Um Método Para Modelagem de Exceções Em Desenvolvimento Baseado em Componentes

Patrick Henrique da Silva Brito

Dissertação de Mestrado

Instituto de Computação
Universidade Estadual de Campinas

Um Método Para Modelagem de Exceções Em Desenvolvimento Baseado em Componentes

Patrick Henrique da Silva Brito

Outubro de 2005

Banca Examinadora:

- Profa. Dra. Cecília Mary Fischer Rubira (Orientadora)
- Prof. Dr. Paulo Cesar Masiero
  ICMC – USP São Carlos
- Profa. Dra. Eliane Martins
  IC – UNICAMP
- Profa. Dra. Ariadne Maria Brito Rizzoni Carvalho (Suplente)
  IC – UNICAMP

FICHA CATALOGRÁFICA ELABORADA PELA BIBLIOTECA DO IMECC DA UNICAMP

Bibliotecário: Maria Júlia Milani Rodrigues - CRB8a / 2116

Brito, Patrick Henrique da Silva

B777m Um método para modelagem de exceções em desenvolvimento baseado em componentes / Patrick Henrique da Silva Brito -- Campinas, [S.P. :s.n.], 2005.

Orientador : Cecília Mary Fischer Rubira

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

1. Tratamento de exceções. 2. Tolerância a falha (Computação). 3. Software – Desenvolvimento – Metodologia. 4. Desenvolvimento baseado em componentes. 5. Engenharia de software. I. Rubira, Cecília Mary Fischer. II. Universidade Estadual de Campinas. Instituto de Computação. III. Título.

Título em inglês: A method for modelling exceptions in component-based software development

Palavras-chave em inglês (Keywords): 1. Exception handling. 2. Fault-tolerant computing. 3. Software development – Methodology. 4. Component-based development. 5. Software engineering.

Área de concentração: Engenharia de Software

Titulação: Mestrado em Ciência da Computação

Banca examinadora: Profa. Dra. Cecília Mary Fischer Rubira (IC/UNICAMP)
Prof. Dr. Paulo César Masiero (ICMC-USP)
Prof. Dra. Eliane Martins (IC/UNICAMP)

Data da defesa: 14/10/2005

Um Método Para Modelagem de Exceções Em Desenvolvimento Baseado em Componentes

Este exemplar corresponde à redação final da Dissertação devidamente corrigida e defendida por Patrick Henrique da Silva Brito e aprovada pela Banca Examinadora.

Campinas, 14 de Outubro de 2005.

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

Devido à grande popularização do Desenvolvimento Baseado em Componentes (DBC), ele vem sendo empregado inclusive no desenvolvimento de sistemas computacionais críticos. O emprego do DBC na construção de sistemas confiáveis evidencia a necessidade de se desenvolver componentes de software que sejam robustos e que possuam uma garantia maior do seu funcionamento correto.

Tratamento de exceções é uma técnica bastante conhecida para a verificação e tratamento de erros em sistemas de software. Porém, apesar da sua popularidade, o seu projeto e a implementação são constituídos de tarefas muito complexas que não recebem uma atenção adequada dos processos de desenvolvimento existentes. A situação é ainda mais crítica se levarmos em consideração os métodos para DBC.

Este trabalho propõe um método para auxiliar a modelagem do comportamento excepcional de sistemas baseados em componentes, chamado MDCE+. Baseado no refinamento da metodologia MDCE, o MDCE+ apresenta dois diferenciais importantes, que reforçam o seu aspecto robusto: (i) o fato dele combinar as abordagens *top-down* e *botton-up* para o desenvolvimento de sistemas confiáveis; e (ii) o fato dele ser centrado na arquitetura. O foco na arquitetura de software contribui para uma melhor definição e análise do fluxo de exceções entre os componentes do sistema. Essa maneira estruturada de detectar e tratar exceções no contexto da ocorrência de falhas é particularmente relevante para sistemas que apresentam requisitos de confiabilidade extrema.

O método MDCE+ é um método genérico que pode ser aplicada a processos de desenvolvimento modernos. Em particular, nesta dissertação o método MDCE+ foi adaptado ao processo *UML Components* e a uma metodologia de testes. Como maneira de avaliar esse método, foi desenvolvido um estudo de caso de um sistema financeiro real, com requisitos de tolerância a falhas. Dada a sua importância, o processo de avaliação do método MDCE+ foi dividido em três etapas: (i) preparação; (ii) execução; e (iii) análise dos resultados. Nesse estudo foi necessário tratar exceções na arquitetura do sistema, com o intuito de aumentar a disponibilidade dos serviços.

Abstract

Due to the large adoption of the Component-Based Development (CBD), it has also been employed in the development of critical software systems. The development of dependable systems using the CBD paradigm evidences the necessity of developing software components that are robust and dependable.

Exception handling is a well known technique for verify and treat errors in software systems. However, despite its popularity, its design and implementation are constituted of very complex tasks that do not receive the adequate attention from the existing development processes. This is still more critical in the context of CBD processes.

This work presents the MDCE+, a method that assists the modeling of the exceptional behavior in component-based software development. Based in the refinement of the MDCE methodology, the MDCE+ presents two important differentials, that strengthen its robustness: (i) it combines the top-down and bottom-up strategies for the development of dependable systems; and (ii) it is centered in the software architecture. As a consequence of the focus given to the software architecture, the exceptions that flow between the system components are better defined and analyzed. This structured way to detect and to treat exceptions in the context of the occurrence of imperfections is particularly needed for developing dependable systems.

The MDCE+ is a generic method that can be applied together with modern development processes. In particular, in this master thesis MDCE+ was adapted to the UML Components process and to a software test methodology. In order to evaluate this method, a case study of a real financial system with fault-tolerance requirements was developed. Given its importance, the evaluation process of the MDCE+ method was decomposed in three stages: (i) preparation; (ii) execution; and (iii) results analysis. In order to increase the services availability, in this study it was necessary to deal with exceptions in the software architecture.

Agradecimentos

Primeiramente, gostaria de agradecer a Deus, por todas as oportunidades, maravilhas e experiências que Ele me proporcionou durante a minha vida e por ter me sustentado nessa árdua jornada. Devo a Ele cada um dos meus passos.

Aos meus Pais (Maria Lucia da Silva Brito e Neusvaldo Henrique de Brito), por todo amor, dedicação, incentivo e exemplo que serviram como motivação fundamental para prosseguir caminhando e superar as dificuldades.

Aos meus irmãos (Papy, Boy e Piu), pela nossa cumplicidade e admiração mútua, que aumenta cada vez mais a minha vontade de melhorar e assim me assemelhar a eles.

À Margareth (Nem), minha namorada, a quem amo e admiro e que será sempre a minha inspiração. Obrigado por ter aceitado com tanta compreensão os sacrifícios da distância e da saudade. Espero sempre retribuir tudo isso com muita dedicação e amor.

À minha família que a Nem me presenteou: D. Margareth, Sr. Pedro, Peu, Pedrinho, Iann (meu afilhado), Ilanne, Tiaguinho, Urânia, Maelson, D. Ester e Vozinha. Muito obrigado pelo apoio, confiança, torcida, e pelo carinho e admiração recíprocos.

A todos os meus familiares: avós, tios, primos, padrinhos e cunhados (Gesse, Diran, Dinho, Peu e Pedrinho), em especial ao meu primo Marcos Henrique. Não sei se ele sabe, mas devo a ele a minha vocação pela ciência da computação.

Em memória dos meus familiares que descansaram eternamente no decorrer do meu mestrado: Vó Júlia, D. Ester, Vovô Zé Henrique, Fabinho e Flávio (primos).

Em memória ao tio Zezinho, a quem devo a alegria de torcer pelo glorioso ASA de Arapiraca :-)

Um agradecimento mais que especial à Cecília Rubira, minha orientadora, pelos ensinamentos, companheirismo e verdadeira amizade. Obrigado pelas conversas, dicas, projetos, correções de textos, exigências por prazos :-), ... tudo isso foram tijolos importantíssimos na construção do nosso objetivo. Com essa convivência tão saudável, me ensina a amadurecer cada vez mais.

A todos os meus professores de graduação, em especial ao Coradine, meu orientador de iniciação científica. O exemplo e o incentivo de vocês nortearam a minha decisão em continuar estudando e pesquisando. Muito obrigado!

Aos meus amigos da terra dos marechais (Alagoas): David, Medeiros, José Ricardo, Rosemeire, Túlio, Moacy, Katiane, André Atanasio, Hítalo, Fernando Rezende, Gilberto, Carlos Alberto... . As minhas sinceras desculpas pela minha ausência total. Espero que ainda lembrem de mim :-)

Aos meus amigos que conheci na Unicamp, que são os principais frutos do meu mestrado, em especial à galera do grupo de pesquisa em engenharia de software (SED), do laboratório de sistemas distribuídos (LSD) e ao Augusto, um dos meus revisores oficiais de inglês :-) Graças a vocês, a jornada se tornou ainda mais gratificante e prazerosa. Muito obrigado mesmo!

Aos companheiros de república e convivência: Eduardo, Felipe, Javier, Márcio, Marcos, além de Clayton e Matias (vizinhos). Por toda amizade, descontração, e como não podia esquecer, horários políticos e novelas assistidas e comentadas ao vivo :-)

Gostaria de agradecer especialmente a cada um dos professores do curso de especialização em Engenharia de Software, onde fui monitor: Ariadne, Arnaldo, Buzatto, Cecília, Célio, Eliane, Hans, Mário Cortes, Thelma e Vanini. Foram dois anos e meio de um convívio sensacional, com muito conhecimento e experiência adquiridos. Eu não poderia deixar de agradecer de maneira especial às secretárias do curso: Cláudia Regina e Olívia. Obrigado pelas conversas amigas e por toda ajuda dentro e fora do curso. Também gostaria de agradecer ao curso em si, na pessoa da coordenadora, a profa. Thelma Chiossi. Obrigado pelo auxílio material fundamental para a melhor condução da pesquisa. Obrigado também ao IC, ao FINEP e ao projeto CompGov, pelos auxílios em viagens.

Não tem como deixar de agradecer aos professores e funcionários do IC que me acompanharam mais de perto: professores Eliane Martins, Ariadne, João Meidanis, Ricardo Anido, Paulo Centoducatte, Edmundo Madeira e Cláudia Bauzer; Flávio (secretaria), Nelson (portaria), Flávio (portaria), Mirian (patrimônio), e Eliane (financeiro). Muito obrigado a todos!

Para fechar com chave de ouro, agradeço também aos professores da banca e a todos que colaboraram diretamente com o trabalho, tanto durante a condução da pesquisa, quanto com a melhoria da qualidade do texto escrito. Ao pessoal da Autbank, em especial ao Paulo Kato, Tomohiro, Michele e Denise. Aos professores Cecília Rubira, Eliane Martins e Paulo Masiero. Aos meus companheiros de pesquisa e amigos: Camila Rocha (companheira inseparável de pesquisa e estudo de caso), Paulo Asterio, Fernando Castor Filho, Rodrigo Tomita, Tiago Moronte, Leonardo Tizzei, Maria Antônia, Regina, Bruno, Leonel Gayard, Ana Elisa, Janaína, Helder, Ivan, Eduardo Gonçalves, Vinícius, Moacir Caetano e Ricardo.

Finalmente, o meu muito obrigado a todos aqueles que trabalham nos bastidores, cujo nome eu desconheço. Um obrigado especialíssimo a todos aqueles que esqueci injustamente de citar o nome aqui. Minhas sinceras desculpas e o meu agradecimento.

"Vencer, crescer... são metas importantes na vida; chegar ao topo, buscar seu lugar ao sol é acima de tudo um sonho a ser conquistado; mas mais importante que tudo isso é chegar através de suas próprias escolhas, fazendo do seu caminho um traço da sua imagem."

(Neusvaldo Júnior)

"Comece pelo necessário, faça o que for possível, ao final você terá feito o impossível."

(São Francisco de Assis)

"Quem nunca errou nunca experimentou nada novo."

(Albert Einstein)

"Sabemos o que somos, mas não o que podemos ser."

(Willian Shakespeare)

"O mundo é um livro e quem fica sentado em casa lê somente uma página."

(Santo Agostinho)

Sumário

Lista de Tabelas

Lista de Figuras

Capítulo 1

Introdução

Sistemas de software são intrinsicamente complexos e esta complexidade é agravada ainda mais com os requisitos impostos pelas aplicações modernas, como por exemplo, confiabilidade, disponibilidade e segurança. Apesar das exigências em se produzir sistemas confiáveis, esse aumento da complexidade, aliado às restrições de tempo de desenvolvimento, comprometem o funcionamento correto do sistema, uma vez que induzem a inserção de um maior número de falhas durante o seu desenvolvimento [AL90]. A fim de lidar com essa complexidade e de reduzir o tempo e o custo de desenvolvimento, a indústria de software vem adotando cada vez mais o desenvolvimento baseado em componentes (DBC), que modulariza o sistema em componentes e a partir dessas unidades básicas, outros sistemas são construídos recursivamente [Szy98].

Com a popularização do DBC, é inevitável a criação de sistemas baseados em componentes para desempenhar atividades críticas, quer seja com requisitos de disponibilidade, como sistemas *web*[1] e bancários; quer seja com requisitos de confiabilidade crítica, como sistemas médicos ou embarcados. Por esse motivo, é necessário se preocupar em produzir sistemas baseados em componentes que sejam robustos.

Apesar do avanço das pesquisas em ciência da computação, ainda não existem meios para garantir a ausência de falhas em sistemas de software [Som01]. Sendo assim, para a construção de **sistemas confiáveis**, devem ser utilizadas técnicas de tolerância a falhas [AL90], que visam manter o sistema em funcionamento mesmo na ocorrência de falhas. Essas técnicas podem ser implementadas através de mecanismos de tratamento de exceções [Fer01], os quais oferecem um arcabouço para a criação de tratadores adequados para cada tipo de exceção, possibilitando o tratamento de possíveis inconsistências, ou até mesmo a continuação da execução das funcionalidades do sistema. Devido à popularidade dos mecanismos de tratamento de exceções, as falhas que podem ser detectadas no sistema também são conhecidas como **situações excepcionais**, enquanto o comportamento cor-

[1] do inglês *World Wide Web*

retivo mediante a ocorrência de situações excepcionais é chamado de **comportamento excepcional**. O comportamento que implementa a execução esperada do sistema é conhecido como **comportamento normal**.

A incorporação efetiva de detecção e tratamento de exceções, visando a implantação de tolerância a falhas, aumenta consideravelmente a complexidade do sistema [Cri89], uma vez que o número de linhas de código adicionais necessário para a execução dessas atividades pode representar mais de dois terços do total [Cri89]. Devido a essa complexidade extra, os desenvolvedores de software sentem dificuldade tanto para utilizar corretamente os mecanismos, quanto para reconhecer e tratar adequadamente as exceções, o que gera sistemas com tratamento excepcional deficiente, com baixa manutenibilidade e baixa capacidade de reutilização.

Um fator agravante na qualidade do tratamento excepcional é o fato de um componente isolado não possuir os recursos necessários para detectar e tratar as exceções de uma maneira efetiva [dLR99]. Sendo assim, é necessário considerar as colaborações entre os componentes do sistema. Essa necessidade de se conhecer o fluxo interativo entre os componentes contribui para que a arquitetura de software assuma um papel fundamental para a melhoria da qualidade e da confiabilidade final dos programas. Isso se deve ao fato da **arquitetura de software** [SG96] explicitar as restrições de comunicação entre os componentes do sistema, uma vez que o representa de maneira abstrata, através dos seus **componentes arquiteturais** e das regras de interação entre eles.

Para se ter uma idéia dessa importância, com a manipulação sistemática desses fluxos de informações, os tratadores excepcionais localizados na arquitetura, chamados aqui de **tratadores arquiteturais** [IB01], são capazes por exemplo, de isolar componentes falhos do sistema em tempo de execução, desviando o fluxo de requisição para componentes redundantes. Outra vantagem de se conhecer a estrutura do sistema é a possibilidade de executar um tratamento excepcional de maneira coordenada, envolvendo diferentes componentes (**tratamento colaborativo coordenado** [dLR99]). Além disso, o tratamento de exceções no contexto da arquitetura de software possibilita o uso de componentes COTS[2] na construção de sistemas confiáveis.

Apesar da necessidade de lidar com o comportamento excepcional de maneira sistemática, por falta de metodologias e ferramentas adequadas, os desenvolvedores costumam adiar essa preocupação para a fase de projeto ou até mesmo para a fase de implementação, executando-a de maneira "ad hoc", guiados apenas pela sua intuição. Porém, a antecipação dessa preocupação para as fases iniciais do desenvolvimento acarreta benefícios consideráveis para o aumento da confiabilidade dos sistemas [RdLeFCF04]. Os principais benefícios alcançados são: (i) evita-se a perda de contexto, devido à identificação antecipada do erro; (ii) melhora-se o tratamento excepcional, tendo em vista o maior

[2]do inglês *Common Off-The-Shelf*

número de informações contextuais disponíveis; (iii) diminui-se o re-trabalho, uma vez que uma fase pode delimitar restrições às fases seguintes; (iv) facilita-se o gerenciamento do desenvolvimento, tendo em vista que a garantia da confiabilidade é distribuída ao longo do processo.

Este trabalho propõe um método para o projeto e implementação do comportamento excepcional no DBC, denominado MDCE+. Nossa solução é um refinamento de um método proposto anteriormente chamado Metodologia para Definição do Comportamento Excepcional (MDCE) [RdLeFCF04], que é uma extensão do processo de DBC Catalysis [DW99]. O método MDCE apresenta diretrizes para a especificação do comportamento excepcional de um sistema nas fases de identificação de requisitos, análise e projeto. O foco do nosso refinamento está principalmente nas fases de projeto arquitetural e implementação, atividades nas quais o MDCE é deficiente.

Além do refinamento, o MDCE+ foi adaptado ao processo de DBC *UML Components* [CD00]. A escolha desse processo se deve principalmente ao fato de possuir uma estruturação simples, de fácil compreensão e utilização prática; ao contrário do processo Catalysis, que é bem complexo. Essa característica torna o *UML Components* mais acessível ao mercado corporativo, principalmente quando comparado a outros processos de DBC, como o Catalysis.

1.1 Objetivos

O objetivo principal deste trabalho é a definição de um método para auxiliar a construção de sistemas de software com requisitos de confiabilidade ligados à tolerância a falhas. Esse método é chamado de MDCE+. Isso deve ser feito através da utilização sistemática dos mecanismos de tratamento de exceções existentes nas principais linguagens de programação atuais. Esse método deve estruturar a identificação de exceções e a especificação dos seus tratadores durante todas as fases de desenvolvimento do software. Outra característica importante do MDCE+ é a preocupação com a informação contextual dos fluxos interativos entre os componentes do sistema. Aproveitando esse enfoque na arquitetura, o método deve guiar o desenvolvedor na especificação de tratadores de exceções localizados na arquitetura de software, a partir de uma análise sistemática das propagações excepcionais.

Além de sistematizar a modelagem das exceções e do comportamento excepcional do sistema, o MDCE+ propõe diretrizes para a estruturação das hierarquias de tipos de exceções. Essas diretrizes devem orientar também sobre a utilização de políticas de *logging*, com o intuito de beneficiar as atividades de manutenção do sistema.

Quanto à notação representativa dos seus artefatos, o método deve utilizar a linguagem

de modelagem UML[3] [RJB99, BRJ99], uma vez que desde o seu lançamento em 1997, tem se mostrado um padrão de fato na indústria de desenvolvimento de software.

De forma complementar ao método MDCE+, é apresentada uma arquitetura para estruturar o comportamento excepcional para sistemas confiáveis. Baseado nessa arquitetura, o método compõe o sistema de maneira estruturada, a fim de, no final do desenvolvimento, se ter todos os componentes arquiteturais materializados.

Após a definição do método, ele é incorporado ao processo *UML Components*. Esse processo adaptado distribui as atividades relativas à modelagem excepcional do sistema em todas as suas fases. Sendo assim, o processo resultante dessa adaptação sistematiza tanto a modelagem dos requisitos funcionais do sistema, quanto dos requisitos não-funcionais relativos a tolerância a falhas, através do uso disciplinado dos mecanismos de tratamento de exceções.

Além de utilizar a linguagem UML como notação para os seus artefatos produzidos,as atividades do processo *UML Components* adaptado com tratamento de exceções são representadas através de diagramas de atividades UML. Essa representação específica deverá proporcionar uma melhor documentação de suas atividades e diretrizes, além de uma distinção clara entre as atividades originais e as incorporadas ao processo adaptado. A principal vantagem decorrente dessa distinção é o aumento da facilidade de adaptação do método MDCE+ para outros processos de desenvolvimento de software.

Como um estudo da viabilidade do método MDCE+, foi feito um estudo de caso real baseado no desenvolvimento de um sistema financeiro com requisitos de confiabilidade[4]. O estudo de caso foi conduzido por terceiros e consistiu na aplicação do processo *UML Components* adaptado com o método MDCE+ para a especificação e implementação do sistema citado. A especificação foi realizada desde a fase de engenharia de requisitos, passando pelo projeto arquitetural e especificação interna dos componentes, culminando na sua implementação e na montagem do sistema.

1.2 A Solução Proposta

O método MDCE+ sistematiza a especificação e a implementação do comportamento excepcional nas fases de desenvolvimento do software com ênfase no projeto da arquitetura de componentes. Na especificação de requisitos, que é baseada no método MDCE (Seção 2.3.8), é feita a antecipação das atividades excepcionais relacionadas ao domínio do problema. Já nas fases de análise e projeto são especificadas as exceções relacionadas às interações entre os componentes, enfatizando os aspectos arquiteturais do sistema.

[3]do inglês *Unified Modeling Language*
[4]do inglês *Dependability*

Para facilitar o entendimento durante as atividades do desenvolvimento, o método MDCE+ apresenta uma distinção entre os conceitos de **componente arquitetural** e **componente de implementação**. Os componentes arquiteturais do sistema são estruturados segundo o modelo do componente tolerante a falhas ideal, apresentado na Seção 2.3.7. Dessa forma, mesmo no nível arquitetural do sistema, se faz uma separação explícita entre os comportamentos normal e excepcional, não adiando apenas para a fase de implementação.

O comportamento de um componente tolerante a falhas ideal, ou simplesmente componente ideal, é categorizado em dois tipos: **normal**, que produz respostas corretas, e **excepcional (ou anormal)**, que é executado quando um erro é detectado.

Pelo fato da solução proposta ser centrada na arquitetura de software, é destacada a importância dos **conectores** [SG96], que materializam as interações entre os componentes arquiteturais. Por esse motivo, eles são os principais responsáveis pelo tratamento das exceções no nível da arquitetura.

A incorporação do MDCE+ ao processo *UML Components* consistiu na distribuição das atividades do método proposto nas fases do processo de DBC. O produto final dessa extensão é um processo iterativo que contém basicamente três novas atividades, ortogonais às etapas definidas pelo processo (Figura 1.1): (i) definição dos cenários excepcionais dos casos de uso e identificação de exceções durante a análise dos requisitos; (ii) análise do fluxo excepcional e detalhamento dos tratadores durante a especificação dos componentes; e (iii) diretrizes para a estruturação excepcional, implementação dos componentes e montagem do sistema.

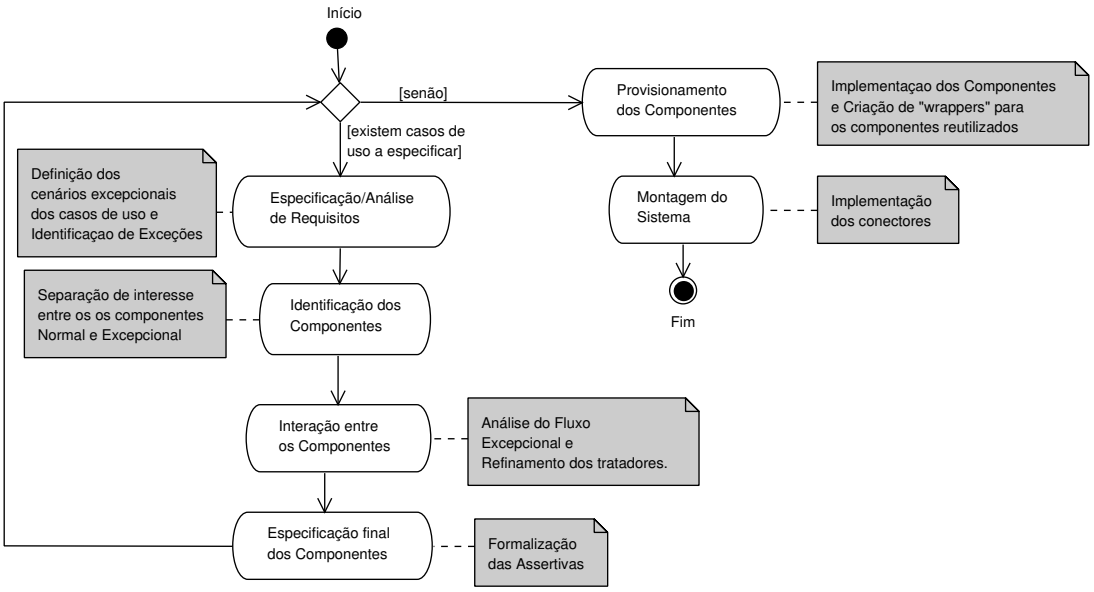

Figura 1.1: Interferência do Método MDCE+ nas fases do *UML Components*

As sub-seções seguintes apresentam as fases do processo *UML Components* adaptado ao método MDCE+. Mais detalhes dessa adaptação estão disponíveis no Capítulo 4 desta dissertação.

1.2.1 Especificação de Requisitos

Nesta fase, os requisitos do sistemas são mapeados através de casos de uso, cujas restrições são especificadas através de assertivas, isto é, pré-, pós-condições, invariantes e outras restrições de negócio. A partir da violação dessas assertivas, são identificadas exceções, que são agendadas nesta fase, a fim de serem especificadas na fase de análise (Seção 1.2.4).

Além de especificar e analisar os requisitos, é elaborado o modelo conceitual do negócio, que representa as principais entidades do negócio e o relacionamento entre elas. Esse modelo é útil para promover a compreensão do domínio que é necessária para a especificação do sistema. Após a elaboração desse modelo, são identificadas as entidades críticas, isto é, as entidades consideradas essenciais para a execução das principais atividades do negócio. Nas fases seguintes da especificação, essas entidades serão o foco central para o tratamento de exceções na arquitetura do sistema, através da utilização de componentes redundantes [IB01, SDUV03].

1.2.2 Definição dos Aspectos Gerenciais

Após a definição dos requisitos desejados para o sistema, o processo MDCE+ apresenta uma fase para a definição de algumas questões gerenciais do desenvolvimento. As principais questões definidas são: (i) análise inicial dos riscos; e (ii) definição das iterações do desenvolvimento. Durante a análise dos riscos são identificadas algumas restrições possíveis para o desenvolvimento. Essas restrições podem interferir no projeto da arquitetura do sistema (Seção 1.2.3) ou até mesmo nas decisões relacionadas à reutilização de componentes (Seção 1.2.5).

Com a definição de iterações para o desenvolvimento, espera-se compensar o *overhead* esperado para o desenvolvimento de sistemas robustos. Essa compensação é decorrente da entrega[5] gradativa do sistema, priorizando as funcionalidades de acordo com a necessidade do cliente.

1.2.3 Projeto Arquitetural

Na fase de projeto arquitetural é especificada a arquitetura adotada para o sistema. Devido ao direcionamento dado pelo processo *UML Components*, a escolha da arquitetura

[5]do inglês *Deployment*

deve obedecer a restrições. Recomenda-se a existência de duas camadas específicas: (i) camada de sistema, que trata de particularidades do sistema em questão; e (ii) camada de negócio, constituída por componentes que implementam as funcionalidades básicas e por isso são candidatos a serem reutilizados. Além das restrições impostas pelo *UML Components*, a arquitetura final adotada deve obedecer às restrições comuns ao ambiente de desenvolvimento utilizado, que são definidas pelo arquiteto.

1.2.4 Especificação do Sistema

Na fase de especificação são realizadas as atividades relativas à análise e projeto dos componentes do sistema. Essa fase é dividida em 3 sub-fases: (i) identificação dos componentes, na qual os componentes são identificados a partir de suas interfaces providas; (ii) interação entre os componentes, na qual são identificadas as operações requeridas pelos componentes de sistema, a partir das interações entre os componentes das camadas de negócio e de sistema; e (iii) especificação final dos componentes, onde são formalizadas as especificações dos componentes.

1.2.5 Provisionamento dos Componentes

A fase de Provisionamento dos componentes tem como objetivo garantir a materialização dos componentes especificados. Isso pode ocorrer de três formas: (i) reutilização de componentes já existentes; (ii) aquisição de componentes de prateleira; (iii) implementação de componentes novos.

Em relação à reutilização de componentes, é necessário realizar um mapeamento entre as suas operações e as operações requeridas especificadas. Esse mapeamento é materializado através de adaptadores [dCGFPR04], como mostrado na Seção 2.4, cuja função é mascarar a assinatura real das operações reutilizadas, a fim de tornar a reutilização transparente para o componente cliente. Além disso, os adaptadores podem identificar a ocorrência de situações excepcionais e conseqüentemente executar o tratamento adequado. Já no caso dos componentes implementados, é necessário especificar as suas estruturas internas de acordo com algum modelo de componentes disponível. O modelo de materialização utilizado para os componentes foi o modelo COSMOS [JGR03], apresentado na Seção 2.2.2, que utiliza estruturas disponíveis em linguagens orientadas a objetos e explicita as abstrações da arquitetura do sistema.

Quanto às exceções, elas devem ser definidas como classes e organizadas de acordo com uma hierarquia predefinida, baseada na proposta apresentada na Seção 2.4. O objetivo principal dessa hierarquia excepcional é relacionar de maneira consistente as exceções internas e as exceções que fluem entre diferentes componentes arquiteturais. Dessa forma,

o método MDCE+ oferece diretrizes para o mapeamento entre os tipos de exceções apresentados na hierarquia e a classificação dada pelo modelo do componente ideal.

1.2.6 Montagem do Sistema

Por se tratar de um método centrado na arquitetura, a fase de montagem é dividida em duas etapas: (i) montagem dos componentes arquiteturais; e (ii) montagem do sistema em si. Na primeira etapa é feita a montagem dos componentes arquiteturais. Dessa forma, os componentes de implementação, que implementam os comportamentos relativos à implementação das funcionalidades e do tratamento de exceções são estruturados de acordo com o modelo do componente tolerante a falhas ideal.

A segunda etapa da fase de montagem do sistema consiste na implementação dos conectores do sistema e na construção do programa principal, que instancia e "monta" os componentes arquiteturais do sistema em tempo de execução. Apesar dos conectores serem refinados nesta fase, eles são especificados gradativamente desde a fase de especificação dos componentes, mais especificamente na sub-fase de interação entre os componentes. A principal atividade excepcional desempenhada pelos conectores é a implementação dos tratadores de exceções na arquitetura do sistema.

1.3 Trabalhos Relacionados

Utilizar um mecanismo de tratamento de exceções é uma das formas mais importantes de detecção e recuperação de erros e estruturação de atividades de tolerância a falhas [AL90, Cri89, GRRX01]. Apesar da complexidade desses mecanismos motivar o uso de metodologias de desenvolvimento específicas, há poucos trabalhos nessa área [Avi97, dLR99, dLR00, Fer01]. O método MDCE, descrito na Seção 2.3.8, orienta o desenvolvimento excepcional desde a especificação dos requisitos, passando pela análise e projeto do sistema. Porém, o MDCE não apresenta diretrizes para tratar as exceções na arquitetura do software, o que impossibilita, entre outras coisas, a reutilização de componentes COTS no desenvolvimento de sistemas confiáveis. Dessa forma, enquanto o MDCE busca especificar sistemas confiáveis a partir do desenvolvimento de componentes tolerantes a falhas (abordagem *botton-up*), o método MDCE+ também possibilita a especificação das atividades de tratamento de exceções durante a ligação entre os componentes, não exigindo que eles possuam esses mecanismos implementados internamente (abordagem combinada *botton-up* e *top-down*).

Tolerância a falhas na arquitetura de software é uma área que vem se destacando pela sua relevância [GRdL03, FdCGR03, SDUV03]. O método MECE [Pag04] propõe uma sistemática com enfoque arquitetural para a especificação do comportamento excepcional

em sistemas baseados em componentes. Sua abordagem propõe o uso conjunto e seqüencial da MDCE, descrita anteriormente, e do processo *UML Components*. Além disso, o MECE sugere um refinamento da arquitetura do sistema através da adoção de conectores explícitos. Apesar dessa melhoria no caráter arquitetural do MDCE, o MECE não sistematiza a especificação dos tratadores arquiteturais [IB01]. Outra deficiência existente nesse método é o fato dela não lidar com o tratamento colaborativo de exceções, isto é, situações onde um conjunto de componentes precisam agir de forma coordenada para a recuperação do estado normal do sistema. Outro diferencial do método MDCE+ é a preocupação com aspectos gerenciais, tais como a definição de *releases* para o desenvolvimento incremental e a identificação e priorização de componentes críticos do sistema.

CO-Actions [dLR99] é uma abstração de modelagem para representação de comportamento colaborativo nas diversas fases da especificação de um sistema. Apesar de ser voltado para sistemas orientados a objetos, seus conceitos podem ser mapeados diretamente para o DBC. Além do comportamento colaborativo, CO-Actions considera a necessidade de se especificar o comportamento excepcional desde as fases iniciais do desenvolvimento do software, propondo uma classificação das exceções entre: (i) exceções relativas à aplicação[6]; (ii) exceções relativas ao projeto[7]; e (iii) exceções relativas ao suporte & implementação[8]. Porém, apesar de oferecer formas de modelar e estruturar melhor o comportamento excepcional, essa abordagem não é apoiada por um processo, isto é, ela não oferece diretrizes para a descoberta e modelagem das exceções e dos seus tratadores.

1.4 Organização deste Trabalho

Este documento foi dividido em seis capítulos e dois apêndices, organizados da seguinte forma:

- **Capítulo 2 – Fundamentos de DBC, Arquitetura de Software e Tolerância a Falhas:** O segundo capítulo apresenta os fundamentos teóricos necessários para embasar este trabalho. Esses conceitos estão classificados em cinco categorias: (i) desenvolvimento baseado em componentes; (ii) arquitetura de software; (iii) tolerância a falhas e tratamento de exceções; (iv) uma estruturação do comportamento excepcional para DBC; e (v) uma arquitetura confiável baseada em tratamento de exceções.

- **Capítulo 3 – O Método MDCE+:** Neste capítulo é apresentado o método para modelagem do comportamento excepcional MDCE+, que utiliza a arquitetura

[6] do inglês *Application-related*
[7] do inglês *Design-related*
[8] do inglês *Implementation & support-related*

proposta na Seção 2.5 e apresenta diretrizes e atividades importantes a serem consideradas no desenvolvimento de sistemas confiáveis. São abordados tanto aspectos do desenvolvimento interno dos componentes quanto aspectos arquiteturais do sistema, distribuídos em todas as fases propostas pelo método.

- **Capítulo 4 – Método MDCE+ Aplicado ao Processo *UML Components*:** Neste capítulo, o método MDCE+ apresentado no capítulo 3 foi aplicado ao processo de DBC *UML Components*. As fases de desenvolvimento específicas desse processo foram estendidas de forma a sistematizar a identificação das exceções e a especificação do comportamento excepcional.

- **Capítulo 5 – Estudo de Caso: Sistema de Controle Bancário** Neste capítulo, o processo *UML Components* modificado, apresentado no Capítulo 4, foi aplicado num estudo de caso real de um sistema financeiro [dSBRMR05].

