). Esta arquitetura passou a ser denominada de Arquitetura FT-SC3[8], devido ao acrônimo em inglês para Composição de Sistemas Tolerante a Falhas utilizando Contratos de Composição.

O uso de contratos de coordenação flexibiliza a evolução dos sistemas, visto que permite a inclusão e adaptação de novas regras de negócios à medida que o sistema evolui. Caso um contrato de coordenação seja criado para impor regras sobre mais de um participante, ele passa a funcionar como um coordenador que sincroniza a atuação de cada participante na realização de uma atividade.

Devido às vantagens apresentadas na Seção 4.5, resolvemos utilizar o conceito de ações atômicas coordenadas que se mostrou um mecanismo bastante adequado para a composição de sistemas-componentes. Introduzimos no contrato de coordenação parte das funcionalidades de coordenação, tal como no coordenador de uma ação atômica coordenada desenvolvido por Beder em [9]. Entre estas funcionalidades foi incluído um esquema de tratamento de exceções baseado no conceito de ação atômica coordenada que foi relaxada do conceito original apresentado no Capítulo 4, pois na composição de sistemas-componentes os participantes são totalmente independentes, podendo existir um objeto externo que seja acessado por mais de um sistema-componente participante desprovido de propriedades transacionais, como seria obrigatório no conceito original de ação atômica coordenada. Este tipo de restrição impede a realização de uma recuperação por retrocesso,

[8] FT-SC3 - **F**ault-**T**olerant **S**ystem **C**omposition using **C**oordination **C**ontracts

obrigando a realização de uma recuperação por avanço que confine todo dano causado ao sistema.

Como descrito por Romanovsky [20], o uso de ações atômicas coordenadas fica limitado na composição de sistemas-componentes, pois cada sistema-componente pode não ter conhecimento de sua participação no SdSs. Devido à independência dos sistemas-componentes, não é permitido nenhum tipo de sincronização adicional a ser imposta sobre a execução dos sistemas-componentes para o mundo exterior.

Tomando por base o que foi feito por Romanovsky [51], utilizamos uma versão relaxada de ações atômicas coordenadas, visto que não poderíamos impor a entrada e a saída sincronizada dos participantes quando os mesmo se tratam de componentes independentes e autônomos. Isto obriga a realização de uma recuperação capaz de envolver todos os participantes afetados pelo erro.

Na Seção 5.1, apresentamos uma descrição geral da Arquitetura FT-SC3. Nas Seções 5.2, 5.3 e 5.4 são descritos detalhadamente todos os componentes que compõem a Arquitetura FT-SC3. E por fim, A Seção 5.5 apresenta alguns comentários pertinentes à arquitetura descrita neste capítulo.

5.1 Descrição Geral da Arquitetura FT-SC3

A Arquitetura FT-SC3 [54, 55] é organizada em três camadas (Figura 7). A *camada computacional* encapsula os serviços de interface do sistema-componente. Cada sistema-componente recebeu a denominação de *componente participante* porque exerce papel similar ao de um participante dentro de uma ação atômica coordenada. Exemplos de componentes participantes incluem componentes COTS [43], sistemas legados e *Web Services* [35]. Estes elementos são adaptados por meio de *wrappers* [29, 62], com o objetivo de incorporá-los ao novo sistema.

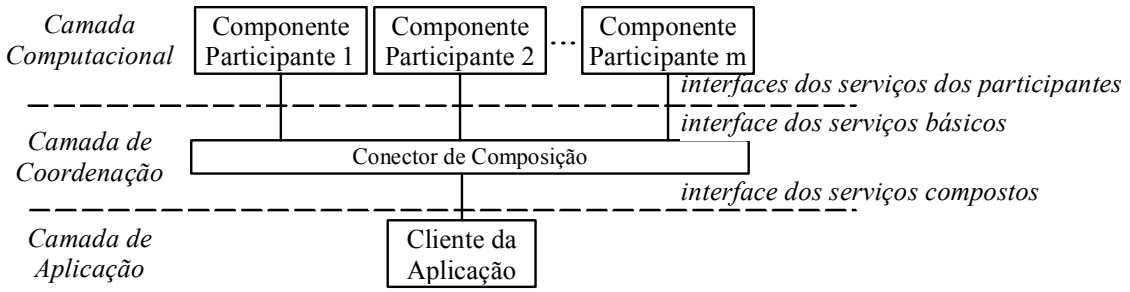

Figura 7. Visão geral da Arquitetura FT-SC3

A *camada de coordenação* consiste de um *conector de composição* que media as interações entre a *camada computacional* e a *camada de aplicação*. A interface superior do conector de composição, ou *interface de serviços básicos*, é a soma de todos os serviços oferecidos pelas *interfaces dos serviços dos participantes*. O conector de composição é capaz de impor regras de negócio sobre a interface de serviços básicos com a finalidade de prover novos serviços compostos. A interface inferior do conector de composição, ou *interface de serviços compostos*, é constituída pela soma dos serviços da interface de serviços básicos adicionados aos novos serviços compostos providos pelo conector de composição. A interface de serviços compostos também incorpora tolerância a falhas a todos os serviços oferecidos por ele através de tratamento de exceções.

A *camada de aplicação* contém os componentes que implementam a lógica da aplicação dos componentes clientes que poderão utilizar os serviços oferecidos pela interface de serviços compostos.

O conector de composição é um conector do estilo arquitetural C2, construído de um componente *interceptador* e um conjunto de *componentes contratos de composição* (Figura 8). Um componente contrato de composição, ou simplesmente contrato, define um conjunto de serviços compostos. Neste contexto, um serviço composto especifica uma ação que pode ser solicitada para impor novas regras de negócios sobre um serviço. O serviço composto pode estender um serviço básico ou compor um serviço mais sofisticado que define uma ação coordenada de dois ou mais componentes participantes. O componente contrato de composição pode ser organizado em várias *camadas de contratos* que são conectados por conectores C2 especializados.

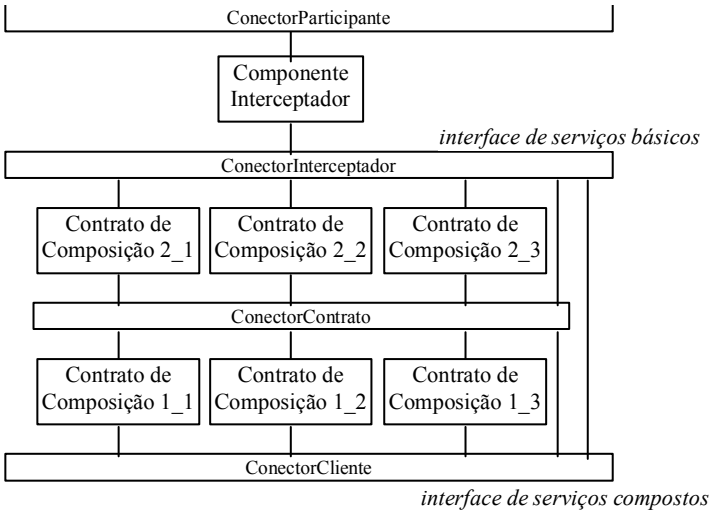

Figura 8. Estrutura básica de um conector de composição

A Figura 8 exibe um conector de composição com duas camadas de contratos. Os contratos de composição são responsáveis por: (i) prover novos serviços compostos; (ii) impor as regras de negócios sobre os serviços básicos e serviços compostos; e (iii) implementar tolerância a falhas para todos serviços oferecidos pelo sistema. Isto é realizado por meio de composições tolerantes a falhas de um ou mais serviços básicos.

O componente contrato de composição de uma camada de contrato pode usar serviços compostos providos pelos contratos localizados numa camada de contrato superior, permitindo aninhamento de ações. O *componente interceptador* é responsável por monitorar o fluxo de eventos da interface de serviços básicos e ativar o contrato de composição quando necessário.

5.2 O Componente Contrato de Composição

Um *componente contrato de composição* implementa um contrato de coordenação como uma ação atômica coordenada relaxada nos requisitos transacionais sobre os componentes participantes, como discutido na introdução deste capítulo.

Um contrato de composição é ativado por uma notificação de um evento associado a um *gatilho do contrato*. Sendo esta notificação enviada pelo *componente interceptador*. A ativação de um contrato de composição implica na invocação implícita de um serviço composto associado. A invocação de um serviço composto normalmente resulta numa ou

mais de solicitações de serviços enviadas de acordo com as regras de coordenação definidas pelo contrato de composição.

As solicitações de serviços enviadas pelos contratos de composição podem ser concorrentemente atendidas pelos componentes participantes da camada computacional, ou podem ativar o gatilho de um ou mais contrato de composição existente numa das camadas de contrato que estejam acima do contrato que emitiu a solicitação, criando ações atômicas coordenadas aninhadas. Entretanto, é importante observar que uma solicitação de serviço enviada por um contrato de composição nunca ativa o gatilho de outro contrato de composição que esteja localizado numa camada abaixo do contrato que emitiu a solicitação.

Quando um serviço composto termina com sucesso, ele pode finalizar com a emissão de uma *notificação normal* ou uma *notificação excepcional*, caso a operação tenha produzido um resultado degradado [18]. Uma operação gera um resultado degradado quando algum fluxo alternativo [15] foi realizado para que a operação tenha sido concluída e o contrato necessitou informar algo ao solicitante do serviço. Um contrato pode lançar uma notificação excepcional, por exemplo, para indicar que a operação foi realizada somente após ter executado alguma recuperação de erro por avanço (ver Capítulo 7).

Se um contrato de composição recebe exceções como respostas dos serviços por ele requisitados, o componente contrato de composição reúne todas respostas dos serviços que fazem parte da ação coordenada e ativa um tratador de exceções. O tratador de exceções resolve as exceções levantadas e coordena a ação de recuperação que deve ser executada pelos participantes da ação coordenada.

Se a condição excepcional não pode ser mascarada, o componente contrato de composição reage com: (i) uma *exceção de aborto*, se os participantes da ação coordenada são deixados num estado consistente igual ao que se encontravam antes do início da ação coordenada pelo contrato, garantindo a consistência do sistema; ou (ii) uma *notificação de defeito*, se um ou mais participantes não têm seus estados recuperados e são deixados num estado inconsistente, não garantindo a consistência do sistema.

A definição de um componente contrato de composição, descrito numa versão simplificada da linguagem Oblog [1], é exibido na Figura 9a. A cláusula attributes especifica os parâmetros de configuração de uma instância de um componente contrato de

composição. A cláusula coordination especifica uma lista de interação que define como os componentes participantes cooperam para executarem um serviço composto (Figura 9b).

contract <nome_contrato> attributes <lista_de_atributos> coordination <lista_de_interações> end contract **(a) A definição de um contrato**	<nome_interação>: when (<condição>) do <conjunto_de_ações> raises <lista_de_exceções> **(b) A definição de uma interação**

Figura 9. Definição do componente contrato de composição.

Na definição de uma interação, a condição depois da cláusula when especifica o *gatilho do contrato* para esta interação. A cláusula "do" especifica o conjunto de ações a serem executadas concorrentemente pelos componentes participantes. A cláusula raises especifica os tipos das exceções que poderão ser lançadas durante a execução de uma interação.

Idealmente, contratos de composição são componentes independentes que especificam serviços e não têm conhecimento sobre o contexto no qual eles estão sendo utilizados. Além disso, componentes contratos de composição podem estar organizados em camadas que permitam o aninhamento de ações dentro de um sistema e facilitam a extensão progressiva de uma regra de negócio implementada por um contrato, promovendo desta forma o baixo acoplamento e a reutilização de contratos.

5.3 O Componente Interceptador

O componente interceptador é responsável por determinar quando uma mensagem de requisição de serviço endereçada a um componente participante pode ser interceptada por um contrato. Ele define quando uma mensagem será, ou não, interceptada e esta decisão é tomada baseada na inspeção dos parâmetros da mensagem e dos gatilhos dos contratos.

Com o objetivo de realizar as responsabilidades atribuídas a ele, o componente interceptador deve ser configurado, de tal forma que ele saiba quais mensagens devem ser entregues para quais componentes contratos de composição. Um componente contrato de

composição somente pode ser ativado se a definição do seu gatilho for inserida na configuração do componente interceptador. As mensagens que não ativam o gatilho de nenhum componente contrato são entregues diretamente à camada computacional, sendo atendidas pelo componente participante capaz de atender a esta solicitação de serviço. O componente interceptador atua como um objeto de meta-nível [41, 9] para o componente participante, uma vez que ele intercepta as mensagens endereçadas aos participantes com a finalidade de impor novas regras de negócios definidas pelos componentes contratos de composição. Além disso, ele também atua como um objeto de meta-nível para os componentes contratos de composição, visto que ele escolhe qual contrato irá prover o serviço requisitado pela mensagem interceptada.

Durante o desenvolvimento desta arquitetura de software a localização do componente interceptador foi bastante discutida [53], pois em muitos momentos parecia óbvio que este poderia ser colocado abaixo das camadas de contratos (Figura 8), permitindo que todas as mensagens que chegassem ao conector de composição pudessem ser prontamente interceptadas e enviadas aos seus devidos destinatários (contratos de composição ou componentes participantes). Entretanto, como o componente interceptador também necessitava interceptar as mensagens oriundas dos contratos e destinadas aos componentes participantes, tudo isto sem incorrer num grande tráfego de mensagens circulando pelo sistema, concluímos que o interceptador deveria se localizar próximo aos componentes participantes, sendo o componente interceptador, então, colocado numa camada entre os componentes contratos de composição e os componentes participantes.

Esta medida ajudou-nos a evitar que a performance do sistema fosse degradada [53], em conseqüência de um grande fluxo de mensagens trafegando através do sistema compostos por componentes potencialmente distribuídos geograficamente.

A organização dos contratos de composição em sub-camadas e a definição dos gatilhos dos contratos, que está dentro do componente interceptador, constituem todo o conhecimento necessário para definir o modo como os contratos de composição serão compostos, isto é, os serviços compostos disponibilizados para a camada de aplicação. O primeiro critério usado pelo interceptador quando mais de um gatilho de contrato é ativado é a camada onde o contrato está localizado, onde é priorizado o contrato que está na camada mais inferior da arquitetura. O interceptador deverá ser configurado de tal maneira

que nunca mais de um contrato localizado na mesma camada sejam ativados simultaneamente.

Além disso, existem algumas situações em que o desenvolvedor precisa criar contratos que possuem o conhecimento sobre o sistema em que eles estão inseridos. Com a finalidade de adicionar esta flexibilidade à arquitetura, foi criado um tipo especial de mensagem de requisição, na qual o destino é definido pelo contrato de composição no momento em que ele solicita um serviço. Quando uma destas mensagens é recebida pelo componente interceptador, ele ignora sua configuração para os gatilhos dos contratos e envia a mensagem para o destino especificado pelo contrato, sendo este destino outro contrato ou um componente participante.