- **Capítulo 6 – Conclusões:** Neste capítulo são apresentadas as conclusões deste trabalho. Além disso, são apontadas as principais contribuições e alguns direcionamentos para trabalhos futuros.

- **Apêndice A – Questionário de Avaliação do Estudo de Caso:** Neste apêndice são apresentados os questionários utilizados para a avaliação do método MDCE+, descrito no Capítulo 5.

Capítulo 2

Fundamentos de DBC, Arquitetura de Software e Tolerância a Falhas

Este capítulo apresenta os fundamentos teóricos necessários para embasar este trabalho. Esses conceitos estão agrupados em cinco seções. As Seções 2.1 e 2.2 tratam de aspectos necessários para o sucesso de um projeto baseado em componentes, tais como o próprio conceito de desenvolvimento baseado em componentes (DBC), arquitetura e processos de desenvolvimento de software. A Seção 2.3 apresenta os princípios de confiabilidade, tolerância a falhas e tratamento de exceções. Essa seção discute as terminologias utilizadas no decorrer do trabalho e como os mecanismos de tolerância a falhas podem ser implementados em sistemas de software. A Seção 2.4 apresenta uma maneira de estruturar o comportamento excepcional no DBC. Essa estruturação contempla tanto as exceções internas aos componentes, quanto as exceções que fluem entre diferentes componentes do sistema. A Seção 2.5 mostra a arquitetura adotada pelo MDCE+ para a estruturação do comportamento excepcional do software.

2.1 Desenvolvimento Baseado em Componentes

A idéia de utilizar o conceito de DBC na produção de software data de 1976 [McI76]. Apesar desse tempo relativamente longo, o interesse pelo DBC só foi intensificado vinte anos depois, após a realização do Primeiro Workshop Internacional em Programação Orientada a Componentes, o WCOP'96 [WCO].

Hoje em dia, a popularização do DBC está sendo motivada principalmente pelas pressões sofridas na indústria de software por prazos mais curtos e produtos de maior qualidade. No DBC, uma aplicação é construída a partir da composição de componentes de software que já foram previamente especificados, construídos e testados, o que proporciona um ganho de produtividade e qualidade no software produzido.

Esse aumento da produtividade é decorrente da reutilização de componentes existentes na construção de novos sistemas. Já o aumento da qualidade é uma conseqüência do fato dos componentes utilizados já terem sido empregados e testados em outros contextos. Porém, vale a pena ressaltar que apesar desses testes prévios serem benéficos, a reutilização de componentes não dispensa a execução dos testes no novo contexto onde o componente está sendo reutilizado.

Apesar da popularização atual do DBC, não existe um consenso geral sobre a definição de um componente de software, que é a unidade básica de desenvolvimento do DBC. Porém, um aspecto muito importante é sempre ressaltado na literatura: um componente deve encapsular dentro de si seu projeto e implementação, além de oferecer interfaces bem definidas para o meio externo. O baixo acoplamento decorrente dessa política proporciona muitas flexibilidades, tais como: (i) facilidade de montagem de um sistema a partir de componentes COTS[1]; e (ii) facilidade de substituição de componentes que implementam interfaces equivalentes.

Uma definição complementar de componentes, adotada na maioria dos trabalhos publicados atualmente, foi proposta em 1998 [Szy98]. Segundo Szyperski, um componente de software é uma unidade de composição com interfaces especificadas através de contratos e dependências de contexto explícitas. Sendo assim, além de explicitar as interfaces com os serviços oferecidos (**interfaces providas**), um componente de software deve declarar explicitamente as dependências necessárias para o seu funcionamento, através de suas **interfaces requeridas**.

Além dessa distinção clara entre interfaces providas e requeridas, os componentes seguem três princípios fundamentais, que são comuns à tecnologia de objetos [CD00]:

1. **Unificação de dados e funções:** Um componente deve encapsular o seu estado (dados relevantes) e a implementação das suas operações oferecidas, que acessam esses dados. Essa ligação estreita entre os dados e as operações ajudam a aumentar a coesão do sistema;

2. **Encapsulamento:** Os clientes que utilizam um componente devem depender somente da sua especificação, nunca da sua implementação. Essa separação de interesse[2] reduz o acoplamento entre os módulos do sistema, além de melhorar a sua manutenção;

3. **Identidade:** Independentemente dos valores dos seus atributos de estado, cada componente possui um identificador único que o difere dos demais.

[1] do inglês *Common-Off-The-Shelf*
[2] do inglês *Separation of concerns*

A fim de contextualizar o componente em relação aos diferentes níveis de abstração que ele pode ser analisado, um componente pode ser observado através de diferentes pontos de vista [CD00]. Em outras palavras, dependendo do papel que o interessado[3] exerce no processo de desenvolvimento, a abstração como o componente é analisado pode variar. A Tabela 2.1 descreve os diferentes pontos de vista de um componente [CD00].

Tabela 2.1: Pontos de Vista de um Componente

Ponto de Vista	Descrição
Especificação	È constituído da especificação de uma unidade de software que descreve o comportamento de um componente. Esse comportamento é definido como um conjunto de interfaces providas e requeridas.
Plataforma	São definidas restrições que o componente deve seguir a fim de ser utilizado em alguma plataforma específica de desenvolvimento. As principais plataformas comerciais existentes são Enterprise Java Beans [MH99], Corba [MM00] e COM+ [Ses98].
Implementação	É a realização de uma especificação. Sendo assim, a implementação de um componente está para a sua especificação, assim como uma classe está para a sua interface. Esse ponto de vista é compartilhado por todos os componentes já implementados em alguma linguagem de programação. Ele representa componentes prontos para serem instalados, como por exemplo, os componentes COTS.
Instalação	É uma instalação ou cópia de um componente implementado. Esse ponto de vista é utilizado na análise do ambiente de execução. Neste caso, pode ser feita uma analogia às classes localizadas no *classpath* do Java. O *classpath* é a lista de diretórios onde a máquina virtual procura as classes a serem instanciadas.
Instanciação	É uma instância de um componente instalado. Esta é a visão de execução do componente, com os seus dados encapsulados e um identificador único. É semelhante ao conceito de objetos no paradigma de orientação a objetos.

2.1.1 Processos de Desenvolvimento de Software

O objetivo de um **processo** de desenvolvimento de software é sistematizar as atividades de construção de programas, distribuindo a sua complexidade em três grupos de atividades gerais [Pre01]: (i) definição do problema; (ii) desenvolvimento do sistema; e (iv) manutenção. O uso de processos disciplinados de desenvolvimento reduz o número de falhas introduzidas no sistema [Avi97, AL90]. A fim de atingir a sistemática necessária para isso, os processos devem possuir um conjunto de métodos estruturados que detalhem e auxiliem o desenvolvimento de sistemas nas suas fases [Pre01]. Esses **métodos** são procedimentos sistemáticos que usam notações bem definidas para alcançar seus objetivos. Assim, os métodos são compostos de atividades e diretrizes de desenvolvimento, normalmente orientadas a um processo específico. De uma maneira geral, os métodos devem incluir informações sobre [Som01]: (i) os modelos produzidos; (ii) as restrições aplicadas a esses modelos; (iii) diretrizes de projeto; e (iv) a seqüência de atividades a ser seguida.

A maioria dos processos de desenvolvimento de software atuais mapeiam o desenvolvimento do sistema em fases, que detalham os quatro grupos de atividades gerais

[3] do inglês *Stakeholder*

apresentados anteriormente:

Grupo 1: Definição do problema:

- **Fase de Especificação de Requisitos.** O objetivo principal desta fase é a identificação dos serviços que devem ser automatizados pelo sistema. Além disso, outras informações e restrições importantes devem ser igualmente documentadas. De acordo com a sua natureza, esses requisitos podem ser classificados em duas categorias: (i)requisitos funcionais, que representam os comportamentos que um sistema deve apresentar diante de certas ações de seus usuários; e (ii) requisitos não-funcionais, que quantificam determinados aspectos do comportamento do sistema [Som01].

Grupo 2: Desenvolvimento do Sistema:

- **Fase de Análise -** Nesta etapa, os desenvolvedores se concentram na identificação das entidades principais para a implementação dos requisitos do sistema. As entidades identificadas são representadas a partir das suas informações de estado (dados internos) e das principais operações a serem oferecidas.
- **Fase de Especificação da Arquitetura -** Baseado principalmente nos requisitos não-funcionais especificados na fase de especificação de requisitos, se dá início à especificação da arquitetura do sistema. Normalmente, essa fase consiste na escolha da arquitetura que melhor ofereça as propriedades arquiteturais desejadas para satisfazer os requisitos não funcionais.
- **Fase de Projeto -** A etapa de projeto consiste no refinamento dos modelos gerados na análise. Esse refinamento visa a aplicações de padrões, por exemplo, os padrões de projetos[4] sugeridos por Gamma *et.al.* [GHJV95]. Ainda nessa etapa, a forma como as entidades interagem entre si é detalhada e especificada.
- **Fase de Implementação -** Esta etapa consiste na materialização dos modelos especificados e detalhados no projeto em uma linguagem de programação. Normalmente é feito um mapeamento direto das estruturas da modelagem para a linguagem de programação. Quando isso não é possível, se faz necessário a utilização de um modelo de mapeamento. Alguns exemplos de modelos para a implementação de componentes de software em linguagens orientadas a objetos podem ser os modelos EJB[5] [MH99], CCM[6] [MM00], DCOM[7] [Ses98] e

[4]do inglês *Design patterns*
[5]do inglês *Enterprise Java Beans*
[6]do inglês *Corba Component Model*
[7]do inglês *Distributed Component Object Model*

COSMOS[8] [JGR03].

- **Fase de Testes -** Na etapa de testes, o sistema é verificado para se certificar de que os requisitos especificados estão implementados de maneira satisfatória.
- **Fase de Distribuição**[9] - Esta etapa consiste na entrega e instalação do sistema nos diversos ambientes onde serão utilizados pelos clientes.

Grupo 3: MANUTENÇÃO:

- **Fase de Manutenções Corretivas -** As atividades desta fase consistem de ações com o objetivo de corrigir inconsistências entre os requisitos especificados e o sistema implementado;
- **Fase de Manutenções Evolutivas (ou perfectivas) -** As ações de manutenções evolutivas adicionam novas funcionalidades aos sistemas já desenvolvidos. Sendo assim, com a aplicação dessas atividades, o tempo de vida dos sistemas costumam aumentar;
- **Fase de Manutenções Adaptativas -** Esse tipo de manutenção é decorrente de mudanças no ambiente tecnológico onde o software atua. Exemplos desse tipo de mudança pode ser a utilização de um novo sistema operacional ou um novo sistema gerenciador de banco de dados [HHK+99];
- **Fase de Manutenções Preventivas -** As atividades de manutenções preventivas são decorrentes de modificações realizadas para melhorar a confiabilidade ou a manutenibilidade futura [Pfl01]. Dessa forma, esse tipo de manutenção tem o intuito de aumentar a vida do software, facilitando futuras evoluções do sistema.

Essas fases apresentadas constituem as etapas de execução do desenvolvimento de um software. Porém, além do processo de desenvolvimento, que especifica e implementa o sistema a partir dos seus requisitos, para se desenvolver um sistema de software é necessário seguir ao mesmo tempo um processo gerencial [CD00]. Um processo gerencial é responsável por esquematizar atividades, planejar liberações, alocar recursos e monitorar os progressos do desenvolvimento. Na literatura encontramos alguns exemplos de abordagens que incluem um dos processos ou ambos. Por exemplo, o processo Catalysis [DW99] e o processo *UML Components* [CD00] são basicamente processos de desenvolvimento; enquanto que o processo unificado (RUP[10]) [JBR99] trata tanto do processo gerencial quanto do processo de desenvolvimento.

[8] do inglês *Component Structuring Model dor Object-oriented Systems*
[9] do inglês *Deployment*
[10] do inglês *IBM-Rational Unified Process*

Este trabalho está concentrado em processos de desenvolvimento de software e para facilitar a leitura, quando o termo simples processo for utilizado, se trata de uma referência a um processo de desenvolvimento.

2.1.2 Processos de Desenvolvimento Baseado em Componentes

Com a popularização do DBC, a necessidade de novos processos voltados para esse paradigma é uma realidade. Isso acontece porque os processos de desenvolvimento tradicionais não se adequam totalmente ao desenvolvimento de sistemas baseados em componentes. Mais especificamente, esses processos devem conter fases e métodos que também ofereçam técnicas que permitam o empacotamento de componentes com o objetivo específico de serem reutilizados [Som01]. Essa reutilização sistemática deve levar em consideração a captura de abstrações que facilitem o entendimento do sistema tanto para seu desenvolvimento, quanto para a reutilização de componentes [Szy98, Som01]. Os métodos também devem auxiliar na definição de como os componentes devem ser conectados uns aos outros para atender aos requisitos especificados. Em outras palavras, processos de DBC devem auxiliar na construção da arquitetura do software [CD00].

Uma técnica bastante utilizada para a estruturação de componentes propícios a serem reutilizados é a de análise de domínio[11] [KCH+90, FH95]. O fundamento básico dessa técnica é a análise da lógica do negócio no qual os componentes se inserem. Sendo assim, os componentes que contêm as funcionalidades das entidades básicas do domínio são considerados candidatos à reutilização.

As principais particularidades dos processos de DBC podem ser observadas tanto no acréscimo de estágios técnicos ao processo convencional, quanto no enfoque dado a algumas práticas já realizadas. Um processo de desenvolvimento de software baseado em componentes geralmente inclui a definição de estratégias para [JBR99, CD00]:

- **Definição explícita da arquitetura do sistema**. A explicitação da arquitetura tem o objetivo principal de enfatizar os aspectos de interação entre os componentes do sistema, com os seus fluxos e restrições;
- **Separação de contextos a partir do modelo de domínios.** Com essa separação, pretende-se classificar os componentes mais propícios à reutilização, de acordo com a lógica do negócio de cada sistema em desenvolvimento;
- **Identificação das interfaces dos componentes.** Um dos objetivos principais do DBC é a construção de sistemas facilmente modificáveis [CD00]. O baixo acoplamento proporcionado pela definição de interfaces providas e requeridas é um meio de alcançar esse objetivo;

[11] do inglês *Domain analysis*

- **Identificação do comportamento interno dos componentes.** A etapa de projeto, comum a todos os processos de desenvolvimento, consiste na modelagem dos serviços oferecidos pelo componente. Por ser uma estrutura altamente encapsulada, deve haver transparência em relação à tecnologia utilizada para a sua implementação interna;

- **Montagem dos componentes do sistema.** Nesta etapa ocorre a materialização da configuração da arquitetura do sistema final. Devido à sua autonomia, um componente de software implementa seus serviços utilizando unicamente as suas interfaces requeridas. Sendo assim, a fase de montagem consiste na indicação dos objetos reais que implementam essas interfaces; e

- **Manutenção de um repositório de componentes.** O objetivo principal da utilização de repositórios é maximizar a reutilização de componentes. Isso acontece através do oferecimento de mecanismos de busca sistemáticos que auxiliam o desenvolvimento do software. Normalmente, essas técnicas são utilizadas em dois momentos principais: (I) no início da especificação; e (ii) antes do projeto interno dos componentes do sistema.

2.2 Arquitetura de Software

2.2.1 Conceito e Terminologia

A arquitetura de software, através de um alto nível de abstração, define o sistema em termos de seus **componentes arquiteturais**, que representam unidades abstratas do sistema; a interação entre essas entidades, que são materializadas explicitamente através dos **conectores**; e os atributos e funcionalidades de cada um [Som01]. Por conhecerem o fluxo interativo entre os componentes do sistema, é possível nos conectores, estabelecer protocolos de comunicação e coordenar a execução dos serviços que envolvam mais de um componente do sistema.

Essa visão estrutural do sistema em um alto nível de abstração proporciona benefícios importantes, que são imprescindíveis para o desenvolvimento de sistemas de software complexos. Os principais benefícios são: (i) a organização do sistema como uma composição de componentes lógicos; (ii) a antecipação da definição das estruturas de controle globais; (iii) a definição da forma de comunicação e composição dos elementos do projeto; e (iv) o auxílio na definição das funcionalidade de cada componente projetado. Além disso, uma propriedade arquitetural representa uma decisão de projeto relacionada a algum requisito não-funcional do sistema, que quantifica determinados aspectos do seu comportamento, como confiabilidade, reusabilidade e modificabilidade [CK03, Som01].

A presença de uma determinada propriedade arquitetural pode ser obtida através da utilização de estilos arquiteturais que possam garantir a preservação dessa propriedade durante o desenvolvimento do sistema [SG96, MKMG97]. Um **estilo arquitetural** caracteriza uma família de sistemas que são relacionados pelo compartilhamento de suas propriedades estruturais e semânticas. Esses estilos definem inclusive restrições de comunicação entre os componentes do sistema. A maneira como os componentes de um sistema ficam dispostos é conhecida como **configuração**.

As propriedades arquiteturais são derivadas dos requisitos do sistema e influenciam, direcionam e restringem todas as fases do seu ciclo de vida. Sendo assim, a arquitetura de software é um artefato essencial no processo de desenvolvimento de softwares modernos, sendo útil em todas as suas fases [CWMY02]. A importância da arquitetura fica ainda mais clara no contexto do desenvolvimento baseados em componentes. Isso acontece, uma vez que na composição de sistemas, os componentes precisam interagir entre si para oferecer as funcionalidades desejadas. Além disso, devido à diferença de abstração entre a arquitetura e a implementação de um sistema, um processo de desenvolvimento baseado em componentes deve apresentar uma distinção clara entre os conceitos de componente arquitetural, que é abstrato e não é necessariamente instanciável; e componente de implementação, que representa a materialização de uma especificação em alguma tecnologia específica e deve necessariamente ser instanciável.

2.2.2 Modelo COSMOS

Para que os benefícios proporcionados pela arquitetura de software sejam efetivamente válidos, é necessário que as abstrações produzidas no projeto arquitetural do sistema sejam consideradas nas várias fases do desenvolvimento, vistas na Seção 2.1.1: análise, projeto arquitetural, projeto, implementação, testes e distribuição [CWMY02]. Porém, as linguagens de programação atuais não oferecem a abstração necessária para o mapeamento direto das estruturas básicas da arquitetura para o código [JGR03]. Sendo assim, se faz necessário o uso de um modelo capaz de viabilizar a implementação dos componentes, conectores e configurações.

O modelo COSMOS [JGR03] é um modelo genérico e independente de plataforma, que utiliza estruturas disponíveis na maioria das linguagens de programação orientadas a objetos, para a implementação de componentes de software. Os principais objetivos do COSMOS são (i) garantir que a implementação do sistema esteja em conformidade com a sua arquitetura e (ii) facilitar a evolução dessa implementação.

Para oferecer esses objetivos, o modelo COSMOS incorpora um conjunto de diretivas de projeto que facilita a evolução dos componentes implementados. Essas diretivas incluem: (i) materialização dos elementos arquiteturais, isto é, componentes, conectores

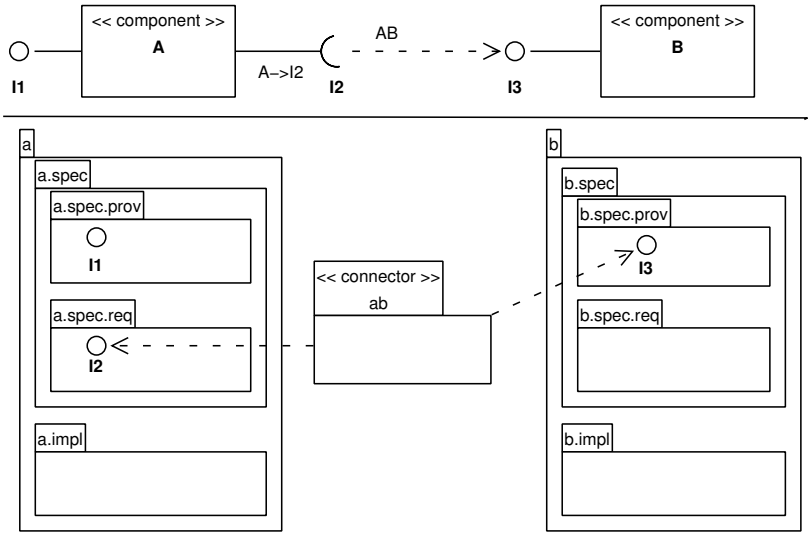

Figura 2.1: Modelo de Especificação do COSMOS

e configurações; (ii) separação entre a especificação e a implementação dos componentes, garantindo que apenas a especificação seja pública; (iii) declaração explícita das dependências entre os componentes, através de interfaces requeridas; e (iv) baixo acoplamento entre as classes da implementação.

O modelo COSMOS define três sub-modelos que tratam diferentes aspectos do desenvolvimento de um sistema baseado em componentes: (i) o **modelo de especificação** define a visão externa de um componente através da especificação das suas interfaces providas e requeridas; (ii) o **modelo de implementação** define como o componente deve ser implementado internamente; e (iii) o **modelo de conector** especifica as conexões entre os componentes, que são materializadas explicitamente através de conectores. Cada um desses modelos possui padrões de modelagem bem definidos que possibilitam a geração automática de códigos-fonte através de alguma ferramenta CASE[12] [TFdCGR04]. A figura 2.1 mostra uma configuração composta de dois componentes (A e B) e uma visão do mapeamento correspondente no modelo COSMOS. Essa visão de implementação é detalhada na Figura 2.2, que representa a implementação do componente A. A implementação do conector AB é apresentada na Figura 2.3.

Na Figura 2.2, é possível perceber mais claramente a separação explícita entre a especificação e a implementação do componente. Nesse modelo, as interfaces providas do componente pertencem ao pacote `spec.prov`, enquanto suas dependências são declaradas no pacote `spec.req`. Todas essas interfaces, sejam elas providas ou requeridas, possuem

[12]do inglês *Computer Aided Software Engineering*

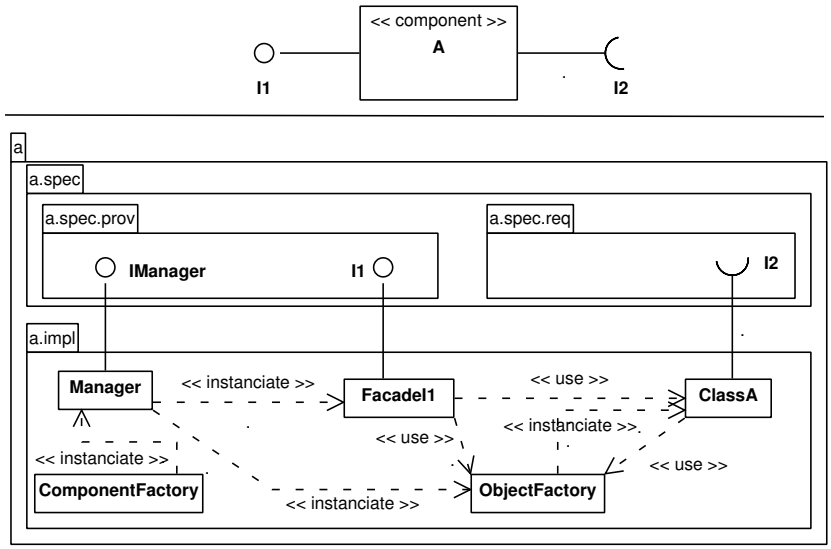

Figura 2.2: Modelo de Implementação do COSMOS

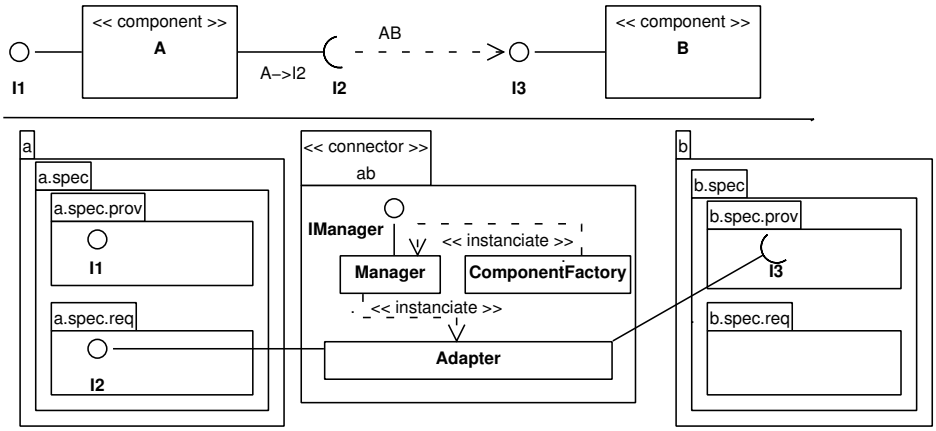

Figura 2.3: Modelo de Conectores do COSMOS

visibilidade pública. O modelo de implementação, por sua vez, tem o papel de materializar os serviços oferecidos pelo componente, utilizando as operações declaradas nas suas interfaces requeridas. A Tabela 2.2 detalha o papel das principais entidades presentes no modelo COSMOS.

Tabela 2.2: Principais entidades do modelo COSMOS

Entidade	Descrição
IManager	É uma interface provida pré-definida pelo modelo. Essa interface oferece as operações relativas à montagem e utilização do componente, isto é, operações de captura das instâncias que materializam as interfaces providas e de definição das instâncias que materializam as interfaces requeridas.
Manager	É a classe que materializa (do inglês *Realizes*) a interface IManager.
ComponentFactory	Esta é a classe responsável pela instanciação do componente, mais especificamente, pela instanciação da classe Manager. Por esse motivo, a ComponentFactory é a única classe com visibilidade pública no modelo de implementação (pacote impl).
Facade(s)	Cada classe Facade materializa o comportamento de uma interface provida pelo componente (exceto IManager). A associação de cada Facade com a sua respectiva interface provida é feita já nos construtores da classe Manager, através do método setProvidedInterface(), especificado em IMaganer.
ObjectFactory	A classe ObjectFactory é responsável pela instanciação das demais classes necessárias para a implementação do componente. O seu papel é facilitar a evolução interna do componente, através da redução do acoplamento entre suas classes.

2.2.3 O Método de Avaliação de Arquitetura ATAM

O ATAM[13] [KKC00] é um método de análise de arquiteturas, que foca na identificação e priorização dos requisitos não-funcionais do negócio relacionados aos atributos de qualidade desejados. A partir da definição dos requisitos não-funcionais, o método ATAM pode ser utilizado para analisar como os diversos estilos arquiteturais podem ser utilizados para alcançar cada um dos atributos de qualidade. O método ATAM apresenta nove atividades, classificadas em quatro grupos:

Grupo 1: Apresentação[14]

1. **Apresentar o ATAM.** Nessa atividade, o método deve ser descrito para as partes interessadas, que tipicamente constituem um grupo formado pelo representante do cliente, o arquiteto de software, o testador, o gerente de projeto, e o responsável pela fase de manutenção.
2. **Apresentar os marcos do negócio**[15]. O gerente de projeto descreve quais são os objetivos pretendidos para o sistema. Essa descrição inicial deve servir de base para a escolha da arquitetura inicialmente proposta. A escolha dessa

[13] do inglês *Architecture Tradeoff Analysis Method*
[14] do inglês *Presentation*
[15] do inglês *Business drivers*

arquitetura deve se basear principalmente na expectativa de disponibilidade, tempo de entrega[16] e confiabilidade.

3. **Apresentar a arquitetura.** A partir da descrição do gerente de projeto, o arquiteto deve descrever a arquitetura inicialmente proposta, focando em como cada um dos objetivos descritos será materializado.

Grupo 2: Investigação e Análise[17]

4. **Identificar as possibilidades para a arquitetura.** A lista de arquiteturas possíveis é identificada pelo arquiteto, mas ainda não são analisadas. Essa lista pode ser criada a partir das arquiteturas semelhantes à arquitetura inicialmente proposta.
5. **Gerar uma árvore com a classificação dos atributos de qualidade desejados.** Os atributos de qualidade especificados devem ser desmembrados em requisitos não-funcionais. Em seguida, são especificados cenários de exemplos que auxiliem a compreensão do que realmente se espera para cada um desses requisitos. Após a especificação dos cenários, esses requisitos devem ser priorizados.
6. **Analisar as possibilidades para a arquitetura.** A partir da lista de prioridades definida na atividade anterior, a lista de arquiteturas possíveis deve ser analisada, por exemplo, uma arquitetura que prioriza o desempenho pode ser menos desejada que outra que priorize a confiabilidade.

Grupo 3: Teste[18]

7. ***Brainstorm* e priorização dos cenários.** Baseado nos cenários especificados na árvore de prioridades, deve-se refinar essa lista de cenários a partir das idéias de cada um dos interessados. Após a identificação dos vários cenários possíveis, esses cenários devem ser priorizados através de uma votação entre os interessados.
8. **Analisar as possibilidades para a arquitetura.** Este passo consiste em uma nova iteração da atividade 6. Mas nesse momento os principais cenários priorizados na atividade anterior devem gerar casos de teste para analisar a arquitetura. Com a execução desses casos de teste, podem ser descobertos riscos e relações de compromisso[19] entre as propostas de arquitetura listadas. Tudo isso deve ser documentado.

[16] do inglês *Time to market*
[17] do inglês *Investigation and Analysis*
[18] do inglês *Testing*
[19] do inglês *Tradeoff*

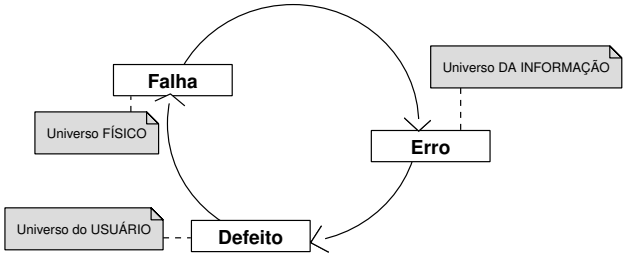

Figura 2.4: Ciclo entre a ocorrência de Falhas, Erros e Defeitos

Grupo 4: DIVULGAÇÃO[20]

9. **Apresentar resultados.** Com a execução das atividades do método ATAM, são coletadas algumas informações importantes, tais como arquiteturas candidatas, requisitos não-funcionais com prioridades, cenários, riscos e relações de compromisso. Com essas informações em mãos, deve ser gerado um relatório detalhando as possíveis estratégias para a estrutura da arquitetura do sistema.

2.3 Tolerância a Falhas e Tratamento de Exceções

2.3.1 Falha, Erro e Defeito

Falha[21] é um evento que causa **erros**[22]. Caso o estado errôneo do sistema não seja tratado em um tempo determinado, ocorrerá um **defeito**[23], que se manifestará pela não execução ou mudança indesejada no serviço especificado. Como pode ser visto na Figura 2.4, além de indicar um estado errôneo irreversível, a ocorrência de um defeito realimenta o ciclo e pode gerar novas falhas. Esse fenômeno é conhecido como propagação da falha e deve ser contido através do seu confinamento.

Existem várias classificações para falhas na literatura técnica [AL90, Lap85, JP89]. Normalmente, essas classificações agrupam as falhas em falhas físicas, que se referem a falhas de componentes de hardware, e falhas humanas. Falhas humanas compreendem falhas de projeto, decorrentes das fases de desenvolvimento do software e falhas de interação, que por sua vez, podem ser acidentais ou intencionais.

Inicialmente, o conceito de falhas se relacionava apenas às falhas de origem física, por exemplo, a mudança de um bit na memória por interferência eletromagnética. Entretanto,

[20] do inglês *Reporting*
[21] do inglês *Fault*
[22] do inglês *Error*
[23] do inglês *Failure*

dado o aumento da complexidade dos sistemas de software, falhas de projetos se tornaram freqüentes e quase inevitáveis.

As falhas também podem ser classificadas segundo a sua duração. Nessa perspectiva, as falhas são distribuídas em três grupos: (i) transientes, quando existe a chance das falhas ocorrem mais de uma vez, mas não obrigatoriamente, por exemplo, a invasão do sistema por um *hacker*; (ii) intermitente, quando a ocorrência da falha se repete, mas não necessariamente em períodos definidos, por exemplo, a manifestação de um vírus presente no sistema; e (iii) permanentes, que são caracterizadas pela sua presença constante no sistema, por exemplo, falhas de especificação e implementação.

No contexto de sistemas distribuídos, as falhas são classificadas basicamente em três tipos [Gär99]: (i) falhas bizantinas, que são arbitrárias e por isso de difícil detecção; (ii) falhas de performance, que ocorrem quando a resposta do serviço solicitado acontece em um tempo inesperado, normalmente após o limite máximo estipulado; e (iii) falhas de omissão, quando a resposta do serviço solicitado não é recebida. As falhas de omissão são também conhecidas como falhas do tipo falha e pára.

É importante observar que uma falha pode resultar em um, mais de um ou nenhum erro. Esse fato, aliado à impossibilidade de se conhecer todas as causas de uma falha dificultam seu o processo de detecção. Outro fator complicante para a detecção de uma falha é o fato de várias falhas distintas poderem acarretar em um mesmo erro. Por esse motivo, a identificação do erro não resulta naturalmente na identificação da falha responsável por ele. Desta forma, faz-se necessário uma análise rigorosa do contexto de manifestação dos erros para que seja possível inferir suas causas e em seguida tratá-las.

2.3.2 Dimensões da confiabilidade de sistemas de software

O grau de **confiança no funcionamento** de um sistema de software[24] depende principalmente do número de falhas contidas no sistema e da maneira como ele se comporta na presença dessas falhas.

Um sistema confiável[25] é um sistema que oferece um grau de confiabilidade de funcionamento mensurável aos seus usuários [Som01]. As principais dimensões dessa confiança são [Lap85, Pra96, Avi97]: (i) **disponibilidade**[26], que é a capacidade do sistema oferecer seus serviços quando requerido; (ii) **confiabilidade**[27], que é a capacidade do sistema oferecer seus serviços conforme a especificação; (iii) **segurança no funcionamento**[28], que é

[24] do inglês *Software dependability*
[25] do inglês *Dependable*
[26] do inglês *Availability*
[27] do inglês *Reliability*
[28] do inglês *Safety*

a capacidade do sistema operar sem apresentar defeitos catastróficos; e (iv) **segurança**[29], que é a capacidade do sistema evitar falhas maliciosas, isto é, falhas provenientes do meio externo e que podem ser intencionais.

As principais abordagens utilizadas para o desenvolvimento de sistemas de software com requisitos críticos de confiabilidade no funcionamento são [Som01]: (i) prevenção de falhas[30], (ii) remoção de falhas[31], (iii) tolerância a falhas[32] e (iv) avaliação de falhas[33]. Essas técnicas são complementares e podem ser combinadas para o desenvolvimento de sistemas de software robustos. A tabela 2.3 descreve sucintamente cada uma delas.

Tabela 2.3: Abordagens para a implementação de sistemas confiáveis

Abordagem	Descrição
Prevenção de Falhas	O sistema é desenvolvido de modo a evitar a inserção de falhas humanas. Neste caso, o processo de desenvolvimento é organizado para detectar e corrigir falhas, antes da entrega do sistema ao usuário final.
Remoção de Falhas	São utilizadas técnicas de verificação e validação (formais ou não) para detectar e corrigir falhas. Essas técnicas também são executadas antes da entrega do sistema ao usuário final.
Tolerância a Falhas	O sistema é projetado de modo que a presença de falhas durante a sua execução não resulte em defeitos visíveis ao usuário.
Avaliação de Falhas	O sistema é simulado, a fim de identificar os estados impossíveis de serem alcançados. Essa identificação é utilizada para restringir o modelo de falhas do sistema, reduzindo os custos e aumentando a eficiência das técnicas de tolerância a falhas.