A idéia de introduzir um componente interceptador entre os participantes e os contratos de composição foi inspirada no padrão de projeto para coordenação desenvolvido por L.F. Andrade et al [5, 1], onde um *proxy* realiza a interceptação das mensagens quando estas chegam aos participantes.

5.4 O Componente Tratador de Exceções

No nosso modelo, um componente arquitetural pode ser um componente participante, um componente interceptador ou um componente contrato de composição. Cada um destes componentes é composto por dois sub-componentes: um componente AtividadeNormal e um componente TratadorExceções (Figura 10). O componente AtividadeNormal implementa o comportamento normal de um componente arquitetural, quando nenhuma exceção é levantada durante a execução dos serviços realizados por este componente. O componente TratadorExceções é responsável por: (i) tratar exceções levantadas pelos componentes AtividadesNormais que estão associados a ele, e (ii) prover *serviços tratadores* que sejam capazes de recuperar o estado do sistema.

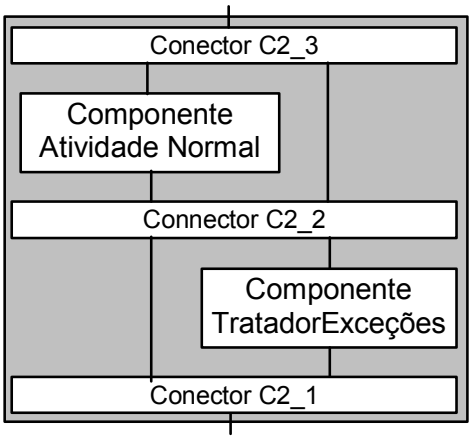

Figura 10. Estrutura interna de um interceptador e de um contrato de composição.

Mais especificamente, o componente TratadorExceções de um componente participante implementa um *serviço tratador* com o objetivo de desfazer operações que afetem o estado do participante. Além disso, o componente TratadorExceções tem o objetivo de fazer um tratamento de exceções pontual sobre o componente participante, de forma a tentar criar um participante confiável a partir de um sistema-componente não-confiável. Desta forma, o componente TratadorExceções permitirá que o desenvolvedor mascare os defeitos dos sistemas-componentes, permitindo, entre outras coisas: (i) contornar os problemas de conexão com o sistema-componente; (ii) detectar a ocorrência de falhas a partir de testes de pré e pós-condições e (iii) substituir um sistema-componente em caso de falhas.

O componente TratadorExceções no componente interceptador trata erros na configuração do sistema. Se um componente interceptador estiver configurado erroneamente para entregar uma solicitação de serviço a um contrato de composição incapaz de atendê-la, o interceptador poderá, através do componente TratadorExceções, ser capaz de se recuperar deste defeito, por exemplo, enviando a mensagem para outro contrato e removendo a configuração errônea.

O componente TratadorExceções implementa um contrato excepcional definido para um contrato de composição (Figura 11) e coordena a ativação de um *serviço tratador* sobre os componentes participantes. Isto tenta mascarar uma exceção e retornar o fluxo de controle para o componente AtividadeNormal usando uma estratégia de recuperação de erro por avanço. Se tal estratégia falha, o componente TratadorExceções pode realizar a

recuperação de erro por retrocesso, através da realização de ações de compensação com o objetivo de desfazer os efeitos indesejáveis sobre os componentes participantes, quando possível.

```
exceptional contract <nome_contrato>
   attributes <lista_de_atributos>
   coordination <lista_de_tratadores>
end contract
```

Figura 11. Definição de um componente tratador de exceções

Assim como na definição de um contrato de composição, o contrato excepcional possui atributos (Figura 11), utilizados para configurar uma instância do TratadorExceções, e uma lista de tratadores (*handlers*) que são interações especiais, as quais definem ações cooperativas sobre os participantes que sofreram os efeitos dos danos causados pelo erro, com o objetivo de recuperar o estado do sistema. Da mesma forma como as interações possuem condições que definem os gatilhos dos contratos, os tratadores também podem possuir gatilhos. Os gatilhos dos tratadores permitem que, quando um contrato intercepte uma solicitação de serviço destinada a um participante, este mesmo contrato seja capaz de interceptar uma solicitação de serviço que faça a compensação do primeiro serviço interceptado. Neste caso, o gatilho de um tratador é ativado para que o serviço de compensação seja interceptado pelo mesmo componente contrato de composição.

5.5 Considerações Finais

Neste capítulo apresentamos a Arquitetura FT-SC3 (***F**ault-**T**olerant **S**ystem **C**omposition using **C**omposition **C**ontracts*), que foi desenvolvida com a finalidade de auxiliar o desenvolvimento de sistemas de softwares confiáveis a partir de sistemas-componentes não confiáveis.

Nesta proposta arquitetural utilizamos um conceito de ação atômica coordenada relaxada na suas características de transação, visto que não se aplica utilizar transações quando os componentes são sistemas-componentes. Mesmo assim, a Arquitetura FT-SC3 utiliza o mecanismo de tratamento exceções de ações atômicas coordenadas, possibilitando

que durante o tratamento de erros possa-se realizar tanto recuperação de erro por avanço quanto por retrocesso.

Além disso, a Arquitetura FT-SC3 conseguiu reutilizar o conceito de contratos de coordenação, adicionando flexibilidade no desenvolvimento de sistemas mais facilmente adaptáveis às evoluções que se fizerem necessárias. Com isto, a Arquitetura FT-SC3 permite que novos requisitos sejam adicionados mais facilmente, através da inserção de contratos. A camada computacional e a camada de coordenação surgiram da proposta em separar os componentes que realizam as operações básicas, dos contratos de coordenação como apresentado na Seção 3.4.

O estilo arquitetural C2 foi utilizado na Arquitetura FT-SC3 visando aproveitar suas características de reutilização e flexibilidade na composição de sistemas, enfatizando a ligação fraca entre seus componentes. Com isto, foi possível criar uma arquitetura de software onde os componentes (com exceção do componente Interceptador) não têm conhecimento sobre os demais, ganhando-se em reuso, manutenção e extensão dos sistemas.

Capítulo 6

Ferramentas de Apoio para Arquitetura FT-SC3

A utilização de ferramentas de apoio tem a finalidade de permitir e guiar a aplicação de técnicas de desenvolvimento de software. No desenvolvimento de sistemas de software do mundo real, a utilização de tais técnicas pelas empresas é essencial para a qualidade do sistema desenvolvido. Neste âmbito, é conveniente a utilização de ferramentas de apoio que permitem a imposição de regras que: (i) facilitam e guiam o processo de desenvolvimento; (ii) permitam reutilizar componentes e artefatos[9] [33] já definidos para uma determinada técnica; e (iii) automatizam etapas do processo de desenvolvimento.

Com isto em mente, foram desenvolvidas algumas ferramentas para apoiar o desenvolvimento de sistemas de software baseados na Arquitetura FT-SC3 apresentada no Capítulo 5. Primeiramente, foi implementado um *framework* orientado a objetos escrito em Java (Seção 6.1) que auxilia o desenvolvimento de sistemas tolerantes a falhas baseados na abordagem apresentada na Arquitetura FT-SC3. Outra ferramenta desenvolvida (Seção 6.2) foi um ambiente de modelagem baseado no *Generic Modeling Environment* (GME) [37]. E por fim, é feito um breve resumo destas ferramentas na Seção 6.3.

6.1 Framework FT-SC3

O Framework FT-SC3 implementado para auxiliar o desenvolvimento de sistemas tolerantes a falhas é uma extensão da versão em Java [30] do Framework C2.FW (Seção 3.2.1) que provê uma infraestrutura para construção de aplicações que utilizam o estilo arquitetural C2 [43, 60]. Além disso, foi adicionado um tratamento de exceções

[9] Artefatos – é o termo usado para qualquer tipo de informação criado, produzido, alterado ou utilizado no desenvolvimento de sistemas.

concorrentes e um esquema de resolução de exceções para o Framework C2.FW, visto que este não oferece um suporte adequado para a construção de sistemas tolerantes a falhas.

O desenvolvimento do Framework FT-SC3 para a composição de sistemas tolerantes a falhas tem a finalidade de implementar parte da Arquitetura FT-SC3, com o objetivo de permitir que o desenvolvedor reutilize a parte da arquitetura já implementada. Além disso, o Framework FT-SC3 define quais partes não foram realizadas permitindo guiar o desenvolvimento das partes ainda não totalmente implementadas. Inicialmente realizaram-se dois trabalhos de análise, a fim de definir o que poderia ser implementado da arquitetura. Os passos foram:

- Identificação das funcionalidades e propriedades comuns a todos os sistemas que sigam a Arquitetura FT-SC3. Nesta etapa foram identificadas as classes que deveriam ser adaptadas ou instanciadas, assim como as interfaces, informando os serviços oferecidos e a serem realizados pelo desenvolvedor da aplicação; e
- Identificação das características das mensagens que são enviadas e recebidas pelos componentes da Arquitetura FT-SC3;

6.1.1 Tipos de Mensagens

Durante a identificação das características das mensagens trocadas entre os componentes da arquitetura, chegamos à conclusão que todas mensagens de solicitação de serviço possuem um atributo contendo o identificador do componente que a emitiu, este atributo é transferido para cada notificação do serviço, facilitando a entrega das notificações ao componente que solicitou o serviço.

As mensagens de solicitação de serviço (*requests*) se enquadravam em três tipos distintos (Figura 12): (i) as solicitações normais; (ii) as solicitações endereçadas; e (iii) as exceções propagadas. A descrição detalhada de cada uma destas mensagens é dada por:

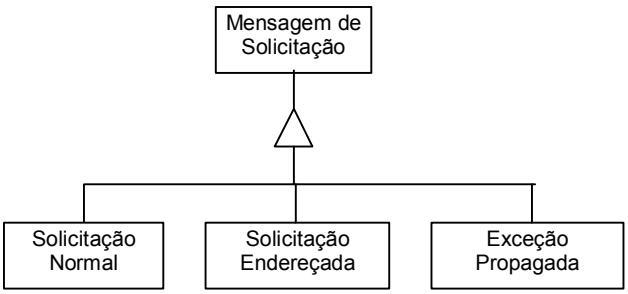

Figura 12. Diagrama de classes das mensagens de solicitações do Framework FT-SC3

- **Solicitação Normal** é uma mensagem de solicitação de serviço enviada por qualquer componente arquitetural destinada a um dos componentes participantes;
- **Solicitação Endereçada** é uma mensagem de solicitação de serviço que dá aos componentes contratos de composição a liberdade de determinar o destino desta mensagem. Nesta mensagem é informado qual o componente contrato de composição que irá interceptar a solicitação de serviço. Este tipo de mensagem tem como finalidade dar a liberdade, em casos muito excepcionais, de criar componentes contratos de composição mais flexíveis, que devido a uma requisição explícita da regra de negócio do sistema, obrigue que tais contratos tenham conhecimento sobre os demais componentes utilizados na composição do sistema. Este tipo de mensagem deve ser utilizado quando o requisito do sistema deixa explícita a necessidade da nova regra de negócio implementada pelo contrato ter tal conhecimento sobre os demais componentes, visto que este tipo de mensagem quando usado indiscriminadamente pode prejudicar a evolução do sistema; e
- **Exceção Propagada** é uma exceção propagada por um componente contrato de composição a todos os participantes envolvidos na ação. Esta mensagem é enviada logo após o contrato de composição ter resolvido as exceções levantadas pelos participantes. A exceção resolvida é propagada a todos os participantes através de uma mensagem de exceção propagada.

As mensagens de notificação (*notifications*) foram classificadas em cinco tipos (Figura 13): (i) as notificações normais; (ii) as notificações de resultado degradado; (iii) as

exceções de aborto; (iv) as exceções de defeito; e (v) as solicitações interceptadas. A descrição detalhada de cada uma destas mensagens é definida como:

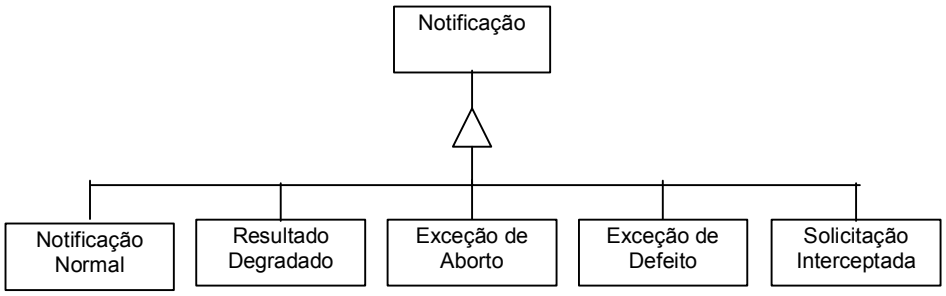

Figura 13. Diagrama de classes das mensagens de notificações do Framework FT-SC3

- **Notificação Normal** é uma mensagem de notificação de serviço enviada pelos componentes participantes ou componentes de contrato de composição como resposta a um serviço solicitado, informando que o mesmo foi realizado com sucesso;
- **Resultado Degradado** é uma mensagem de notificação de serviço enviada pelos componentes participantes ou componentes de contrato de composição indicando que o serviço solicitado foi realizado, mas que algum fluxo alternativo [15] foi realizado durante a realização de serviço, ou algum tipo de recuperação de erro por avanço foi realizado para que a operação fosse executada com sucesso;
- **Exceção de Aborto** é uma mensagem de notificação de serviço enviada pelos componentes participantes ou componentes de contrato de composição com o objetivo de informar que o serviço solicitado foi abortado e o sistema permanece no mesmo estado que se encontrava antes da realização do serviço solicitado;
- **Exceção de Defeito** de notificação de serviço enviada pelos componentes participantes ou componentes de contrato de composição quando não foi possível realizar o serviço solicitado nem realizar uma recuperação de erro que conseguisse restaurar o estado do sistema. Neste ponto o sistema se encontra num estado errôneo e o envio desta mensagem indica que a responsabilidade

pela recuperação do estado do sistema passou a ser do componente que solicitou o serviço; e

- **Solicitação Interceptada** é uma notificação que possui embutido uma solicitação de serviço. Esta mensagem é gerada apenas pelo componente interceptador sempre que uma solicitação de serviço é interceptada por um contrato de composição.

6.1.2 Protocolo dos Componentes

Todos componentes do Framework FT-SC3 foram implementados de forma que uma *thread*[10] [17] é executada para cada um dos componentes. Os componentes identificados e implementados para o Framework FT-SC3 são descritos nas próximas subseções.

Componente Participante

Cada componente participante é capaz de receber mensagens de solicitações de serviços e processá-las, gerando mensagens de notificações. O parâmetro da solicitação de serviço contendo o nome do contrato é copiado para a nova mensagem de notificação gerada. Este componente deve ser adaptado pelo desenvolvedor durante a criação dos participantes do sistema, para permitir que um sistema-componente seja reutilizado durante a composição do sistema.

Componente Interceptador

O componente interceptador possui um conjunto de configuração de contratos que é projetado para definir quando e a qual contrato o interceptador deverá redirecionar uma mensagem que era destinada a um participante.

O componente interceptador reage da seguinte forma, quando recebe um dos tipos de mensagens descritos a seguir:

- **Mensagem de solicitação endereçada –** O interceptador realiza os seguintes passos: (i) ignora a configuração dos contratos; (ii) cria uma solicitação interceptada; (iii) embute o identificador do contrato contido na mensagem

[10] Thread – Processo leve.

endereçada; (iv) embute a mensagem de solicitação endereçada; e (v) a envia ao ConectorInterceptador (Figura 8);

- **Demais mensagens de solicitação** – O interceptador procura se alguma configuração de contrato determina se a mensagem deve, ou não, ser interceptada por um componente contrato de composição. Se a mensagem for interceptada por um contrato, o interceptador: (i) cria uma solicitação interceptada; (ii) embute o identificador do contrato contido na configuração do contrato selecionado; (iii) embute a mensagem de solicitação; e (iv) a envia ao ConectorInterceptador. Caso nenhum contrato intercepte a solicitação de serviço, ela é enviada ao ConectorParticipantes; e
- **Mensagem de notificação** – As mensagens de notificações recebidas pelo interceptador são enviadas ao ConectorInterceptador.

Componente Contrato de Composição

Quando um componente contrato de composição recebe uma mensagem, ele pode gerar uma ou mais mensagens de solicitação de serviço ou uma mensagem de notificação.

Sempre que o contrato de composição envia uma notificação de serviço, ele preenche o parâmetro de identificação do contrato indicando quem solicitou o serviço atendido pelo contrato. Caso o contrato tenha interceptado uma mensagem do cliente da aplicação, o parâmetro de identificação do contrato não recebe nenhum valor.

Cada contrato de composição reage da seguinte forma, quando recebe os tipos das mensagens descritas a seguir:

- **Mensagem de solicitação interceptada** - O contrato de composição processa a solicitação de serviço que veio embutida nesta mensagem; e
- **Mensagem de notificação** - O contrato de composição processa a mensagem de notificação.

Componente Cliente da Aplicação

O componente cliente da aplicação envia mensagens de solicitação ao ConectorCliente com o objetivo de solicitar a realização de serviços oferecidos pelo sistema composto. A resposta esperada para o serviço solicitado ao sistema composto é dada por

meio de uma mensagem de notificação normal, uma resposta degradada, uma exceção de aborto ou uma exceção de defeito.

6.1.3 Protocolo dos Conectores

Todos conectores do Framework FT-SC3 foram implementados de forma que uma *thread* é executada para cada um deles. Os conectores identificados e implementados para o Framework FT-SC3 são descritos nas próximas subseções.

ConectorParticipante

ConectorParticipante é um conector C2 padrão, onde todas as mensagens de solicitação e notificação são enviadas respectivamente para a interface superior e inferior.

ConectorInterceptador

O ConectorInterceptador tem dois atributos: (i) **contratos inferiores**, atributo que contém a lista com todos os contratos de composição conectados imediatamente abaixo deste contrato; e (ii) **conector inferior**, atributo que contém o ConectorContrato ou o ConectorCliente que esteja conectado abaixo deste contrato.

O protocolo do ConectorInterceptador reage da seguinte forma, quando recebe os tipos das mensagens descritas a seguir:

- **Mensagem de solicitação** - Reenvia para o componente interceptador;
- **Mensagem de notificação** - Reenvia para: (i) o ConectorCliente, se a notificação não vem com parâmetro de identificação do contrato; e (ii) o contrato de composição especificado no parâmetro da identificação do contrato, caso o contrato seja um dos contratos inferiores; caso contrário, (iii) o conector inferior; e
- **Mensagem de solicitação interceptada** - Reenvia para: (i) o contrato de composição especificado no parâmetro da identificação do contrato, caso o contrato seja um dos contratos inferiores; caso contrário, (ii) o conector inferior.

ConectorContrato

ConectorContrato possui três atributos: (i) **contratos inferiores**, atributo contendo todos os contratos de composição conectados imediatamente abaixo deste conector; (ii) **conector inferior**, atributo contendo o ConectorContrato ou o ConectorCliente conectado abaixo deste conector; e (iii) **conector superior**, atributo contendo o ConectorContrato ou o ConectorInterceptador conectado acima deste conector.

O protocolo do ConectorInterceptador reage da seguinte forma quando recebe os tipos das mensagens descritas a seguir:

- **Mensagem de solicitação** – Reenvia para conector superior;
- **Mensagem de notificação** – Reenvia para: (i) o conector inferior, se a notificação não tiver o parâmetro da identificação do contrato; (ii) o contrato de composição especificado no parâmetro da identificação do contrato, caso o contrato seja um dos contratos inferiores; caso contrário, (iii) o conector inferior; e
- **Mensagem de solicitação interceptada –** Reenviada para: (i) o contrato de composição especificado no parâmetro da identificação do contrato, caso o contrato seja um dos contratos inferiores; caso contrário, (iii) o conector inferior.

ConectorCliente

O ConectorCliente reage da seguinte forma, quando recebe os tipos das mensagens descritas a seguir:

- **Mensagem de solicitação** – Reenvia para o ConectorInterceptador; e
- **Mensagem de notificação** – Reenvia para os componentes que estejam conectados na sua interface inferior.

6.1.4 Como utilizar o Framework FT-SC3

A utilização do Framework FT-SC3 na criação de um sistema exige a realização de quatro passos por parte do desenvolvedor, são eles:

1. **Criação de participantes confiáveis** – Implementar *wrappers* [29, 62] para cada sistema-componente, com a finalidade de criar os participantes que exercem os papéis necessários para a realização da ação coordenada pelos

contratos de composição. Os *wrappers* criados para os sistemas-componentes devem realizar um tratamento para as exceções locais levantadas, com o objetivo de criar participantes confiáveis a partir de sistemas-componentes não confiáveis;

2. **Criação dos componentes contratos de composição** – Implementar todas regras de negócios que coordenam a composição dos serviços básicos oferecido pela camada computacional. Além disso, estes contratos de coordenação devem realizar o tratamento das exceções levantadas por um ou mais participantes, provendo serviços compostos confiáveis. Cada contrato de composição deve ser capaz de resolver as exceções levantadas concorrentemente e propagar a exceção resolvida a todos os participantes envolvidos na ação coordenada pelo contrato. Nesta etapa, o projetista pode utilizar alguns dos padrões de projeto para tratamento de exceções concorrentes [26, 25];
3. **Modelagem da arquitetura do sistema** - Conectar todos os participantes, o interceptador e os contratos de composição na arquitetura, definindo o número e os contratos que estarão inseridos em cada sub-camadas de contratos. Para esta etapa foi elaborado um ambiente de modelagem, que é usado para construção do modelo de uma Arquitetura FT-SC3. Na Seção 6.2 iremos descrever em maiores detalhes este ambiente de modelagem; e
4. **Configuração do interceptador** – Projetar e configurar o interceptador com os gatilhos de ativação dos contratos de composição implementados no passo dois.

6.2 Um Ambiente para Modelagem da Arquitetura FT-SC3

O ambiente para modelagem é baseado no *Generic Modeling Environment* (GME) [37]. O GME é usado inicialmente para a construção de modelos. Ele possibilita e descreve um meta-modelo para modelagem. Baseado nesta descrição, ele é capaz de funcionar como um ambiente de modelagem para modelos que são instâncias do meta-modelo especificado.

O uso de um ambiente para modelagem baseado no GME permite a construção de modelos de forma mais fácil do que começar a construção de um modelo a partir do zero.

Segundo definição apresentada em [32], GME é uma ferramenta gráfica para sintetizar programas para integração de modelos, pertencentes a um domínio específico, e que possibilita a criação e evolução de modelos para sistemas de software de larga escala. GME é configurável, o que significa que ele pode funcionar em diversos domínios de aplicação. GME oferece várias propriedades que são úteis para modelar sistemas complexos, tais como: checagem de restrições baseadas em OCL [45], integrações com geradores de código, modelos hierárquicos e a capacidade para importar e exportar modelos como arquivos XML [61].

O ambiente de modelagem desenvolvido neste trabalho permite que desenvolvedores modelem sistemas baseados na Arquitetura FT-SC3 por meio de diagramas. Um modelo pode ter sua consistência checada e posteriormente pode-se armazenar um arquivo XML, que é usado como entrada para um gerador de código, sendo o modelo em si uma documentação bastante útil para a arquitetura de software. Por termos apenas desenvolvido a primeira versão do ambiente para a Arquitetura FT-SC3, muitas propriedades não foram incorporadas, tais como a especificação dos gatilhos dos contratos de composição e a definição dos componentes TratadorExceções (Figura 9).

Por fim, foi criado um gerador de código simples que é capaz de gerar, a partir do arquivo XML com a descrição do modelo arquitetural, o esqueleto da implementação das classes dos elementos arquiteturais e dos conectores que os interligam. O código gerado é adaptado ao Framework FT-SC3 definido na Seção 6.1 e pode ser utilizado da forma como foi criado sem necessidade de configurações.

Os diagramas dos modelos do estudo de caso apresentado na Seção 7.2. foram criados a partir da utilização do ambiente de modelagem da arquitetura FT-SC3. Cada diagrama teve sua consistência checada, o que permitiu identificar as desconformidades dos modelos do estudo de caso com a Arquitetura FT-SC3. A partir destes diagramas, foi gerado um arquivo XML para cada modelo, que permitiu a utilização do gerador de código para implementar automaticamente as assinaturas das classes de todos os componentes e conectores definidos nos modelos.

6.3 Considerações Finais

Neste capítulo apresentamos duas ferramentas de apoio criadas para auxiliar o desenvolvimento de sistemas baseados na Arquitetura FT-SC3.

A primeira ferramenta, o Framework FT-SC3 (Seção 6.1), implementa partes comuns da Arquitetura FT-SC3, permitindo que o desenvolvedor reutilize alguns componentes que já foram implementados. Este *framework* também permite ao desenvolvedor um maior conhecimento sobre os detalhes da Arquitetura FT-SC3, tendo em vista que são definidos: (i) todos tipos das mensagens que são trocadas entre os componentes da arquitetura; (ii) o protocolo de todos os componentes; e (iii) o protocolo de todos os conectores utilizados na arquitetura. Por fim, foi apresentado um roteiro com as quatro atividades a serem realizadas durante a criação de sistemas a partir da utilização do Framework FT-SC3. As atividades são: (i) criação de participantes confiáveis; (ii) criação dos componentes de contratos de composição; (iii) modelagem da arquitetura do sistema; e (iv) configuração do interceptador. Pode-se observar por este curto roteiro que a composição de sistemas com o uso do Framework FT-SC3 é, na medida do possível, uma atividade que não exige grandes esforços de programação.

Na Seção 6.2 foi apresentado um ambiente de modelagem baseado no *Generic Modeling Environment* (GME). Este ambiente modelagem permite a construção de um modelo a partir de digramas que podem ser exportados e importados a partir de arquivos XMLs. Estes arquivos XMLs podem ser utilizados por um gerador de código, que realiza parte do trabalho de programação que seria realizado manualmente. Além disso, os arquivos XMLs são, por si só, uma documentação da arquitetura de software do sistema.

Capítulo 7

Estudos de Casos

Com o objetivo de avaliar a abordagem para composição de sistemas-componentes apresentado nesta dissertação, foram realizados três estudos de casos. Todos os estudos de casos foram implementados utilizando o Framework FT-SC3 para composição de sistemas tolerantes a falhas apresentado no Capítulo 6.

O primeiro estudo de caso, apresentado na Seção 7.1, teve como objetivo realizar um sistema simples para controle de contas bancárias tolerantes a falhas que auxiliasse o desenvolvimento da Arquitetura FT-SC3, permitindo confirmar a viabilidade das idéias e opções estudadas durante o processo de criação e melhoria da Arquitetura FT-SC3.

Na seção 7.2 é apresentado um segundo estudo de caso que foi desenvolvido para observarmos como a utilização da Arquitetura FT-SC3 se comportaria numa nova situação, onde o maior desafio era resolver um problema encontrado no cotidiano dos sistemas bancários, onde freqüentemente novas regras de negócios são incorporadas aos requisitos dos sistemas. Neste caso, foi comprovado, entre outras coisas, o aninhamento de ações coordenadas pelos contratos.

No terceiro e último estudo de caso é realizado a integração de sistemas-componentes para venda de produtos de um sistema varejista. Este estudo de caso tem por objetivo resolver um problema real encontrado pelas empresas de software no desenvolvimento destes sistemas. Atualmente, quase todos os sistemas varejistas possuem problemas na atualização de preços nos pontos de vendas, como pode ser comprovado pelos constantes erros nos preços dos produtos vendidos nos supermercados. A grande dificuldade desta atualização de preço deve-se aos diversos tipos de concorrência a serem tratados. Na Seção 7.3 mostramos um estudo de caso que soluciona o problema de sincronismo na atualização de preços utilizando a Arquitetura FT-SC3.

Na Seção 7.4 fazemos apanhado geral das conclusões obtidas com a realização dos três estudos de casos.

7.1 Estudo de Caso 1 – Sistema Bancário Simples

Este estudo de caso ilustra o comportamento de um sistema bancário tolerante a falhas aplicando a Arquitetura FT-SC3. Este estudo de casos integra dois sistemas-componentes: (i) O componente ContaCorrente e um componente ContaPoupança, que encapsulam componentes pré-existentes e provêem operações para sacar e para consultar o saldo bancário. Tais componentes, assim como todo componente participante da Arquitetura FT-SC3, oferecem seus serviços independentemente de estarem, ou não, ligados ao sistema bancário.