O objetivo principal das técnicas de prevenção e remoção de falhas é evitar a inserção das falhas no sistema durante a sua construção, isto é, antes dele ser entregue ao usuário final. Enquanto isso, as técnicas de tolerância a falhas oferecem mecanismos que possibilitam uma convivência amistosa com determinados tipos de falhas. Em outras palavras, o objetivo das técnicas de tolerância a falhas é não permitir que erros acarretem defeitos no sistema [AL90]. Já as técnicas de avaliação de falhas, como o próprio nome indica, possuem uma característica mais analítica, buscando restringir o modelo de falhas através da identificação de estados não alcançáveis ou menos importantes.

Os sistemas de software que possuem requisitos críticos de confiabilidade e/ou disponibilidade devem, de alguma forma, implementar atividades de tolerância a falhas [Som01]. Essa preocupação se baseia no princípio de que não se pode garantir a corretude dos modelos especificados, uma vez que mesmo utilizando técnicas de especificação formal, a intervenção humana é imprescindível.

[29] do inglês *Security*
[30] do inglês *Fault prevention*
[31] do inglês *Fault removal*
[32] do inglês *Fault tolerance*
[33] do inglês *Fault forecasting*

2.3.3 Fases da Tolerância a Falhas

Existem várias propostas para a divisão das fases de implementação de tolerância a falhas. Uma das propostas mais utilizadas é a classificação em quatro fases de aplicação [AL90]: (i) detecção do erro[34], (ii) confinamento e avaliação[35], (iii) recuperação de erros[36], e (iv) tratamento de falhas[37].

Uma falha primeiro se manifesta como um erro, para então ser detectada. A detecção do erro consiste na verificação da consistência do estado do sistema. De forma complementar à detecção, a fase de confinamento e avaliação tem o objetivo de conhecer e isolar as entidades inconsistentes do sistema. Após conhecer os seus componentes errôneos, é necessário recuperar o estado normal do sistema. A recuperação do erro pode ocorrer de duas maneiras: (i) recuperação por retrocesso[38], que consiste na restauração de um estado anteriormente válido; e (ii) recuperação por avanço[39], que consiste na construção de um novo estado válido a partir das informações contextuais disponíveis. Duas técnicas bastante utilizadas para a recuperação de estados errôneos são a definição de marcos de execução[40] e o tratamento de exceções. Essas técnicas, são utilizadas respectivamente para implementar técnicas de recuperação por retrocesso e recuperação por avanço. As Seções 2.3.4 a 2.3.6 detalham a segunda abordagem.

A última fase sugerida para as atividades de tolerância a falhas é reservada para o tratamento de falhas propriamente dito. O objetivo desta fase é identificar e eliminar as causas dos erros do sistema, de maneira que eles não voltem a acontecer. Uma técnica muito utilizada para essa tarefa é a de reconfiguração arquitetural através do uso de componentes redundantes [IB01]. Com a disponibilidade de componentes críticos sobressalentes, é possível, por exemplo, reconfigurar a arquitetura do sistema de forma a evitar o envio de mensagens aos componentes falhos. Esse tipo de correção de erro é conhecido como mascaramento da falha[41].

Vale a pena ressaltar a diferença entre componentes replicados e componentes redundantes. No contexto do desenvolvimento de software, o conceito de réplica significa que existe uma outra instância de um mesmo componente. Enquanto isso, um componente redundante, além de ser materializado por uma instância distinta, possui estrutura interna igualmente diferente, já que se trata de um outro componente, projetado possivelmente

[34] do inglês *Error detection*
[35] do inglês *Damage assessment*
[36] do inglês *Error recovery*
[37] do inglês *Fault treatment*
[38] do inglês *Backward error recovery*
[39] do inglês *Forward error recovery*
[40] do inglês *Checkpoints*
[41] do inglês *Fault masking*

por uma equipe distinta de desenvolvedores (diversidade de projeto[42]). Em termos de tolerância a falhas, componentes redundantes são mais expressivos, apesar do seu reflexo negativo no custo do sistema.

Apesar das técnicas de tolerância a falhas se basearem na distribuição e na redundância de elementos [AL90], muitos fatores devem ser levados em conta ao se empregar redundância, seja ela de componentes, de projeto ou temporal. Os principais fatores são os custos financeiro, de tempo, de sobrecarga e de espaço. A utilização desses recursos deve ser analisada baseado no seu impacto nas fases de projeto, implementação e utilização do sistema. Deve ser feita uma análise da relação entre esses custos e as conseqüências causadas por problemas relativos ao mal funcionamento ou à indisponibilidade dos serviços prestados pelo sistema.

Uma maneira de se implementar técnicas de tolerância a falhas é utilizando a estrutura oferecida pelos mecanismos de tratamento de exceções [Fer01] das linguagens de programação. Esses mecanismos oferecem um arcabouço para a criação de tratadores adequados para cada tipo de exceção, possibilitando o tratamento de possíveis inconsistências, ou até mesmo a continuação da execução das funcionalidades do sistema.

2.3.4 Visão Geral de Tratamento de Exceções

O **comportamento normal** de um sistema é responsável por oferecer os serviços especificados nos seus requisitos. Porém existem circunstâncias que impedem a execução adequada desses serviços. Como se espera que estas circunstâncias ocorram raramente, programadores referem-se a elas como **exceções** [Cri89]. Apesar de se esperar uma ocorrência rara, na prática não é bem isto que se observa. Devido à grande influência do meio externo nos sistemas computacionais, as situações excepcionais são bastante freqüentes [Cri89].

Tratamento de exceções [Goo75] é um mecanismo conhecido para a implementação de tolerância a falhas em sistemas de software. Os Mecanismos de Tratamento de Exceções (MTE) capacitam os desenvolvedores a definirem o **comportamento excepcional** dos sistemas, que consiste tanto das atividades de detecção e lançamento das exceções, quanto dos seus tratadores correspondentes.

Quando um erro é detectado, uma exceção é gerada, ou **lançada** e o MTE da linguagem de programação ativa automaticamente o tratador mais próximo que se adequa ao tipo da exceção. Dessa forma, se a mesma exceção puder ser lançada em diferentes partes do programa, é possível que diferentes tratadores sejam executados. Por esse motivo, o local de lançamento da exceção também é conhecido como Contexto de Tratamento da Exceção (CTE). Um CTE é uma região de um programa onde exceções de um mesmo tipo são tratadas de maneira uniforme. Cada CTE possui um conjunto de tratadores

[42] do inglês *Design diversity*

associados a ele e cada tratador, por sua vez, está associado a um tipo de exceção em particular. Uma exceção lançada dentro de um CTE pode ser **capturada** por um dos seus tratadores. Quando a exceção é tratada, o sistema pode retomar o seu comportamento normal de execução; caso contrário, a exceção é **sinalizada** para o CTE de nível imediatamente superior. A Figura 2.5 ilustra o funcionamento de um MTE. Apesar da Exceção E1 ter sido lançada no CTE2 (Linha 4), ela só é capturada no nível imediatamente superior (CTE1, Linha 13). Por esse motivo, essa exceção é tratada pelo tratador T3.

```
1  try{ //CTE1 (bloco de contexto excepcional 1)
2     //... ócdigo implementado
3     try{ //CTE2 (bloco de contexto excepcional 2)
4        throw new E1(); //lancamento de uma excecao do tipo E1
5     }
6     catch(E2 e){ //Tratador1 - T1
7        //tratamento para excecoes do tipo E2
8     }
9     catch(E3 e){ //Tratador2 - T2
10       //tratamento para excecoes do tipo E3
11    }
12 }
13 catch(E1 e){ //Tratador3 - T3
14    //tratamento para excecoes do tipo E1
15 }
```

Figura 2.5: Funcionamento de um Mecanismo de Tratamento de Exceções (MTE)

2.3.5 Tratamento de Exceções em Sistemas Concorrentes

Atividades executadas de forma cooperativa podem ser estruturadas como um conjunto de ações coordenadas [XRR+99]. As atividades que implementam essas ações são conhecidas como ações atômicas coordenadas[43]. No contexto específico deste trabalho, essas ações serão chamadas de **colaborações** [dLR99].

Uma colaboração oferece um mecanismo para gerenciar de maneira coordenada, a execução de ações concorrentes. Os componentes **participantes** de uma colaboração só podem trocar informações entre si, de modo a evitar a propagação de algum erro para outros componentes do meio externo. Por esse motivo, apesar dos participantes poderem entrar na colaboração de maneira assíncrona, eles só podem sair dela sincronamente, após a confirmação da corretude do estado da colaboração como um todo.

[43]do inglês *Coordinated atomic actions (CA-Actions)*

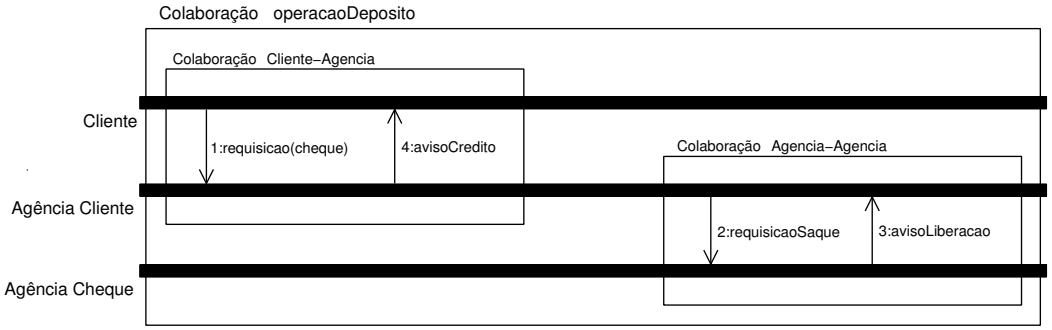

Figura 2.6: Operação de Depósito em Cheque (Colaboração)

A Figura 2.6 mostra parte de um sistema bancário, mais precisamente a estruturação de um tratamento de exceções coordenado, decorrente do lançamento de duas exceções concorrentes. Os participantes da colaboração são representados por barras horizontais. A comunicação entre esses participantes está representada por setas direcionadas, que indicam o sentido da comunicação. As colaborações, por suas vez, são representadas por retângulos que envolvem os participantes.

Esse exemplo simula uma operação de depósito de um cheque. O cliente deseja efetuar o depósito e leva o cheque consigo. Primeiramente, ele entra em contato com a sua agência e inicia a operação. Em seguida, a agência onde está sendo efetuada a transação envia o cheque para a agência que detém a conta do cheque. Após o processamento dos dados, ocorre a liberação do valor para a agência do cliente, que o repassa para a conta apropriada. Como demonstrado nesta figura, uma colaboração pode ser vista como um componente e por isso pode assumir o papel de participante de outras colaborações recursivamente (aninhamento de colaborações).

No caso do lançamento de exceções concorrentes, é necessário identificar qual o tratador que deve ser executado em cada um dos participantes. Para isso, devido à relação estreita entre o tratamento e o tipo da exceção, é necessário identificar um tipo que seja equivalente às exceções concorrentes lançadas. Para isso, é possível utilizar uma técnica conhecida como grafo de resolução [CR86], que estrutura as exceções de maneira hierárquica. Havendo o lançamento concorrente de exceções, o tipo da exceção equivalente nesse grafo é o ancestral mais próximo, que seja comum a todas as exceções lançadas. Sendo assim, as exceções de um grafo podem ser classificadas em dois tipos: (i) exceções simples, que são representadas pelas folhas do grafo; e (ii) exceções compostas, que se referem aos demais nós da hierarquia. A Figura 2.7 mostra um exemplo de um grafo de resolução para o exemplo da operação de depósito em cheque, mostrado na Figura 2.6. Esse exemplo está estruturado na forma de uma árvore.

Para exemplificar o seu funcionamento, supõe-se que durante a execução da operação

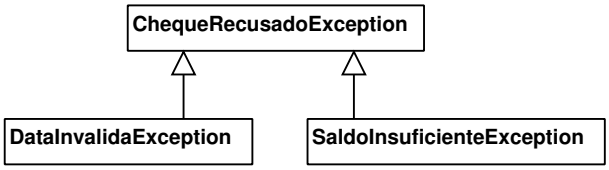

Figura 2.7: Grafo de Resolução da Colaboração agência-agência

verificarCheque() (colaboração agência-agência) são lançadas as exceções `DataInvalidaException` e `SaldoInsuficienteException`. De acordo com o grafo de resolução da Figura 2.7, existe uma exceção composta equivalente a esse estado errôneo do sistema (a exceção `ChequeRecusadoException`). A colaboração agencia-agencia tentará recuperar o estado normal do sistema. Caso não seja possível, é lançada uma exceção ao seu nível exatamente superior (colaboração operacaoDeposito), que tentará tratar de forma recursiva.

2.3.6 Tratamento de Exceções em Sistemas Baseados em Componentes

Tendo em vista as diferenças existentes entre as abstrações do modelo orientado a objetos e do DBC, o tratamento de exceções em sistemas baseados em componentes possui algumas particularidades que não são totalmente satisfeitas pelas linguagens de programação atuais.

Uma das dificuldades na construção de sistemas tolerantes a falhas baseados em componentes é a falta de separação entre o tratamento excepcional e a implementação das funcionalidades do sistema, o que prejudica o seu entendimento e principalmente a sua reutilização [LR01]. Uma outra necessidade para os MTE no DBC é a preocupação com a propagação de exceções de acordo com o fluxo interativo entre os componentes. Durante a propagação do fluxo excepcional, pode ser necessário por exemplo, converter os tipos de algumas exceções entre componentes que adotam modelos de falhas distintos [dCGFPR04, Fer01, SUVD03]. Esse tipo de manipulação intermediária nas mensagens do fluxo interativo só pode ser feito na arquitetura do sistema, mais especificamente nos conectores arquiteturais.

A versatilidade e a autonomia dos componentes faz com que um sistema baseado em componentes fique sujeito a alguns problemas característicos de sistemas concorrentes [Szy02]. Por exemplo, é possível que componentes distribuídos que façam parte de um sistema lancem exceções de maneira autônoma, o que pode significar um estado errôneo diferenciado, que não é equivalente à soma do tratamento das exceções lançadas inicialmente [RdLeFCF04, dLR00]. Assim, se faz necessário a execução de uma função que, a

partir das exceções lançadas, possibilite a descoberta do seu tratador equivalente. Esse tipo de tratamento que envolve a análise do fluxo excepcional de mais de um componente do sistema também só é possível na arquitetura do sistema.

Como já pôde ser percebido, considerar o tratamento de exceções na arquitetura de software é imprescindível para a implementação dos MTE no DBC. Nos conectores arquiteturais é possível, por exemplo: (i) gerenciar o fluxo excepcional dos componentes; (ii) implementar heurísticas para melhorar a escolha do tratador apropriado; (iii) possibilitar o uso de COTS, através de possíveis conversões entre os tipos de dados; e (iv) implementar o tratamento de falhas (Seção 2.3.2) através do isolamento de componentes defeituosos. Além do mais, um componente isolado não é capaz de oferecer os recursos necessários para identificar e tratar os erros de forma efetiva [dLR99].

Além da necessidade de enfatizar o aspecto arquitetural do sistema, estudos mostram alguns requisitos desejáveis e outros essenciais para a implementação efetiva de tratamento de exceções no DBC. Os requisitos essenciais devem ser levados em consideração no projeto do comportamento excepcional de qualquer sistema baseado em componentes. São eles [LS98]:

- **Reusabilidade.** Componentes de software devem ser reutilizáveis. Sendo assim, as exceções e os tratadores excepcionais devem ser modelados separadamente e devem ser propícios à reutilização.

- **Encapsulamento.** Componentes de software possuem requisitos críticos de encapsulamento. Assim, as exceções propagadas entre componentes distintos devem poder ser alteradas no decorrer do fluxo. Essa alteração visa possíveis adaptações entre os diversos modelos de falhas utilizados.

- **Flexibilidade.** Os mecanismos de tratamento de exceções devem ser flexíveis. Isso implica que o refinamento das exceções e tratadores deve se adequar à criticidade do sistema.

- **Consistência.** Um erro significa uma inconsistência no estado do componente. Assim, os tratadores de exceções, mesmo que não consigam retomar a execução normal, devem tornar o estado do componente consistente, ou até mesmo isolá-lo do restante do sistema.

2.3.7 Componente Tolerante a Falhas Ideal

O conceito de componente tolerante a falhas ideal, ou simplesmente **componente ideal** foi introduzido por Anderson e Lee [AL90] para denominar um componente tolerante a falhas cujo comportamento é categorizado em dois tipos: **comportamento normal** e

comportamento excepcional (ou anormal). Com o comportamento normal, o componente executa adequadamente os serviços, produzindo respostas corretas. Entretanto, na ocorrência de alguma falha, o componente assume o comportamento excepcional, tentando tratá-la de maneira adequada.

A divisão do comportamento é refletida na sua estrutura, como ilustrado na Figura 2.8. A estrutura do componente ideal é dividida em duas partes: uma define o seu comportamento normal, enquanto a outra é responsável pelo tratamento de exceções, implementando o comportamento excepcional [AL90]. Um componente ideal pode ser constituído de outros componentes ideais, caracterizando uma estruturação recursiva.

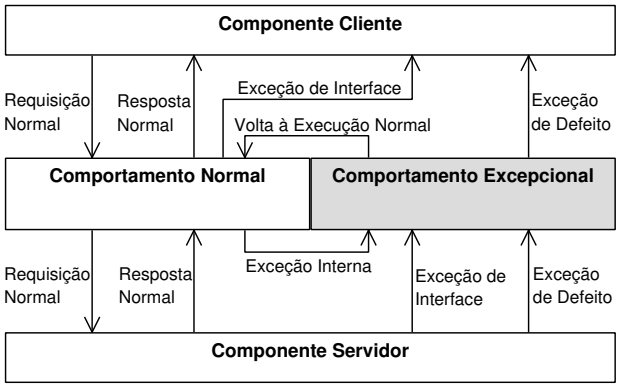

Figura 2.8: Estrutura do Componente Tolerante a Falhas Ideal

As exceções ocorridas em um componente ideal podem ser classificadas como **internas** ou **externas**. As exceções externas, por sua vez, podem ser **exceções de defeito**, quando indicam a incapacidade de fornecer um serviço, ou **de interface**, decorrentes de requisições incorretas por parte dos clientes. Na ocorrência de qualquer um dos tipos, a execução normal do componente é interrompida e é ativado o tratador específico para a exceção ocorrida. Caso seja possível, após a execução do tratamento adequado, o fluxo de execução deve retornar ao componente normal. Caso não seja possível a recuperação do estado normal do sistema, é retornada uma exceção de defeito à sua camada superior, que pode tratá-la ou continuar propagando. Porém, devido à importância das informações contextuais, quanto mais próximo do local de origem do erro, mais eficiente pode ser o tratamento [AL90].

2.3.8 O Método MDCE

A MDCE é um método genérico que apresenta diretrizes para a especificação do comportamento excepcional de um sistema nas fases de identificação de requisitos, análise,

projeto e implementação do sistema. Neste método, grande parte dos esforços estão concentrados na fase de análise de requisitos, na busca da antecipação de possíveis exceções, para a implantação de tolerância a falhas. Segue uma descrição de cada uma das fases desse método.

1. **Especificação de Requisitos.** Durante esta fase, os requisitos são modelados sob a forma de casos de uso que possuem a descrição tanto do seu comportamento normal, quanto do comportamento excepcional associado a ele. Estes casos de uso são refinados com assertivas e seus cenários são representados através de diagramas de atividades da UML. A especificação de assertivas consiste na definição de condições a serem satisfeitas antes da execução da funcionalidade (pré-condições), após a sua execução (pós-condição) e das condições válidas durante toda a execução do serviço (invariantes) [JM97].

2. **Análise e Projeto.** As fases de análise e projeto do sistema são responsáveis pela modelagem dos seus componentes internos, englobando os comportamentos normal e excepcional. Durante as atividades de projeto, a MDCE destaca a importância de se representar exceções e tratadores no projeto arquitetural do sistema, propondo que cada componente arquitetural seja estruturado segundo o modelo do componente tolerante a falhas ideal, visto na Seção 2.3.7. No entanto, o detalhamento da sistemática necessária para a modelagem do fluxo de exceções e dos tratadores arquiteturais fica como trabalho futuro.

   Ainda nas fases de análise e projeto do sistema, são sugeridos modos de representação para as exceções e seus tratadores na forma de classes. Apesar de focar no contexto das classes do sistema, a MDCE sugere a separação entre as classes que implementam o comportamento normal (classes normais) e as que implementam os tratadores de exceção (classes excepcionais). Além disto, as exceções também são representadas como classes UML. Essas classes possuem o estereótipo $<< exception >>$ e são associadas às classes que a lançam.

3. **Implementação.** Na fase de implementação foram propostas duas diretrizes para orientar o trabalho dos programadores: (i) manter a separação entre as classes normais e excepcionais, que possibilita uma melhor reutilização de código; e (ii) explicitar na assinatura das operações o contrato estabelecido entre o cliente e o servidor, inclusive o comportamento excepcional. Porém, a MDCE não apresenta diretrizes para a implementação de componentes em nenhum modelo específico.

Com essas atividades básicas definidas e descritas conforme mostrado, o método MDCE é genérico o suficiente para ser aplicado aos processos de desenvolvimento pre-

sentes na literatura. Inclusive esse método já foi adaptado ao processo de DBC Catalysis [DW99].

2.4 Uma Estruturação do Comportamento Excepcional para DBC

Por mais cuidadosa que seja a forma como o sistema possa ser especificado, é possível que durante a implementação, surjam situações excepcionais não previstas na especificação. O fato das saídas excepcionais fazerem parte do contrato estabelecido com os clientes do componente faz com que qualquer implementação correta dessa especificação inclua atividades de detecção e tratamento adequados. Apesar disso, no desenvolvimento de sistemas robustos, as exceções não previstas na especificação também devem ser consideradas. Para esses erros não antecipados, o desenvolvedor deve definir novas exceções que os representem.

Por se tratar de uma tarefa intuitiva, a definição de exceções não declaradas na especificação se mostra problemática, uma vez que dada uma mesma especificação, programadores distintos podem descobrir exceções diferenciadas. Um outro fator agravante é o fato dos tipos dessas exceções serem definidos normalmente de maneira *ad hoc* e arbitrária. Essa falta de preocupação com o tipo da exceção dificulta a sua contextualização na arquitetura e com o comprometimento da semântica do erro, a possibilidade de introduzir mecanismos eficientes de tolerância a falhas é reduzida.

Asterio *etal*. [dCGFPR04, dCG04] propõem uma solução para esse problema através de uma abordagem de estruturação dessas exceções. Essa estratégia compreende duas visões de tratamento: uma local, ou intra-componente; e outra global, ou inter-componentes. A abordagem intra-componente é utilizada durante o desenvolvimento interno do componente e pode ser utilizada tanto para o desenvolvimento de componentes novos, quanto para componentes reutilizados. Já a abordagem inter-componentes é implementada durante a integração dos componentes do sistema, mais especificamente na implementação dos conectores arquiteturais. Para possibilitar a utilização conjunta dessas duas abordagens, Asterio *etal*. definiu uma hierarquia comum de tipos de exceções. A Seção 2.4.1 apresenta essa hierarquia excepcional, enquanto as Seções 2.4.2 e 2.4.3 apresentam respectivamente as abordagens intra- e inter-componentes.

2.4.1 Hierarquia de Exceções

Segundo a proposta de Asterio *etal*. [dCGFPR04, dCG04], as exceções devem ser definidas como classes e organizadas de acordo com a hierarquia apresentada na Figura 2.9. Essa hierarquia abrange tanto as exceções previstas na especificação, quanto as não previstas ou

específicas da linguagem de programação. Seu objetivo principal é relacionar de maneira consistente as exceções internas e as exceções arquiteturais que fluem entre diferentes componentes.

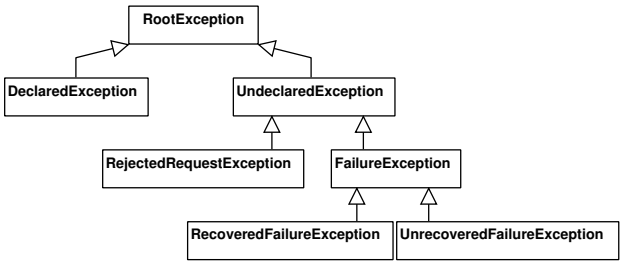

Figura 2.9: Hierarquia Excepcional Proposta

No topo da hierarquia, existe a classe `RootException`, a superclasse de todas as exceções do sistema. A execução de um componente termina com uma `DeclaredException` quando se trata de uma saída excepcional declarada na sua especificação. Todas as exceções do tipo `DeclaredException` ou de um de seus sub-tipos, que possam ser propagadas, devem ser declaradas explicitamente na assinatura da operação, caso a linguagem de programação adotada permita.

A hierarquia de `UndeclaredException` é utilizada para mapear as exceções que sejam específicas do ambiente utilizado, como por exemplo exceções do sistema operacional ou da linguagem de programação adotada. Esses tipos abstratos de exceções possibilitam que os responsáveis pela integração do sistema adicionem tratadores que lidem de uma maneira sistemática com exceções imprevistas. O tipo `UndeclaredException` possui dois sub-tipos: `RejectedRequestException` e `FailureException`. Exceções que herdam de `RejectedRequestException` são utilizadas para sinalizar violações de pré-condições, desde que essas violações não interfiram no estado do sistema.

Exceções do tipo `FailureException` indicam uma falha no atendimento a uma requisição válida. O tipo `FailureException` possui dois sub-tipos: `RecoveredFailureException` e `UnrecoveredFailureException`. Exceções do tipo `RecoveredFailureException` são utilizadas para indicar que, apesar da ocorrência do erro, o componente continua em um estado consistente. De forma complementar, exceções do tipo `UnrecoveredFailureException` são utilizadas para indicar que o estado do sistema pode ter sido afetado.

Fazendo um comparativo entre a estruturação proposta por Asterio *etal*. e o modelo do componente ideal apresentado na Seção 2.3.7, o mapeamento entre as duas abordagens poderia ser feito como segue [dSBFR04]:

- As exceções de interface podem herdar de `DeclaredException`, se previstas na especificação; ou de `RejectedRequestException`, se não previstas.

- As exceções internas podem ser do tipo `DeclaredException`, se previstas, e `RecoveredFailureException`, se imprevista.
- O tipo de uma exceção de defeito depende da interferência do erro no estado do sistema. Caso o estado permaneça consistente, ela pode ser classificada como `DeclaredException`, se foi prevista na especificação, ou `RecoveredFailureException`, se não prevista. Caso não haja garantia de consistência, são implementadas como subclasses de `UnrecoveredFailureException`.
- Além disso, as exceções específicas da linguagem, por exemplo, uma exceção que sinaliza uma divisão por zero, devem pertencer à hierarquia de `FailureException`.

2.4.2 Abordagem Intra-Componente

A abordagem intra-componente consiste na estruturação dos tratadores excepcionais internos ao componente. Para lidar com essas exceções internas, Asterio *etal*. propõe uma estruturação dos tratadores em dois níveis complementares, como mostrado na Figura 2.10: (i) tratamento no nível de aplicação (ALE)[44]; e (ii) tratamento no nível de fronteira (BLE)[45]. Os tratadores ALE se referem às exceções que podem ser lançadas pelas classes de implementação do componente. Já os tratadores BLE são responsáveis pelo tratamento das exceções lançadas pela classe `Facade` [GHJV95], que define um ponto de acesso às classes que implementam os serviços oferecidos do componente.

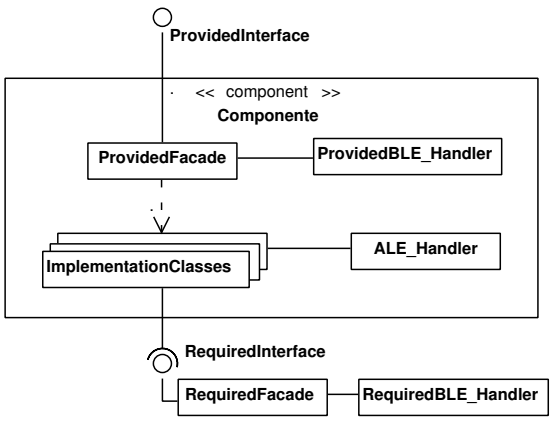

Figura 2.10: Estruturação Interna do Componente

A divisão de responsabilidades entre cada um dos módulos internos do componente pode ser feita da seguinte forma:

[44] do inglês *Application-level exception (ALE) handling*
[45] do inglês *Boundary-level exception (BLE) handling*

- **Classe `ProvidedFacade`.** Por representar um acesso centralizado ao componente, a classe `Facade` também é responsável por serializar todas as requisições feitas. Do ponto de vista do comportamento excepcional do sistema, essas classes também são responsáveis por detectar as condições excepcionais relativas às violações das pré-condições necessárias para uma requisição válida dos serviços. Essas violações podem abranger tanto as assertivas antecipadas na especificação, quanto as que não foram previstas. Essas violações devem ser sinalizadas através do lançamento de exceções internas. No caso das exceções especificadas, suas classes devem herdar de `DeclaredException`, caso contrário, ela deve ser de um dos sub-tipos de `UndeclaredException`. Ou `RejectedRequestException`, ou `UnrecoveredFailureException`, dependendo do estado do sistema.

- **Tratador de exceção ProvidedBLE.** Os tratadores de fronteira são responsáveis por tratar as exceções lançadas pela classe `ProvidedFacade` do componente. Exceções do tipo `DeclaredException` e `RejectedRequestException` são simplesmente propagadas para seus respectivos clientes. Para outros tipos de exceções é indicado executar alguma atividade de recuperação por retrocesso[46], como visto na Seção 2.3.3. Nesses casos, se o estado do sistema for recuperado, é lançada uma exceção do tipo `RecoveredFailureException`. Caso contrário, é lançada uma exceção do tipo `UnrecoveredFailureException`.

- **Classes de implementação.** Do ponto de vista do comportamento excepcional do sistema, essas classes são responsáveis por: (i) detectar as condições excepcionais antecipadas na especificação e sinalizar a violação dos contratos através do lançamento de exceções internas; (ii) sinalizar outras condições excepcionais específicas da linguagem de programação, também através de exceções internas; e (iii) executar atividades de "limpeza" sistemática, um papel equivalente ao desempenhado pelos blocos `finally` de Java. Os tipos das exceções internas lançadas pelas classes de implementação podem tanto ser subtipos de `UndeclaredException`, quanto de `DeclaredException`, dependendo do erro.

- **Tratador de exceção ALE.** Esses tratadores são responsáveis basicamente por tratar dois tipos de exceções: (i) exceções externas, que são propagadas pelas interfaces requeridas do componente; e (ii) exceções internas, que são lançadas pelas classes de implementação, sejam elas previstas ou não. Sempre que possível, esses tratadores devem mascarar as exceções, através da implementação de mecanismos de recuperação por avanço[47], como visto na Seção 2.3.3.

[46] do inglês *Backward error recovery*
[47] do inglês *Forward error recovery*

- **Classe `RequiredFacade`.** Por representar a fronteira entre o componente e a sua interface requerida, essa classe é responsável por repassar as requisições dos serviços requeridos. Sendo assim, caso um desses serviços lance uma exceção, a classe `RequiredFacade` deve capturá-la e ativar o respectivo tratador de fronteira (classe `RequiredBLE_Handler`). No caso de algum serviço não estar disponível, a classe deve lançar uma exceção do tipo `UnrecoveredFailureException`. Da mesma forma, se o serviço executado lançar uma exceção não prevista no modelo de falhas do componente, deve ser lançada uma exceção do tipo `UnrecoveredFailureException`

- **Tratador de exceção RequiredBLE.** Este tratador tem uma responsabilidade semelhante à da classe `ProvidedBLE_Handler`. Porém, o seu papel é tratar as exceções lançadas pela classe `RequiredFacade` do componente.

No caso de componentes reutilizados onde as classes de implementação não possam ser modificadas, a abordagem propõe que as classes de fachada (`ProvidedFacade` e `RequiredFacade`) desempenhem também os papéis de interceptadores. Dessa forma, são atribuídas a elas duas funcionalidades adicionais, muito importantes do ponto de vista do comportamento excepcional do sistema: (i) verificar as possibilidades de violações das assertivas especificadas (pré- e pós-condições); e (ii) adaptar os tipos das exceções lançadas pela implementação do componente à hierarquia comum de tipos abstratos de exceções, mostrada na Seção 2.4.1. Nesse processo de adaptação, as exceções lançadas que não sejam dos tipos declarados na especificação das suas interfaces são convertidas para um subtipo de `UndeclaredException`. Finalmente, tratadores de fronteira (BLE) são associados aos interceptadores, com o intuito de tratar as exceções descobertas por eles.

A Figura 2.11 mostra a estruturação do componente reutilizado após a aplicação da abordagem intra-componente. Como pode ser visto, o interceptador de fachada é estruturado como um invólucro[48], que altera o comportamento visível externamente do componente através da manipulação das mensagens, sem interferir na implementação do componente em si.

2.4.3 Abordagem Inter-Componentes

A abordagem inter-componentes lida com aspectos relativos à integração de componentes pré-existentes para formar uma nova configuração. Por essa razão, essa abordagem de tratamento se baseia nos conectores arquiteturais e os tratadores inter-componentes são denominados tratadores no nível de conectores (CLE)[49]. A Figura 2.12 mostra a estru-

[48] do inglês *Wrapper*
[49] do inglês *Connector-level exeption handlers*

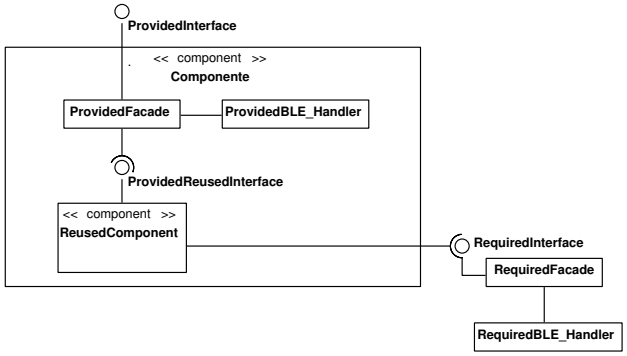

Figura 2.11: Estruturação Interna do Componente com Reuso

turação interna de um conector arquitetural e como ele se relaciona com os componentes cliente e servidor.

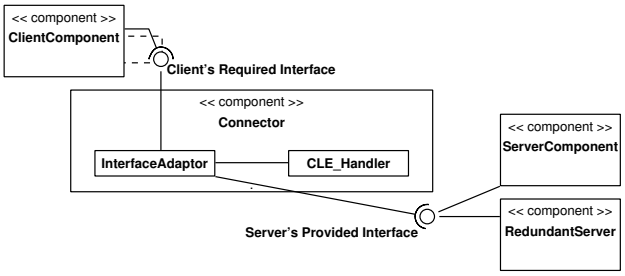

Figura 2.12: Estruturação de um Tratador CLE

As responsabilidades de um tratador CLE são duas: (i) oferecer as ações de mascaramento e correções de erros no nível arquitetural do sistema; e (ii) resolver incompatibilidades semânticas entre os componentes envolvidos na ligação, mais precisamente entre os componentes servidores e seus clientes associados. Essas incompatibilidades semânticas dizem respeito às conversões de tipos de exceções entre esses dois componentes. Dessa forma, quando forem integrados componentes com modelos de falha distintos, o tratador CLE deve encapsular a exceção proveniente do componente servidor. O tipo de uma exceção lançada pelo tratador CLE deve ser preferencialmente um dos tipos de exceções declaradas pelo componente cliente, isto é, um subtipo de `DeclaredException`.

As diretrizes apresentadas na Tabela 2.4 podem ser utilizadas para auxiliar nas tomadas de decisão relativas ao mapeamento dos tipos de exceções entre um componente servidor e um componente cliente [dCG04].