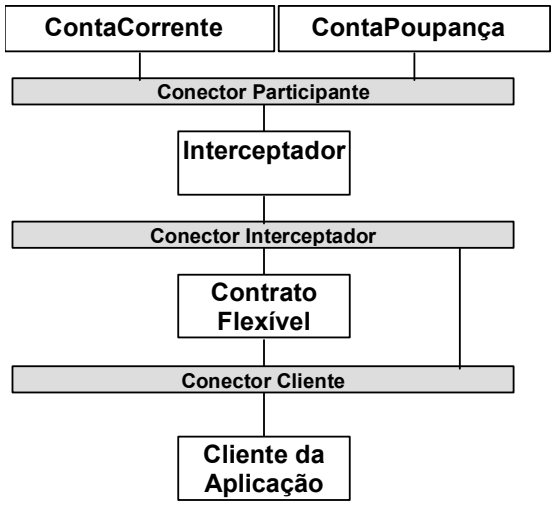

Figura 14. Arquitetura do sistema bancário do Estudo de Caso 1

A regra de negócio específica do sistema é realizada através de um único contrato de composição, denominado de ContratoFlexível (Figura 14). O ContratoFlexível tenta evitar que a ContaCorrente fique com saldo negativo transferindo fundos da Conta Poupança, quando necessário.

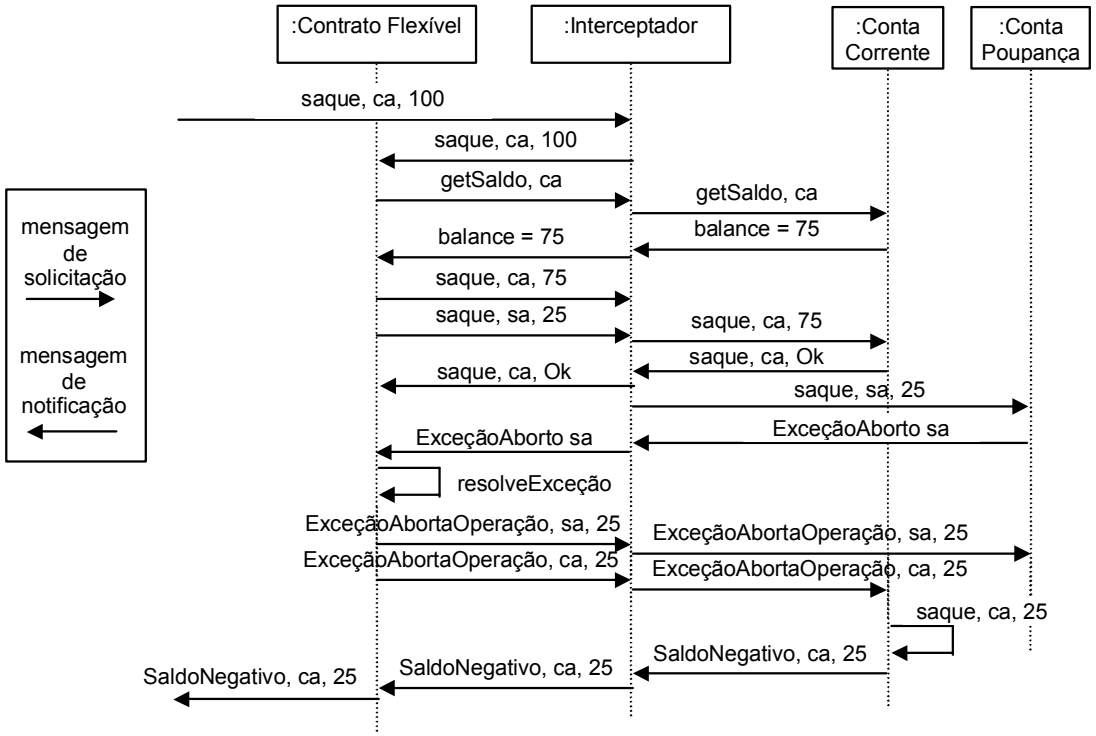

Figura 15. Diagrama de Seqüência do Estudo de Caso 1.

O diagrama de seqüência[11] em UML [23, 11, 46] da Figura 15 mostra o processamento de uma solicitação de saque de $100,00 na conta corrente identificada por "ca". A conta corrente "ca" está associada a uma poupança identificada por "sa" por meio de um ContratoFlexível. Inicialmente, a solicitação de saque é interceptada pelo componente ContratoFlexível. Neste estudo de caso, o Interceptador está configurado apenas para interceptar as solicitações de saques a serem realizadas sobre a conta "ca", sendo a operação de saque a única solicitação de serviço passível de interceptação pelo ContratoFlexível. O ContratoFlexível checa o saldo da conta corrente "ca", o qual é observado que somente há $75,00. O ContratoFlexível tenta evitar que a ContaCorrente fique com saldo negativo e para satisfazer o pedido de saque de $100,00 resolve enviar concorrentemente duas solicitações de saque, um saque no valor de $75,00 da conta corrente "ca" e outro no valor restante de $25,00 da poupança "sa". Neste caso, nenhuma

[11] Por questão de simplicidade, todos os diagramas de seqüência desta dissertação suprimem os conectores, visto que eles apenas entregam as mensagens recebidas.

destas duas solicitações de saque será interceptada, pois não há contrato localizado numa camada acima do ContratoFlexível configurado para interceptar solicitações de saque.

Neste estudo de caso a ContaPoupança apresenta um erro e recusa-se a realizar o saque de $25,00, abortando assim a operação de saque. Neste momento a ContaPoupança levanta uma ExceçãoAborto. O ContratoFlexível resolve as exceções levantadas (no caso apenas uma ExceçãoAborto) e envia uma exceção resolvida de ExceçãoAbortaOperacao para todos os componentes participantes. A ContaPoupança ignora esta exceção, enquanto que a ContaCorrente trata esta exceção com uma invocação do serviço de saque no valor de $25,00. Esta recuperação de erro leva a conta corrente "ca" a ficar com saldo negativo. Isto leva o componente ContaCorrente a levantar uma resposta degradada indicando que o saque foi realizado, mas a conta corrente "ca" ficou com seu saldo negativo.

7.2 Estudo de Caso 2 – Sistema Bancário Evolutivo

O segundo estudo de caso teve como enfoque ver o comportamento do uso da Arquitetura FT-SC3 para uma situação do mundo real, onde vários novos participantes são inseridos no sistema e as regras de negócio evoluem. Este estudo de caso é a integração dos sistemas de dois bancos, A e B. Tais sistemas bancários foram implementados utilizando-se EJB [57, 10]. A camada de coordenação utiliza os participantes para oferecer novos serviços que envolvam um ou mais bancos.

Durante o desenvolvimento deste estudo de caso foi utilizado o ambiente para modelagem da arquitetura FT-SC3 (Seção 6.2). Por este motivo, todos os modelos apresentados aqui estão em inglês e com formato diferente dos demais estudos de casos, visto que foram extraídos do ambiente de modelagem.

Este estudo de caso é constituído por três cenários ordenados pela complexidade. As Subseções 7.2.1, 7.2.2 e 7.2.3 descrevem cada um dos cenários.

7.2.1 Cenário 1: Um Participante e nenhum Contrato

O Cenário 1 consiste de um sistema bastante simples em que o cliente da aplicação é capaz de fazer um depósito em uma conta bancária do Banco A. A Figura 16 mostra a configuração arquitetural para este sistema, composto por três componentes: um

componente participante correspondente ao sistema bancário do Banco A (Bank_A), o interceptador (Interceptor), e o componente cliente (ClientComponent).

Neste cenário, o cliente da aplicação envia uma solicitação de serviço que é recebida pelo componente interceptador e redirecionada posteriormente para o Banco A. Quando o Banco A recebe esta solicitação, ele executa o depósito na conta corrente especificada e responde com uma mensagem de notificação. Esta mensagem de notificação é entregue ao cliente da aplicação, informando-o que o depósito foi realizado com sucesso.

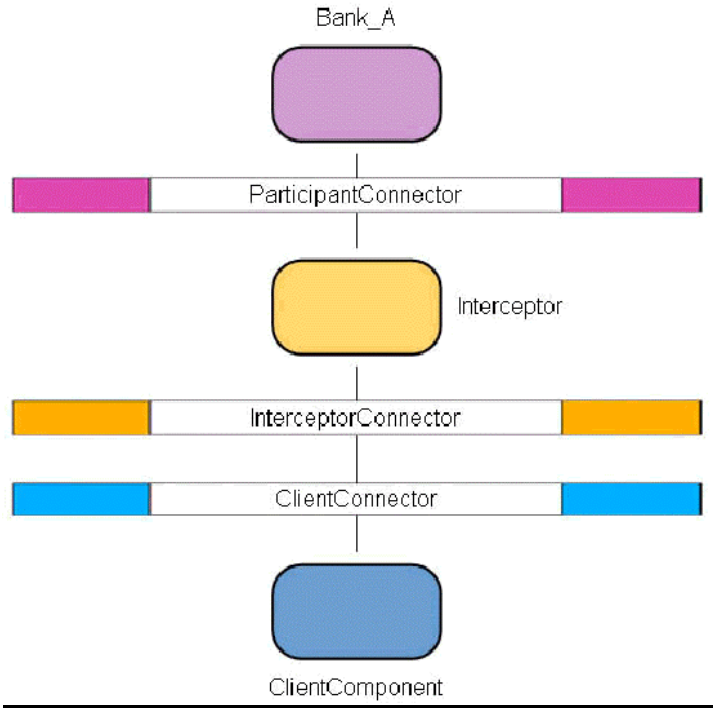

Figura 16. Sistema Bancário com um Participante e nenhum Contrato

7.2.2 Cenário 2: Dois Participantes e um Contrato

Com a necessidade dos Bancos A e B integrarem seus sistemas para permitirem que a construção um sistema composto seja capaz de realizar transferência de fundos entre contas de um mesmo correntista. No cenário 2, foram introduzidos um novo participante e um novo contrato para realizar uma operação de transferência de fundos. A Figura 17 mostra a nova configuração da arquitetura deste novo sistema.

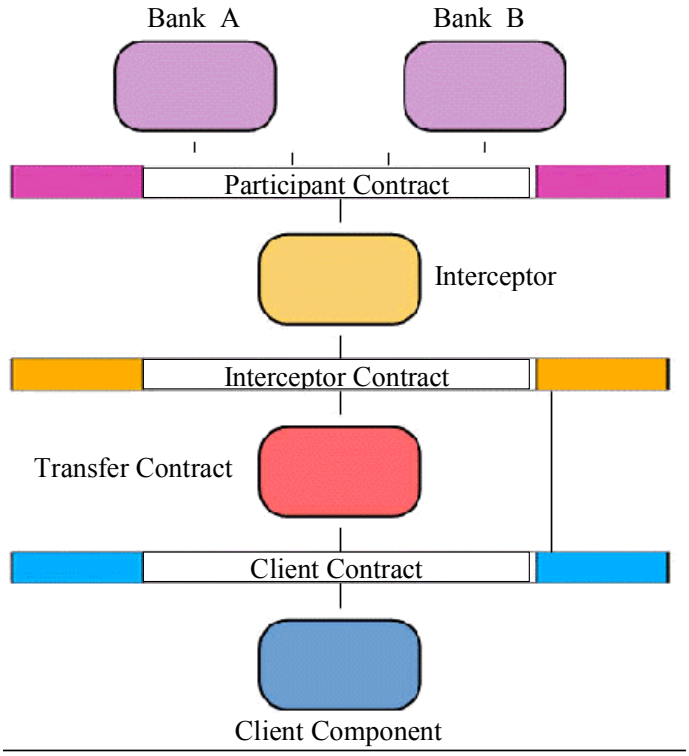

Figura 17. Sistema Bancário com dois Participantes e um único Contrato

O novo participante corresponde ao sistema bancário do Banco B (Bank_B). A nova operação de transferência de fundos é implementada pelo contrato TransferContract, que coordena o uso das operações básicas de saque e depósito oferecidas pelos participantes do sistema.

Quando o cliente deseja fazer uma transferência de $100,00 de uma conta bancária do Bank_A para o Bank_B, o ClientComponent envia uma mensagem de solicitação de transferência de fundos que é recebida pelo Interceptor. O Interceptor está configurado para redirecionar todas as mensagens de transferência de fundos para o componente TransferContract. Tão logo o TransferContract receba a solicitação de transferência, ele envia simultaneamente um pedido de saque e outro de depósito.

Como nenhum gatilho de contrato está configurado para as mensagens de saque e depósito, estas mensagens são prontamente entregues aos dois componentes participantes. Se ambos os participantes executarem sua tarefa como esperado, o TransferContract receberá duas notificações informando que as operações foram realizadas com sucesso. O TransferContract, neste caso, enviará uma outra notificação, informando ao ClientComponent

que a transferência ocorreu com sucesso. Este cenário é ilustrado no diagrama de seqüência em UML da Figura 18.

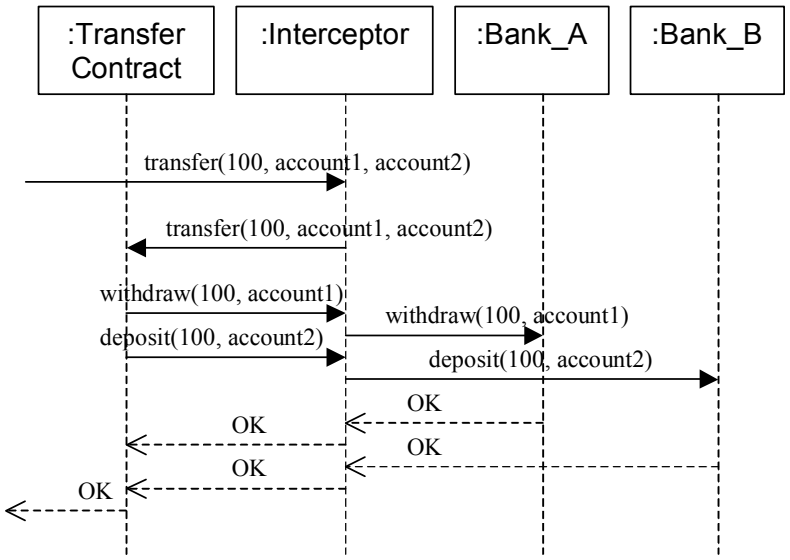

Figura 18. Diagrama de Seqüência do Cenário 2

O TransferContract também é responsável por realizar o tratamento de exceções coordenado, caso a transferência não possa ser realizada com sucesso. Quando um dos participantes levanta uma exceção, o TransferContract recolhe todas as notificações enviadas pelos participantes, resolve as exceções, nos casos onde mais de uma exceção tenham sido lançadas, e inicia o tratamento da exceção resolvida em conjunto com os participantes envolvidos na ação.

A Figura 19 apresenta um cenário onde o participante Bank_B não consegue realizar o depósito e aborta a operação. O TransferContract trata esta exceção através de um depósito de $100,00 na conta do participante Bank_A, equivalente aos $100,00 que haviam sido sacados. Uma vez realizada esta ação de recuperação corretamente, o sistema volta a um estado válido. Uma mensagem de exceção de aborto é lançada pelo TransferContract para o ClientComponent, indicando que apesar da operação de transferência não ter sido realizada, o sistema se encontra num estado consistente.

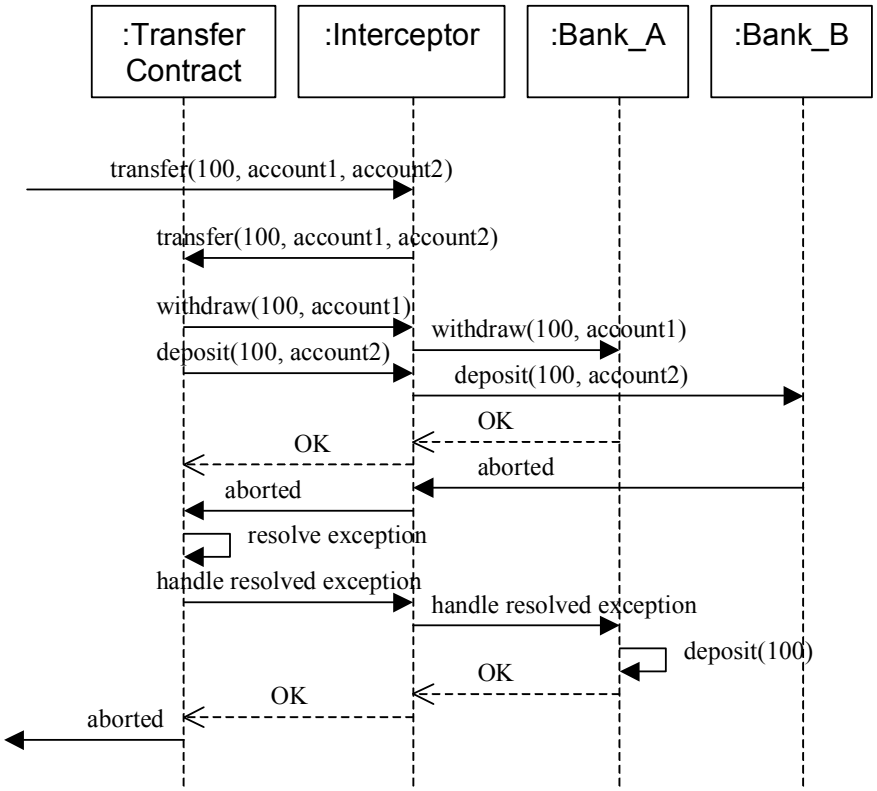

Figura 19. Diagrama de Seqüência para um Tratamento de Exceções no Cenário 2.

7.2.3 Cenário 3: Três Participantes e dois Contratos

Em 1993, o governo brasileiro criou um novo imposto, denominado CPMF[12]. Este imposto recolhe um pequeno percentual sempre que é realizada uma movimentação financeira numa conta bancária. No caso de uma transferência de fundos, a taxa de CPMF somente é cobrada quando é realizada uma movimentação financeira entre contas bancárias pertencentes a indivíduos diferentes, ou seja, transferências de fundos entre contas pertencentes a uma mesma pessoa não são tributadas. Na criação deste imposto as instituições bancárias foram obrigadas a inserirem uma nova funcionalidade em seus sistemas, para que fosse possível calcular o montante transferido entre contas bancárias pertencentes a pessoas distintas.

Para atender a estas novas exigências, o Cenário 2 foi adaptado para conseguir se adequar às imposições ocasionadas pela criação da CPMF. A Figura 20 apresenta a arquitetura para este cenário.

[12] CPMF - Um acrônimo para **C**ontribuição **P**rovisória sobre **M**ovimentação **F**inanceira.

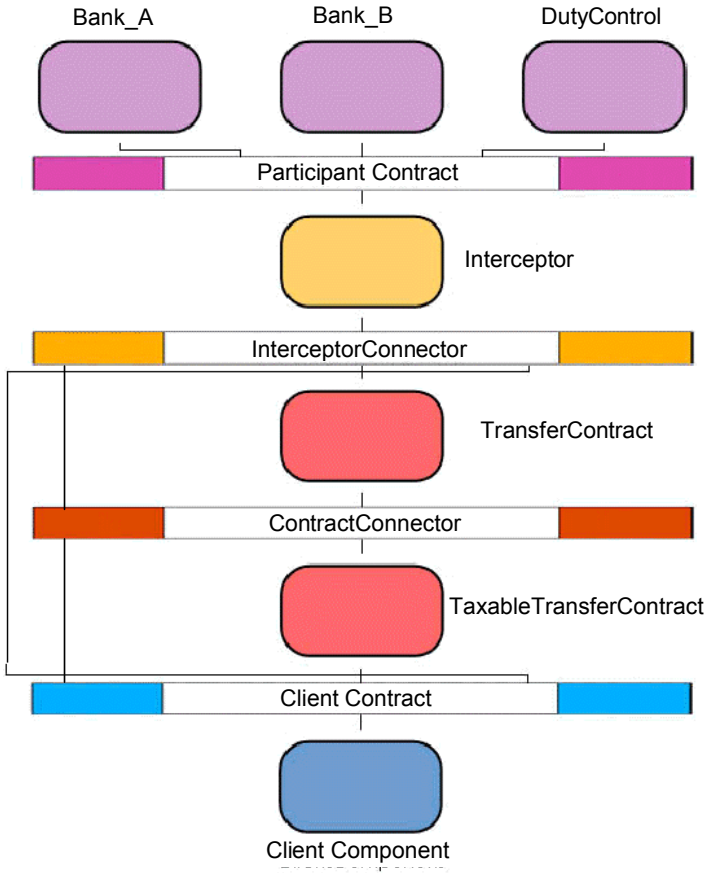

Figura 20. Sistema Bancário com três Participantes e dois Contratos

Um novo contrato, TaxableTransferContract, e um novo participante, DutyControl, foram adicionados. O participante DutyControl automatiza a tarefa de armazenar todas operações financeiras. O TaxableTransferContract utiliza o TransferContract e o DutyControl para realizar as transferências de fundos que devem estar sujeitas a tributação da CPMF.

Neste cenário, uma nova configuração é adicionada ao componente interceptador, para que todas mensagens de solicitação de transferência de fundos sejam interceptadas pelo TaxableTransferContract. O diagrama de seqüência apresentado na Figura 21 ilustra o comportamento do sistema quando este recebe uma solicitação de transferência de fundos.

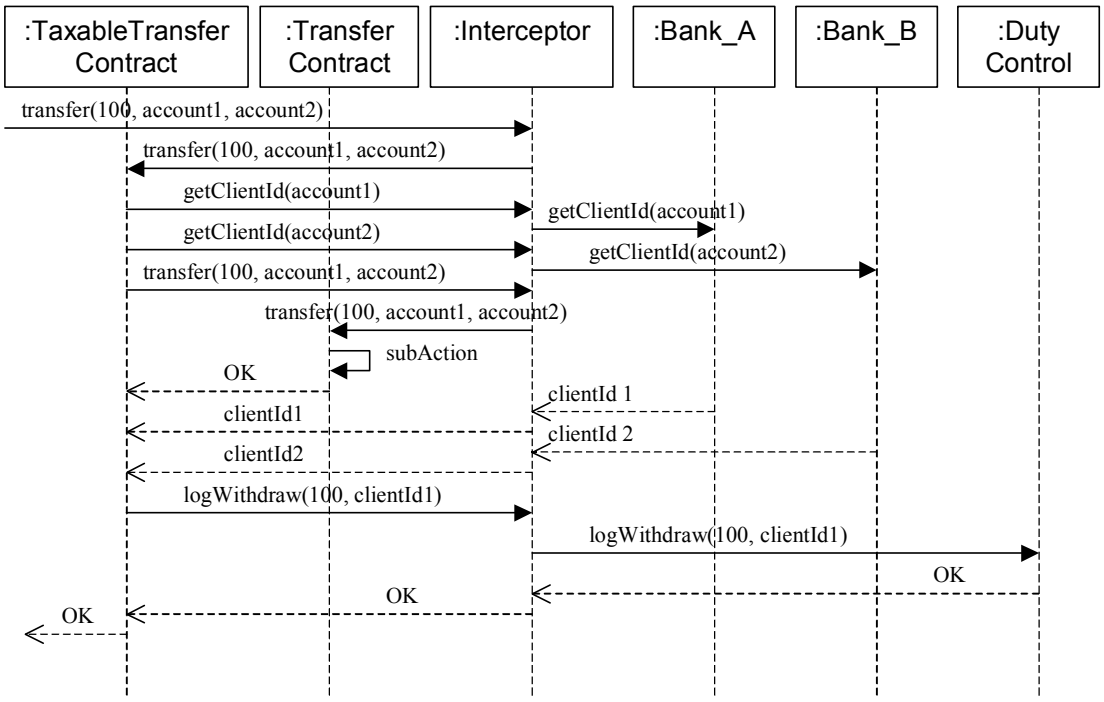

Figura 21. Diagrama de Seqüência para o Cenário 3

Inicialmente, o interceptador recebe a solicitação para transferir fundos de uma conta bancária do participante Bank_A para outra conta do participante Bank_B. O interceptador checa se esta mensagem ativa o gatilho de algum contrato. Neste caso, os gatilhos de ambos os contratos (TransferContract e TaxableTransferContract) são ativados, mas a solicitação é interceptada apenas pelo TaxableTransferContract, pois o interceptador foi especificado para priorizar os contratos que estejam localizados nas camadas mais inferiores.

Depois de receber a mensagem interceptada, o TaxableTransferContract envia três mensagens de solicitação. Duas destas mensagens são para solicitar informações aos participantes Bank_A e Bank_B sobre as contas bancárias envolvidas na transferência. A terceira mensagem solicita uma transferência de fundos. Esta terceira mensagem será então interceptada pelo TransferContract que realiza uma transferência como descrita na Seção 7.2.2. Neste caso, o TransferContract realiza uma sub-ação para o TaxableTransferContract e qualquer resultado desta sub-ação é entregue para este último. Se a transferência de fundos for realizada com sucesso TaxableTransferContract usa as informações obtidas do Bank_A e Bank_B sobre as contas bancárias envolvidas e checa se as duas pertencem a um mesmo indivíduo. No caso da contas pertencerem a indivíduos distintos, o TaxableTransferContract

solicita que o componente DutyControl armazene a operação para posterior cobrança da CPMF. Se tudo correr bem, o TaxableTransferContract envia uma notificação ao ClientComponent informando que a transferência foi realizada com sucesso.

Se o componente TransferContract abortar a transferência de fundos, o TaxableTransferContract recebe uma exceção de aborto levantada pelo TransferContract (Sessão 7.2.2) e a resolve. A ação de recuperação de erro realizada pelo TaxableTransferContract para tratar esta exceção de aborto consiste em não solicitar o armazenamento da operação pelo DutyControl e enviar uma exceção de aborto para componente ClientComponent, informando que a transferência de fundos foi abortada.

7.3 Estudo de Caso 3 – Sistema Varejista

O terceiro estudo de caso que desenvolvemos tem por objetivo resolver um problema real encontrado pelas empresas de software na criação de um sistema varejista.

O sistema para uma empresa varejista é constituído por dois tipos de componentes, os Sistemas de Pontos de Vendas, mais conhecido por PDVs, e o Gerenciador do Sistema. Cada PDV tem a função primordial de vender os produtos mesmo sob as mais desfavoráveis condições, ou seja, independente do que ocorra, o PDV não pode parar a venda para realizar qualquer uma outra tarefa. O Gerenciador do Sistema é formado por um banco de dados que concentra todas as informações provenientes das vendas dos produtos pelos PDVs, tais como: os preços praticados pelos PDVs, os PDVs que estejam em funcionamento, o que foi vendido nos PDVs, entre outras. Estas informações serão utilizadas para prover dados aos administradores da loja, e permitirá que atualizações de preços de produtos sejam executadas simultaneamente em todos os PDVs.

Em um sistema varejista, entende-se por uma venda, a operação de cadastro da venda de vários produtos para um determinado cliente. Uma operação de venda inicia na abertura do cupom de venda e finaliza com o fechamento do cupom indicando o pagamento dos produtos comprados.

Cada PDV é formado por dois sub-componentes: (i) um componente de gerenciador de persistência de dados, Gerente de Persistência, que insere, altera, remove e desabilita a

leitura do registro de um produto contido num arquivo local; e (ii) um componente gerenciador de vendas, Gerente de Vendas, que controla as vendas realizadas pelo PDV.

A Figura 22 apresenta o esboço de um sistema varejista constituídos por três PDVs interligados a um Gerenciador do Sistema. Em cada PDV podemos ver o Gerente de Persistência e o Gerente de Vendas que o constitui. Ambos gerenciadores competem pelo arquivo local que contém as informações sobre os preços dos produtos que podem ser vendidos pelo PDV.

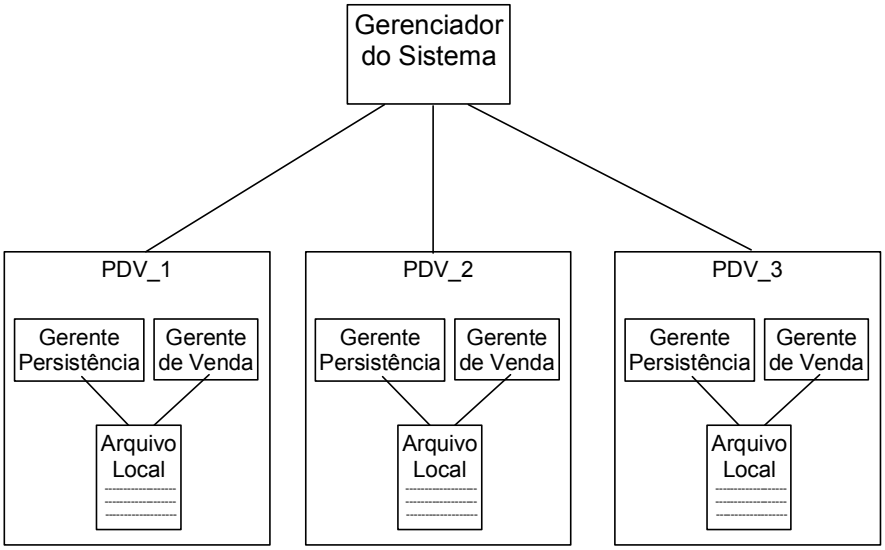

Figura 22. Esboço do Sistema Varejista

A atualização de preços de uma lista de produtos em um sistema varejista como o apresentado na Figura 22 é um processo de natureza concorrente, visto que o preço do produto deve ser atualizado simultaneamente em todos os PDVs, sob pena de termos o mesmo produto com preços diferentes entre os PDVs do sistema. Logo, sempre que o Gerenciador do Sistema for atualizar os preços de uma lista de produtos, este deverá solicitar atualizações nos arquivos de preço de produtos em todos os PDVs, que deverão realizar esta operação paralelamente.