Tabela 2.4: Diretrizes para o Mapeamento entre Exceções de Componentes Distintos

Exceção lançada no Servidor	Exceção Propagada para o Cliente
Uma exceção `E1`, cujo tipo está especificado tanto na interface provida pelo servidor, quanto na requerida pelo cliente.	A própria exceção do tipo `E1` - propagação direta.
Uma exceção do tipo `E1`, declarado na especificação da interface provida pelo servidor. Além disso, existe um tipo `E2` com semântica equivalente, declarado na especificação da interface requerida pelo cliente. Um exemplo dessa equivalência é quando `E2` é um supertipo de `E1`.	`E2`.
Uma exceção do tipo `E1`, declarado na especificação da interface provida pelo servidor. Além disso, não existe nenhum tipo com semântica equivalente, declarado na especificação da interface requerida pelo cliente.	Um subtipo de um dos tipos abstratos `RejectedRequestException`, `RecoveredFailureException` ou `UnrecoveredFailureException`, dependendo da semântica de defeito especificada para o tipo `E1`.
Uma exceção do tipo `E1`, que é um subtipo de `UndeclaredException`.	A própria exceção do tipo `E1` - propagação direta.
Uma exceção de um tipo não declarado na especificação da interface provida pelo servidor e que não seja um subtipo de `UndeclaredException`.	Um subtipo de `UnrecoveredFailureException`.

2.5 Uma Arquitetura Confiável Baseada em Tratamento de Exceções

A arquitetura apresentada nesta seção é uma arquitetura genérica que auxilia a modelagem do tratamento de exceções. A Figura 2.13 apresenta os quatro componentes que a compõem e os seus relacionamentos. As responsabilidades de cada um desses componentes arquiteturais foram classificadas em dois tipos:

1. **Dependentes da aplicação (DA).** Essas responsabilidades são relacionadas às funcionalidades que normalmente são específicas para cada aplicação, como por exemplo o lançamento de exceções. Um outro exemplo poderia ser a especificação de colaborações, que depende dos requisitos do sistema. Para maiores detalhes sobre colaborações, consulte a Seção 2.3.5.

2. **Independentes da aplicação (IA).** Essas responsabilidades incluem funcionalidades básicas do gerenciamento de exceções e por isso utilizadas em qualquer sistema. Por exemplo, facilidades para gerenciamento de informações contextuais das exceções e o desvio do fluxo de controle no momento do lançamento de uma exceção.

A seguir, cada um dos componentes arquiteturais presentes na Figura 2.13 e suas respectivas interfaces são apresentados brevemente. Adiante, nas Seções 2.5.1 a 2.5.4, eles são detalhados.

- **Exception.** Este componente possui duas responsabilidades principais: (i) modelagem e instanciação das classes de exceções, sejam elas locais ou cooperativas (DA); e (ii) gerenciamento das informações contextuais das exceções (IA).

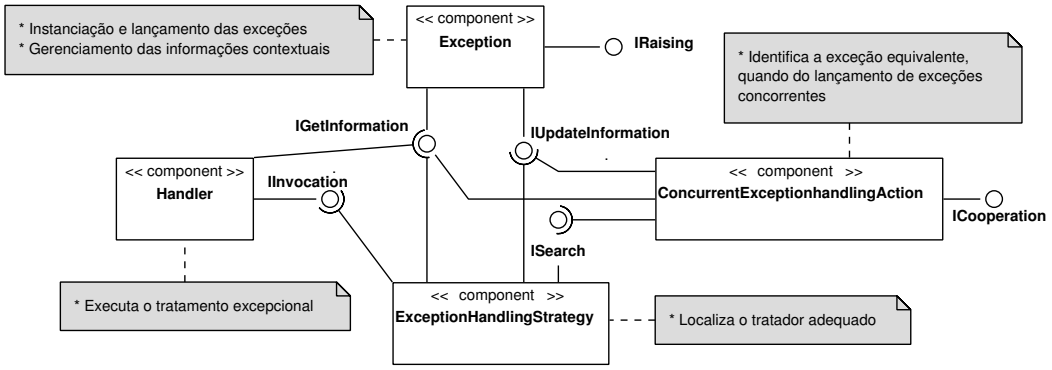

Figura 2.13: Arquitetura de Componentes Genérica para Tratamento de Exceções

- **Handler.** As principais responsabilidades deste componente são: (i) modelagem dos tratadores excepcionais (DA); e (ii) invocação dos tratadores (IA).

- **Exception Handling Strategy.** Devido ao papel genérico que este componente desempenha, as suas responsabilidades são todas independentes da aplicação (IA). São elas: (i) busca por tratadores; e (ii) desvio do fluxo de controle após o lançamento de uma exceção.

- **Concurrent Exception Handling Action.** Este componente trata dos aspectos complementares necessários para o tratamento de exceções concorrentes. As suas principais responsabilidades são: (i) especificação das colaborações (DA); (ii) sincronização dos participantes; e (iii) resolução da exceção equivalente às exceções concorrentes (Seção 2.3.5). A especificação das colaborações consiste na identificação de quais os componentes serão participantes de cada colaboração; a sincronização dos componentes é o processo de garantia da permanência de todos os participantes, até que a colaboração termine; e por fim a identificação da exceção equivalente consiste na implementação de uma das técnicas existentes , como por exemplo o método do grafo de resolução explicado na Seção 2.3.5.

Como pode ser visto na Figura 2.13, o componente `Exception` implementa os serviços relacionados a um repositório de informações contextuais das exceções, tornando essas informações disponíveis para os outros componentes. Esses componentes podem interagir com o componente `Exception` tanto para obter as informações, quanto para atualizá-las. O componente `ExceptionHandlingStrategy` tem um papel central na arquitetura e interage com todos os outros: captura as informações contextuais de exceções (componente `Exception`), antes de selecionar o tratador adequado. Após encontrar o tratador, ele solicita a sua execução (componente `Handler`). Quando exceções concorrentes são lançadas

durante uma cooperação, o componente `ConcurrentExceptionHandlingAction` é acionado. Após a descoberta da exceção equivalente, o componente `ExceptionHandlingStrategy` entra mais uma vez em ação, afim de localizar os diferentes tratadores dos componentes participantes que devem ser executados.

2.5.1 Componente Exception

Para aumentar o poder de modelagem de sistemas confiáveis, os desenvolvedores de sistemas tolerantes a falhas baseados em tratamento de exceções devem estar aptos a especificar tanto exceções simples, quanto compostas (Seção 2.3.5). Qualquer uma dessas exceções podem ser lançadas pelos seus componentes normais em tempo de execução. Esses componentes normais devem armazenar as informações contextuais dessas exceções, como por exemplo o local do seu lançamento, uma vez que elas são úteis para a escolha do tratador mais adequado.

Para o componente `Exception`, as exceções devem ser representadas através de classes que encapsulam suas próprias informações contextuais. Além disso, esse componente implementa três interfaces: (i) `IUpdateInformation`, responsável pela atualização das informações contextuais das exceções; (ii) `IGetInformation`, que disponibiliza as operações para a consulta das informações contextuais; e (iii) `IRaising`, que contém as operações de instanciação e lançamento da exceção. Essas três interfaces providas pelo componente `Exception` são públicas, isto é, disponíveis para todos os componentes do sistema.

2.5.2 Componente Handler

Outra necessidade dos desenvolvedores que utilizam os mecanismos de tratamento de exceções para implementar técnicas de tolerância a falhas é a implementação de tratadores excepcionais. Esses tratadores podem se referir tanto a exceções locais, quanto a exceções tratadas de maneira colaborativa. Além de possibilitar a definição dos tratadores, a infraestrutura da arquitetura de software deve oferecer uma separação de interesse entre os tratadores, que implementam o comportamento excepcional do sistema, e os componentes normais, que implementam as funcionalidades especificadas.

Dessa forma, o fato da arquitetura possuir um componente com o papel específico de tratar exceções possibilita uma separação explícita entre os comportamentos normal e excepcional do sistema. O componente `Handler` implementa a interface `IInvocation`. O papel dessa interface é possibilitar a execução dos tratadores excepcionais por parte do componente `ExceptionHandlingStrategy`. Dessa forma, essa interface possui visibilidade pública, onde cada um dos tratadores excepcionais especificados é materializado como uma operação provida.

Os principais benefícios decorrentes da utilização do componente `Handler` são:

- **Expressividade.** Os tratadores das exceções podem ser tanto locais, quanto arquiteturais, podendo envolver mais de um componente do sistema como parte de um tratamento coordenado mais elaborado.

- **Legibilidade e Manutenibilidade.** Esta abordagem proporciona uma separação explícita entre as atividades normais do sistema e as relativas à recuperação de erros.

- **Reusabilidade.** A adoção de componentes normais e excepcionais possibilitam a visualização dos dois comportamentos de maneira ortogonal entre si. Essa separação de interesse, além de possibilitar a reutilização de tratadores entre os componentes de um mesmo sistema, ela torna os componentes normais mais propícios a serem reutilizados.

2.5.3 Componente ExceptionHandlingStrategy

O papel do componente `ExceptionHandlingStrategy` é desviar o fluxo de execução do componente no momento em que uma exceção é lançada. Esse fluxo deve ser direcionado para o tratador mais adequado para a exceção e o contexto em questão. Dessa forma, o papel exercido por esse componente se mostra essencial no contexto de qualquer sistema que envolva tratamento excepcional, seja ele crítico ou não.

A identificação do tratador ideal em um fluxo excepcional é uma tarefa recursiva, que inicia no local de lançamento da exceção e segue propagando para o cliente imediato do serviço solicitado, isto é, para o componente que solicitou o serviço, seguindo recursivamente. Essa busca só é finalizada quando o componente `ExcepcionHandlingStrategy` localiza um tratador adequado para a exceção, ou quando ele atinge o final da pilha de requisições (cliente iniciador da chamada). No primeiro caso, o tratador excepcional deve ser executado. No segundo, a exceção provoca a parada repentina do sistema sem que haja a execução de um tratamento prévio. Uma boa prática de programação é evitar a existência de exceções sem tratadores associados, principalmente no contexto de sistemas críticos, onde até mesmo o processo de parada deve ser controlado.

O componente `ExceptionHandlingStrategy` implementa a interface pública `ISearch`. Esta interface possibilita que o componente `ConcurrentExceptionHandlingAction` execute o serviço de busca para obter e executar o tratador de cada participante da colaboração, dados um tipo de exceção e as suas informações de contexto. A principal informação contextual levada em consideração nesse momento é o ponto da execução onde a exceção foi lançada.

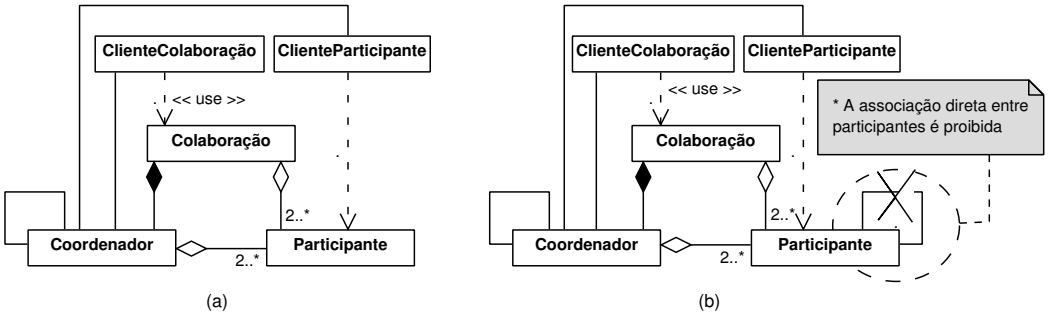

Figura 2.14: Meta-modelo da Estruturação de uma Colaboração

2.5.4 Componente ConcurrentExceptionHandlingAction

Ao desenvolver sistemas reais baseados em componentes concorrentes, percebe-se que a possibilidade de ocorrência de mais de uma exceção concomitantemente deve ser considerada. Essas exceções devem ser tratadas de maneira colaborativa, sendo necessário para isso a implementação de colaborações e de funções de resolução, como visto na Seção 2.3.5. O componente `ConcurrentExceptionHandlingAction` implementa uma única interface pública (`ICooperation`), que oferece as operações necessárias para a implementação dessas duas funcionalidades.

A Figura 2.14 (a) mostra o meta-modelo definido para a estruturação das colaborações. Cada colaboração é formada por um coordenador, que possui associado a ele os componentes participantes. O papel do participante, por sua vez, pode ser desempenhado tanto por componentes normais e excepcionais, quanto por outras cooperações recursivamente. Uma cooperação é representada pelo seu coordenador.

Para facilitar a implementação das colaborações de um sistema, foram adotadas algumas restrições de comunicação entre os componentes envolvidos. Todas as mensagens recebidas pelos participantes devem ser intermediadas pelo respectivo coordenador. Por exemplo, a associação em destaque na Figura 2.14 (b) não é permitida. Além disso, o componente coordenador deve necessariamente ser a única porta de acesso entre o meio externo e os participantes da uma colaboração. Esse ponto central de acesso visa facilitar a implementação, de modo a evitar a propagação de erros para componentes externos à colaboração. A propagação de erros para o meio externo inviabiliza atividades de recuperação por retrocesso, uma vez que o controle do estado do sistema seria perdido.

Os principais benefícios da utilização do componente `ConcurrentExceptionHandling-Action` são:

- **Uniformidade.** Apesar das suas características próprias, a estratégia de tratamento de exceções concorrentes é uma extensão consistente das estratégias convencionais existentes.

- **Simplicidade.** O fato de uma colaboração ser controlada de maneira centralizada (um único coordenador) proporciona uma maior facilidade de entendimento e implementação. Apesar da simplicidade proporcionada, a centralização é uma vulnerabilidade em termos de confiabilidade, uma vez que representa um ponto único de falhas. Uma maneira de amenizar essa questão é a utilização de coordenadores redundantes. Porém, há uma relação de compromisso estreita entre o número de componentes redundantes e o custo final do sistema. Além disso, o envio de mensagens para os participantes da colaboração deve necessariamente ser intermediado pelo coordenador, o que proporciona uma implementação simples para garantir o confinamento do erro. Porém, dependendo da natureza da aplicação, isso pode se tornar um "gargalo" e comprometer o desempenho.

- **Controle da complexidade.** A complexidade geral do sistema pode ser melhor controlada com a utilização de colaborações aninhadas[50], isto é, coordenadores que são participantes de outras colaborações.

2.6 Resumo

Este capítulo apresentou os conceitos necessários para o entendimento do método MDCE+, bem como dos seus objetivos e aplicabilidade. Entre os fundamentos de DBC apresentados na Seção 2.1, foi possível ver quais as características necessárias para que um processo de desenvolvimento possa se adequar ao desenvolvimento de sistemas baseados em componentes. Além disso, na Seção 2.2, foi ressaltada a importância da arquitetura de software para o desenvolvimento de sistemas com requisitos de qualidade críticos. Na Seção 2.2.2 foi apresentado o modelo COSMOS, que proporciona um mapeamento das estruturas da arquitetura de software para o código, facilitando o entendimento e a execução das atividades de manutenção. O modelo ATAM, apresentado na Seção 2.2.3, é utilizado pelo método MDCE+ durante o projeto da arquitetura do sistema. A Seção 2.3 apresentou os conceitos básicos de tolerância a falhas e tratamento de exceções. O objetivo de tê-los apresentado como fundamentos teóricos é proporcionar uma melhor compreensão do problema, antes de mostrar a proposta de solução, que é o método MDCE+. A Seção 2.4 apresenta uma abordagem para a estruturação do comportamento excepcional no DBC. Essa estruturação permite inclusive a utilização de componentes COTS para o desenvolvimento de sistemas confiáveis, através do conceito de adaptadores. Finalmente, a Seção 2.5 apresentou uma divisão dos papéis necessários para a estruturação de sistemas confiáveis. Essa divisão de papéis foi apresentada na forma de uma arquitetura, cujos componentes

[50] do inglês *Nested collaborations*

são materializados pelo método MDCE+. As atividades que detalham essa materialização são mostradas no Capítulo 3.

Capítulo 3

O Método MDCE+

Este capítulo apresenta o método MDCE+, suas principais características e atividades. O objetivo principal do MDCE+ é estender o método MDCE, apresentada na Seção 2.3.8, para sistematizar a modelagem e a implementação do comportamento excepcional no desenvolvimento de sistemas baseados em componentes. O foco desse refinamento se concentrou basicamente nas fases de projeto arquitetural e implementação. A ênfase na arquitetura de software possibilitou uma melhor análise dos fluxos de exceções que fluem entre os componentes arquiteturais do sistema. Essa análise melhorada permite a construção de tratadores mais eficientes, além de antecipar a correção de possíveis falhas de especificação.

Este capítulo não trata detalhes das atividades relativas ao comportamento normal. Uma versão bem mais detalhada que abrange a especificação dos comportamentos normal e excepcional está disponível como um relatório técnico do Instituto de Computação da Unicamp [dSBR05]. Se for do interesse do leitor, a leitura desse capítulo pode ser substituída pela leitura dessa versão mais detalhada. Porém, vale a pena salientar que os detalhes adicionais dizem respeito principalmente às atividades relativas à especificação do comportamento normal.

3.1 Visão Geral do Método

Existem basicamente duas abordagens para a construção de sistemas confiáveis: (i) abordagem ascendente[1]; e (ii) abordagem descendente[2]. A abordagem ascendente, proposta por Avizienis [Avi97], busca compor sistemas confiáveis a partir da construção de componentes também confiáveis. Dessa forma, deve-se ter a preocupação de manter os requisitos

[1] do inglês *botton-up*
[2] do inglês *top-down*

de confiabilidade durante a integração dos componentes, onde as partes internas do sistema são interligadas. A abordagem descendente, por sua vez, possibilita a reutilização de componentes não-confiáveis na construção de sistemas confiáveis [dCGFPR04]. Dessa forma, no desenvolvimento *Top-Down*, a confiabilidade do sistema depende muito dos conectores arquiteturais. Essa dependência é justificada, já que o controle dos requisitos de tolerância a falhas se concentra nas interações entre os componentes arquiteturais, através da implementação de funções de monitoramento e tratamento das falhas.

O método MDCE+ visa sistematizar a modelagem e o desenvolvimento de sistemas confiáveis que sejam baseados em componentes, através do uso dos mecanismos de tratamento de exceções das linguagens de programação existentes. Diferentemente do método MDCE [Fer01], o método MDCE+ combina a utilização das duas abordagens, *Botton-Up* e *Top-Down*, mencionadas anteriormente, enfatizando a importância da arquitetura do sistema na especificação do comportamento excepcional. Com essa combinação de abordagens, o MDCE+ busca melhorar a eficiência do tratamento através de uma análise cuidadosa do fluxo de informações da arquitetura do sistema.

Como visto na Seção 2.2, os principais benefícios desse destaque dado à arquitetura de software são as possibilidades de detecção e tratamento de exceções envolvendo simultaneamente mais de um componente arquitetural do sistema. Com o aumento da complexidade dos sistemas de software modernos, a necessidade de se antecipar a especificação do comportamento interativo entre os seus componentes vem se tornando evidente [SDUV03, dLR99, Szy02]. Como conseqüência dessa antecipação, é possível analisar o sistema de forma abstrata e independente de linguagem de programação, ao contrário do que acontece nas fases de projeto e implementação. Dessa forma, a estruturação do comportamento excepcional fica facilitada e conseqüentemente mais clara e eficiente. Seguindo essa tendência, Souchon [SDUV03] propôs uma técnica para detecção de condições excepcionais dependentes do contexto interativo entre vários componentes do sistema. Nesse trabalho, ele considera o contexto de um sistema multi-agentes, onde uma condição excepcional pode ser vista a partir de um determinado percentual de componentes falhos, isto é, dado o número de componentes distribuídos, a falha isolada de um componente pode ser ignorada pelo sistema, caso o percentual de segurança que foi especificado para os servidores redundantes continue funcionando.

Além disso, o método MDCE+ possibilita a antecipação da descoberta de falhas de projeto a partir da análise do fluxo de exceções dos componentes e dos seus respectivos tratadores, que é uma conseqüência do fato dele considerar a arquitetura desde as fases iniciais do desenvolvimento. Essas falhas podem ser detectadas automaticamente, através do uso de ferramentas de análise, tais como o *framework* Aereal [FdSBR05a]. A partir da definição do fluxo excepcional dos componentes arquiteturais, esse *framework* analisa a existência de erros de especificação, como por exemplo, situações onde uma exceção possa

ser lançada e não tratada, ou vice-versa. Dessa forma, essa análise automática antecipa a detecção e a conseqüente correção das falhas para a fase de especificação, o que reduz o custo final do sistema [Pre01].

Em relação ao tratamento de exceções na arquitetura, que envolve mais de um componente do sistema, existem basicamente dois tipos: (i) **tratamento que envolve reconfiguração** [IB01], que consiste na reconfiguração dos componentes arquiteturais para isolar um ou mais componentes falhos e substituí-los, em tempo de execução, por outros componentes que ofereçam serviços equivalentes; e (ii) **tratamento colaborativo coordenado** [dLR99], que é caracterizado pelo envolvimento de mais de um componente do sistema, executando o tratamento concorrentemente e de maneira colaborativa. Em outras palavras, esse tratamento é uma utilização prática do conceito de ações atômicas coordenadas [XRR+99], descrito na Seção 2.3.5. As principais características de um grupo de execução colaborativa, ou simplesmente uma **colaboração** são: (i) a garantia de consistência entre os seus membros, isto é, todos os membros de uma colaboração devem aprovar o resultado final para que a execução possa ser considerada bem sucedida; e (ii) as exceções lançadas concorrentemente por mais de um membro de uma colaboração devem ser convertidas em uma exceção equivalente, que seja compreendida e tratada de forma colaborativa pelos integrantes da colaboração.

Porém, todos esses requisitos modernos aumentam consideravelmente a complexidade dos sistemas e dificultam o seu desenvolvimento. Esse aumento da complexidade, aliado à dependência cada vez maior da sociedade em relação à automação, evidencia a necessidade de se utilizar técnicas e mecanismos que aliem o aumento da confiabilidade e o gerenciamento sistemático da complexidade do software. Uma forma bastante conhecida para se alcançar esse aumento da confiabilidade é considerar o comportamento excepcional dos sistemas, utilizando de forma adequada os mecanismos de tratamento de exceções das linguagens de programação.

Apesar da popularidade dos mecanismos de tratamento de exceções, o projeto e a implementação do comportamento excepcional de um sistema são tarefas muito complexas que não recebem a atenção devida de metodologias de desenvolvimento existentes [dLR99, RdLeFCF04]. A situação é ainda mais crítica se forem levados em consideração os métodos para desenvolvimento baseado em componentes. Conseqüentemente, os projetistas e desenvolvedores de software sentem dificuldade tanto para utilizar corretamente os mecanismos de tratamento de exceções, quanto para projetar o comportamento excepcional, o que compromete a confiabilidade dos sistemas construídos [RS03]. Um outro fator que dificulta ainda mais a qualidade do tratamento de exceções é o fato dele ser considerado apenas na fase de projeto do sistema ou até mesmo somente durante a sua implementação. A principal conseqüência disso é a baixa qualidade dos artefatos de projeto, acarretando o aumento da complexidade das estruturas de exceções e uma menor

eficiência dos tratadores.

O objetivo principal do método MDCE+ é auxiliar na construção de sistemas de software com algum requisito de confiabilidade ligado à tolerância a falhas. Dessa forma, o MDCE+ deve estruturar a identificação de exceções e a especificação dos seus tratadores durante as fases de análise, projeto e implementação do software. Outra característica importante desse método é o foco dado às informações contextuais dos fluxos interativos entre os componentes do sistema. Aproveitando esse enfoque arquitetural, o método deve guiar o desenvolvedor na especificação de tratadores de exceções da arquitetura, a partir de uma análise responsável das propagações excepcionais.

De acordo com o que foi apresentado na Seção 2.3.2, as dimensões da confiança no funcionamento[3] atendidas pelo MDCE+ são: **confiabilidade**[4], através de uma abordagem sistemática de desenvolvimento, que também é conhecido como projeto rigoroso[5] [XRR+99]; e **disponibilidade**, através da especificação de tratadores estruturados que mascaram falhas, evitando a parada do serviço. Isso será alcançado através da implementação de mecanismos de tolerância a falhas, que serão estruturados utilizando o mecanismo de tratamento de exceções da linguagens de programação.

Conforme ilustrado na Figura 3.1, o método MDCE+ distribui suas atividades em sete fases: (i) especificação e análise dos requisitos; (ii) definição dos aspectos gerenciais do sistema; (iii) projeto arquitetural; (iv) análise do sistema; (v) projeto do sistema; (vi) materialização dos componentes; e (vii) integração dos componentes do sistema. Essas fases constituem o ciclo de desenvolvimento, atuando desde a especificação dos requisitos até a construção final do software. O caráter genérico dessas fases, no sentido de estarem presentes em um ciclo de desenvolvimento clássico (Seção 2.1.1), facilita a utilização conjunta do MDCE+ com processos de desenvolvimento de software existentes na literatura, como por exemplo o processo *UML Components*, ao qual o MDCE+ foi incorporado. Os detalhes dessa adaptação são mostrados no Capítulo 4.

A seguir, é feito um comparativo entre as fases de desenvolvimento de um processo tradicional, mostradas na Seção 2.1.1 e as fases do método MDCE+:

1. **Especificação de Requisitos.** No método MDCE+, essa fase é dividida em duas: (i) especificação e análise dos requisitos, onde são especificados os requisitos funcionais do sistema e são descobertas as exceções relacionadas à lógica do negócio; e (ii) definição dos aspectos gerenciais, onde são definidas algumas restrições ligadas ao desenvolvimento, tais como a linguagem de programação utilizada e a prioridade de entregas[6] do sistema. Além disso, a fase de definição dos aspectos gerenciais

[3] do inglês *Dependability*
[4] do inglês *Deliability*
[5] do inglês *Rigorous design*
[6] do inglês *Release*

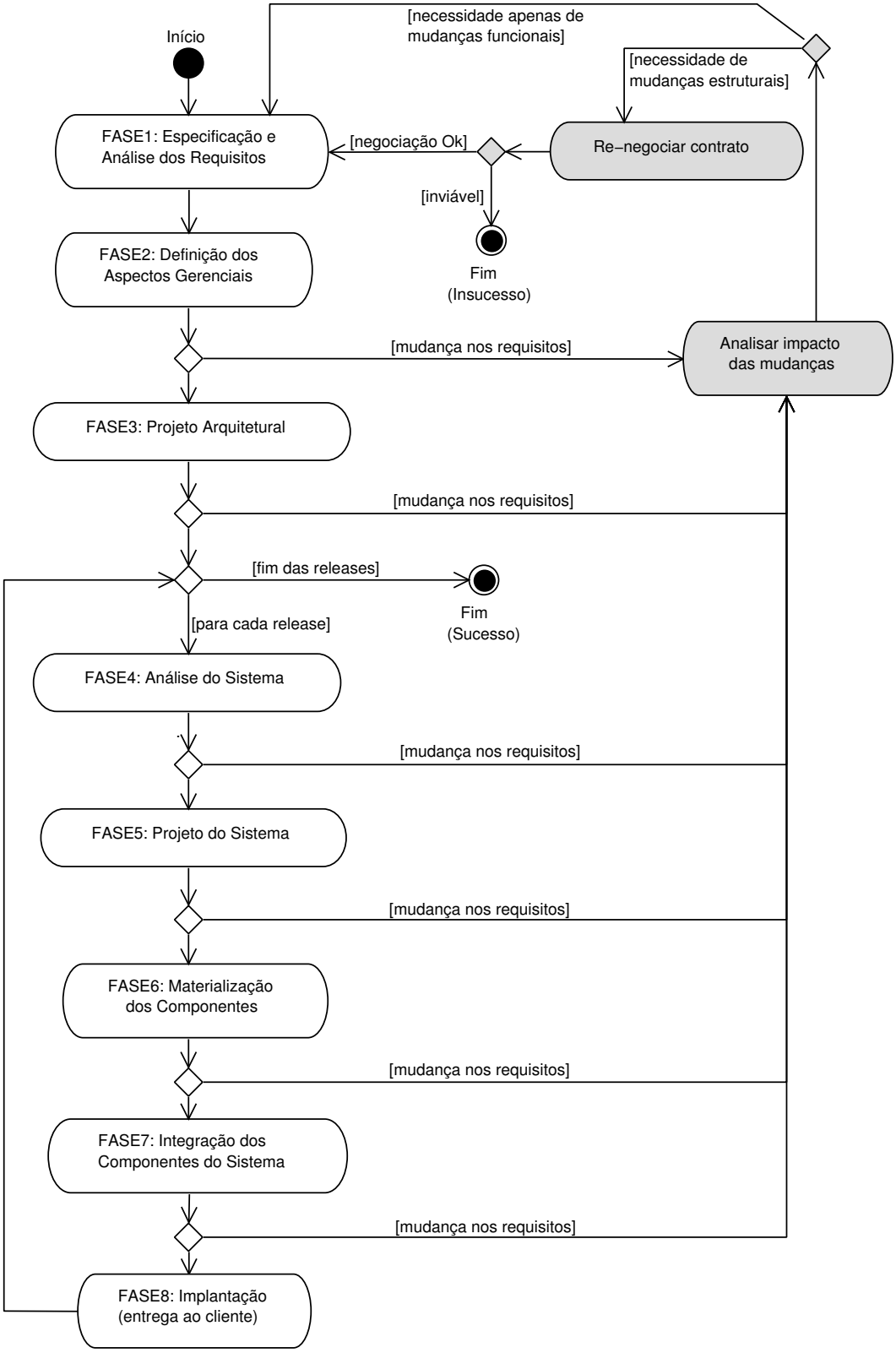

Figura 3.1: Fases do Método MDCE+

auxilia na identificação dos componentes críticos do sistema, que podem ser alvo de tratamento de exceções na arquitetura de software.

2. **Análise.** Assim como em um processo tradicional, na fase de análise são identificadas as principais entidades envolvidas na solução do problema, que no caso de um processo de desenvolvimento baseado em componentes, são os componentes. Por se tratar de um método de desenvolvimento centrado na arquitetura, cada um dos componentes identificados deve ser posicionado na arquitetura. Por esse motivo, no método MDCE+ a fase de projeto arquitetural deve ser executada antes da fase de análise (Figura 3.1).

3. **Projeto Arquitetural.** Como discutido anteriormente, apesar de ser uma fase comum a um processo de desenvolvimento tradicional, no método MDCE+ a fase de projeto arquitetural é executada antes da fase de análise. Apesar dessa antecipação representar uma mudança em relação aos processos de desenvolvimento tradicionais, essa alteração não deve dificultar a adaptação do método MDCE+ aos processos de desenvolvimento existentes. Isso acontece porque o MDCE+ propõe a escolha da arquitetura a partir dos requisitos e restrições especificados, em especial os requisitos não-funcionais, que são especificados no decorrer do projeto da arquitetura do sistema.

4. **Projeto.** Assim como nos processos tradicionais, o objetivo da fase de projeto é refinar o modelo da análise de modo a torná-lo mais próximo às particularidades das tecnologias adotadas.

5. **Implementação.** Por se tratar de um método voltado para o desenvolvimento de sistemas baseados em componentes, o MDCE+ divide a fase de implementação em duas: (i) materialização dos componentes, onde os componentes são reutilizados ou implementados isoladamente; e (ii) integração dos componentes do sistema, onde os componentes isolados são integrados para constituir o sistema.

6. **Testes.** Apesar desta fase não ser detalhada pelo método MDCE+, esse método foi integrado a um método de testes para o comportamento excepcional de componentes [dSBRMR05].

7. **Implantação (entrega ao cliente).** Apesar do procedimento de execução dessa fase não ser detalhado pelo método MDCE+, esse método sugere uma diretriz importante relacionada a ele. Como pode ser visto na Figura 3.1, o método MDCE+ propõe um desenvolvimento iterativo e incremental, baseado em entregas sucessivas de versões do sistema.

8. **Manutenção.** Esta fase também não é detalhada pelo método MDCE+, porém espera-se que ela seja facilitada. Devido à visão abstrata oferecida pela arquitetura e à separação de interesse explícita entre os comportamentos normal e excepcional, a complexidade dos sistemas deve ser controlada mais facilmente.

As seções seguintes detalham cada fase do método MDCE+, descrevendo suas características gerais relativas tanto à especificação do comportamento normal, quanto à especificação do comportamento excepcional dos sistemas de software baseados em componentes.

3.2 FASE 1: Especificação e Análise dos Requisitos

A especificação de requisitos é a primeira fase a ser executada durante o desenvolvimento de um software. Em geral, os requisitos podem ser vistos como os objetivos esperados para o sistema, assim como as condições e capacidades necessárias para tal. Por se tratar de um método voltado para o desenvolvimento de sistemas confiáveis, o MDCE+ oferece já nessa fase, meios de especificar as possíveis situações excepcionais previstas, incluindo as suas condições de ativação (condições excepcionais). Nas outras fases do desenvolvimento, essas situações e condições excepcionais darão origem às exceções e suas respectivas condições de lançamento.

Como visto na Seção 2.1.1, de acordo com as suas características, um requisito pode ser classificado como **funcional** ou **não-funcional** [Pre01]. Os requisitos funcionais podem ser mapeados para os serviços, tarefas ou funções que o sistema deve oferecer. Já os requisitos não-funcionais, apesar de não representarem funcionalidades diretamente, eles podem interferir na maneira como o sistema deve executá-las. Dessa forma, a organização estrutural do sistema, materializada na arquitetura de software (Seção 2.2), é diretamente relacionada a esses requisitos. Em todo este trabalho, os termos **requisito de qualidade** e **requisito não-funcional** são utilizados como sinônimos.

Conforme mostrado na Figura 3.2, para a especificação das funcionalidades do sistema, a fase de especificação de requisitos abrange cinco atividades: (i) compreender o problema através de descrições textuais; (ii) analisar e representar o domínio do problema; (iii) definir o escopo do sistema; (iv) especificar os requisitos funcionais; e (v) especificar o comportamento excepcional. De acordo com o perfil recomendado para a execução dessas tarefas, o MDCE+ sugere a necessidade de três papéis bem definidos: (i) cliente; (ii) analista de domínio; e (iii) engenheiro de requisitos. As próximas sub-seções descrevem brevemente cada uma das atividades dessa fase. Existe disponível uma versão mais detalhada do método MDCE+ [dSBR05], na forma de um relatório técnico do Instituto de Computação da Unicamp.

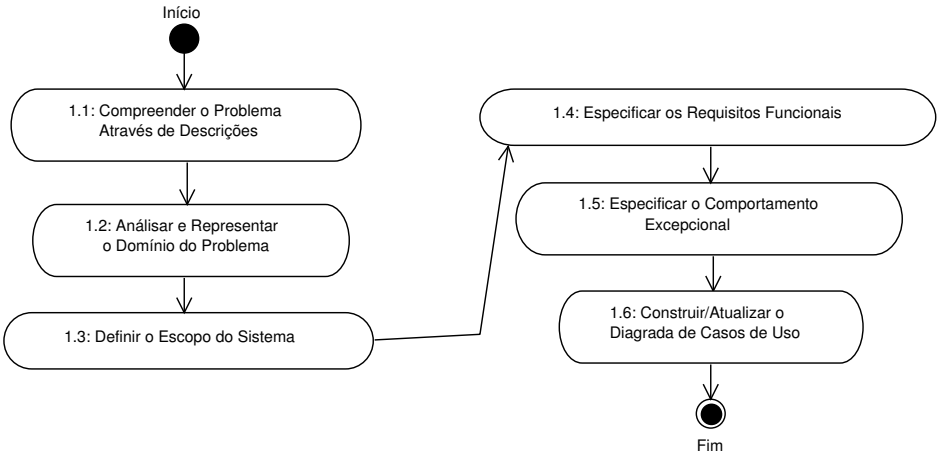

Figura 3.2: Fase de Especificação de Requisitos

3.2.1 Compreender o Problema Através de Descrições (ativ. 1.1)

Antes de mais nada, se faz necessário conhecer o problema que se pretende resolver. Esse esclarecimento prévio visa evitar diferentes interpretações que possam ocorrer naturalmente devido às ambigüidades inerentes às linguagens falada e escrita [CdSC01, Pre01]. Dessa forma, o método MDCE+ busca reduzir o re-trabalho e mudanças tardias de requisitos, além de aumentar as chances de aceitação e conseqüente sucesso do sistema. Na versão detalhada do método MDCE+ [dSBR05] são apresentadas duas formas básicas de proporcionar essa compreensão refinada: (i) especificação formal dos requisitos; e (ii) interação extrema entre os interessados do sistema.

3.2.2 Analisar e Representar o Domínio do Problema (ativ. 1.2)

A identificação do domínio de um sistema é uma atividade importante, que auxilia durante todo o desenvolvimento do software, ajudando inclusive a aumentar a qualidade final do produto [FH95]. O objetivo das técnicas de análise de domínio é identificar as características da lógica do negócio, que sejam comuns a um grupo de sistemas. Por esse motivo, as entidades e funcionalidades identificadas a partir dessa análise são consideradas básicas, uma vez que são mais propícias a serem reutilizadas posteriormente. Os principais aspectos de qualidade do sistema que podem ser melhorados com o conhecimento do domínio são: (i) facilidade de gerenciamento; (ii) maior índice de sucesso e maior confiabilidade ; (iii) facilidade de evolução ; e (iv) maior grau de reutilização.

Mais detalhes sobre a melhoria dos aspectos de qualidade e sobre os procedimentos de identificação, análise e representação do domínio podem ser vistos na versão detalhada do MDCE+ [dSBR05].

Identificar as Entidades Críticas

Tendo em vista o objetivo de aumentar a confiabilidade do sistema, após a representação do modelo do negócio se faz necessário classificar as entidades de acordo com a sua relevância para o domínio. Mais adiante, durante o projeto e a implementação do sistema (Seções 3.6.1 e 3.7), essa identificação das entidades críticas servirá de base para que o gerente e o projetista restrinjam o número de componentes redundantes para a implementação de técnicas de mascaramento de falhas. Essa restrição é necessária, uma vez que normalmente é inviável dos pontos de vista financeiro e estratégico (custo x benefício), que todo o sistema seja alvo desse esforço adicional.

Por se tratar da classificação da criticidade das entidades principais do domínio, essa atividade deve se basear nos documentos de compreensão do domínio, que são artefatos produzidos na primeira atividade da especificação de requisitos (Seção 3.2.1). Esses documentos apresentam uma lista das principais características do negócio. O método MDCE+ sugere dois critérios básicos para auxiliar na tomada de decisão relacionada à seleção de uma entidade crítica:

- Essa entidade se relaciona (direta ou indiretamente) com a maioria das entidades do domínio? Normalmente, entidades com um grande número de relacionamentos são meios de comunicação entre segmentações (áreas) do domínio. Essas entidades podem ser consideradas críticas, desde que a sua possível parada possa comprometer a comunicação entre partes importantes do domínio.

- Essa entidade é um ponto de interação com o meio externo (outros domínios)? Normalmente, as entidades que desempenham um papel de fronteira entre domínios distintos é considerada crítica, desde que haja a possibilidade de interferência direta do meio externo no domínio através dela.

3.2.3 Definir o Escopo do Sistema (ativ. 1.3)

Após conhecer o domínio no qual o sistema se insere, é necessário explicitar qual é o seu papel específico, em outras palavras, é necessário restringir os limites do sistema a fim de deixar claro o que exatamente se espera dele. Por esse motivo, a definição do escopo do sistema deve ser executada em conjunto, tanto pelo engenheiro de requisitos, quanto pelo cliente e pelo analista do domínio.

O método MDCE+ sugere uma sistemática baseada nas definições textuais produzidas durante a aquisição do conhecimento do domínio, já realizada anteriormente (Seção 3.2.1). Nesse método, o escopo do sistema é definido através de três atividades: (i) identificar as funcionalidades do domínio; (ii) determinar as responsabilidades do sistema; e (iii) associar

o texto descritivo da funcionalidade. O documento detalhado do método [dSBR05] explica cada uma dessas atividades.

O artefato final produzido é uma lista das funcionalidades esperadas para o sistema, juntamente com os respectivos textos de definição do domínio de onde as funcionalidades foram identificadas. Após essa definição do papel real do sistema, já se pode iniciar a especificação das suas funcionalidades. Essa atividade é mostrada na próxima seção (Seção 3.2.4).

3.2.4 Especificar os Requisitos Funcionais (ativ. 1.4)

Com o escopo do sistema bem definido, o próximo passo é o detalhamento das funcionalidades identificadas. O método MDCE+ propõe a representação desses requisitos funcionais através de casos de uso UML, que são especificados segundo o *workflow* apresentado na Figura 3.3. Dessa forma, os casos de uso são especificados a partir do documento de definição dos limites do sistema. Essa especificação segue o padrão sugerido pelo método MDCE (Seção 2.3.8). A seguir, as especificações dos casos de uso serão complementadas com a especificação do comportamento excepcional do sistema.

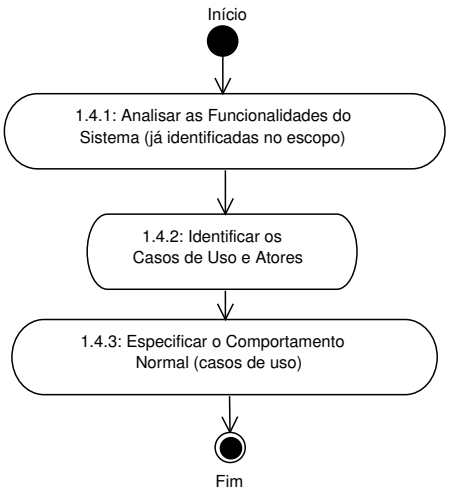

Figura 3.3: Atividades da Especificação dos Requisitos Funcionais

O detalhamento das três atividades apresentadas na Figura 3.3 está disponível na versão detalhada do método [dSBR05].

Para a especificação do comportamento normal de um caso de uso, o método MDCE+ sugere o mesmo modelo proposto pelo método MDCE (Tabela 3.1). Sendo assim, esse modelo deve guiar a especificação, uma vez que essa atividade consiste basicamente no preenchimento de cada um dos campos solicitados. O comportamento de um caso de

uso é descrito por uma seqüência de atividades (ou passos) denominada **cenário**. O **comportamento normal** de um caso de uso possui um cenário primário e zero ou mais cenários alternativos. O cenário primário representa o fluxo mais provável para execução da funcionalidade. Variações desse fluxo, como condições e iterações, são descritas como cenários alternativos. Dada a importância dos cenários para o modelo de casos de uso, o relatório técnico que detalha o método MDCE+ [dSBR05] mostra um *workflow* para orientar a especificação do cenário principal e dos cenários alternativos do caso de uso.

Tabela 3.1: Especificação Normal de um Caso de Uso

Nome do Caso de Uso	
Descrição:	Breve descrição do comportamento do caso de uso.
Participantes:	Lista dos atores participantes do caso de uso.
Pré-condições:	Conjunto de condições que devem ser satisfeitas antes que o serviço inicie sua execução (início do caso de uso).
Invariantes:	Condições que devem ser mantidas durante toda a execução do serviço (início e fim do caso de uso).
Cenário primário:	Atividades executadas pelo caso de uso. 1- Atividade 1; 2- Atividade 2.
Cenários alternativos:	Atividades opcionais que podem ser executadas pelo caso de uso. 1.1- Atividade executada após atividade 1; 1.2- Atividade executada após atividade 1.1;
Pós-condições:	Condições que devem ser satisfeitas após a execução do serviço (final do caso de uso).

3.2.5 Especificar o Comportamento Excepcional (ativ. 1.5)

Os passos de um cenário podem ser bem ou mal sucedidas. O comportamento normal de um caso de uso é a sua execução livre de erros, isto é, o sucesso na execução de todos os passos do cenário. De maneira complementar, o seu **comportamento excepcional** é representado por desvios do curso normal, decorrentes da necessidade de se tratar situações excepcionais ocorridas durante a execução de um dos passos de seus cenários. Sendo assim, além de descrever a funcionalidade do caso de uso, é necessário apurar todas as situações excepcionais que podem impedir o seu sucesso. Em seguida, devem ser especificadas as formas como tais situações devem ser tratadas.

Existem alguns trabalhos que enfatizam a importância de se considerar as situações excepcionais de um caso de uso no momento em que ele está sendo especificado [Coc03, SW98]. No entanto, essas abordagens descrevem o comportamento excepcional como uma extensão do fluxo normal do caso de uso, sem haver uma separação explícita entre eles. Não separar explicitamente o comportamento normal do excepcional torna os casos de uso muito complexos e conseqüentemente difíceis de serem entendidos, estendidos e mantidos.

O modelo de caso de uso proposto pelo método MDCE inclui a descrição completa de um caso de uso, isto é, a definição dos seus comportamentos normal e excepcional.

Além disso, mantém essas representações separadas, o que é de suma importância para manter a legibilidade do caso de uso, como será mostrado a seguir. No método MDCE+, a especificação do comportamento excepcional é composta de duas atividades: (i) identificar exceções; e (ii) especificar os cenários excepcionais.

Identificar Exceções (ativ 1.5.1)

Com os casos de uso já especificados, a identificação das exceções consiste na verificação de possíveis violações das assertivas definidas (pré-condições, pós-condições e invariantes). Como mostrado na Figura 3.4, essa verificação é composta de dois passos. No primeiro, são verificadas as pré-condições e as invariantes; no segundo, as invariantes são verificadas juntamente com as pós-condições. Dessa forma, fica claro que uma invariante deve ser satisfeita antes da execução e que deve permanecer válida após ela [JM97].

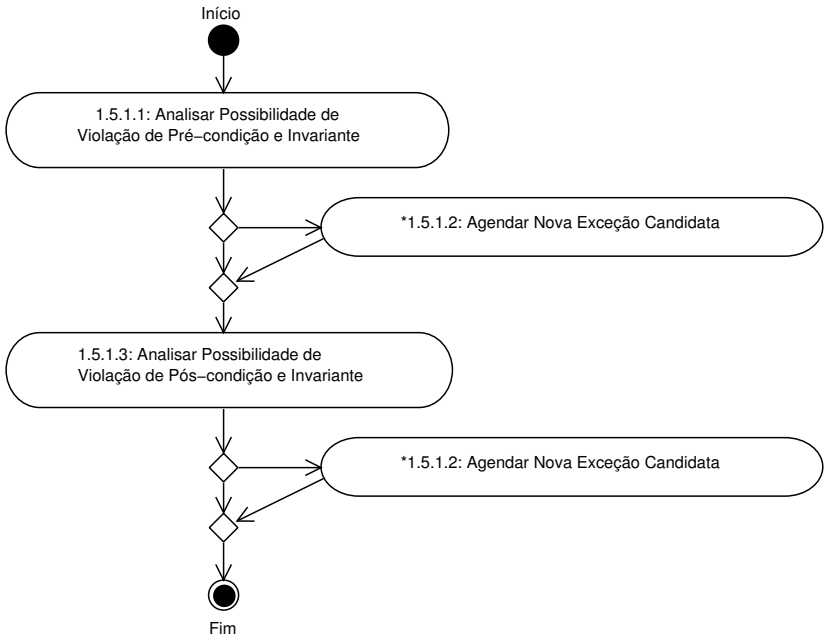

Figura 3.4: Verificação de Assertivas

Durante qualquer uma das etapas da verificação, a identificação de uma nova possibilidade de violação acarreta no agendamento de uma **exceção candidata**. Vale a pena salientar que essas violações ainda não são consideradas exceções; isso só acontecerá durante a fase de análise do sistema (Seção 3.5.1). A utilidade desse procedimento de agendamento se torna mais evidente se analisarmos o aspecto iterativo do processo de desenvolvimento. Em outras palavras, o processo prevê várias atividades de agendamento e especificação de exceções candidatas durante todas as fases do desenvolvimento

do software; porém, as exceções descobertas a partir da fase de análise só podem se tornar efetivamente exceções na iteração seguinte do desenvolvimento. Dessa forma, como mostrado na Seção 3.3, um dos critérios para a finalização do desenvolvimento é o fato de não haver exceções candidatas agendadas para especificação. Dada a importância dessa atividade nas diversas fases do processo, o agendamento de exceções candidatas é detalhado na Figura 3.5. Em seguida, cada uma das atividades é explicada brevemente.

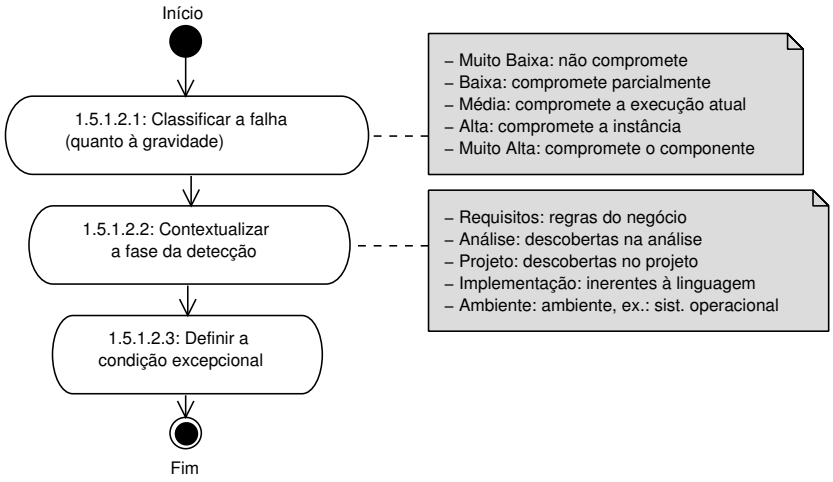

Figura 3.5: Agendamento de Exceções Candidatas

1. **Classificar a falha (quanto à gravidade) [Gil03].** De acordo com a sua gravidade, uma falha pode ser classificada em cinco níveis: (i) muito baixa, quando não compromete a execução; (ii) baixa, quando compromete parcialmente; (iii) média, quando compromete a execução atual, isto é, um *re-try* resolveria o problema; (iv) alta, quando compromete as execuções feitas à instância atual; e (v) muito alta, quando compromete as execuções feitas a qualquer instância do componente.

2. **Contextualizar a fase da detecção [dLR99].** Nessa atividade, ocorre a classificação das exceções, de acordo com a fase do desenvolvimento onde elas são detectadas: (i) requisitos, que se referem a violações das regras do negócio; (ii) análise, quando descobertas durante a identificação das entidades do sistema; (iii) projeto, descobertas durante o refinamento da especificação interna das entidades; (iv) implementação, exceções normalmente inerentes à linguagem de programação utilizada; e (v) ambiente, que normalmente são difíceis de serem previstas, uma vez que são lançadas pela infra-estrutura do sistema, tais como sistema operacional e plataforma de componentes.

3. **Definir a condição excepcional.** Devido à maior contextualização existente no momento da detecção da exceção, a condição excepcional deve ser fornecida durante o agendamento das exceções candidatas. Basicamente, o desenvolvedor deve descrever sucintamente o sinal de "disparo", isto é, a condição de lançamento da exceção. Normalmente, essa condição é ou a indicação da assertiva que foi violada, ou alguma informação que possa justificar o porquê do estado do sistema ter sido considerado errôneo, de modo a inviabilizar a execução do serviço.

Especificar os Cenários Excepcionais (ativ. 1.5.2)

Como já dito anteriormente, o comportamento de um sistema é representado através dos cenários dos casos de uso. Quando esses cenários representam um comportamento relativo ao tratamento de exceções, eles são denominados **cenários excepcionais**. Esses cenários constituem os desvios do comportamento normal do caso de uso quando situações excepcionais acontecem e representam as medidas de recuperação do sistema. Cenários excepcionais podem ser de dois tipos [Fer01]: (i) cenários de falha; e (ii) cenários recuperáveis. A diferença entre estes dois cenários está no tipo de falha que cada um trata.

Os **cenários de falha** se referem a situações onde o sistema não pode voltar a um estado normal válido, isto é, situações errôneas irreversíveis. Por esse motivo, esse tipo de cenário representa ações paliativas normalmente com o objetivo de confinar uma falha, evitando o agravamento da situação e proporcionando uma saída segura do sistema. Por exemplo, salvar o documento de texto antes do editor travar. Já os **cenários recuperáveis** representam ações corretivas com o objetivo de construir um estado normal válido para o sistema. Dessa forma, as ações corretivas representadas nesses cenários caracterizam implementações da técnica de recuperação por avanço, apresentada na Seção 2.3.3.

A Tabela 3.2 sugere o modelo de representação desses cenários excepcionais, que também é baseado no método MDCE. Como pode ser visto nessa tabela, o comportamento relativo ao cenário excepcional é especificado em um caso de uso específico, que é indicado no campo Tratador. Dessa forma se faz uma separação explícita entre os comportamentos normal e excepcional. Os casos de uso que implementam o comportamento relativo ao tratamento da exceção deve ser classificado com o estereótipo $<< handler >>$.

A execução do sistema inicia no cenário principal e pode alternar entre os cenários alternativos relacionados a ele. Por outro lado, caso um erro ocorra em qualquer um dos cenários normais, um cenário excepcional será executado. Se o erro foi uma falha prevista e permitida, um cenário recuperável será iniciado. Caso contrário, se a falha identificada representar uma situação inaceitável para a segurança do caso de uso, um cenário de falha será executado e o caso de uso sempre terminará de forma excepcional. Caso o cenário recuperável consiga tratar o erro com sucesso, os cenários normais voltarão a ser

Tabela 3.2: Especificação Excepcional de um Caso de Uso

Nome do Caso de Uso	
Cenário Recuperável/de Falha 1:	Breve descrição do cenário, incluindo uma síntese do seu comportamento.
Condição Excepcional:	Condição para a ativação do cenário (sinal de disparo).
Ponto de identificação:	O passo do cenário normal onde a condição de ativação pode ser verificada.
Tratador:	Informa o caso de uso que especifica o comportamento do tratador (<< *handler* >>).
Pós-condições:	Condições que devem ser satisfeitas após a execução do cenário (final do tratamento).

executados.

A Figura 3.6 mostra as atividades para a especificação dos tratadores das exceções agendadas. Inicialmente, cada exceção agendada é classificada segundo a natureza da falha, que pode ser: (i) recuperável; ou (ii) não recuperável (cenário de falha). Em seguida, após descrever sucintamente o tratamento que deverá ser executado, o tratador é classificado de acordo com a natureza do seu tratamento, que pode ser de dois tipos: (i) interno, quando um único componente é capaz de tratar a exceção; e (ii) externo (ou arquitetural), quando envolve mais de um componente do sistema no tratamento. Fazendo uma analogia ao modelo do componente tolerante a falhas ideal apresentado na Seção 2.3.7, as exceções tratadas na arquitetura devem necessariamente ser exceções externas. Após essa classificação, deve-se escolher o caso de uso excepcional responsável pelo tratamento. Caso ele não tenha sido especificado anteriormente, é necessário fazê-lo. Para isso, deve-se seguir a mesma seqüência de passos sugerida na Seção 3.2.4 para a especificação dos casos de uso normais. Mas o detalhamento dessas atividades só estão disponíveis na versão detalhada do método [dSBR05].

Finalmente, pode ser especificada a pós-condição do cenário excepcional. Após sua conclusão, esses cenários devem ser atualizados nos casos de uso. Essa atualização possibilita que haja uma associação entre os comportamentos normal e excepcional, apesar da separação explícita entre eles. Essa separação proporciona um aumento no grau de reutilização interna dos tratadores, isto é, componentes de um mesmo projeto poderão compartilhar o mesmo tratador. Isso é comum em sistemas que implementam tratamentos mais simples, como por exemplo abordagens de *logging* de mensagens, apresentada na versão detalhada do método [dSBR05].

O fato do comportamento excepcional ser descrito desde os casos de uso propicia uma análise que vai além do impacto das falhas sobre o sistema propriamente dito. Por se tratar de um artefato que auxilia em todas as fases do desenvolvimento [JCJv92], a especificação precoce do comportamento excepcional aumenta a qualidade das especificações excepcionais em todas as atividades do processo. Dessa forma, uma vez que as falhas e seus impactos já foram especificados, tem-se a base para uma especificação robusta, com

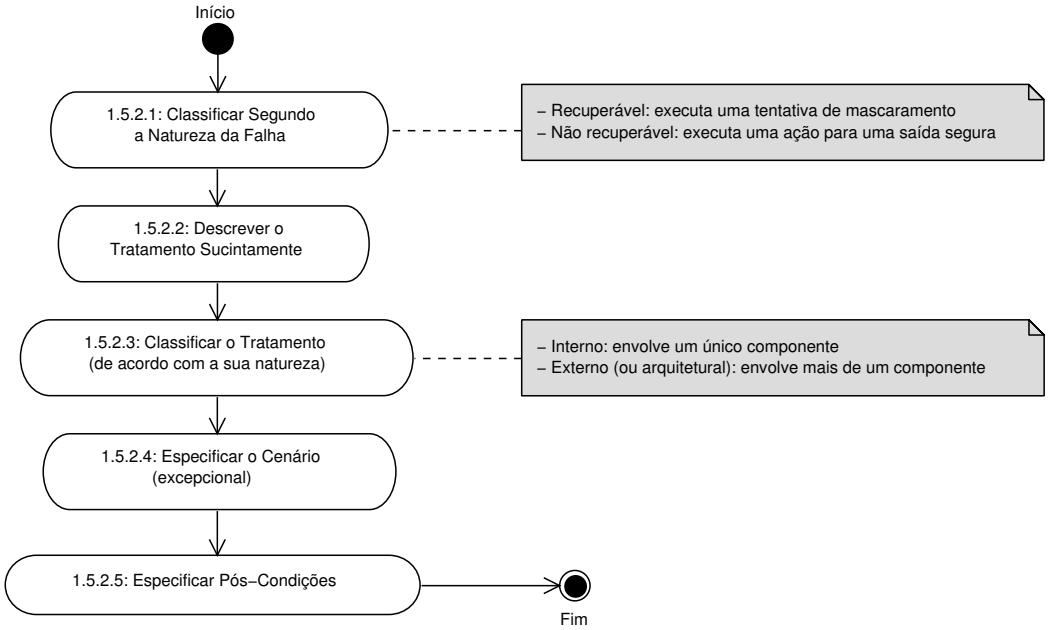

Figura 3.6: Especificação do Comportamento Excepcional

os mecanismos de proteção contra falhas e seus relacionamentos sendo incorporados desde as fases iniciais do desenvolvimento.

3.2.6 Construir/Atualizar o Diagrama de Casos de Uso (ativ. 1.6)

Para finalizar a construção do modelo de casos de uso, só nos falta modelar um diagrama represente graficamente o relacionamento entre os casos de uso identificados até o momento. Esse diagrama é o diagrama de casos de uso UML. Um **diagrama de casos de uso** representa no contexto de um sistema específico, um conjunto de funcionalidades (casos de uso), o relacionamento entre elas e a lista de todas as entidades externas que interagem com o sistema (atores). Sendo assim, esse diagrama modela aspectos dinâmicos do sistema e possui quatro elementos básicos: (i) atores; (ii) casos de uso; (iii) associações; e (iv) sistema. O documento detalhado do MDCE+ [dSBR05] apresenta um *workflow* detalhado dessa atividade, acompanhado de um exemplo ilustrativo.

3.3 Fase 2: Definição dos Aspectos Gerenciais

Nesse momento, a fase de especificação e análise dos requisitos está quase finalizada. Mas antes de seguir adiante com o desenvolvimento do software, é necessário verificar a existência de restrições para o desenvolvimento. Essas restrições podem tanto ser impostas

pelo cliente, quanto por limitações tecnológicas da atualidade. Além disso, antes que o desenvolvimento possa ser continuado, também é necessário fazer uma avaliação, do ponto de vista de criticidade, das funcionalidades especificadas até o momento. Dessa forma, espera-se que as funcionalidades críticas do sistema sejam identificadas, no intuito de otimizar a aplicação dos recursos disponíveis para tolerância a falhas. Finalmente, por se tratar de um processo iterativo, se faz necessário especificar juntamente com o cliente o cronograma de entrega[7] dos módulos do sistema. Essas entregas sucessivas visam atender a um requisito dos processos de desenvolvimento modernos, que é o desenvolvimento dirigido por característica[8] [PF01, Bec99].

Cada uma dessas três atividades é explicada a seguir: (i) definir as restrições para o desenvolvimento; (ii) identificar as funcionalidades críticas; (iii) definir as *releases* para iterações.

3.3.1 Definir as Restrições para o Desenvolvimento (ativ. 2.1)

Com o intuito de extrair o maior número de informações importantes a respeito das exigências do usuário, além dos requisitos funcionais do sistema, é necessário definir as restrições existentes para o seu desenvolvimento. Normalmente, essas restrições se referem a particularidades do ambiente de execução, como por exemplo o sistema operacional utilizado e a necessidade de se integrar a uma plataforma específica. A Tabela 3.3 mostra as restrições mais comuns, que são sugeridas pelo método MDCE+ como *check list* [Boe81, WBGK].

Analisando especificamente o contexto do desenvolvimento baseado em componentes (DBC), foi possível ver que são necessárias outras restrições adicionais baseadas nas particularidades desse paradigma de desenvolvimento. Devido ao reuso de componentes ser uma característica marcante do DBC, o MDCE+ oferece uma lista de ítens que pode ser utilizada como uma *check list* para especificar restrições relacionadas à reutilização de componentes prontos (Tabela 3.4) [AFC01]. Essas restrições serão levadas em consideração no momento de decidir entre implementar ou reutilizar os componentes que compõem o sistema (Seção 3.7.2).

Após a especificação das restrições para o desenvolvimento do sistema, é necessário realizar uma análise cuidadosa dos riscos envolvidos, o que determinará a continuidade ou não do projeto. Essa análise de riscos se baseia principalmente na identificação de situações prováveis de insucesso, tais como viabilidade técnica e problemas de tempo e recurso insuficientes. Com isso, o gerente do projeto é alertado para a necessidade de se ter planos de contingência relacionados aos problemas mais críticos identificados. Apesar

[7] do inglês *release schedule*
[8] do inglês *feature driven development*

Tabela 3.3: Principais Focos de Restrições

Restrição	Descrição
Tempo:	Tempo de desenvolvimento crítico. Dependendo dessa restrição, pode-se priorizar a reutilização a ponto de relevar ou negociar algumas inconsistências de requisitos.
Custo:	Limitação do custo do projeto. Normalmente essa restrição é em relação ao valor máximo estipulado pelo cliente. Essa restrição pode influenciar por exemplo, na criticidade do método em relação à implementação de tolerância a falhas.
Qualidade:	Dependendo do processo ou certificação, a empresa pode necessitar produzir uma lista específica de artefatos. Essa restrição é importante e pode adicionar novas atividades ao processo.
Estrutura:	A empresa pode adotar alguma arquitetura específica. Essa é uma restrição importante e levada em consideração na fase de projeto arquitetural. Porém, ela não impede que o arquiteto sugira novas possibilidades de solução.
Sistema Operacional:	Sistema operacional onde o sistema será executado.
Middleware:	Obrigatoriedade de adotar uma plataforma já utilizada na empresa.
Linguagem de Programação:	Dependendo do corpo técnico, pode-se exigir uma linguagem específica. Essa restrição pode ser justificada pela facilidade de manutenções posteriores.
Framework:	Algum arcabouço utilizado pela empresa. Apesar de ser uma restrição importante, que pode interferir no projeto arquitetural do sistema, o arquiteto pode sugerir novas soluções ao cliente.

Tabela 3.4: Principais Restrições Ligadas ao Reuso de Componentes

Restrição	Descrição
Restrições do vendedor:	Tempo de entrega, reputação.
Restrições do fabricante:	Garantia, experiência, maturidade, estabilidade, reputação.
Restrições de treinamento:	Disponibilidade, qualidade, custo adicional.
Restrições de custo:	Relação custo x benefício.
Restrições de pagamento:	Facilidades de pagamento.
Restrições contratuais:	Aspectos legais, licenças de uso.

de aparentemente representar um custo adicional ao tempo de desenvolvimento, processos que possuem fases explícitas de análise de riscos e de viabilidade proporcionam uma redução considerável dos custos finais do sistema. Essa economia é decorrente principalmente da redução do tempo de identificação e correção de problemas e da melhoria da pontualidade dos prazos negociados. Esses benefícios são percebidos principalmente no contexto de sistemas maiores e mais complexos [Pre01], que é o caso dos sistemas críticos.

3.3.2 Identificar as Funcionalidades Críticas (ativ. 2.2)

Com o intuito de melhorar a aplicação dos recursos disponíveis para a implementação efetiva de técnicas de tolerância a falhas, o método MDCE+ sugere a classificação das funcionalidades especificadas de acordo com a sua criticidade para a aplicação. Mais especificamente, as funcionalidades devem ser classificadas em três níveis: (i) essenciais; (ii) importantes; e (iii) desejáveis. Por se tratar de uma atividade influenciada tanto por características do domínio, quanto por particularidades da solução, essa análise crítica deve

ser feita em conjunto, tanto pelo cliente, quanto pelo analista de domínio e o engenheiro de requisitos responsáveis. Para isso, poderá ser utilizado o documento de compreensão do problema, produzido no início da análise de requisitos (Seção 3.2.1).

A importância dessa análise está no fato dela guiar, no método MDCE+, a escolha do nível de robustez exigido para cada componente do sistema. Esse ajuste das atividades do método tem o intuito de tornar flexível a escolha da melhor relação de compromisso[9] entre o tempo e o custo de desenvolvimento, o desempenho e a confiabilidade do sistema como um todo. Dessa forma, pretende-se direcionar a utilização dos recursos disponíveis de acordo com a criticidade de cada componente de software. Mais adiante, nas fases de análise e projeto (Seções 3.5 e 3.6), a criticidade do componente ajudará a decidir se ele será ou não alvo de tolerância a falhas ascendente, no caso de componentes novos, ou descendente, para os componentes reutilizados.

3.3.3 Definir as *Releases* para Iterações (ativ. 2.3)

Por mais cautelosa que possa ser a fase de especificação de requisitos, é inevitável que ainda durante o desenvolvimento do sistema surja a necessidade de evoluir esses artefatos [Pre01]. Essa evolução normalmente se refere a dois aspectos principais: (i) alteração de uma funcionalidade já especificada; e (ii) adição de novas funcionalidades. Dessa forma, para que o processo de desenvolvimento seja utilizado na prática, ele deve oferecer meios de revisar os requisitos constantemente, de forma a reduzir o esforço de manutenção do sistema. Além disso, devido às restrições de tempo cada vez mais presentes atualmente, existe uma tendência de parcelar a entrega do sistema de acordo com a necessidade do cliente.

Com o intuito de oferecer meios de lidar com essas necessidades do mercado de software, o método MDCE+ propõe que o desenvolvimento seja iterativo e baseado em entregas gradativas. Como visto no início deste capítulo (Figura 3.1), a cada iteração do desenvolvimento é realizada uma entrega de uma nova versão do sistema ao cliente. Esse desenvolvimento incremental favorece a evolução contínua dos requisitos, o que adequa o processo para o desenvolvimento de sistemas inovadores, uma vez que nesses casos, os desenvolvedores não têm muita familiaridade com a lógica do negócio [CdSC01]. Além disso, o fato de desenvolver o sistema através de iterações sucessivas facilita o paralelismo entre as atividades de desenvolvimento e testes, pois enquanto se especifica uma *release*, a anterior já estará sendo testada.

A Figura 3.7 mostra o *workflow* para a priorização das entregas (*releases*) que serão feitas ao cliente. Em seguida, cada uma dessas atividades é explicada rapidamente.

[9] do inglês *Tradeoff*

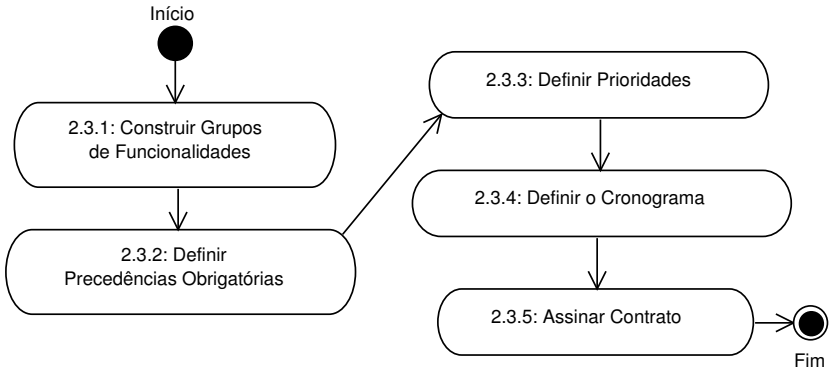

Figura 3.7: Definição das *Releases* do Sistema

1. **Construir grupos de funcionalidades (ativ. 2.3.1).** Devido ao fato dos requisitos do sistema terem sido mapeados através de casos de uso UML, o agrupamento das funcionalidades correlatas fica facilitado. Basicamente, cada grupo funcional é constituído de um determinado número de casos de uso, que normalmente são relacionados entre si.

2. **Definir precedências obrigatórias (ativ 2.3.2).** Essas dependências entre as funcionalidades do sistema podem ser percebidas de duas formas: (i) analisando-se os documentos de compreensão do problema (Seção 3.2.1); ou (ii) analisando-se os relacionamentos entre os casos de uso.

3. **Definir prioridades (ativ. 2.3.3).** De acordo com as suas necessidades, o cliente deve definir as funcionalidades mais urgentes. Essas funcionalidades podem ser tanto atividades importantes do ponto de vista do negócio (funcionalidades críticas, Seção 3.3.2), quanto prioridades logísticas, como por exemplo as funcionalidades de cadastro, a fim de adiantar a popularização das bases de dados. Vale a pena ressaltar que no caso de haver dependências obrigatórias ainda não implementadas, elas deverão ser agrupadas na mesma *release*.

4. **Definir o cronograma (ativ. 2.3.4).** Após a definição das *releases* do sistema, é necessário definir um cronograma para cada uma das versões previstas para entrega.

5. **Assinar contrato (ativ. 2.3.5).** Com a definição do cronograma de liberação do sistema, o contrato final entre os desenvolvedores e o cliente já pode ser assinado. Vale a pena salientar a importância em se definir algumas questões, como por exemplo o número máximo de refinamentos para cada *release*. Caso contrário, corre-se o risco de se desenvolver o sistema indefinidamente, uma vez que com a utilização

do sistema e com as mudanças inerentes à lógica do negócio, existe a tendência de evolução contínua dos requisitos [Bar05].

Para conciliar a flexibilidade do desenvolvimento iterativo com a robustez exigida para o desenvolvimento de sistemas confiáveis, a cada mudança tardia de requisitos, o método MDCE+ prevê uma análise do impacto dessas alterações (Figura 3.1). Nos casos onde esse impacto possa representar alterações na arquitetura do sistema ou a perda de um grande volume de trabalhos anteriores, essas mudanças devem ser re-negociadas com o cliente, que por sua vez pode confirmar a sua necessidade, desistir, ou o que é mais comum, adaptar os novos requisitos com o auxílio do arquiteto de software e do engenheiro de requisitos.

3.4 Fase 3: **Projeto Arquitetural**

Com o aumento crescente do tamanho e principalmente da complexidade dos sistemas de software, o projeto arquitetural assumiu um papel decisivo para o sucesso ou falha no atendimento dos requisitos, principalmente dos requisitos não-funcionais, que estão relacionados a aspectos qualitativos do sistema (Seção 2.1.1). Por esse motivo, no contexto de sistemas críticos, quer seja criticidade de tempo, de disponibilidade ou de confiabilidade; a arquitetura de software deve ter um lugar de destaque no seu processo de desenvolvimento [YSYY03, CWMY02].