Caso a lista de produtos a terem seus preços alterados seja muito grande, pode ocorrer diferença de preço entre os PDVs, visto que cada PDV tem performances diferentes. Devido a este problema, os PDVs devem cooperar para que a atualização de

preços seja sincronizada para cada produto, atualizando o mesmo produto aproximadamente no mesmo instante em todos os PDVs.

Dentro de cada PDV o Gerente de Persistência e o Gerente de Vendas também trabalham concorrentemente, mas neste caso, eles competem pelo recurso do arquivo local onde estão persistidas as informações sobre os preços dos produtos. Além de competir, estes dois gerenciadores cooperam, visto que o Gerente de Vendas deve verificar se o produto a ter seu preço alterado pelo Gerente de Persistência de dados já foi vendido durante a venda corrente, com a finalidade de evitar que um mesmo produto seja vendido por preços diferentes em uma única venda.

7.3.1 Motivação e Solução

Dentre as funcionalidades de um sistema varejista, escolhemos resolver o problema de alteração de preços dos produtos sincronizadamente, visto que se trata de um problema onde o emprego da Arquitetura FT-SC3 comprovadamente agrega valor na composição de sistemas que competem e cooperam, oferecendo um serviço que seja tolerante a falhas.

Grande parte dos sistemas varejistas implantados nos supermercados e demais lojas possuem dificuldades para atualizar sincronizadamente os preços dos produtos durante o funcionamento do sistema. Esta dificuldade se deve principalmente porque não se consegue resolver concorrência competitiva e cooperativa utilizando apenas o conceito de transação.

Entre as várias soluções que tivemos contato, as soluções propostas não atendiam a contento todos os requisitos. As soluções encontradas funcionavam das seguintes formas: (i) o sistema garante preços iguais, mas simplesmente não atualiza os preços dos produtos durante o funcionamento da loja; (ii) o sistema garante preços iguais, mas somente atualiza todos registros dos produtos de uma única vez ou atualiza um único produto por vez; ou (iii) o sistema permite que uma lista de produtos tenha seus preços atualizados, mas dependendo da performance de cada PDV, os produtos são atualizados em momentos diferentes, gerando inconsistência de preço de um produto entre os PDVs. Outro problema encontrado foi a falta de estruturação do sistema para tratar o problema de sincronismo na atualização de preços.

Para solucionar o problema, resolvemos fazer duas composições de sistemas. Na primeira, é feita a coordenação entre os PDVs, para isto utilizamos: (i) o Gerenciador do

Sistema como cliente da aplicação, que solicita a atualização de produtos; e (ii) o ContratoSincronizador, o contrato de composição que sincroniza os PDVs no momento em que eles estiverem atualizando uma lista de preços. Na segunda composição, é feita a coordenação dentro dos PDVs. Para isto criamos o ContratoCoordenador para coordenar: (i) a colaboração entre o Gerente de Persistência e o Gerente de Vendas; e (ii) a competição entre o Gerente de Persistência e o Gerente de Vendas pelo acesso ao arquivo local.

7.3.2 Coordenação entre Gerenciador do Sistema e PDVs

O ContratoSincronizador foi criado para controlar a atualização de uma lista de produtos para que cada produto seja atualizado ao mesmo momento em todos PDVs. A Figura 23 apresenta o modelo da arquitetura definido para a coordenação entre o Gerenciador do Sistema e os PDVs, onde se pode observar que os PDVs foram inseridos na arquitetura como componentes participantes, e o Gerenciador do Sistema atuando como o componente cliente da aplicação.

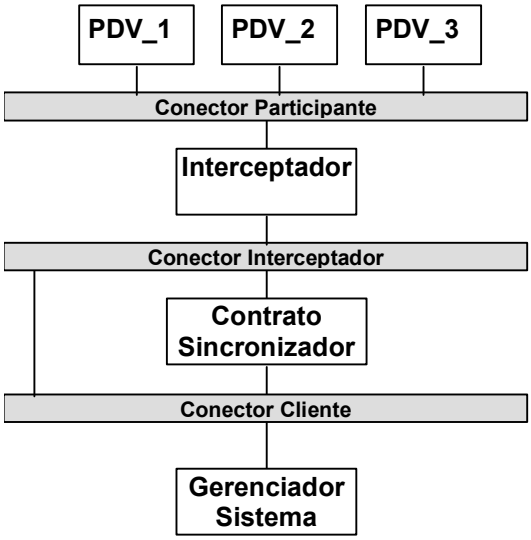

Figura 23. Coordenação entre o Gerenciador do Sistema e os PDVs

A Figura 24 mostra o diagrama de seqüência da composição do Gerenciador do Sistema com os PDVs. Inicialmente, o Gerenciador do Sistema solicita que seja feita uma atualização numa lista (listaProd) contendo novos preços para o produto1 e o produto2. O

Interceptador verifica que esta solicitação de serviço ativa o gatilho do ContratoSincronizador e redireciona a mensagem para este contrato de composição.

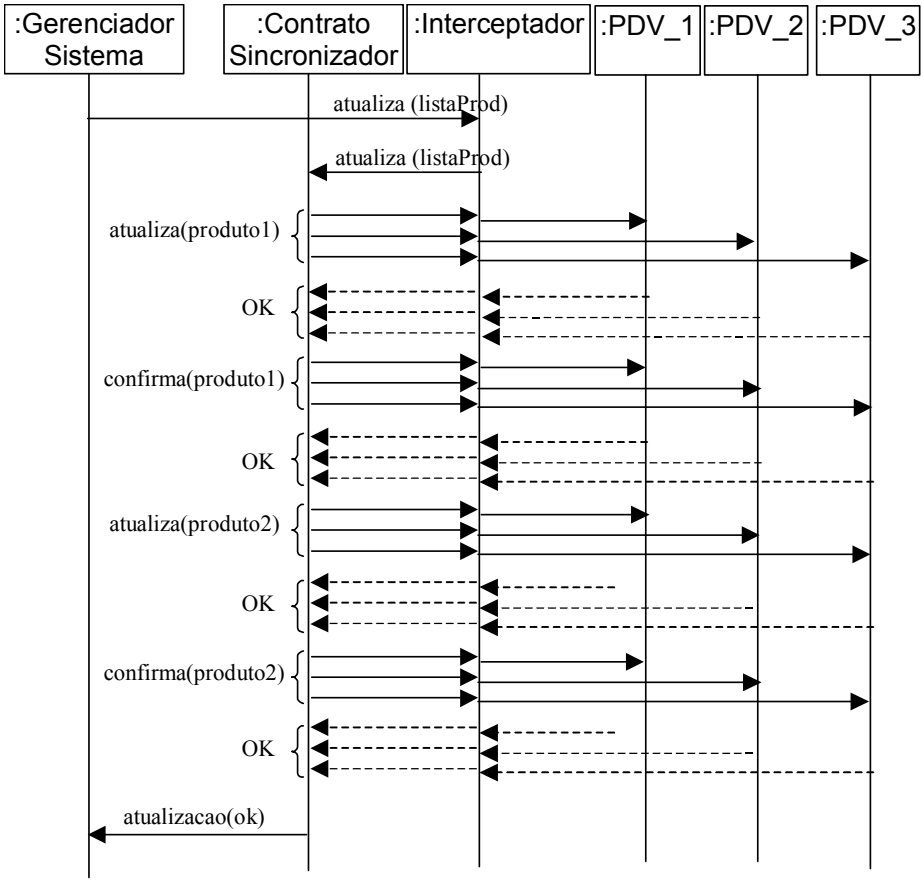

Figura 24. Diagrama de Seqüência entre Gerenciador do Sistema e PDVs

O ContratoSincronizador armazena a lista com as novas informações dos dois produtos e inicia a atualização produto a produto nos PDVs. A atualização de cada produto é feita em dois passos. Primeiro, o ContratoSincronizador envia uma solicitação de atualização de produto aos PDVs. Depois que todos PDVs atualizam o produto, o ContratoSincronizador envia uma mensagem de confirmação para confirmar aos PDVs que o produto foi atualizado corretamente em todos os demais PDVs e que não há possibilidades de abortar a operação.

Caso um dos PDVs levante uma exceção indicando que não atualizou o produto, o ContratoSincronizador propaga uma exceção de aborto aos demais PDVs para que desfaçam a atualização do produto. Desfeito a atualização, o ContratoSincronizador

continua atualizando os demais produtos que restam na lista. Terminado a atualização da lista, o ContratoSincronizador retorna uma exceção ao Gerenciador do Sistema informando qual produto não foi atualizado.

A grande vantagem de utilizar o ContratoSincronizador está em deixar o Gerenciador do Sistema livre da responsabilidade de atualizar produto por produto. Neste caso, o ContratoSincronizador pode estar localizado numa máquina que possua acesso mais rápido que o Gerenciador do Sistema. Imagine o caso onde o Gerenciador do Sistema fica em outra cidade, o ContratoSincronizador poderia ficar numa máquina dentro da loja, diminuindo o tempo gasto e o tráfego de mensagens caso o Gerenciador do Sistema fizesse a sincronização.

7.3.3 Coordenação dentro dos PDVs

Dentro de cada PDV é necessário um contrato de composição que seja responsável por coordenar a cooperação entre o Gerente de Persistência e o Gerente de Venda, além da competição pelo acesso ao arquivo local. A Figura 25 apresenta o modelo da arquitetura definido para a coordenação dentro de cada PDV, onde se pode observar o Gerente de Persistência e o Gerente de Vendas como componentes participantes, e o Controlador do PDV atuando como o componente cliente da aplicação.

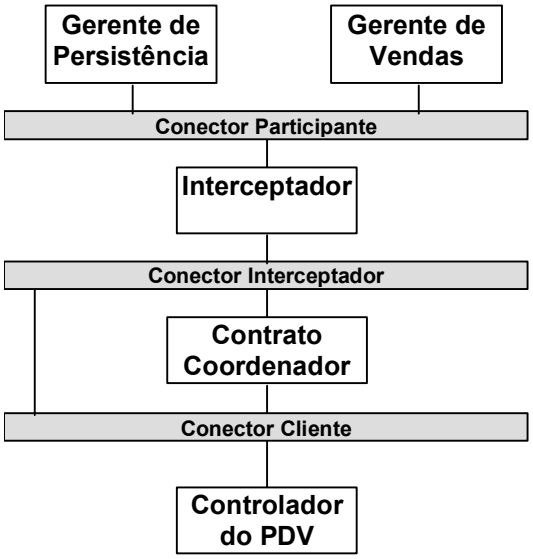

Figura 25. Coordenação dentro dos PDVs

A Figura 26 mostra que o ContratoCoordenador intercepta a solicitação de atualização do produto1, e ao interceptar esta mensagem, o ContratoCoordenador envia duas solicitações de atualizações para o Gerente de Persistência e o Gerente de Venda.

Ao receber a solicitação de atualização de um produto, o Gerente de Persistência desabilita a leitura do produto1 e atualiza o produto1, modificando o preço deste produto. Já o Gerente de Vendas trava o fim da venda, evitando que se feche a venda corrente no PDV, e caso o produto a ser alterado já tenha sido vendido na venda corrente, é dado um desconto, se o novo preço for menor, ou continua vendendo o produto1 pelo preço inicial até o fim da venda corrente, caso o produto1 tenha aumentado de valor. Feito isto, o ContratoCoordenador notifica que a atualização do produto1 foi realizada. Como visto na Seção 7.3.2 o ContratoSincronizador deverá logo em seguida enviar uma mensagem de confirmação da atualização em processo para efetivá-la.

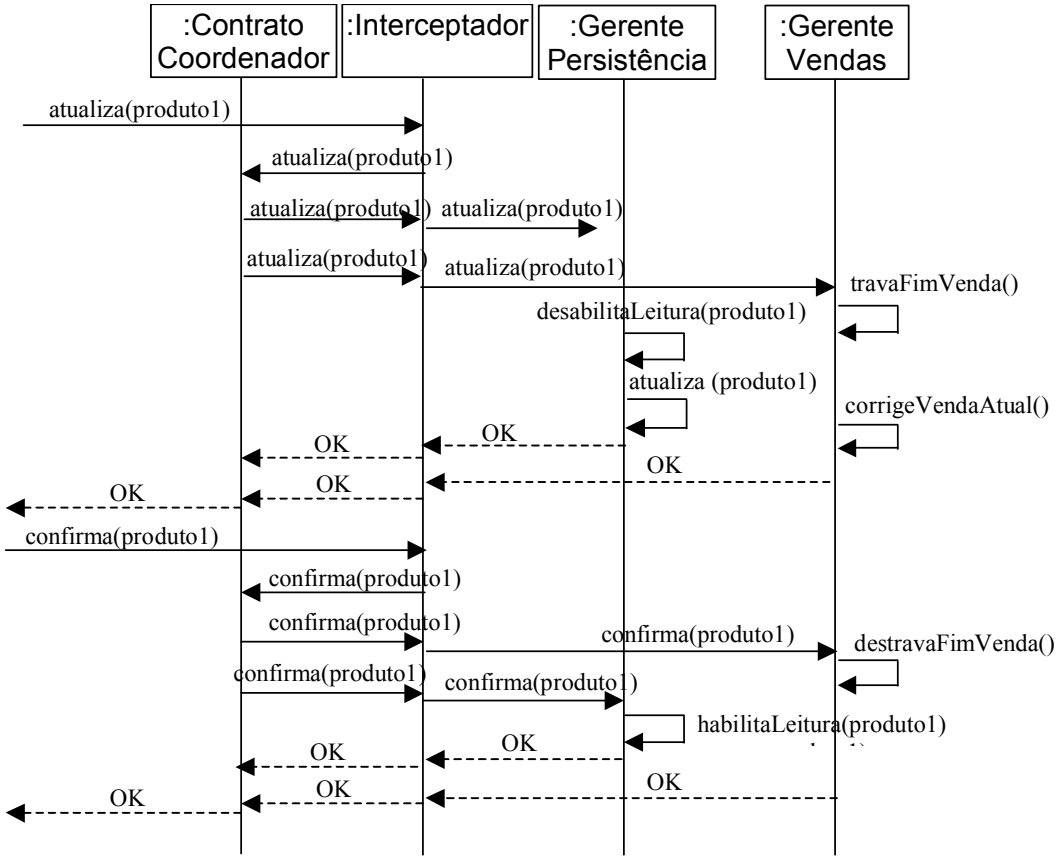

Figura 26. Diagrama de seqüência entre os sub-componentes do PDV

O ContratoCoordenador intercepta a mensagem de confirmação e envia duas mensagens de confirmação ao Gerente de Persistência e ao Gerente de Vendas. O Gerente de Persistência habilita o registro do produto1 para que possa ser lido e o Gerente de Vendas destrava o fim da venda permitindo que se possa fechar a venda.