Diferentemente dos processos clássicos de desenvolvimento, no método MDCE+ a fase de projeto arquitetural é executada antes da fase de análise. A principal razão para isso é o fato de que na fase de análise acontece a identificação dos componentes que constituem o sistema. Além disso, durante essa identificação, é feito um posicionamento inicial de cada um dos componentes de acordo com as respectivas camadas da arquitetura. Dessa forma, a antecipação da fase de projeto arquitetural possibilita uma maior contextualização dos componentes identificados com a estrutura geral do sistema, o que reforça o fato do MDCE+ ser um método centrado na arquitetura.

As principais vantagens de se enfatizar o papel arquitetural do sistema são: (i) ter uma estruturação abstrata, que facilita o entendimento e faz da arquitetura um eficiente veículo de comunicação entre as partes interessadas[10] do sistema; (ii) alta granularidade de reutilização, que aliado ao DBC pode ser determinante para a redução do tempo de desenvolvimento, o que atende a um requisito importante do mercado desenvolvedor de software atual [Ran04]; e (iii) maior facilidade para adaptar, manter, evoluir e portar o sistema para outras plataformas, como conseqüências do baixo acoplamento proporcionado pelo uso de conectores arquiteturais, que explicitam a comunicação entre os componentes da

[10] do inglês *Stakeholders*

arquitetura. Essa comunicação explícita facilita a substituição dos componentes, uma vez que torna possível a adaptação das mensagens que fluem entre eles [YSYY03, dCGFPR04].

Um processo de desenvolvimento para sistemas confiáveis, além de usufruir dessa abstração arquitetural em todas as fases do desenvolvimento, deve fornecer meios de especificar a arquitetura do sistema, obedecendo as restrições impostas pelos seus requisitos não-funcionais [YSYY03].

Durante a escolha dos estilos arquiteturais que serão utilizados, os interessados podem inclusive reutilizar arquiteturas de sistemas anteriores, baseado nas semelhanças entre os modelos conceituais, as suas restrições e os seus atributos de qualidade. Seguindo essa tendência, pesquisadores de várias partes do mundo estudam maneiras de sistematizar essa reutilização estrutural do sistema como forma de agilizar o desenvolvimento do software. Um exemplo muito em voga atualmente é a abordagem de desenvolvimento em linhas de produção de software[11] [ABM00, Nor04, Fei96]. Devido à interferência da arquitetura em todo o desenvolvimento do sistema, a Seção 3.4.1 apresenta um método para a escolha de uma arquitetura adequada.

3.4.1 Método para a Especificação da Arquitetura do Sistema

A relação direta entre a arquitetura de um sistema e seus requisitos não funcionais é uma característica bem conhecida entre os arquitetos de software [CK03, Som01]. Apesar disso, não faz muito tempo que vem sendo investido esforço para sistematizar a maneira como essas estruturas podem ser materializadas, de forma a maximizar a satisfação desses requisitos [BKB00]. Além disso, a possibilidade de existir requisitos antagônicos, como por exemplo confiabilidade extrema aliada ao desempenho crítico, evidencia a necessidade de alguma abordagem para alcançar o melhor ponto de equilíbrio de satisfação, onde não necessariamente todos os requisitos sejam satisfeitos.

O método MDCE+ propõe uma simplificação da abordagem adotada pelo método ATAM [CK03], que foi apresentado na Seção 2.2.3. Essa abordagem é baseada no entendimento de como os atributos de qualidade do sistema podem ser satisfeitos pela arquitetura. Basicamente, essa fase é constituída de cinco atividades principais, que seqüencialmente, orientam o arquiteto na escolha de uma arquitetura adequada. A Figura 3.8 mostra o *workflow* referente a esse método e em seguida, cada uma das atividades é explicada em maiores detalhes.

1. **Avaliar restrições arquiteturais (ativ. 3.1).** Nessa atividade, o arquiteto de software avalia as restrições definidas para o sistema (Seção 3.3.1), a fim de descartar as arquiteturas consideradas claramente inviáveis. O resultado final dessa atividade

[11] do inglês *Software product lines*

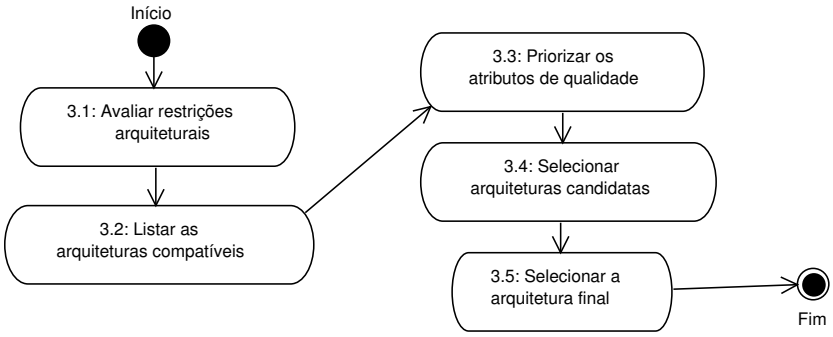

Figura 3.8: Especificação da Arquitetura do Sistema

é uma lista de fatores que podem interferir diretamente na escolha da arquitetura, podendo inclusive inviabilizar alguma adoção específica. Alguns exemplos de fatores a serem analisados são: a adoção de um determinado *framework* ou de uma plataforma específica; ou até mesmo a preferência pela utilização de alguma arquitetura em particular.

2. **Listar as arquiteturas compatíveis (ativ. 3.2).** Após a identificação dos principais fatores que influenciam a escolha da arquitetura, o arquiteto deve restringir a lista de estilos arquiteturais candidatos. Essa restrição tem o intuito de retirar os estilos considerados inviáveis de acordo com os novos fatores levantados. O resultado final dessa atividade é uma lista de estilos arquiteturais que são compatíveis com as restrições identificadas na atividade anterior. Esses estilos são considerados candidatos a satisfazer os atributos de qualidade do sistema. Alguns exemplos de estilos arquiteturais são [CK03]: arquiteturas cliente-servidor, arquiteturas em camadas e arquiteturas *publish-subscribe*.

3. **Priorizar os atributos de qualidade (ativ. 3.3).** Depois de se ter a listagem dos estilos arquiteturais que podem ser utilizados, é chegada a hora de escolher entre eles, o estilo (ou combinação de estilos) mais adequado para o sistema.

   Para isso, o método MDCE+ sugere uma análise criteriosa dos atributos de qualidade desejados. O primeiro passo dessa análise não-funcional do sistema é estipular os critérios da avaliação, isto é, a relação de precedência entre os atributos de qualidade. Seguindo a proposta utilizada pelo método ATAM, o cliente juntamente com o arquiteto de software devem estabelecer uma árvore de precedência que represente as prioridades de escolha de uma maneira hierárquica *top-down*. Dessa forma, como mostrado na Figura 3.9, os requisito mais gerais são representados como nós de mais alto nível. No exemplo específico dessa figura, os requisitos de confiabilidade, disponibilidade, manutenibilidade e performance são considerados requisitos gerais. No

modelo de árvore utilizado pelo MDCE+, a ordem em que os requisitos são dispostos (da esquerda para a direita) indica a sua precedência em relação aos demais. Por exemplo, na mesma árvore da Figura 3.9, a ordem de precedência dos atributos de qualidade é a seguinte: 1^o- confiabilidade; 2^o- disponibilidade; 3^o- manutenibilidade; e 4^o- desempenho.

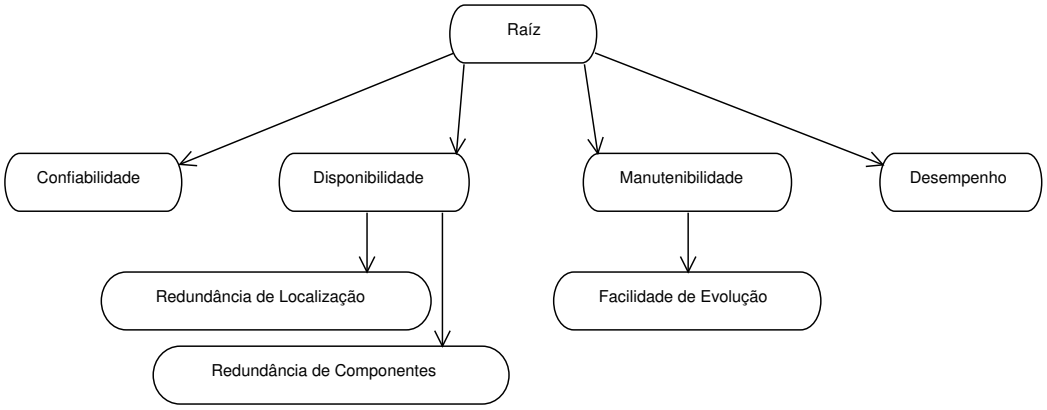

Figura 3.9: Árvore de Precedência dos Requisitos Não-Funcionais

Em seguida, a partir das descrições dos casos de uso (Seção 3.2.4), são selecionadas as afirmações que evidenciam exemplos da necessidade real de cada um desses requisitos. Essas frases são utilizadas para detalhar os interesses das partes interessadas e ajudam a repensar a ordem de precedência entre os atributos de qualidade. Por exemplo, atributos exigidos por alguma funcionalidade crítica tem, normalmente, uma prioridade maior. Para reduzir o número de cenários a serem analisados, o método MDCE+ sugere a análise apenas dos cenários pertencentes às funcionalidades críticas do sistema. A Figura 3.10 mostra a árvore de precedência da Figura 3.9 com alguns cenários representados.

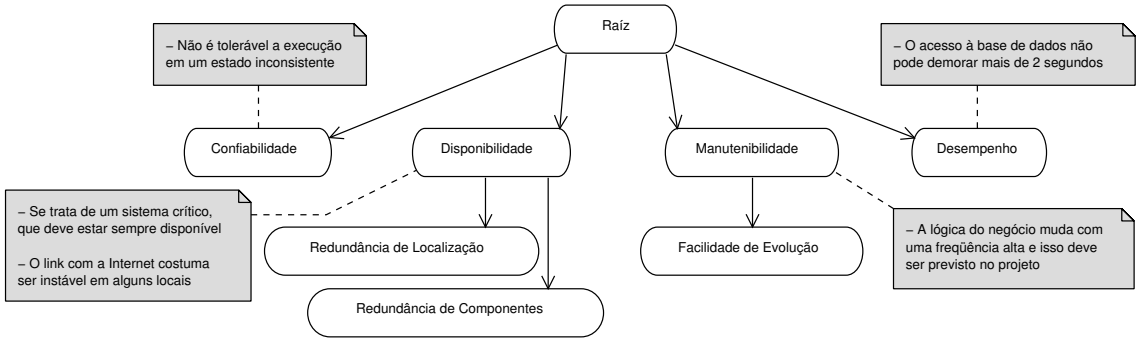

Figura 3.10: Árvore de Precedência de Requisitos Não-Funcionais com Cenários

4. **Selecionar arquiteturas candidatas (ativ. 3.4).** Com a definição das prioridades entre os atributos de qualidade do sistema, o arquiteto deve fazer uma triagem entre os estilos arquiteturais existentes na saída da Atividade 2. Essa triagem tem o intuito de restringir a lista de arquiteturas candidatas que satisfaçam os requisitos não-funcionais do sistema. Para isso, a partir da avaliação da árvore de precedência com cenários, devem ser procurados estilos arquiteturais indicados para cada um dos requisitos desejados. Para esse mapeamento, o método MDCE+ recomenda a utilização do método ABAS[12] [KK99], desenvolvido pelo Instituto de Engenharia de Software (SEI) da Universidade de Carnegie Mellon (CMU). Em seguida, deve ser feita uma análise dos riscos envolvidos com a utilização de cada um deles. Essa análise deve considerar as relações de compromisso[13] entre os requisitos; lembrando que a relação de precedência deve ser levada em consideração nessa análise. O produto final dessa atividade é uma lista restrita dos estilos arquiteturais (ou combinações de estilos) mais adequados para o sistema.

5. **Selecionar a arquitetura final (ativ. 3.5).** Esta é a última atividade referente à escolha da arquitetura do sistema. Por se tratar de uma das decisões mais importantes de todo o desenvolvimento, ela deve ser desempenhada em conjunto por toda a equipe de projeto: cliente, arquiteto de software, engenheiro de requisitos, testador, e mantenedor. Normalmente, essa atividade consiste de uma ou mais seções de *brainstorm* moderadas pelo arquiteto. Nessas reuniões, cada um dos participantes escolhe um dos estilos arquiteturais pré-selecionados e argumenta a respeito da sua decisão; em seguida, o grupo tenta chegar a um consenso, a partir da combinação de estilos sugerida pelo arquiteto. No final, após a escolha da arquitetura mais indicada, essa informação deve ser documentada no contrato.

Vale a pena salientar o fato de nem sempre ser possível satisfazer todos os requisitos desejados para o sistema. Nos casos onde os requisitos não são atendidos totalmente, o método MDCE+ define que deve-se iterar novamente, a fim de refinar os requisitos, a arquitetura ou ambos e se necessário, o contrato com o cliente e a documentação inicial do sistema também deve ser atualizado.

3.5 FASE 4: **Análise do Sistema**

A partir da fase de análise do sistema, o foco se volta para aspectos mais específicos relacionados ao domínio da solução do problema [JBR99]. Por esse motivo, como pode

[12] do inglês *Attribute-Based Architectural Styles*
[13] do inglês *Tradeoff*

ser visto na Figura 3.1, os artefatos de entrada para essa fase são: (i) o diagrama de casos de uso (com as especificações de cada um); (ii) o modelo de representação do domínio; e (iii) a arquitetura definida para o sistema. Os dois primeiros são provenientes da análise de requisitos (Seção 3.2) e o último é fruto da fase de projeto arquitetural (Seção 3.4). O detalhamento das atividades de análise pode ser visto na Figura 3.11, que mostra o *workflow* de execução dessa fase. Cada uma dessas atividades serão detalhadas nas seções seguintes (3.5.1 a 3.5.4).

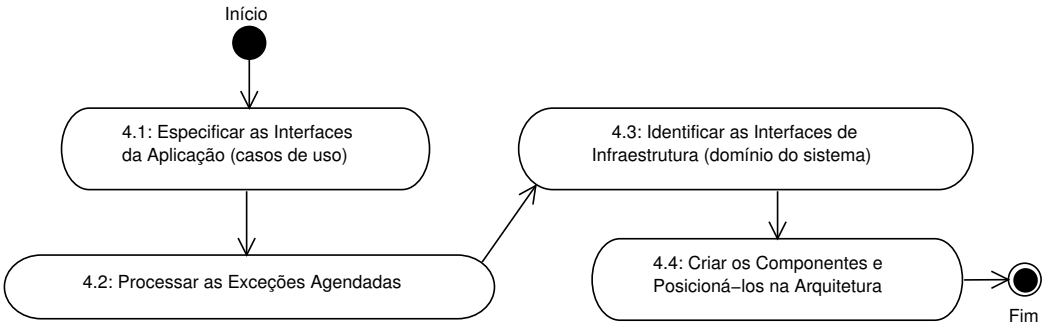

Figura 3.11: *Workflow* da Fase de Análise

De uma maneira geral, o passo principal a ser executado durante a análise de um sistema é a identificação das entidades envolvidas na resolução do problema em questão. Por se tratar de um método para desenvolvimento baseado em componentes, essas entidades são os componentes em si. Porém, nesta fase do desenvolvimento é necessário enxergar os componentes do ponto de vista de especificação, como descrito na Seção 2.1 (Tabela 2.1). De uma maneira geral, isso significa que a atenção deve ser voltada para a modelagem das interfaces (providas e requeridas), sem se preocupar com qual componente específico realmente desempenhará esse papel. Além disso, não se pode esquecer em nenhum momento o contexto da arquitetura, isto é, durante a identificação dos componentes do sistema, deve-ser ter em mente o posicionamento de cada um na arquitetura adotada.

As principais fontes de identificação de componentes são: (i) os casos de uso; e (ii) o modelo de representação do domínio do negócio. Devido aos casos de uso serem diretamente relacionados à maneira como as funcionalidades serão implementadas, os componentes derivados deles implementam normalmente particularidades do domínio, o que dificulta a sua reutilização. Por esse motivo, esses componentes são considerados candidatos a uma nova implementação. Já os componentes relativos à lógica do negócio, que são derivados do modelo de representação do domínio, constituem a infra-estrutura necessária para que diversos sistemas relacionados ao mesmo domínio funcionem corretamente. Por esse motivo, esses componentes são considerados candidatos à reutilização.

O método MDCE+ diferencia esses dois tipos de componentes: (i) de funcionalidades

da aplicação; e (ii) de funcionalidades de infra-estrutura. Essa classificação tem os objetivos de auxiliar no processo de reutilização (concentração do esforço) e de auxiliar na escolha do mecanismos de tolerância a falhas adequado, como será visto adiante na Seção 3.7. Devido a essa distinção, as interfaces derivadas dos casos de uso são consideradas como interfaces da aplicação, enquanto que as interfaces que conterão as funcionalidades de infra-estrutura são consideradas interfaces de infra-estrutura.

3.5.1 Especificar as Interfaces da Aplicação (ativ. 4.1)

Essa atividade consiste na identificação das interfaces e operações que representarão as funcionalidades especificadas. Por esse motivo, esse comportamento normal esperado para o sistema é identificado a partir dos casos de uso definidos nos requisitos (Seção 3.2.4). A versão detalhada do método MDCE+ [dSBR05] apresenta o *workflow* dessa atividade.

3.5.2 Processar as Exceções Agendadas (ativ. 4.2)

Esta atividade da análise é responsável tanto pela criação das exceções, quanto pela criação das interfaces excepcionais, cujas operações corresponderão aos tratadores dessas exceções. Essa análise excepcional do sistema se baseia na lista de exceções que foram agendadas tanto na iteração anterior, quanto no refinamento dos requisitos feito na iteração atual.

Como pode ser visto na Figura 3.12, inicialmente para cada exceção agendada, deve-se analisar a necessidade de se criar uma classe que a represente, isto é, deve-se verificar se já existe a classe da exceção. Essa classe será utilizada para encapsular através de atributos, as informações contextuais que serão utilizadas para melhorar a qualidade do tratamento [GRRX01]. Apesar de parte dessas informações já serem descobertas neste momento, elas serão complementadas durante o desenvolvimento do sistema, mais especificamente na fase de descoberta das operações de infra-estrutura (Seção 3.6.1). Alguns exemplos de informações contextuais podem ser: nome da exceção, descrição, gravidade, local de lançamento, etc. Mais detalhes a respeito do gerenciamento das informações contextuais estão disponíveis na versão detalhada do método [dSBR05] dessa dissertação. Independentemente do fato da classe já existir ou dela ter sido criada agora, deve-se associar a exceção à operação que a chamou. Essa informação pode ser extraída a partir do passo do cenário que foi responsável pelo lançamento da exceção, já que existe um mapeamento entre esses passos e as operações das interfaces da aplicação.

Após a criação da classe da exceção, o próximo passo é verificar a necessidade de se criar uma nova interface excepcional. Essa verificação é executada para cada caso de uso excepcional (<< *handler* >>) e consiste basicamente de duas etapas: (i) verificar se existe alguma interface excepcional capaz de conter os tratadores da exceção em questão; e (ii) verificar a possibilidade de agrupar algumas interfaces em uma interface mais genérica,

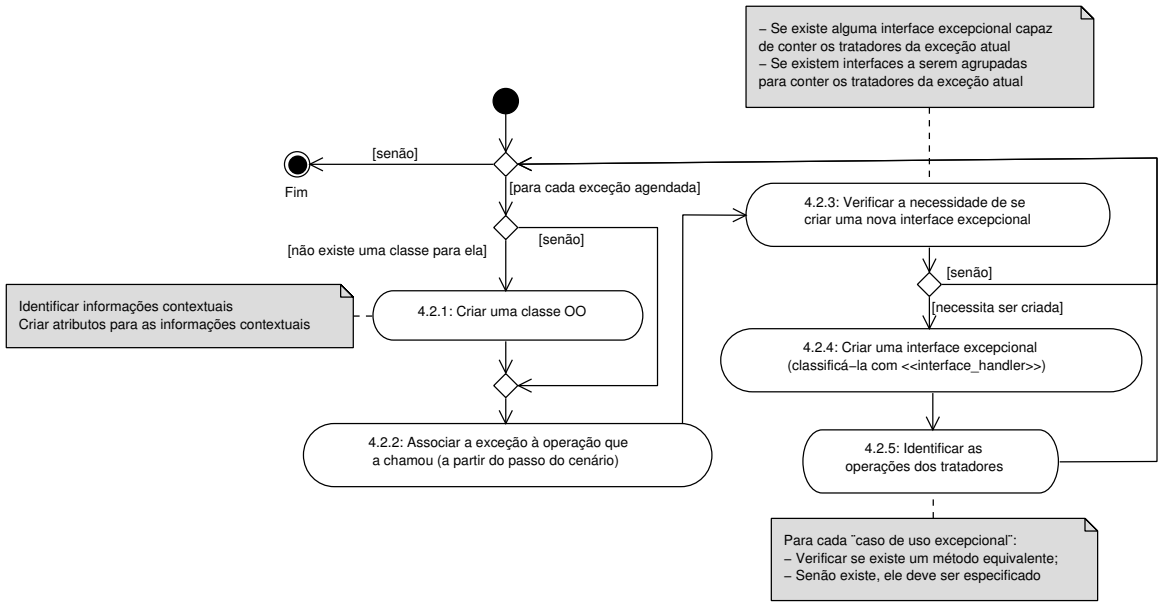

Figura 3.12: Processamento das Exceções Agendadas

capaz de conter os tratadores da exceção atual. Essa análise, que é uma etapa importante que influenciará todo o desenvolvimento do sistema, é desempenhada pelo analista. O analista deve executá-la tendo sempre em mente os objetivos de manter a alta coesão do sistema e de alcançar o equilíbrio na relação de compromisso existente entre a complexidade das interfaces (número de operações) e a facilidade de implementação (número de interfaces). Para oferecer uma distinção clara desde o nível de modelagem, as interfaces excepcionais são classificadas com o estereótipo $<< interface\_handler >>$.

Com as interfaces excepcionais criadas, o próximo passo é a identificação das suas operações. Para isso, deve-se verificar a necessidade de se criar um método na interface, relativo a cada um dos tratadores especificados nos casos de uso excepcionais ($<< handler >>$). Essa verificação consiste na checagem da existência prévia de uma operação que implemente um tratador equivalente ao tratamento desejado. Caso não exista ou a sua assinatura necessite ser modificada, o analista deve proceder a atualização. Vale salientar que diferentemente das operações das interfaces providas normais, as operações das interfaces excepcionais não são derivadas dos passos dos cenários, mas somente para cada um dos casos de uso excepcionais. Dessa forma, cada operação representa um tratador, que pode ser reutilizado em diferentes pontos do projeto.

Essa possibilidade de reutilização do comportamento excepcional é uma das vantagens de se ter uma separação explícita entre o código normal do sistema e seus tratadores excepcionais. Essa vantagem é evidenciada pelo fato de num mesmo projeto, muitas vezes ser comum a existência de tratamentos semelhantes em diferentes contextos. Um

exemplo pode ser a abordagem de *logging* de exceções, mostrada na versão detalhada do método [dSBR05].

Adiante, na fase de implementação (Seção 3.7), ocorrerá a materialização dos tratadores em código. Dependendo da sua complexidade, o método MDCE+ prevê duas formas de implementação: (i) como um componente, no caso de sistemas confiáveis, onde os tratamentos costumam ser mais complexos; ou (ii) como uma classe, no caso de sistemas sem requisitos críticos de confiabilidade, que apresentam um tratamento simplificado.

3.5.3 Identificar as Interfaces de infra-estrutura (ativ. 4.3)

Como discutido anteriormente, no método MDCE+, as interfaces de infra-estrutura são aquelas que oferecem os serviços necessários para que os componentes da aplicação funcionem adequadamente. Sendo assim, para a identificação dessas interfaces será necessário ter em mente a contextualização de cada funcionalidade no domínio do problema. Dessa forma, as interfaces de infra-estrutura são identificadas a partir das entidades desse domínio. Como pode ser visto na versão detalhada do método MDCE+ [dSBR05], os únicos artefatos de entrada necessários são: (i) o modelo de representação do domínio, produzido na Seção 3.2.2; e (ii) a definição do escopo do sistema, definida na Seção 3.2.3. Ambos os artefatos são provenientes da fase de análise de requisitos do sistema.

3.5.4 Criar os Componentes e Posicioná-los na Arq. (ativ. 4.4)

Após a definição das interfaces dos componentes do sistema, pode-se iniciar a criação dos componentes propriamente ditos. Nessa atividade, serão criados tanto os componentes normais, quanto os componentes excepcionais, que oferecerão as operações relativas aos tratadores de exceção. Devido à similaridade entre os *workflows* de criação desses dois tipos de componentes, a Figura 3.13 mostra as atividades relativas à criação de ambos. Como pode ser visto nessa figura, o procedimento de criação dos componentes se baseia unicamente nas interfaces já identificadas, sejam elas normais (da aplicação e de infra-estrutura) ou excepcionais. As diferenças entre a criação de componentes normais e excepcionais estão representadas por condições. Os parágrafos seguintes explicam melhor a figura.

Para cada uma das interfaces, deve existir um componente que a ofereça. Mas antes de se proceder a criação de um novo componente, é necessário verificar a possibilidade de associar a interface a algum componente já existente. No caso de não haver um componente que possa oferecê-la, ele deve ser criado para que a associação seja feita. Caso a interface seja proveniente de um caso de uso "coordenador" de uma cooperação, o componente deve ser classificado da mesma forma. A necessidade dessa classificação será compreendida mais tarde, na fase de implementação (Seção 3.7), uma vez que os

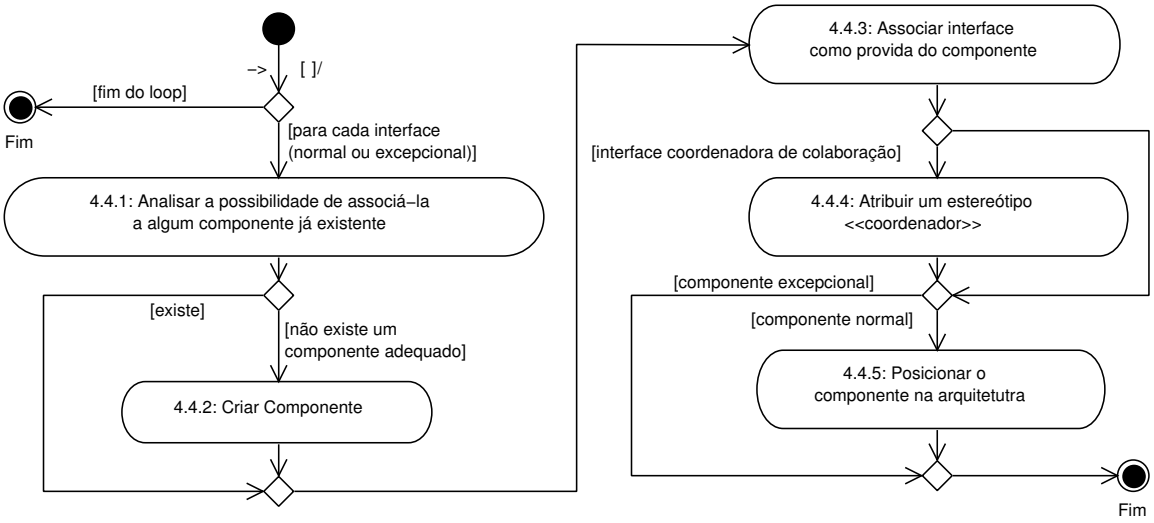

Figura 3.13: Criação dos Componentes (Normais e Excepcionais)

componentes coordenadores necessitam de informações contextuais de vários componentes do sistema e por isso são implementados como um invólucro[14] entre eles.

Em relação ao posicionamento na arquitetura, como pode ser visto na Figura 3.13, apenas os componentes normais são posicionados. A exclusão dos componentes excepcionais dessa atividade é justificada pelo fato de assim como sugerido pelo método MDCE, no MDCE+ os componentes arquiteturais do sistema são estruturados segundo o modelo do componente tolerante a falhas ideal, descrito na Seção 3.5.2 e agregam tanto o comportamento normal, quanto o excepcional. Dessa forma, o posicionamento atribuído aos componentes normais implicará no posicionamento dos respectivos componentes arquiteturais, com comportamento normal e excepcional. Os componentes excepcionais criados nesse momento representam componentes de projeto (CBD) e não são representados diretamente na arquitetura do sistema, podendo ser utilizados por mais de um componente normal, devido ao reuso do comportamento excepcional discutido na Seção 2.3.7.

Apesar do posicionamento dos futuros componentes arquiteturais do sistema, ainda não se tem as informações necessárias para a representação das dependências entre eles. Em outras palavras, embora se tenha o conhecimento prévio de que as operações oferecidas pelos componentes de infra-estrutura serão requeridas pelos componentes da aplicação, ainda não se tem o conhecimento e o contexto suficientes para saber quais são as operações requeridas de cada um deles. Isso é detalhado na fase de projeto do sistema, apresentada na Seção 3.6.

[14]do inglês *Wrapper*

3.6 FASE 5: **Projeto do Sistema**

O objetivo principal da fase de projeto do software é refinar os modelos provenientes da análise, com o objetivo de aproximá-los da implementação [JBR99]. Em outras palavras, apesar de na fase de análise já se pensar na solução do problema, é somente a partir do projeto que se analisa do ponto de vista específico da implementação. Por se tratar de desenvolvimento baseado em componentes, o projeto será conduzido utilizando os componentes como unidade básica de abstração, mas a forma como eles serão materializados, se utilizando programação estruturada, orientada a objetos, etc., será decidida na próxima fase, a fase de implementação (Seção 3.7).

Para estruturar melhor as suas atividades, o projeto do sistema foi dividido em dois passos principais: (i) análise do aspecto interativo entre os componentes; e (ii) formalização da especificação dos componentes. As sub-seções a seguir (3.6.1 e 3.6.2) apresentam cada um deles em detalhes. Dessa forma, será possível perceber a importância da fase de projeto no desenvolvimento de sistemas críticos.

3.6.1 Analisar a Interação entre os Componentes

Analisar o aspecto interativo entre as entidades que compõem um sistema é imprescindível durante o seu desenvolvimento, principalmente do ponto de vista de tolerância a falhas, uma vez que um componente isolado não é capaz de oferecer os recursos necessários para identificação e tratamento dos erros de forma efetiva [dLR99]. Essa importância foi evidenciada ainda mais com o advento da arquitetura de software, que visa explicitar essas interações através de uma visão do sistema em um alto nível de abstração.

Como pode ser visto na Figura 3.14, esse passo do projeto é constituído das atividades relacionadas diretamente com o aspecto interativo entre os componentes. Portanto, além da identificação das operações requeridas pelos componentes do sistema e um conseqüente refinamento da arquitetura. Do ponto de vista da melhoria dos atributos de qualidade relacionados à sua criticidade e facilidade de manutenção, se faz necessário refinar o projeto em oito perspectivas complementares: (i) detalhar os tratadores; (ii) estruturar os componentes arquiteturais; (iii) refinar as entidades críticas e definir a rigorosidade do método; (iv) analisar o fluxo excepcional; (v) revisar o modelo de falhas; (vi) refinar as colaborações; (vii) revisar as interfaces providas dos componentes e (v) definir as interfaces requeridas. Apesar do grande número de atividades, a sua maioria é executada apenas uma vez por cada *release*. As atividades que compõem essa etapa de análise interativa são descritas nas sub-seções seguintes.

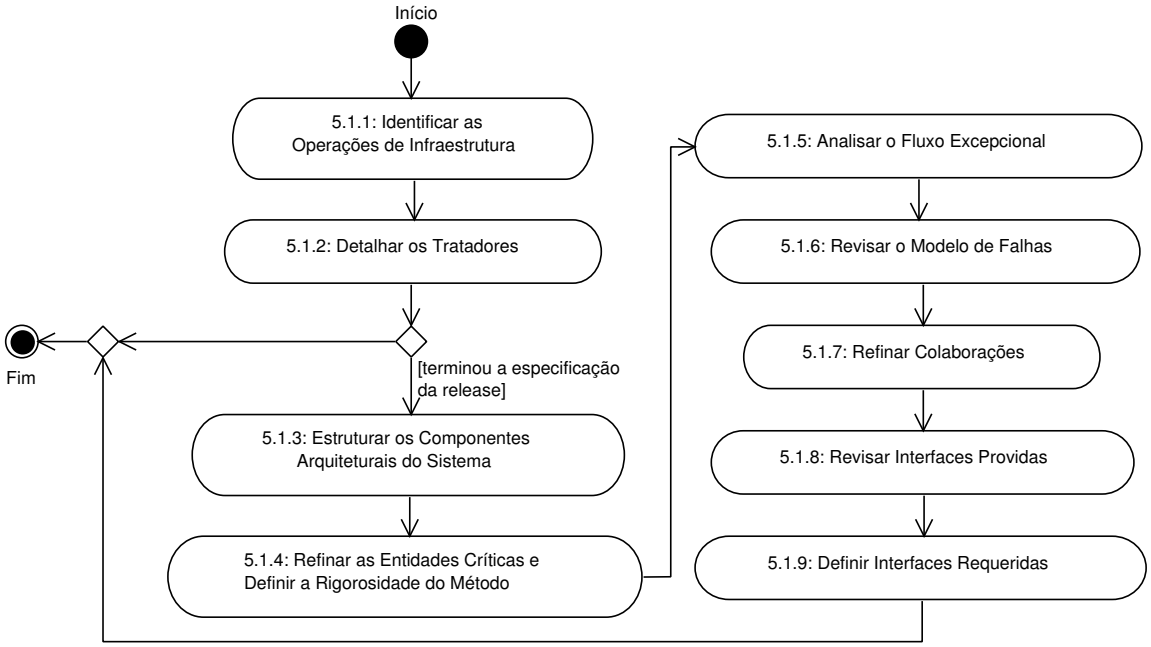

Figura 3.14: Analisar a Interação entre os Componentes

Identificar as Operações de Infra-estrutura (ativ. 5.1.1)

Como definido na Seção 3.5, as operações oferecidas pelos componentes de infra-estrutura são as operações requeridas para a execução dos serviços da aplicação. Por essa razão, a identificação dessas operações se dá a partir da simulação dos cenários dos casos de uso, já que as interfaces da aplicação são derivadas deles. Sendo assim, para cada uma das operações da aplicação que representam funcionalidades de casos de uso, deve-se analisar os passos dos seus cenários normais. Com essa análise seqüencial, deve-se perceber a necessidade de execução de serviços gerais do domínio (infra-estrutura), como por exemplo uma consulta aos dados de alguma entidade. Em seguida, todas as novas operações requeridas devem ser adicionadas às interfaces de infra-estrutura correspondentes, de acordo com a definição das responsabilidades das entidades principais, que é produzida na Seção 3.5.3 e explicada em detalhes na versão estendida do método [dSBR05]. As colaborações resultantes da análise dos passos dos cenários são representados por diagramas de atividades UML, onde cada interface requerida é representada em uma *swimlane*.

Detalhar os Tratadores de Exceções (ativ. 5.1.2)

Assim como na identificação das operações de infra-estrutura, o detalhamento dos tratadores deve ser analisado para cada uma das operações que representam um casos de uso. Porém, por lidar com o comportamento excepcional do sistema, as operações analisadas

são as relativas aos casos de uso que especificam os tratadores excepcionais.

Antes de iniciar a simulação dos fluxos de execução, o método MDCE+ propõe uma revisão da lista de exceções lançadas por cada caso de uso normal. Essa revisão tem o intuito de perceber a necessidade de refinar os tratadores existentes ou de especificar novos tratadores. Por exemplo, o projetista pode julgar necessário algum pré-processamento antes que uma dessas exceções seja propagada. Nesses casos, os requisitos do sistema devem ser refinados, a fim de adicionar mais um cenário excepcional ao caso de uso, ou de refinar o comportamento de um tratador já especificado. As diretrizes propostas na versão detalhada do método MDCE+ [dSBR05] podem auxiliar a tarefa de especificação dos cenários excepcionais. Por consistir de uma atualização dos requisitos, o processo deve iterar novamente para que essas mudanças possam ser percebidas (em termos de interfaces e operações).

Após esse refinamento do comportamento excepcional, deve-se proceder com a simulação dos fluxos de execução. Como pode ser visto na Figura 3.15, para cada operação das $<< interface\_handler >> s$, deve-se identificar as operações requeridas para a execução do tratamento. Esse processo de identificação de operações já é nosso conhecido: é semelhante ao utilizado na sub-seção anterior, para a identificação das operações de infra-estrutura. Mas desta vez são utilizados os cenários dos casos de uso excepcionais ($<< handler >>$). As operações identificadas devem ser associadas como requeridas dos respectivos tratadores.

Seguindo a seqüência de atividades, após a identificação das operações necessárias para o tratamento, deve-se prosseguir com a simulação do cenário, que também será documentado através de um diagrama de atividades UML. Por se tratar de um artefato referente ao comportamento excepcional do sistema, ele deve ser distinguido dos demais, a fim de haver uma separação de interesse[15] entre os artefatos normais e excepcionais especificados. Aproveitando a contextualização do projetista em relação ao tratamento especificado, os tratadores devem ser classificados quanto à natureza do seu comportamento excepcional. Para isso, além do tratamento, deve ser levada em consideração a forma de detecção da condição excepcional. A natureza do comportamento excepcional pode ser classificada em: (i) simples; (ii) arquitetural; ou (iii) colaborativo. Para auxiliar a classificação do tratamento, o projetista pode se basear na gravidade atribuída à falha durante a especificação dos casos de uso excepcionais (Seção 3.2.4), fazendo o seguinte mapeamento:

Falhas com gravidades **muito baixa**, **baixa** e **média**. Provavelmente possuem um tratamento simples;

Falhas com gravidades **alta** e **muito alta**. Provavelmente possuem um tratamento

[15] do inglês *Separation of concerns*

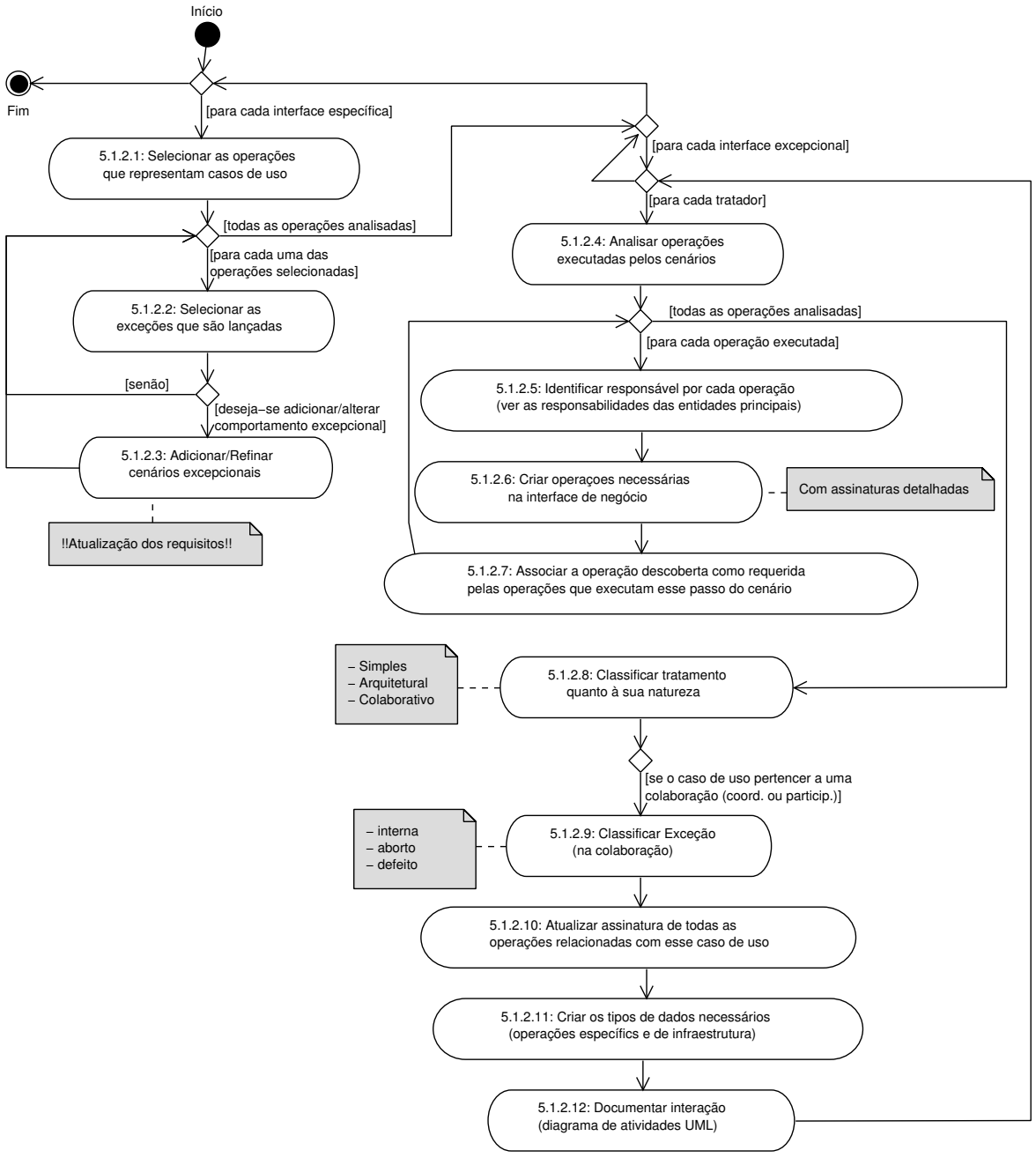

Figura 3.15: Detalhamento dos Tratadores

arquitetural, uma vez que o seu tratamento efetivo envolve algum tipo de reconfiguração ou reinstanciação dos componentes do sistema.

Além da gravidade da falha, existem outros fatores que podem ser analisados para auxiliar a classificação. Por exemplo, se para a detecção de alguma exceção for necessário ter informações a respeito de vários componentes, ainda que o tratamento da exceção seja interno, ele deve ser classificado como arquitetural. Um exemplo dessa situação é um problema típico de sistemas que implementam técnicas de garantia de qualidade de serviço [SDUV03]. Nesses sistemas, é comum somente considerar um erro no momento que um determinado percentual de componentes deixa de oferecer os seus serviços. Dessa forma, a meta-informação do número total de componentes e do número de componentes falhos só pode ser obtida através da arquitetura do sistema.

Para finalizar, no caso da exceção pertencer a uma colaboração (cenário de um caso de uso participante), a exceção deve ser classificada segundo sugerido por Xu et al. [XRR+99]. Nessa classificação, as exceções são agrupadas de acordo com a efeito que elas causam à colaboração: (i) interna, quando ainda é possível oferecer um tratamento dentro da colaboração; (ii) aborto, quando apesar de não possuir um tratamento corretivo, é possível cancelar a execução garantindo a consistência do estado do sistema; e (iii) defeito, quando além de não possuir um tratamento corretivo, não se pode garantir a consistência do estado do sistema. Vale a pena salientar que essa classificação só é considerada no contexto da colaboração; para o restante do sistema ela é irrelevante.

O detalhamento dos tratadores está finalizado. As sete atividades de projeto restantes só são executadas na última iteração de cada *release*. Elas são detalhadas nas próximas sub-seções.

Estruturar os Componentes Arquiteturais do Sistema (ativ. 5.1.3)

Esta atividade consiste na composição dos componentes arquiteturais, que como sugerido pelo método MDCE (Seção 2.3.8), são estruturados segundo o modelo do componente tolerante a falhas ideal (Seção 2.3.7). O *workflow* proposto para a estruturação desses componentes pode ser visto na Figura 3.16.

Dessa forma, para cada tratador que possa ser executado por uma operação normal do sistema (da aplicação ou de infra-estrutura), deve-se criar uma associação entre o componente normal e o excepcional. Apesar da associação ser feita entre os componentes, vale a pena salientar a necessidade de se preservar a informação de qual operação normal executa qual tratador. Essa preocupação adicional traz benefícios para a execução das atividades de manutenção do sistema.

A partir deste momento, ao se referir a um componente ideal ou a um componente arquitetural, estará fazendo uma referência ao componente abstrato, formado por um com-

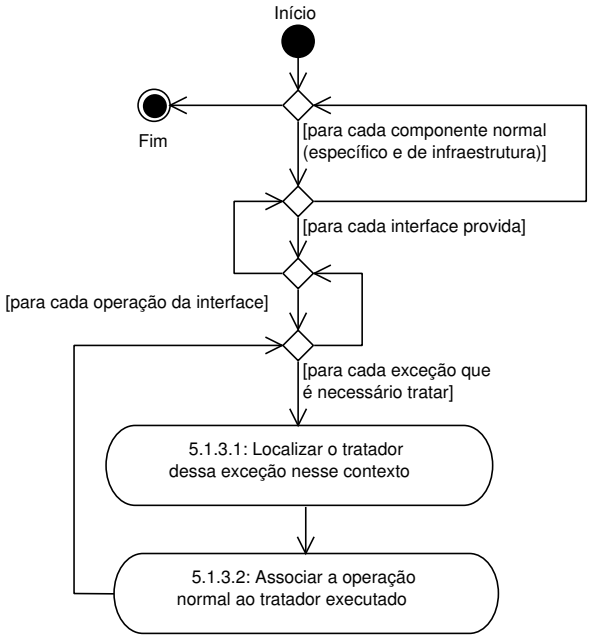

Figura 3.16: *Workflow* de Estruturação dos Componentes Arquiteturais do Sistema

ponente normal associado aos componentes excepcionais que oferecem os comportamentos dos tratadores necessários.

Refinar as Entidades Críticas e Definir a Rigorosidade do Método (ativ. 5.1.4)

A definição das entidades críticas feita inicialmente (Seção 3.2.2) foi produzida a partir de uma análise unicamente do ponto de vista do domínio do problema. Apesar de importante, para o desenvolvimento de sistemas que envolvem a possibilidade de perda de vidas humanas, essa relação de criticidade necessita ser revisada. Esse refinamento deve ser feito com base na lista de funcionalidades críticas, definida na Seção 3.3.2. Assim, a partir dos diagramas de atividades produzidos durante a identificação das operações de infra-estrutura (ativ 5.1.1), deve-se analisar todas as operações requeridas pelas operações classificadas como sendo funcionalidades críticas. Essas operações requeridas devem necessariamente ser consideradas igualmente críticas, uma vez que fazem parte da execução das funcionalidades vitais do sistema.

As definições de entidades, operações requeridas e funcionalidades críticas do sistema serão utilizadas para regular a rigorosidade do método MDCE+ em relação ao tratamento arquitetural de exceções, que envolve mais de um componente arquitetural do sistema. Por essa razão, nesse momento da especificação do sistema, o projetista deve classifi-

car a criticidade da disponibilidade e/ou confiabilidade[16] do sistema em uma das quatro categorias pré-definidas, baseadas na proposta de classificação de Gilbert [Gil03]: (i) criticidade extrema; (ii) criticidade alta; (iii) criticidade média; e (iv) criticidade baixa. Essa definição da criticidade influenciará diretamente as atividades das fases de implementação (Seção 3.7) e de integração dos componentes (Seção 3.8). Na fase de implementação, a interferência consistirá da necessidade de aquisição de novos componentes para possibilitar a redundância. Já na fase de integração dos componentes, durante a finalização da especificação dos conectores, deve-se definir a forma como os componentes poderão ser reconfigurados, de acordo com a classificação das falhas do sistema.

Sendo assim, dependendo da criticidade escolhida, o método se comportará diferentemente no que diz respeito aos componentes redundantes do sistema. Vale à pela ressaltar o fato de que a redundância de componentes afetará diretamente os aspectos de disponibilidade e confiabilidade[17]. A seguir, é mostrada a heurística adotada pelo método MDCE+ para selecionar os componentes que serão foco de redundância, de acordo com a sua categoria escolhida:

1. **Criticidade extrema.** É indicada para sistemas muito críticos com disponibilidade de investimentos altos. Nesse caso, deve-se considerar os componentes relacionados a todas as entidades, operações requeridas e funcionalidades críticas (componentes da aplicação e de infra-estrutura);

2. **Criticidade alta.** É indicada para sistemas muito críticos sem disponibilidade de investimentos altos. Nesse caso, deve-se considerar apenas os componentes relacionados às funcionalidades críticas do sistema (componentes da aplicação);

3. **Criticidade média.** É indicada para sistema menos críticos com a possibilidade de investimento nos aspectos de disponibilidade e/ou confiabilidade do sistema. Nesse caso, deve-se considerar apenas os componentes das entidades críticas (componentes de infra-estrutura);

4. **Criticidade baixa.** É indicada para sistema menos críticos sem a possibilidade de investimento nos aspectos de disponibilidade e/ou confiabilidade do sistema. Nesse caso, não haverá a preocupação com a replicação de componentes, a menos que o projetista defina explicitamente o seu desejo. Nesse caso, essa definição deve ser explícita e deve indicar quais componentes serão foco de replicação.

Após a definição da criticidade do aspecto de disponibilidade do sistema, se faz necessário definir o número de cópias desejado para cada um dos componentes com neces-

[16] do inglês *Reliability*
[17] do inglês *Reliability*

sidade de replicação. Por padrão, esse número é igual a um, isto é, se o componente foi destacado como foco de redundância, deve haver pelo menos uma cópia adicional sua.

Vale a pena ressaltar o fato de que no caso de ser possível dispor os componentes do sistema em locais distintos, essa redundância física possibilita uma garantia a mais, no sentido de prevenir eventuais falhas de comunicação [Jal94].

Como mostrado na Seção 2.2, a arquitetura de software é considerada o lugar ideal para implementar técnicas que aumentam a confiabilidade do sistema. Esse aumento da confiabilidade pode ser alcançado através da execução concorrente dos componentes redundantes. Em seguida, os resultados dessas execuções devem ser avaliados para em seguida decidir um valor satisfatório de retorno. Por envolver mais de um componente do sistema, a implementação dessas técnicas também é feita nos conectores arquiteturais, durante a materialização da configuração do sistema (Seção 3.8.2).

Analisar o Fluxo Excepcional (ativ. 5.1.5)

Para que o tratamento de exceções possa ocorrer de forma efetiva, é necessário considerar a propagação de exceções entre os componentes do sistema [IB01]. Por esse motivo, o método MDCE+ oferece uma atividade para analisar o fluxo excepcional. Os principais objetivos dessa análise são: (i) refinar a lista de exceções a serem tratadas de acordo com as exceções propagadas pelas operações requeridas de cada componente; e (ii) prevenir falhas de projeto, através da análise automática do fluxo excepcional;

Por se tratar de exceções propagadas entre componentes arquiteturais distintos, essas exceções são consideradas **exceções arquiteturais** e a utilidade da sua descoberta vai além da definição de novos contextos de tratamento. As exceções arquiteturais descobertas podem implicar em três atualizações importantes na especificação do sistema: (i) atualização da lista de exceções que devem ser tratadas por cada componente; (ii) alteração das assinaturas das operações, decorrente da possibilidade de uma nova propagação da exceção; e (iii) especificação dos conectores arquiteturais, baseado na possibilidade de conversões de tipos entre componentes arquiteturais distintos. Devido ao caráter recursivo do método MDCE+, a análise do fluxo excepcional pode ser feita tanto para os componentes normais, quanto para os excepcionais. Dessa forma é possível especificar as exceções a serem tratadas pelas operações dos tratadores excepcionais, no caso de alguma operação requerida para o tratamento propagar uma execução.

Durante o procedimento de análise do fluxo, é processado um componente de cada vez. Para cada exceção propagada por uma de suas operações requeridas, é verificada a necessidade de algum tratamento ou propagação no novo contexto da exceção. Devido à importância dessa atividade, que também interfere na especificação dos conectores arquiteturais, o seu *workflow* de execução é mostrado na Figura 3.17. Como pode ser visto nessa figura, se a exceção for proveniente de uma outra camada da arquitetura, pode ser ne-

cessário realizar uma conversão de tipos, a fim de possibilitar a compatibilidade entre dois modelos de falhas distintos. Um exemplo de uma conversão recomendada é a propagação de exceções inerentes à linguagem de programação. Como sugerido na Seção 2.4.3, essas exceções devem ser convertidas em exceções genéricas do modelo de falhas (exceções não declaradas). Essa conversão de tipos faz parte da especificação do conector arquitetural responsável por essa ligação. Mais detalhes sobre diretrizes para propagação de exceções podem ser vistos na versão detalhada do método [dSBR05].

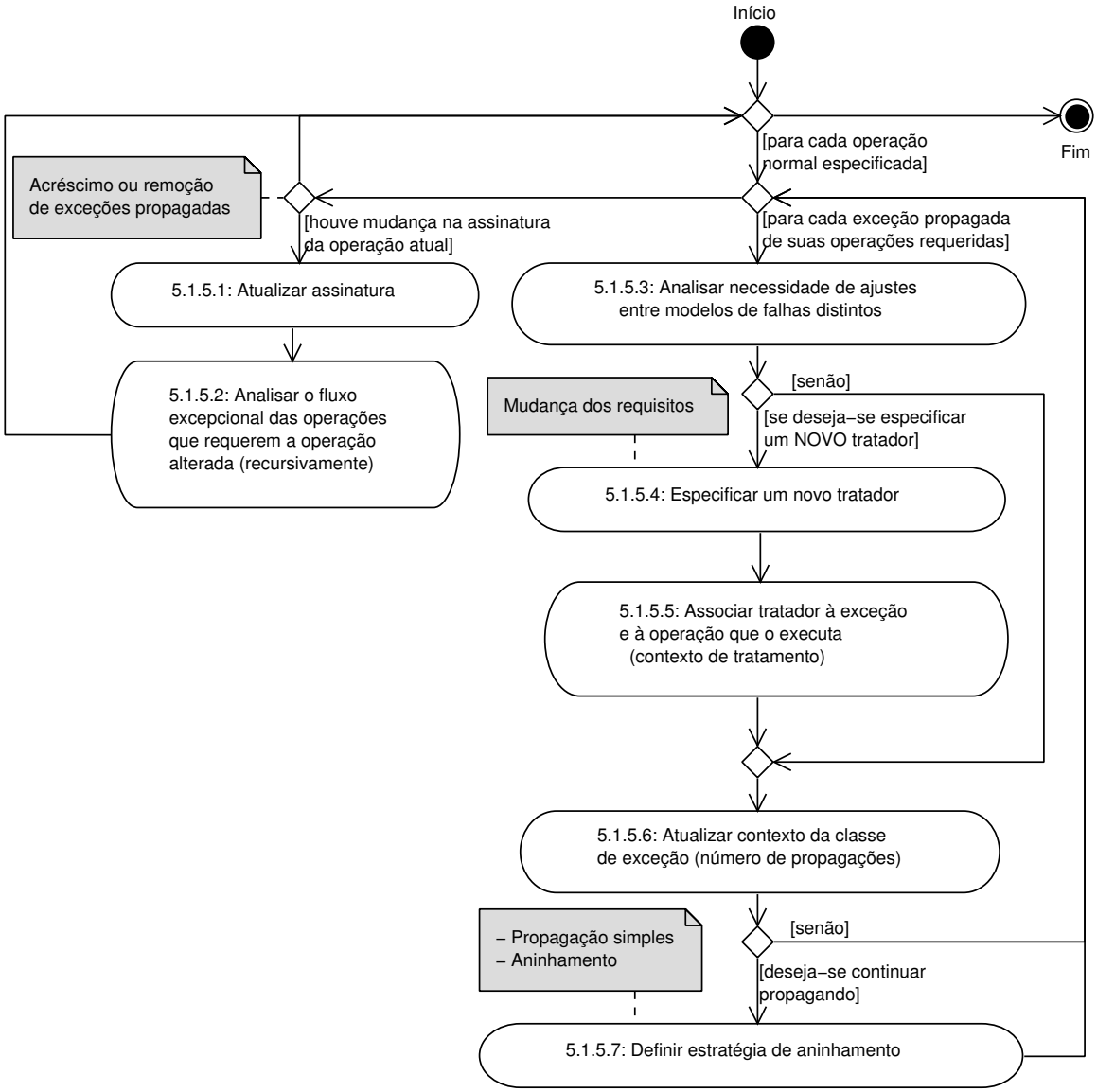

Figura 3.17: *Workflow* da Verificação de Tratamento ou Propagação

Após a especificação das possíveis traduções entre diferentes modelos de falhas, o

projetista deve analisar a necessidade de executar algum comportamento para tratar a exceção. Caso se julgue necessário, novos tratadores podem ser especificados através da atualização dos requisitos (cenários excepcionais), como apresentado na Seção 3.2.4. Vale a pena relembrar que a gravidade da falha, definida na Seção 3.2.5 pode auxiliar na especificação do tratamento, por exemplo, no caso de uma falha de gravidade média, um *re-try* poderia ser uma tentativa de solucionar o problema. Além disso, se a exceção tiver sido lançada de uma operação crítica que seja oferecida por mais de um componente (caso possua réplicas), como definido na sub-seção anterior, pode-se especificar algum tipo de reconfiguração para uma nova tentativa da execução. Mais uma vez, os tratamentos que envolvem *re-try* ou reconfiguração são considerados arquiteturais e por isso fazem parte da especificação dos conectores. Após a sua especificação, deve-se definir o contexto de tratamento, isto é, esses tratadores devem ser associados à exceção e à operação onde ele será executado. Em seguida, para se ter a informação da distância em relação ao contexto de tratamento inicial, é atualizado o número de propagações efetuadas desde a criação inicial da exceção.

Para finalizar a primeira análise do fluxo excepcional, o projetista deve decidir sobre a necessidade de continuar propagando a exceção. Caso se julgue necessário, como sugerido nas diretrizes do método (versão detalhada [dSBR05]), deve ser definida a estratégia de aninhamento[18] das exceções. Terminada a análise das propagações de todas as operações requeridas, é necessário verificar se houve mudança na assinatura do serviço. Neste caso, uma mudança de assinatura pode ser caracterizada tanto pelo acréscimo, quanto pela retirada de uma exceção propagada. Em caso de haver mudanças, a atividade 5.1.5 (analisar o fluxo excepcional) deve ser executada recursivamente para todas as operações que requerem a operação alterada.

Devido às várias atualizações da especificação durante as iterações do desenvolvimento, após analisar o fluxo excepcional dos componentes, se faz necessário verificar a existência de possíveis falhas de projeto, como por exemplo exceções lançadas e não tratadas, ou até mesmo o inverso: tratadores especificados para tratar exceções que nem sequer são lançadas. Porém, por ser considerada uma tarefa complexa e com um alto custo de tempo, a sua execução manual não é recomendada, uma vez que é grande a chance de ocorrência de erros humanos. Para automatizar essa tarefa, podem ser utilizadas ferramentas de análise do fluxo da arquitetura do sistema, tais como o *framework* Aereal [FdSBR05a], apresentado brevemente no ítem 3 da Seção 6.3, que é voltada para a análise do fluxo de exceções entre os componentes arquiteturais.

[18] do inglês *Nesting*

Revisar o Modelo de Falhas (ativ 5.1.6)

Revisando o modelo de falhas, o método MDCE+ visa analisar a possibilidade de simplificá-lo, a fim de adequar o método para o desenvolvimento de sistemas sem requisitos críticos de confiabilidade[19]. Essa revisão consiste basicamente do agrupamento de exceções em tipos mais genéricos (superclasses). Esse agrupamento pode se basear na existência de tratadores semelhantes em exceções semanticamente próximas, porém distintas. Por levar em consideração a semântica dos tipos excepcionais, espera-se reduzir o nível de especificacao do modelo de falhas sem perder todo o conhecimento da origem real do erro. Algumas diretrizes a respeito da criação de tipos genéricos de exceções podem ser consultadas na versão detalhada do método MDCE+ [dSBR05].

Uma observação importante do ponto de vista da implementação dos conectores, é que se cada um dos componentes for executado em uma *thread* de execução independente, o desenvolvedor deve estar apto a lidar com o lançamento concorrente de exceções. Por essa razão, além da simplificação do modelo de falhas, essa revisão deve abranger também a resolução de exceções lançadas concomitantemente (Seção 2.3.5). Outro motivo para o MDCE+ lidar com esse tipo de exceções é o fato da preocupação com o tratamento de exceções concorrentes ser presença obrigatória no desenvolvimento de sistemas robustos [XRR00].

Como visto na Seção 2.3.5 do Capítulo 2, uma maneira de se tratar o lançamento concorrente de exceções é através de grafos de resolução. Nesses grafos, enquanto os nós "filhos" representam as exceções simples que são lançadas, os nós "pais" representam as exceções equivalentes ao lançamento concorrente de alguma combinação de seus descendentes. A Figura 3.18 mostra o *workflow* de criação dos grafos de resolução de exceções concorrentes. Devido à necessidade de se conhecer o estado de vários componentes do sistema, a execução dessas resoluções é papel dos conectores arquiteturais e por isso faz parte da sua especificação.

Com o intuito de evitar um esforço muito alto, o método MDCE+ só especifica grafos de resolução para as funcionalidades críticas do sistema. Sendo assim, a Figura 3.18 mostra o procedimento executado para cada uma dessas operações. Antes de iniciar a criação do grafo, deve-se estipular o número máximo de exceções que possam ser lançadas concomitantemente. Após essa restrição, deve-se listar todas as exceções propagadas das operações requeridas. Cada uma dessas exceções é representada como um nó do grafo. Se for uma exceção simples (Seção 2.3.5), será um nó folha (extremidade do grafo). No caso das exceções compostas, deve-se associá-las a todas as exceções simples que deram origem a ela. No caso dessas exceções não estarem listadas, deve-se fazer isso nesse momento.

Em seguida, o projetista deve analisar e estipular quais são as combinações de exceções

[19] do inglês *Reliability*

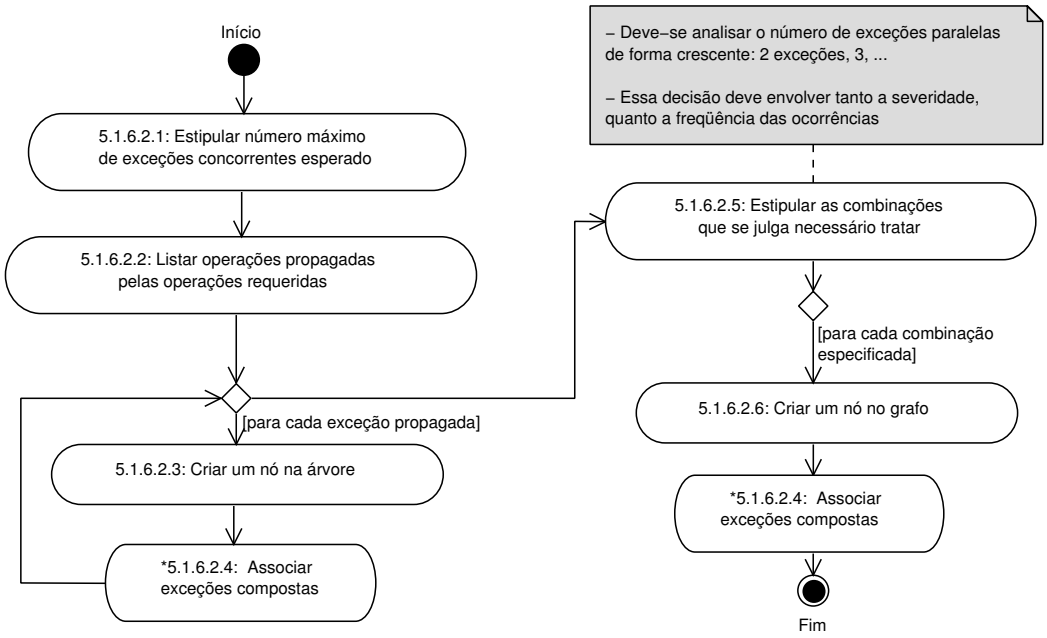

Figura 3.18: *Workflow* de Especificação do Grafo de Resolução

que ele julga necessário tratar. Essa decisão deve envolver tanto a severidade, quanto a freqüência das ocorrências. Cada uma dessas combinações deve ser representada como um nó "não-folha" do grafo, que possui como descendentes os nós relativos às exceções lançadas concorrentemente. A representação deve seguir a ordem crescente de cardinalidade de exceções. No caso dos nós que representem mais de duas exceções concomitantes, sempre que possível ele deve ser atribuído como pai dos outros nós que representem exceções compostas com o maior número de descendentes possível. Essa atribuição tem o objetivo de garantir a execução do algoritmo, que diz que a exceção equivalente é considerada o nó de maior grau do grafo (mais distante da raiz), que contiver todas as exceções lançadas como suas descendentes. Por fim, deve ser criada a raíz do grafo, isto é, uma exceção genérica que será lançada quando uma combinação não prevista acontecer.

A partir das exceções equivalentes, deve-se refinar os requisitos do sistema, a fim de especificar os cenários excepcionais do tratamento de cada uma delas. As exceções e tratadores serão especificados na próxima iteração do desenvolvimento. Seguindo a classificação proposta por Alessandro Garcia [GR01], as classes de exceções que representam algum tipo de concorrência (nós "não-folha") devem ser classificadas com o estereótipo $<< structured\_excepcion >>$.

Devido à explosão combinatória do número de exceções, em sistemas sem requisitos críticos de paralelismo, confiabilidade e robustez, a definição do grafo de resolução pode ser considerada opcional, exceto nas colaborações.

Com o modelo de falhas já refinado e estável, as exceções devem ser classificadas de acordo com a natureza da sua propagação: (i) exceções arquiteturais, quando são propagadas entre componentes arquiteturais do sistema; e (ii) exceções não-arquiteturais, quando são tratadas internamente ao componente arquitetural, ou no contexto do seu conector imediatamente superior, não fluindo até outro componente arquitetural.

Refinar Colaborações (ativ. 5.1.7)

Devido às características especiais atribuídas às colaborações, elas necessitam de uma atividade complementar para a finalização da sua especificação. Nesse momento, devem ser especificadas: (i) a condição de aceitação; e (ii) o encapsulamento dos participantes no coordenador.

Uma **condição de aceitação**, nada mais é do que uma condição de teste para julgar se o resultado da execução foi bem sucedido ou não [RX95]. A incapacidade de satisfazer essa condição implica necessariamente no lançamento de uma exceção interna à colaboração. Se apesar do insucesso da execução, os estados dos demais componentes arquiteturais do sistema não foram afetados, é lançada uma exceção de aborto, o que indica que a operação pode ser simplesmente desfeita pelos participantes da colaboração. Após isso, o coordenador avisa ao cliente da impossibilidade de oferecer o serviço, mas como o estado do sistema continua consistente, seria possível realizar uma nova tentativa ou utilizar um componente redundante.

Porém, se não existir uma garantia da consistência do estado do sistema, o coordenador deverá lançar uma exceção de defeito, com o intuito de avisar aos seus participantes que eles devem providenciar uma forma de parar a sua execução. Nesse caso, mais uma vez o coordenador necessita avisar ao componente cliente, que desta vez não poderá continuar a sua execução, já que a consistência do estado do sistema não é garantida.

Para finalizar a especificação das colaborações, é necessário associar os participantes ao respectivo coordenador responsável. Esse agrupamento é necessário, tendo em vista o papel centralizador desempenhado pelo coordenador dentro da colaboração. Como mostrado na Figura 3.19, para cada participante da colaboração, deve-se identificar seus serviços oferecidos (operações das interfaces providas). Essas operações devem também ser oferecidas pelo coordenador, a quem se destinarão inicialmente todas as requisições, seguindo o padrão de projeto *Facade* [GHJV95]. Dessa forma, através da interceptação das mensagens, o coordenador poderá por exemplo, bloquear a requisição de serviços até que a execução da colaboração termine. O bloqueio mostrado no exemplo é uma das formas de se evitar a propagação para outros componentes dos erros internos à colaboração. Sendo assim, é possível desfazer a execução (exceção de aborto), sem prejudicar a integridade do sistema como um todo.

Finalmente, como se deseja evitar o acesso direto aos componentes participantes de

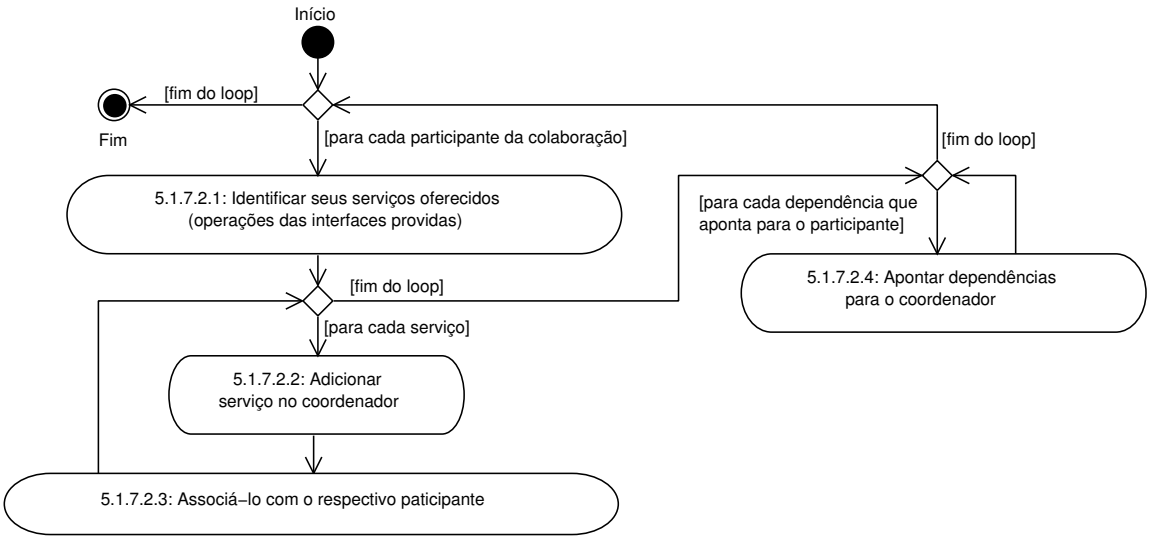

Figura 3.19: Encapsulamento dos Participantes Através do Coordenador

colaborações, todas as dependências para os serviços desses componentes devem ser substituídas por referências aos serviços equivalentes dos respectivos coordenadores. Sendo assim, o único componente que pode depender de operações de um participante é o seu coordenador.

Revisar Interfaces Providas (ativ. 5.1.8)

Para evitar a modelagem de um sistema com um número exagerado de interfaces simples ou com um número reduzido de interfaces muito complexas, o método MDCE+ oferece no final do projeto de cada *release*, a possibilidade de refatorar as interfaces providas pelos componentes normais e excepcionais do sistema. A refatoração de interfaces consiste basicamente no agrupamento ou desagrupamento de interfaces similares, que normalmente pertencem a um mesmo componente. Com isso, espera-se atingir um equilíbrio entre o número de interfaces e as suas complexidades.

Durante a identificação das interfaces (Seções 3.5.1, 3.5.2 e 3.5.3), elas já foram agrupadas de acordo com critérios de coesão. Neste momento, o projetista pode refinar essa formação inicial, através de outros agrupamentos ou até mesmo divisões das interfaces existentes. Nessa refatoração das interfaces, o projetista tem inclusive a liberdade de alterar os nomes das interfaces resultantes e dos seus componentes, de modo a generalizá-los ou especializá-los para oferecer os novos serviços sem degradar a coesão. Se durante essa revisão das interfaces providas do sistema, algum componente "perder" todas as suas interfaces, não faz sentido que ele continue a existir.