Até a confirmação da atualização, é possível desfazer a atualização de um produto em ambos gerentes de venda e persistência de um PDV. Com isto, caso não seja enviado a mensagem de confirmação para a conclusão da atualização ou algum dos sub-componentes do PDV levante uma exceção durante a realização da operação de atualização, o ContratoCoordenador envia uma exceção de aborto aos sub-componentes que estejam à espera da confirmação. Ao receber uma exceção de aborto, o Gerente de Vendas volta a vender o produto com o preço antigo, corrigindo novamente a venda corrente, e o Gerente de Persistência recupera as informações que o produto tinha antes do início da atualização e a grava no arquivo local.

7.4 Considerações Finais

Neste Capítulo foram apresentados três estudos de casos implementados com o intuito de comprovar o comportamento da Arquitetura FT-SC3 na solução de problemas práticos. Para a realização destes estudos de casos utilizamos o Framework FT-SC3, que se comportou a contento, mesmo tendo sido utilizado no desenvolvimento de sistemas de contextos distintos.

Na Seção 7.1, foi apresentado um estudo de caso simples para o controle de contas bancárias tolerante a falhas que nos auxiliou durante o desenvolvimento da Arquitetura FT-SC3. Neste estudo de caso pudemos verificar o trabalho de interceptação de mensagens e como o contrato de composição pode ser utilizado ao mesmo tempo para coordenar e tratar as possíveis exceções levantadas pelos participantes.

A Seção 7.2 apresentou um sistema bancário onde os requisitos foram evoluindo com o tempo. Inicialmente, havia apenas um sistema bancário onde operações de saque e depósito podiam ser realizadas. Com a evolução do sistema, desejou-se compor dois sistemas bancários para permitir a realização de transferência de fundos entre bancos distintos. A terceira evolução ocorreu com o advento de um imposto sobre movimentação

financeira (CPMF) que tributava toda transferência de fundos entre contas correntes pertencentes a pessoas diferentes. Neste estudo de caso, a Arquitetura FT-SC3 se mostrou bastante útil durante a evolução de sistemas onde novas regras de negócios foram acrescentadas, tendo em vista que a adição de novos contratos de composição exigiu pouca implementação. Outro importante ponto deste estudo de caso foi o aninhamento de ações coordenadas por contratos, além de observar a viabilidade da realização de tratamento de exceções quando existe um aninhamento de ações.

A Seção 7.3 mostra um estudo de caso que soluciona de forma apropriada um problema encontrado nos sistemas varejista, que constantemente causam erros nos preços dos produtos quando é realizada uma venda. Neste estudo de caso, mostramos como a Arquitetura FT-SC3 é mais eficiente na solução de um problema que possua concorrência competitiva e cooperativa entre os componentes participantes. Além disso, foi mostrada uma forma de aninhamento diferente do apresentado na Seção 7.2, visto que na composição de componentes um dos participantes é formado por um sistema composto.

Capítulo 8

Conclusões e Trabalhos Futuros

Neste capítulo é feita uma análise dos resultados obtidos, onde são expostas as lições aprendidas. Além disso, são comentadas as possíveis melhorias para o trabalho desenvolvido, acrescentando algumas sugestões para trabalhos a serem realizados no futuro. Por fim, é feito um breve resumo das contribuições deste trabalho.

8.1 Análise Global dos Resultados

Apesar da grande abrangência de problemas que podem ser solucionados pela abordagem arquitetural proposta neste trabalho, deve-se ter o cuidado de avaliar se o sistema confiável a ser composto realmente necessita de um esquema de tratamento de exceções tão sofisticado, onde os sistemas-componentes competem por recursos ao mesmo tempo em que cooperam durante a realização de novos serviços. Observamos que a maioria dos problemas não necessita de todos estes recursos. Logo, a primeira análise a ser feita é se todo o aparato exposto neste trabalho é realmente necessário para resolver o problema, ou se este mesmo problema pode ser resolvido de forma mais simples.

Os estudos de casos foram bastante ilustrativos, pois se pôde observar mais claramente as vantagens e desvantagens do uso da Arquitetura FT-SC3. Isto permitiu que elaborássemos uma síntese dos benefícios e limitações encontradas na abordagem adotada nesta pesquisa.

8.1.1 Benefícios

Os estudos de casos mostraram que independência entre os componentes clientes (que fazem as solicitações dos serviços) e os componentes participantes (que provêem os serviços) é uma meta alcançável quando se usa a Arquitetura FT-SC3. Em todos estudos de casos do Capítulo 7 os participantes funcionavam independentemente do fato de serem

parte de um sistema composto por meio da integração de participantes. Além disso, o cliente da aplicação não dependia de como os serviços oferecidos pelo sistema eram implementados. O cliente da aplicação apenas tinha conhecimento dos papéis que o sistema composto era capaz de realizar.

Os participantes se mostraram totalmente sem conhecimento da regra de negócio, visto que estas últimas são implementadas pelos contratos de composição. Tais contratos de composição são capazes de interceptar as solicitações de serviços enviadas aos participantes sem que os participantes tomem conhecimento. Os contratos de composição se mostraram capazes de impor novos requisitos aos serviços oferecidos pelos participantes, além de facilitarem a extensão e introdução de novas regras de negócios, simplesmente com a adição de novo contratos a camada de coordenação. A experiência nos estudos de casos mostrou que a adição de novos contratos de composição num sistema pré-existente exige quase nenhuma implementação, com exceção da própria implementação do contrato.

O uso do ambiente de modelagem baseado no GME durante a descrição da configuração da arquitetura para os estudos de casos se mostrou benéfico, devido a sua praticidade se comparado com o trabalho de programação que seria necessário para obter a construção dos esqueletos das classes do sistema, gerando uma documentação arquitetural de fácil manutenção.

8.1.2 Limitações

Durante a composição de sistemas-componentes verificamos que os serviços compostos oferecidos pela camada de coordenação ficavam limitados aos serviços oferecidos pelas interfaces dos componentes participantes, ou seja, durante a evolução do sistema novas regras de negócio são inseridas e novos serviços providos pelos participantes se fazem necessários. Neste caso, é preciso adaptar os sistemas-componentes para atenderem as novas necessidades. É importante ressaltar que, às vezes, é mais econômico inserir novos participantes do que modificar um sistema desenvolvido por terceiros que não é capaz de realizar todos os papéis exigidos.

O esquema de tratamento de exceções mostrou-se eficiente nos estudos de casos apresentados, onde os contratos eram sempre capazes de recuperar os participantes retornando para o estado original antes do início da ação coordenada por contratos. Porém

não se pode esquecer que quando este tipo de recuperação não é possível, os participantes são deixados em um estado errôneo.

As limitações até agora expostas são inerentes ao modelo computacional assumido na abordagem adotada neste trabalho, visto que os componentes participantes podem encapsular sistemas inteiros que podem ter sido construídos sem pretensão de serem tolerantes a falhas.

A organização dos contratos de composição de forma hierárquica foi um dos aspectos para a diminuição da complexidade na comunicação da Arquitetura FT-SC3. Esta estrutura hierárquica ajuda o interceptador no momento em que ele necessita decidir qual contrato de composição irá receber a mensagem quando múltiplos gatilhos são ativados simultaneamente. O estudo de caso da Seção 7.2.3 mostra um exemplo claro desta situação. Entretanto, esta organização dos contratos de composição em camadas impõe uma ordem parcial sobre a execução dos contratos.

Uma das limitações da abordagem arquitetural proposta neste trabalho ocorre quando pensamos num gigantesco SdSs, formado por centenas de contratos e participantes distribuídos numa rede, onde a manutenção é dada por diversas pessoas que não possuem o conhecimento detalhado de todo o sistema. Neste caso, um sistema-componente pode realizar um papel numa determinada ação e, ao mesmo tempo, realizar solicitações de serviços ao SdSs. Ou seja, um sistema-componente exerce ao mesmo tempo tanto o papel de participante como de cliente da aplicação. Este tipo de situação, apesar de ser útil em alguns casos, não é permitido pelo estilo C2. Além disso, uma série de dificuldades seriam impostas quando pensamos no controle de ciclos durante a composição de sistemas-componentes que participam da realização de um serviço solicitado por ele. Acreditamos que a criação de uma camada de configuração, como proposta na Seção 8.3, venha a ser a solução para tratar esta limitação.

8.2 Contribuições

Este trabalho apresenta as seguintes contribuições:

- **A Arquitetura FT-SC3:** Com o objetivo de guiar o desenvolvimento de sistemas tolerantes a falhas, realizados a partir da composição de sistemas-

componentes, desenvolvemos uma arquitetura de software, denominada Arquitetura FT-SC3, que utiliza o estilo C2 e o conceito de contratos de coordenação, além de incluir um esquema de tratamento de exceções baseado em ações atômicas coordenadas;

- **O Framework FT-SC3:** Para apoiar o desenvolvimento de sistemas baseado na Arquitetura FT-SC3, algumas ferramentas foram criadas. Uma destas ferramentas foi um *framework* orientado a objetos escrito em Java, denominado de Framework FT-SC3. Este *framework* implementa partes da Arquitetura FT-SC3. Além disso, foi criado um pequeno roteiro de como utilizar este *framework* durante a composição de sistemas;
- **Um Ambiente de Modelagem:** Outra ferramenta criada para auxiliar a elaboração de sistemas baseado na Arquitetura FT-SC3 foi o desenvolvimento de um ambiente de modelagem baseado no *Generic Modeling Environment* (GME). Este ambiente de modelagem permite a construção de modelos a partir de diagramas, que são checados e podem ser exportados e importados a partir de arquivos XML, além de gerar o código das assinaturas das classes modeladas; e
- **Projeto e Implementação de três Estudos de Casos:** Com o objetivo de comprovar a aplicabilidade do uso da Arquitetura FT-SC3 na elaboração de Sistemas-de-Sistemas, desenvolvemos três estudos de casos.

Publicações

Este trabalho resultou em duas publicações em conferências internacionais. São elas:

- Component Integration using Composition Contracts with Exception Handling.

  Autor: Ricardo M. Da Silva, Paulo A. C. Guerra e Cecília M. F. Rubira.

  Conferência: Exception Handling in Object Oriented Systems Workshop (EHOOS'2003) – ECOOP'2003 – Darmstadt, Alemanha, Agosto de 2003.
- An Architectural Solution for Component-System Integration with Exception Handling.

  Autor: Ricardo M. Da Silva.

Conferência: Workshop de Teses e Dissertações (WTD'2003) – LADC'2003 – São Paulo, Brasil, Outubro de 2003.

Submissões

- An Architectural Approach for Fault-Tolerant System Composition.

  Autor: Ricardo M. Da Silva, Fernando Castor Filho, Paulo A. C. Guerra e Cecília M. F. Rubira.

  Conferência: Submetido ao 8th IEEE International Symposium on High Assurance Software Engineering (HASE'2004) – Tampa, Flórida, EUA, Março de 2004.

8.3 Trabalhos Futuros

Na Arquitetura FT-SC3, somente um componente interceptador é usado para todo o sistema. Isto pode se tornar um gargalo para aplicações que ofereçam serviços com alto grau de disponibilidade e que exijam grande quantidade de processamento. Uma possível solução a ser estudada é a criação de instâncias distribuídas do interceptador para que seja feito um balanceamento de cargas entre estas instâncias.

A versão atual do Framework FT-SC3 descrito na Seção 6.1 somente permite configuração estática para o componente interceptador. Como as configurações arquiteturais dos contratos e participantes dependem da configuração do interceptador, estes elementos não poderão ser inseridos dinamicamente no sistema. Uma extensão do Framework FT-SC3 poderia aumentar a disponibilidade, visto que o sistema não necessitaria interromper a sua execução para que novos contratos e participantes fossem introduzidos. Para esta modificação no Framework FT-SC3 será necessário reavaliar a Arquitetura FT-SC3, pois muito provavelmente será necessária a introdução de uma camada de configuração e de contexto (Seção 3.4) que ofereçam serviços de reconfiguração sobre a configuração do sistema.

O estilo arquitetural C2 apresentou uma série de vantagens no que diz respeito à reutilização e extensibilidade. Em contrapartida, e como descrito na seção 8.1.2, o estilo C2 limitou a arquitetura no tocante à flexibilidade na composição do sistema. O estudo de uma nova arquitetura de software que não utilize o estilo C2 seria um trabalho de grande

contribuição, visto que poderia eliminar várias das restrições impostas por este estilo arquitetural.

Nos componentes arquiteturais descritos na Seção 5.4, existe uma separação explícita entre o comportamento normal e o comportamento excepcional. Entretanto, até o presente momento esta separação é puramente estrutural e nenhuma restrição sobre as interações das atividades normais e excepcionais foi definida. Portanto, um dos trabalhos futuros a serem realizados, consiste na introdução do conceito de componente C2 ideal (iC2C) como um meio de definir componentes tratadores de exceção, e regiões de confinamento e avaliação do dano [38]. A integração dos sistemas tratadores de exceções desenvolvidos por Castor et al [13] com os componentes da Arquitetura FT-SC3 (Seção 5.4) irão garantir que as partes que se interagem e a partes interativas no sistema sejam confiáveis.

8.4 Conclusões

Neste trabalho foi proposta uma solução arquitetural para o desenvolvimento de sistemas de software confiáveis desenvolvidos a partir de sistemas-componentes autônomos, heterogêneos e não confiáveis que executam concorrentemente (Sistemas-de-Sistemas). A solução aqui descrita privilegiou os atributos de adaptabilidade, extensibilidade e confiabilidade, colocando estes como os principais requisitos para a Arquitetura FT-SC3.

Inicialmente, a intenção era desenvolver uma infraestrutura conceitual onde vários sistemas-componentes pudessem ser conectados com o mínimo de esforço, de forma que depois de conectados estes sistemas-componentes passassem a interagir, produzindo assim novos serviços confiáveis aos clientes desta nova aplicação.

Devido aos resultados apresentados no Capítulo 7, acredita-se que este objetivo foi alcançado. O conceito de ações atômicas coordenadas se mostrou um mecanismo bastante adequado para a composição de sistemas-componentes. Após avaliar os resultados empíricos, observamos que a Arquitetura FT-SC3 também permite que os contratos sejam facilmente compostos, promovendo o reuso de regras de negócios existentes e simplificando a tarefa de introduzir novas regras.

A separação clara entre a camada computacional e a camada de coordenação na Arquitetura FT-SC3, permitiu que os participantes não tivessem nenhum conhecimento das regras de negócios. A substituição e atualização dos componentes participantes e as inconformidades arquiteturais [29] podem ser tratadas por meio de *wrappers* [16]. Além disso, novos contratos podem ser adicionados na camada de coordenação, acompanhando a evolução do sistema. Esta característica é de grande importância para sistemas coorporativos que constantemente necessitam alterar sua regra de negócio sem que haja modificação nas funcionalidades mais importantes do sistema.

Bibliografia

[1] L. F. Andrade & J. L. Fiadeiro. Evolution by Contract. Em 14th *ECOOP'00 – 2nd Workshop on Object-Oriented Architectural Evolution,* Sophia Antipolis and Cannes, France Junho 2000. (versão expandida submetida para publicação está disponível em http://www.fiadeiro.org/jose sob o título: Contracts: Supporting Architecture-Based Evolution).

[2] L. Andrade & J. L. Fiadeiro. Coordination for Orchestration. Em *5th International Conference on Coordination Languages and Models.* Lecture Notes in Computer Science 2315, 5-13. Springer-Verlag, 2002.

[3] L. F. Andrade & J. L. Fiadeiro. Coordination: the Evolutionary Dimension. Em *TOOLS Europe 2001*, 136-147, W.Pree (ed), IEEE Computer Society Press 2001, 2001.

[4] L. F. Andrade, J. L. Fiadeiro. Feature Modelling and Composition with Coordination Contracts. Em *Feature Interaction in Composed System*, ECOOP'2001, 49-54, Budapest, Hungria, Junho 2001.

[5] L. F. Andrade, J. L. Fiadeiro, J. Gouveia, G. Koutsoukos, A. Lopes & M. Wermelinger. Coordination Patterns for Component-Based Systems. Em *5th Brazilian Symposium on Programming Languages (SBPL'2001)*, B29-B39, Universidade Federal do Paraná, Brasil, Maio 2001.

[6] A. Avizienis. *Toward Systematic Design of Fault-Tolerant Systems*. IEEE Computer, 1997.

[7] L. Bass, P. Clements & R. Kazman. *Software Architecture in Practice.* Addison-Wesley, 1998.

[8] D. M. Beder, *Uma Arquitetura de Software baseada em Padrões de Projeto para o Desenvolvimento de Aplicações Concorrentes Confiáveis.*

Tese de Doutorado, Instituto de Computação – Universidade Estadual de Campinas, Maio 2001.

[9] D. M. Beder, B. Randell, A. Romanovsky, and C. M. F. Rubira. On Applying Coordinated Atomic Actions and Dependable Software Architectures for Developing Complex Systems. Em *4th IEEE International Symposium on Object-Oriented Real-time Distributed Computing (ISORC'2001)*, 103-112, Magdeburg, Germany, Maio 2001.

[10] S. Bodoff, D. Green, K. Haase, E. Jendrock, M. Pawlan & B. Stearns. *Tutorial do J2EE*. Ed. Campus, 2002.

[11] G. Booch, J. Rumbaugh & I. Jacobson. *The Unified Modeling Language User Guide*. Addison-Wesley, 1999.

[12] A. W. Brown. *Large-Scale, Component-Based Development*. Prentice Hall, 2000.

[13] F. Castor Filho, P. A. C. Guerra & C. M. F. Rubira. An Architectural-Level Exception-Handling System for Component-Based Applications. Em *1st Latin-American Symposium on Dependable Computing (LADC'2003),* Lecture Notes in Computer Science 2847, 321-340, Springer-Verlag, São Paulo, Brasil, Outubro 2003.

[14] J. Cheesman, J. Daniels. *UML Components - A Simple Process for Specifying Component-Based Software*. Addison-Wesley, 2000.

[15] A. Cockburn. *Writing Effective Use Cases.* Addison-Wesley, 2000.

[16] F. M. Coelho. *Uso de Componentes de Software no Desenvolvimento de Frameworks Orientados a Objetos*. Dissertação de Mestrado, Instituto de Computação - Universidade Estadual de Campinas, Brasil, Dezembro 2002.

[17] A. S. Tanenbaum. Modern Operating Systems, Second Edition, Prentice Hall, 2001.

[18] F. Cristian. Exception Handling. Em Dependability of Resilient Computers (ed. T. Anderson), 68-97, Blackwell Scientific Publications, 1989.

[19] I. Crnkovic & M. Larsson. *Building Reliable Component-Based Software Systems*. Artech House, 2002.

[20] Dependable Systems of Systems Project. *Architecture and Design - Initial Results on Architectures and Dependable Mechanisms.* Deliverable IC2, (IST– 1999–11585), Abril 2000.

[21] Dependable Systems of Systems Project. *Preliminary Version of Conceptual Model: Basic Concepts 1.* Deliverable BC1, (IST– 999–11585), Setembro 2000.

[22] Dependable Systems of Systems Project. *State of the Art Survey*. Deliverable BC2, (IST– 999–11585), Abril 2000.

[23] H. E. Ericksson & M. Penker. *UML Toolkit*. John Wiley & Sons, 1997.

[24] G. R. M. Ferreira. *Tratamento de Exceções no Desenvolvimento de Sistemas Confiáveis Baseados em Componentes*. Dissertação de Mestrado, Instituto de Computação - Universidade Estadual de Campinas, Brasil, Dezembro 2001.

[25] A. F. Garcia. *Tratamento de Exceções em Sistemas Concorrentes Orientados a Objetos*. Dissertação de Mestrado, Instituto de Computação – Universidade Estadual de Campinas, Brasil, Março 2000.

[26] A. F. Garcia, D. M. Delano & C. M. F. Rubira. An Exception Handling Software Architecture for Developing Fault-Tolerant Software. Em *5th IEEE International Symposium on High Assurance Systems Engineering (HASE'00)*, 311-320, Albuquerque, New Mexico, USA, Novembro 2000.

[27] A. F. Garcia, D. M. Delano & C. M. F. Rubira. An Object-Oriented Exception Handling Mechanism for Developing Dependable Component-Based Object-Oriented Software. Em *10th International Symposium on Software Reliability Engineering*, 52-61, Florida, USA, Novembro 1999.

[28] A. F. Garcia, D. M. Delano & C. M. F. Rubira. Unified Meta-Level Software Architecture for Sequential and Concurrent Exception Handling. Em *The Computer Journal, Special Issue on High Assurance Systems Engineering*, 44(6):569-587, 2001.

[29] D. Garlan, R. Allen, and J. Ockerbloom. Architectural mismatch: Why reuse is so hard. *IEEE Software*, 12(6):17-26, Novembro 1995.

[30] J. Gosling, B. Joy, and G. Steele. *The Java Language Specification*. Addison-Wesley, 1996.

[31] P. Guerra, C. Rubira, and R. de Lemos. An Idealized Fault-Tolerant Architectural Component. Em *24th International Conference on Software Engineering (ICSE'02) - Workshop on Architecting Dependable Systems*, 16-20, Orlando, FL, Maio 2002.

[32] Institute for Software Integrated Systems. *GME 3 User's Manual*. Vanderbilt University, Maio 2003.

[33] I. Jacobson, G. Booch & J. Rumbaugh. *The Unified Software Development Process.* Addison-Wesley, 1999.

[34] J. C. Knight, SafetyCritical Systems: Challenges and Directions (summary of state-of-the-art presentation). Em *24th International Conference on Software Engineering (ICSE'02)*, 547-550, Orlando, FL, Maio 2002.

[35] H. Kreger. Web Services Conceptual Architecture (WSCA 1.0). IBM Software Group White Paper. Maio 2001.

[36] B. W. Lampson. Atomic Transactions. Em *Distributed Systems: Architecture and Implementation*, Lecture Notes in Computer Science 105, 254-259. Springer-Verlag, Berlin, 1981.

[37] A. Ledeczi, M. Maroti, A. Bakay, G. Karsai, J. Garret, C. Thomason IV, G. Nordstrom, J. Sprinkle & P. Volgyesi. The Generic Modeling environment. Em *Workshop on Intelligent Signal Processing*, Budapest, Hungria, Maio 2001.

[38] P.A. Lee e T. Anderson. *Fault Tolerance: Principles and Practice.* Springer-Verlag, 2ª edição, 1990.

[39] R. Lemos. Describing Evolving Dependable Systems using Co-operative Software Architectures. Em *21th IEEE International Conference on Software Maintenance (ICSM'01)*, 320-329, Florence, Itália, Novembro 2001.

[40] R. Lemos & J. L. Fiadeiro. An Architectural Support for Self-Adaptative Software for Treating Faults. Em *1st ACM SIGSOFT Workshop on Self-Healing Systems (WOSS'02)*, 39-42, Charleston, SC, USA, Novembro 2002.

[41] P. Maes. Concepts and Experiments in Computational Reflection. Em *ACM SIGPLAN Notices*, 22(12):147-155, Orlando, Florida, Março 1987.

[42] M. Maier. Architecture Principles for Systems-of-Systems. System Engineering, 6th Annual Symposium of INCOSE, 567-574, 1996.

[43] N. Medvidovic, P. Oreizy & R. N. Taylor. Reuse of Off-the-Shelf Components in C2-Style Architectures. Em *19th International Conference on Software Engineering (ICSE'97) - Symposium on Software Reusability (SSR'97)* ,692 - 700, Boston, Massachusetts, 1997.

[44] R. de Lemos & A. Romanovsky. Exception Handling in a Cooperative Object-Oriented Approach". Em *2nd IEEE International Symposium on*

Object-oriented Real-time Distributed Computing, 3-13, Saint Malo, France, 1999.

[45] Object Management Group. *Response to the UML 2.0 OCL RfP (ad/2000-09-03 "UML 2.0 OCL",* 2nd revised submission, 2003. http://www.omg.org/cgi-bin/doc?ad/2003-01-07.

[46] Object Management Group, *The Unified Modeling Language v1.5, 2003*. http://www.omg.org/cgi-bin/doc?formal/03-03-01

[47] D. E. Perry and A. L. Wolf. Foundations for the Study of Software Architecture. *SIGSOFT Software Engineering Notes*, 17(4):40–52, Outubro 1992.

[48] P. F. Pires. Building Reliable Web Services Compositions. Em *NET. Object Days Conference (WS-RDS'02)*, 551-562, Erfurt, Alemanha, Outubro 2002.

[49] B. Randell, A Romanovisky, R.J. Stroud, J. Xu & A.F. Zorzo. Coordinated Atomic Actions: from Concept to Implementation. *Relatório Técnico 595*, Department of Computer Science, University of Newcastle upon Tyne, UK, 1997.

[50] B. Randell. System Structures for Software Fault Tolerance. Em *IEEE Transactions on Software Engineering*, SE--1(3):221-232, Junho 1975.

[51] A. Romanovsky. Looking Ahead in Atomic Actions with Exception Handling. Em *20th IEEE Symposium on Reliable Distributed Systems (SRDS-2001)* - IEEE CS, New Orleans, USA, 142-151, Outubro 2001.

[52] A. Romanovsky. On Structuring Cooperative and Competitive Concurrent Systems. Em *The Computer Journal*, 42(8):627-637 Oxford University Press/British Computer Society, ISSN: 0010-4260, 1999.

[53] R. M. Silva. An Architectural Solution for Component-System Integration with Exception Handling. Em *1st Latin American Symposium on Dependable Computing (LADC'2003)* - Workshop de Teses e Dissertações, 37-42, São Paulo, Brasil, Outubro 2003.

[54] R. M. Silva, F. Castor Filho, P. A. C. Guerra & C. M. F. Rubira. *An Architectural Approach for Fault-Tolerant System Composition.* Submetido ao IEEE International Symposium on High-Assurance Systems Engineering (HASE'04), 2004.

[55] R. M. Silva, P. A. C. Guerra & C. M. F. Rubira. Component Integration using Composition Contracts with Exception Handling. Em *ECOOP'2003* - Exception Handling in Object Oriented Systems Workshop, 1-20, Darmstadt, Alemanha, Agosto 2003.

[56] J. Sometinger. *Software Engineering with Reusable Components*. Springer, 1997.

[57] Sun Microsystems. *Enterprise Java Beans (EJB)*, 1998. http://java.sun.com/products/ejb/docs.html

[58] F. Tartanoglu, V. Issarny, A. Romanovsky & N. Levy. Dependability in the Web Services Architecture. Em *24th International Conference on Software Engineering (ICSE'02) - Workshop on Architecting Dependable Systems*, Lecture Notes in Computer Science 2677, Springer-Verlag, 6-10, Orlando, FL, Maio 2002.

[59] R. N. Taylor, N. Medvidovic, K. M. Anderson, E. J. Whitehead & J. E. Robbins. A Component- and Message-Based Architectural Style for GUI Software. Em *17th International Conference on Software Engineering (ICSE'95)*, 295-304, Seattle, WA, Abril 1995.

[60] UCI. ArchStudio 3.0 homepage, 2003. http://www.isr.uci.edu/projects/archstudio.

[61] W3 Consortium. *Extensible Markup Language (XML) homepage*, 2003. http://www.w3.org/XML/

[62] G. M. Weiss. Adaptação de Componentes de Software para o Desenvolvimento de Sistemas Confiáveis. Dissertação de Mestrado, Instituto de Computação - Universidade Estadual de Campinas, Março 2001.

[63] M. Wermelinger, J. L. Fiadeiro, L. Andrade, G. Koutsoukos & J. Gouveia. Separation of Core Concerns: Computation, Coordination and Configuration. Em *OOPSLA 2001*, *Workshop on Advanced Separation of Concerns in Object-Oriented Systems*, 14-18, Tampa Bay, Florida, USA, Outubro 2001.

[64] M. Wermelinger, G. Koutsoukos, J. L. Fiadeiro, L. Andrade & J. Gouveia. Evolving and Using Coordinated Systems. Em *ICSE'2002, International Workshop on Principles of Software Evolution*, 43-47, Orlando, Flórida, USA, Maio 2002.

[65] K. Whitehead. *Component-Based Development – Principles and Planning for Business Systems*. Addison-Wesley, 2002

[66] J. Xu, B. Randell, A. Romanovsky, C. Rubira, R. Stroud, & Z. Wu. Fault Tolerance in Concurrent Object-Oriented Software Through Coordinated Error Recovery. Em *25th International Symposium on Fault-Tolerant Computing*, 499–508, Pasadena, USA, 1995.

[67] J. Xu, B. Randell, A. Romanovsky, R.J. Stroud, A.F. Zorzo, A. Burns, S.E. Mitchell & A.J. Wellings. Cooperative and Competitive Concurrency in Fault-Tolerant Distributed Systems. Em *De Va 1st Year Report*, 21-42, França, Janeiro 1997.

[68] A. Zorzo and R. J. Stroud. A distributed Object-Oriented Framework for Dependable Multiparty Interactions. Em *OOPSLA'99*, 435-446, Denver, CO, USA, Novembro 1999.