Existe uma outra diretriz que pode ser utilizada para saber quando agrupar inter-

faces da aplicação (derivadas dos casos de uso). Normalmente, as interfaces derivadas de casos de uso relacionados, quer seja através de << *include* >>, << *extend* >> ou generalização/especialização, são candidatas a serem agrupadas. Essa dica se torna praticamente uma regra nas situações onde algum dos casos de uso não se relaciona diretamente com atores. Nessas situações, as interfaces dos casos de uso que não possuem atores associados podem ser agrupadas à interface do outro caso de uso.

Um lembrete importante relacionado aos agrupamentos e desagrupamentos de interfaces: é necessário atualizar as dependências relativas a cada um dos serviços modificados. Em outras palavras, cada operação que depende de um serviço refatorado deve ser avisada da sua nova localização, se necessário.

Definir Interfaces Requeridas (ativ 5.1.9)

Antes de finalizar a fase de projeto, é necessário definir as interfaces requeridas de cada componente arquitetural do sistema. Um número muito reduzido de interfaces requeridas de um componente pode provocar um agrupamento de operações pouco coesas. Por outro lado, um número exagerado de interfaces requeridas aumenta consideravelmente o custo da implementação. Isso é agravado mais ainda no contexto do método MDCE+, que orienta a criação de um conector arquitetural para cada interface requerida (Seção 3.8). Assim, além do maior custo de implementação, ocorre um aumento no custo da integração dos componentes do sistema (Seção 3.8.2).

Tendo em vista essa situação de extremos, o método MDCE+ propõe uma solução intermediária, que implica na criação de uma interface requerida para cada camada da arquitetura com a qual o componente interage, inclusive a sua própria camada. Por exemplo, supondo que o sistema possua uma arquitetura em camadas como a mostrada na Figura 3.20. Um componente arquitetural A, que pertence à camada de sistema, requer quatro operações de outros componentes. Dessas quatro, duas são oferecidas por componentes da própria camada de sistema e as outras duas por componentes da camada de negócios. Neste caso, o componente A teria duas interfaces requeridas, uma para cada camada (Figura 3.21).

Para não perder a informação de quais os componentes que oferecem cada uma das interfaces requeridas, é definido um relacionamento de dependência entre cada operação das interfaces requeridas e as respectivas operações providas dos outros componentes (requeridas dependem das providas). Lembrando que no caso de haver componentes redundantes, as operações requeridas devem referenciar cada uma delas. Essa informação pode ser representada através de tabelas, que facilitarão o rastreamento da informação nos dois sentidos: (i) saber quem oferece alguma operação requerida; e (ii) saber quem depende de alguma operação provida.

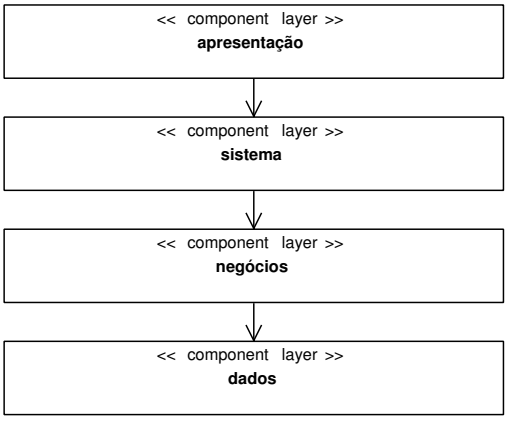

Figura 3.20: Uma Arquitetura em Quatro Camadas

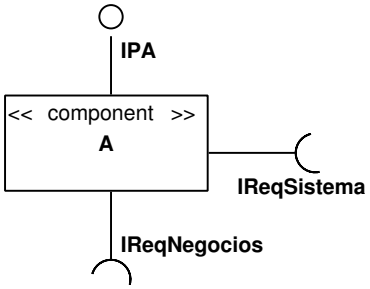

Figura 3.21: Exemplo do Componente A

3.6.2 Formalização da Especificação dos Componentes

Com os componentes do sistema já especificados, essa fase consiste no refinamento das suas especificações através da formalização das assertivas do componente. Por se tratar de uma etapa custosa do ponto de vista do tempo de desenvolvimento, essa atividade é considerada opcional. Apesar de opcional, o método MDCE+ recomenda a sua execução nos casos em que esse tipo de formalismo possa ser utilizado para automatizar alguma etapa posterior do desenvolvimento. Um exemplo dessa automação pode ser a geração de casos de testes [dSBRCF+05], que no contexto de sistemas críticos é essencial.

Existem várias linguagens e notações que podem ser utilizadas para formalizar as assertivas, como por exemplo *statecharts* [Har87] e OCL [EFLR99]. O método MDCE+ recomenda a utilização da linguagem OCL[20], que é a linguagem formal adotada pela UML. Em OCL, as pré-condições e invariantes podem ser escritas como expressões simples. Já as pós-condições, por poderem representar os vários retornos possíveis de um serviço,

[20]do inglês *Object Constraint Language*

devem ser compostos pelo valor de retorno e pela respectiva condição necessária para que ele seja retornado.

3.7 FASE 6: Materialização dos Componentes

Neste ponto do desenvolvimento, os componentes que implementam a *release* atual já estão especificados. Após essa especificação, o próximo passo é materializar cada um desses componentes, sejam eles normais ou excepcionais. Vale a pena salientar que de acordo com a rigorosidade do método em relação aos aspectos de disponibilidade do sistema[21], que foi definida na Seção 3.6.1, pode ser necessário providenciar mais de um componente que implemente a mesma especificação.

O método MDCE+ define três formas de se materializar um componente: (i) reutilização de um componente já utilizado pela empresa; (ii) aquisição do componente a partir de um catálogo de terceiros; e (iii) implementação do componente. Mas antes de escolher a forma mais adequada para a materialização de cada componente do sistema, é necessário executar algumas atividades gerais, que são comuns às três abordagens. Essas atividades são mostradas na Seção 3.7.1. Na seqüência, a Seção 3.7.2 mostra os critérios que devem ser seguidos para a escolha da forma mais adequada de se materializar o componente. Finalmente, as Seções 3.7.3 e 3.7.4 mostram as atividades específicas, de acordo com a forma de materialização adotada.

Após a materialização dos componentes individuais (normais e excepcionais isolados), dependendo do método de testes adotado, pode-se proceder atividades de teste unitário [Bin99], que visam a verificação da conformidade das funcionalidades de cada componente isoladamente. Adiante, no início da fase de integração dos componentes (Seção 3.8), será possível realizar essa mesma verificação para os componentes arquiteturais do sistema, já estruturados segundo o modelo do componente tolerante a falhas ideal.

3.7.1 Atividades Comuns (reutilização ou implementação)

Independente da forma como os componentes serão materializados, os contratos estabelecidos entre eles durante a especificação do sistema devem ser mantidos. Essa necessidade de se satisfazer os contratos torna obrigatória a especificação dos tipos de dados[22] dos parâmetros e do retorno, presentes na assinatura especificada durante a análise interativa entre os componentes (Seção 3.6.1). Além disso, devem ser definidas as classes das exceções arquiteturais, que podem fluir entre componentes distintos do sistema. Conforme

[21] do inglês *System availability*
[22] do inglês *Data types*

definido na análise das propagações excepcionais (Seção 3.6.1), essas exceções devem ser explicitadas nas assinaturas dos métodos. A Figura 3.22 mostra as atividades executadas nessa etapa.

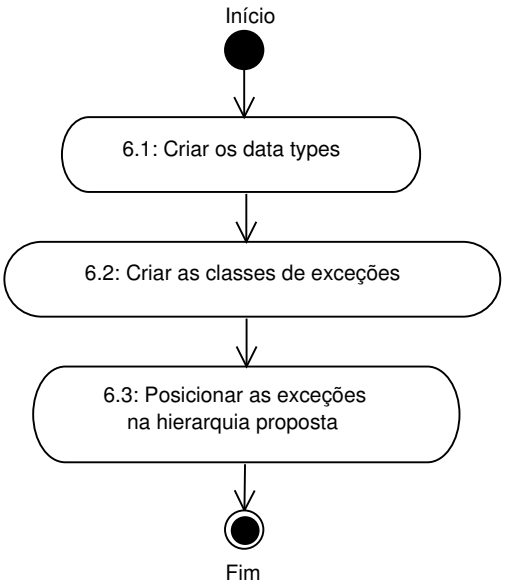

Figura 3.22: Atividades Comuns (reutilização e implementação)

A criação dos tipos de dados utilizados pelo componente consiste de um procedimento simples. Para isso, deve-se analisar a assinatura de cada um dos serviços oferecidos pelo componente. Em seguida, para cada tipo de dado utilizado, deve-se analisar a possibilidade de criação ou atualização da sua classe correspondente. Essa atualização pode ser necessária no contexto de refinamentos feitos em iterações posteriores do desenvolvimento. A classe que representa a estrutura de dados deve possuir os atributos relativos aos seus campos especificados. Com o intuito de conciliar a facilidade de implementação e o encapsulamento da informação, caso a linguagem ofereça esse recurso, a visibilidade dos atributos dos *data types* deve ser definida como públicas para leitura, mas na ausência desse tipo, deve ser considerada pública. Esse "relaxamento" visa facilitar a implementação dos conectores, que realizam conversões de tipos constantemente. Em relação à visibilidade da classe, ela deve ser definida como visibilidade de pacote, uma vez que a sua instanciação deve ocorrer por intermédio da classe `DatatypeFactory`, que por sua vez deve ser visível publicamente.

Da mesma forma que aconteceu com os tipos de dados, mesmo que um componente seja reutilizado, é necessário especificar as classes das exceções que cada um dos serviços do componente pode propagar. Deve-se atentar à necessidade de se implementar os atributos relativos às informações contextuais de cada exceção, conforme especificado durante o

processamento das exceções agendadas (Seção 3.5.1) e refinado durante a análise das propagações excepcionais, feita na Seção 3.6.1. Diferentemente do caso dos *data types*, esses atributos não podem possuir visibilidade de pacote, uma vez que essas classes terão seus atributos consultados constantemente pelos seus tratadores, que são implementados fora do componente. Assim, na impossibilidade de se especificar a visibilidade como sendo pública para leitura, ela deve ser considerada pública.

Em relação ao procedimento de atualização do contexto, normalmente ele é efetuado no momento do lançamento da exceção e nos conectores arquiteturais, que fazem o mapeamento entre os diferentes modelos de falhas dos sub-sistemas (Seção 3.8.2). Com essa abordagem adotada pelo método, as informações contextuais acompanham os próprios objetos das exceções. Assim, elas podem ser acessadas e atualizadas através da interface pública dessas classes.

Independentemente da visibilidade de seus atributos, as classes de exceções devem possuir visibilidade de pacote. Isso acontece porque elas são posicionadas em um pacote `excep`, interno ao pacote `impl` do COSMOS (Figura 3.23). Da mesma forma como acontece com as classes de implementação, o acesso às exceções deve acontecer através de um *factory*, denominado `ExceptionFactory`.

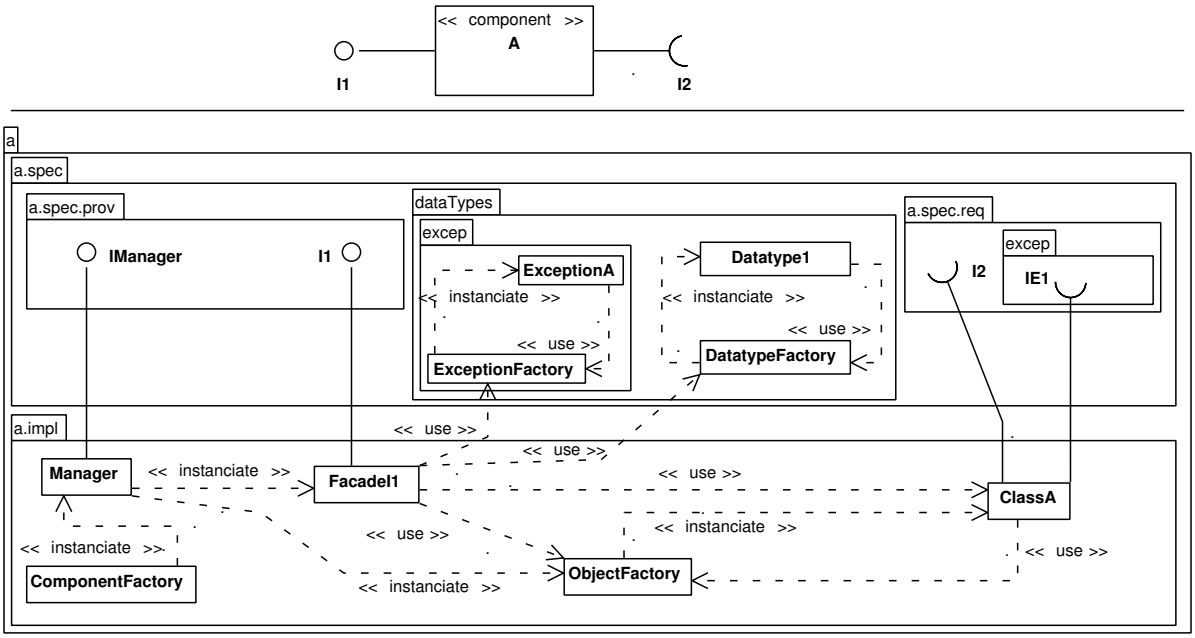

Figura 3.23: Novo Modelo de Implementação do COSMOS

Após a criação da classe e a atualização da `ExceptionFactory`, cada exceção deve ser posicionada na hierarquia adotada pelo método MDCE+. Para possibilitar a estruturação de exceções simples e compostas, o método MDCE+ propõe uma hierarquia excepcional.

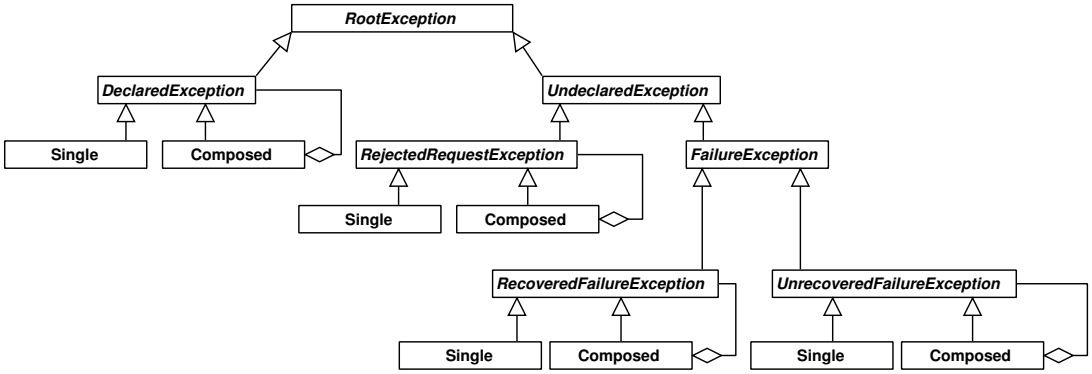

Figura 3.24: Modelagem de Classes de Exceções Simples e Compostas

Essa hierarquia, mostrada na Figura 3.24, concilia a sistemática de classificação dos tipos excepcionais apresentada na Seção 2.4.1, com a possibilidade de declarar exceções compostas através do padrão de projeto *Composite* [GHJV95].

Os principais benefícios decorrentes da utilização da hierarquia excepcional sugerida são:

- **Uniformidade.** Graças à estruturação mostrada na Figura 3.24, as exceções simples e compostas são especificadas de maneira uniforme.
- **Simplicidade.** Os grafos de resolução são definidas mais facilmente.
- **Reusabilidade e Extensibilidade.** A representação de exceções simples e compostas na forma de classes proporcionam a reutilização dos grafos de resolução em várias colaborações e/ou conectores.
- **Legibilidade e Manutenibilidade.** A representação de exceções através de objetos é mais fácil de entender. Principalmente quando comparados com outras representações, como por exemplo a representação através de $flags$, onde o valor de retorno pode representar um significado normal ou excepcional [GRRX01].

3.7.2 Escolher a Forma de Materialização (ativ. 6.4)

As duas primeiras formas de reutilização mostradas envolvem de alguma forma a reutilização de componentes prontos, que não necessariamente obedecem aos mesmos contratos especificados para o componente. Por esse motivo, nesses casos pode ser necessário adaptar o componente reutilizado, a fim de satisfazer os contratos estabelecidos na especificação. Já no terceiro caso, deve ser especificado um modelo de componentes que possa ser implementado utilizando os recursos presentes nas linguagens de programação atuais.

Para maximizar a redução do tempo e do custo de desenvolvimento, é necessário priorizar a reutilização de componentes durante a fase de materialização. Dessa forma, antes de mais nada é necessário analisar a possibilidade de reutilizar algum componente já existente que satisfaça o maior subconjunto possível dos serviços oferecidos pelo componente. Inicialmente, essa atividade consiste basicamente da busca em repositórios internos da empresa, à procura de componentes já utilizados em outros projetos. Por representarem características comuns ao domínio do negócio, os componentes de infra-estrutura são os mais propícios para serem reutilizados. Além disso, por estarem intimamente relacionados ao domínio, a busca desses componentes pode ser otimizada através da adição de informações extras, como por exemplo a entidade do modelo do domínio que eles se relacionam, além da tradicional lista de serviços que se espera que ele implemente.

Após a identificação dos componentes que já foram utilizados no domínio da empresa, a busca pode ser estendida para catálogos externos de componentes. Esses catálogos são constituídos de componentes de softwares que podem ser adquiridos de terceiros, quer seja através de uma compra, quer seja através da adoção de softwares livres (*open source* ou *freeware*). Nesses casos, a busca de componentes nem sempre pode levar em consideração as entidades do domínio, uma vez que além das particularidades próprias de cada empresa, não se pode esperar uma padronização na representação dos modelos.

Com a lista dos possíveis candidatos a serem reutilizados, deve-se proceder a escolha do modo como o componente será materializado. Essa escolha consiste basicamente numa análise do custo/benefício entre as três formas existentes. A análise de custos deve abranger limitações de tempo e recursos. Esses custos podem ser calculados a partir das variáveis mostradas na Tabela 3.6. Outras vantagens, tais como a disponibilidade de código e o oferecimento de serviços extra, como garantia e manutenção também devem ser levados em consideração.

Tabela 3.5: Custos das Formas de Materialização

Modo de Materialização	Custo
Reutilização Interna	Custo da integração de componentes (se existe mais de um componente) + custo da adaptação + custo da materialização dos outros serviços (se não foram materializados todos).
Aquisição Externa	O mesmo da reutilização + o custo da aquisição.
Implementação	Estimativa do custo da implementação dos componentes. Podem ser utilizadas técnicas tradicionais, tais como pontos por função e estimativas baseadas em casos casos de uso.

O restante do desenvolvimento é dependente da forma como o componente será materializado. As seções seguintes mostram as atividades necessárias para cada uma das opções disponíveis.

3.7.3 Reutilizar Componentes (ativ. 6.5)

Se a equipe de desenvolvimento decidir por uma das formas de reutilização, pode ser necessário construir adaptadores, cujo objetivo é mascarar a verdadeira interface do componente reutilizado. Dessa forma, busca-se tornar transparente aos seus clientes a maneira como ele foi materializado. Em outras palavras, devem ser feitas adaptações entre as interfaces reais do componente e as interfaces especificadas para o sistema em questão.

Devido à maneira como as mensagens enviadas e retornadas do componente reutilizado são interceptadas pelos adaptadores, eles exercem um papel de invólucro[23] [KJ97b]. A Figura 3.25 ilustra a estrutura de um *wrapper* que envolve dois componentes. As atividades de adaptação executadas pelos *wrappers* consistem basicamente de conversões de tipos entre os parâmetros de entrada e as estruturas de retorno, incluindo os retornos excepcionais. Dessa forma, as requisições de serviços devem chegar ao *wrapper* seguindo o contrato especificado. Em seguida a classe `InputAdapter` adapta os tipos dos parâmetros e repassa a solicitação ao componente propriamente dito. Ao receber a resposta da solicitação, acontece um processo semelhante, mas no sentido inverso.

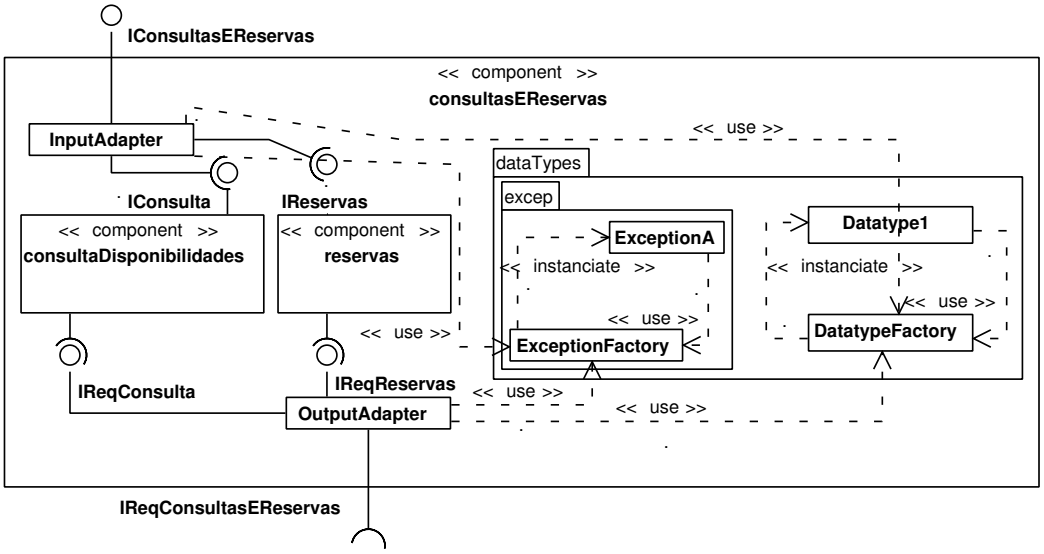

Figura 3.25: Ilustração do Papel do *Wrapper*

Devido às diversas exceções que o componente reutilizado pode lançar, ao receber um retorno excepcional, o *wrapper* deve analisar a necessidade de conversão para algum tipo excepcional equivalente do modelo de falhas especificado para o sistema. Conforme proposto por Asterio et al. [dCGFPR04] e apresentado na Seção 2.4.2, nesse processo de

[23] do inglês *Wrapper*

adaptação, as exceções lançadas que não sejam de um dos tipos declarados na especificação das suas interfaces são convertidas para um subtipo de UndeclaredException na hierarquia adotada.

Além desse papel conciliador entre as duas especificações, o *wrapper*, também assume o papel do interceptador na abordagem de tratamento de exceções intra-componente, apresentada na Seção 2.4.2. Dessa forma, ele também é responsável por detectar novas exceções, desempenhando uma função semelhante aos componentes sentinelas[24], propostos por Dellarocas [Del98]. Essa detecção de exceções acontece a partir das condições excepcionais especificadas, que se baseiam na violação das pré- e pós-condições das operações. Ao descobrir novas exceções que podem ser detectadas pelos *wrappers*, deve-se criar uma interface que contenha as dependências dos tratadores que serão executados para os respectivos tratamentos. A ligação entre os componentes normais e os excepcionais acontecerá na primeira etapa da fase de integração dos componentes, onde são montados os componentes arquiteturais do sistema (Seção 3.8.1). Nessa etapa, a interface de dependências excepcionais será implementada por conectores internos que desempenharão os papéis dos tratadores de fronteira (BLE), apresentados na Seção 2.4.2.

Assim como as exceções que são propagadas pelo componente, para cada uma das exceções detectadas deve ser criada uma classe que encapsule suas informações contextuais. Essa classe também deve ser posicionada na hierarquia de exceções apresentada na Figura 3.24.

3.7.4 Implementar Componentes Novos (ativ. 6.6)

Na ausência de componentes prontos que motivem a reutilização, os componentes devem ser implementados de acordo com a especificação produzida no decorrer do desenvolvimento. Devido à inexistência de linguagens de programação que utilizem o paradigma do desenvolvimento baseado em componentes, o primeiro passo para a criação de um novo componente é escolher um modelo que possibilite o mapeamento das suas abstrações para as estruturas das linguagens de programação atuais. Alguns exemplos de modelos de componentes existentes na literatura são: EJB [MH99], DCOM [Ses98], CCM [MM00] e COSMOS (Seção 2.2.2). Após a escolha do modelo, deve-se adaptar a especificação de acordo com as suas restrições. Um exemplo comum de uma restrição a ser verificada é a impossibilidade de um componente oferecer mais de uma interface. Finalmente, conhecendo-se as limitações do modelo adotado, o projetista deve procurar formas de contorná-las. No exemplo da inviabilidade de se oferecer mais de uma interface, poderia ser criada uma nova interface a ser oferecida pelo componente. Em seguida, essa interface deveria herdar de todas as interfaces providas especificadas.

[24] do inglês *Sentinel components*

Entre os modelos já citados, o método MDCE+ sugere particularmente a adoção do COSMOS. Como pode ser visto na Figura 3.26, a implementação de um componente COSMOS consiste basicamente de quatro passos: (i) criar a estrutura de pacotes mostrada na Figura 3.23; (ii) criar as interfaces e classes definidas no COSMOS para o gerenciamento de meta-informação (Seção 2.2.2, Tabela 2.2); (iii) criar as classes `Facade` que materializarão as interfaces providas do componente e (iv) especificar as classes que implementam os serviços do componente (utilizadas pelos `Facade`s). Essa especificação pode ser feita utilizando qualquer processo orientado a objetos existente, como por exemplo o RUP[25] [JBR99], da IBM.

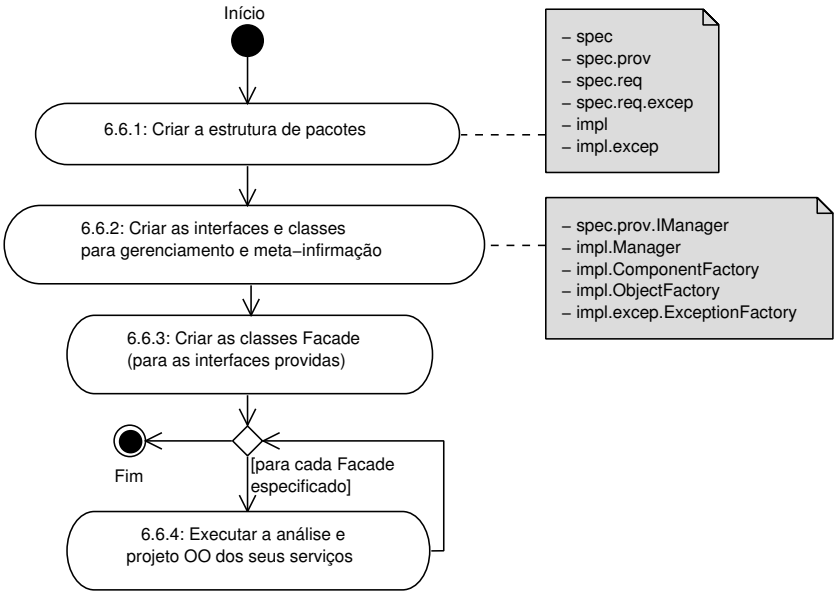

Figura 3.26: Implementação de um Componente no COSMOS

Após a especificação do comportamento normal do sistema, deve-se especificar os três níveis de tratamento de exceções intra componente proposto por Asterio et al. e apresentado na Seção 2.4.2. Dessa forma, a interface excepcional requerida já especificada deve ser dividida em três: (i) tratadores requeridos pelas classes de fachada, que representa o tratador de fronteira `ProvidedBLE`; (ii) tratadores requeridos pelas classes de implementação, que representa o tratador da aplicação `ALE`; e (iii) tratadores das exceções propagadas pelas operações requeridas, que representa o tratador de fronteira `RequiredBLE`. Assim como no caso dos componentes reutilizados, essas interfaces darão origem aos conectores internos que constituirão os componentes arquiteturais do sistema (Seção 3.8.1).

Dependendo da complexidade dos tratadores do sistema, o método MDCE+ define duas formas de se implementar os componentes excepcionais do sistema: (i) como um

[25]do inglês *IBM/Rational Unified Process*

componente, no caso de sistemas confiáveis, onde os tratamentos costumam ser mais complexos; ou (ii) como uma classe, no caso de sistemas sem requisitos críticos de confiabilidade, que normalmente possuem tratadores simples.

Em relação aos componentes colaboradores, eles são implementados como *wrappers* que envolvem os participantes e intermediam o acesso a eles. Numa implementação concorrente, cada participante deve ser executado em uma *thread* distinta dos demais. A execução dessas *threads* é coordenada e sincronizada pelo próprio coordenador. A Figura 3.25, que mostra a estruturação de um *wrapper* pode ser utilizada para exemplificar uma colaboração, onde os componentes reutilizados assumem o papel de participantes e o *wrapper* assume o papel de coordenador.

3.8 FASE 7: Integração dos Componentes do Sistema

Apesar de nesse ponto do desenvolvimento os componentes da *release* atual já estarem disponíveis para serem utilizados, esses componentes isoladamente não são capazes de oferecer os seus serviços especificados. Para que isso seja possível, é necessário indicar para cada um dos componentes, quem supre as suas dependências. Essa ligação entre as interfaces requeridas de um componente e as respectivas interfaces providas de outro é feita através dos conectores, que conseqüentemente são dependentes dos componentes envolvidos.

O método MDCE+ divide a fase de integração dos componentes em duas etapas conceitualmente distintas: (i) montar os componentes arquiteturais, onde é realizada a ligação entre os componentes normais e excepcionais; e (ii) materializar a configuração do sistema, onde é realizada a ligação entre os componentes arquiteturais do sistema e a implementação do programa principal. As seções seguintes detalham cada uma dessas etapas.

3.8.1 Montar os Componentes Arquiteturais (ativ. 7.1)

Como discutido anteriormente, a fase de montagem dos componentes arquiteturais do sistema consiste basicamente na estruturação dos componentes tolerantes a falhas ideais (Seção 2.3.7). Para isso, para cada componente do sistema que possui alguma dependência excepcional (interface requerida excepcional), o método MDCE+ prevê a criação de conectores especiais que realizem essa ligação. Devido ao caráter interno desses conectores em relação ao componente arquitetural do sistema, eles são denominados conectores internos.

Do ponto de vista da abordagem de tratamento de exceções intra componente apresentada na Seção 2.4.2, os conectores internos materializam os três níveis de tratamento: de fronteira (ProvidedBLE e RequiredBLE) e de aplicação (ALE), uma vez que para o

componente, são eles que oferecem os seus tratadores. Deve ser criado um conector interno para cada nível de tratamento presente no componente: um conector ProvidedBLE, um RequiredBLE e um conector ALE, onde cada um implementa a sua interface correspondente, que foi definida durante a materialização do componente (Seção 3.7). No caso dos componentes reutilizados, apenas as interfaces BLEs podem estar presentes. Já no caso dos componentes novos, pode existir uma interface ProvidedBLE, uma ALE, uma RequiredBLE ou todas, conforme a natureza de detecção da exceção (Seção 2.4.2).

Do ponto de vista prático, cada conector interno é implementado como uma classe simples. Essas classes, por sua vez, devem materializar[26] a sua respectiva interface excepcional requerida, de acordo com o nível de tratamento de exceções que o conector se refere. A Figura 3.27 mostra a estrutura de um componente arquitetural que possui os três níveis de tratamento excepcional. Em seguida, a Figura 3.28 mostra o código relativo a parte da implementação em Java do conector interno ALE. Nesse código, o trecho sublinhado destaca a implementação da interface provida. Os atributos das classes, por sua vez, representam as interfaces providas dos componentes excepcionais dos quais o conector depende.

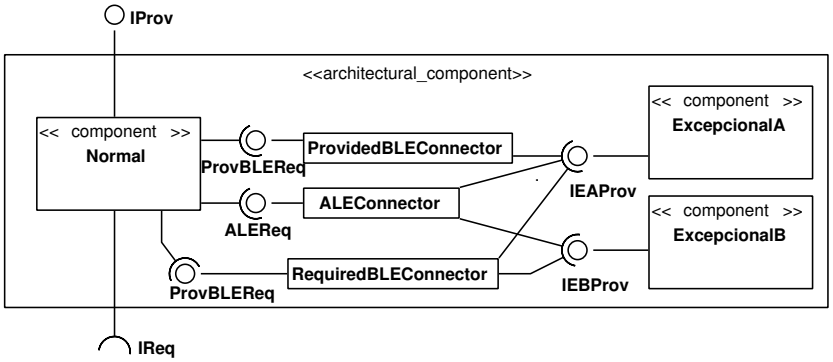

Figura 3.27: Estrutura do Componente Ideal Projetado

```
1 public class ALEConnector implements ALEReq {
2     IEAProv tratadorA;
3     IEBProv tratadorB;
4     ...
5 }
```

Figura 3.28: Implementação do Conector Interno ALE

Após a implementação dos conectores internos, deve-se proceder a criação dos componentes tolerantes a falhas ideais. Para isso, é necessário implementar o método do sistema

[26] do inglês *Realize*

que será responsável pela montagem dos componentes arquiteturais. Em resumo, esse método deve instanciar o componente normal, os excepcionais e os conectores internos. Em seguida, deve realizar a ligação desses componentes através dos conectores. Por se tratar de apenas uma classe, a ligação entre os componentes excepcionais e os conectores internos é feita através da passagem das referências ao construtor da própria classe. Em relação aos componentes do sistema, as dependências são supridas através da execução da operação `setRequiredInterface(String nome, Object facade)`, da interface IManager do COSMOS. A Figura 3.29 mostra um exemplo desse procedimento de ligação entre o componente normal `CN` e o componente excepcional `CE`, através do conector interno `ConInt`.

```
1     ...
2 CN.spec.prov.IManager cn = CN.impl.ComponentFactory.createInstance();
3 CE.spec.prov.IManager ce = CE.impl.ComponentFactory.createInstance();
4 ConInt conInt = new ConInt(ce.getProvidedInterface(''IPCE''));
5 cn.setRequiredInterface(conInt);
```

Figura 3.29: Exemplo em Java da Ligação entre dois Componentes

Por apresentar uma visão integrada dos componentes normal e excepcional, a partir desse momento é possível realizar testes de unidade relativos aos componentes arquiteturais do sistema, isto é, testar os componentes tolerantes a falhas ideais como unidades de implementação.

3.8.2 Materializar a Configuração do Sistema (ativ. 7.2)

Apesar das dependências excepcionais terem sido supridas na atividade anterior de montagem dos componentes arquiteturais (Seção 3.8.1), ainda nos falta materializar as conexões entre esses componentes do sistema. Essas conexões são realizadas através de ligações entre as interfaces requeridas normais de um componente e as respectivas interfaces providas de outros. Dessa forma, a materialização das configurações do sistema consiste de três atividades: (i) finalizar a especificação dos conectores arquiteturais; (ii) implementar esses conectores; e (iii) implementar o programa principal.

Por representar o elo de comunicação entre componentes arquiteturais, os conectores são os únicos componentes do sistema que possuem o conhecimento dos fluxos interativos entre os componentes. Dessa forma, como discutido no decorrer desse capítulo, eles são considerados os locais ideais para a implementação dos requisitos de qualidade do sistema, tais como tolerância a falhas e disponibilidade. Boa parte da especificação dos conectores foi definida durante a análise do aspecto interativo entre os componentes (Seção 3.6.1), mas dada a sua importância, ele ainda deve ser refinado.

No contexto específico do desenvolvimento de sistemas robustos, a especificação dos conectores deve ser refinada para oferecer o tratamento inter componentes, proposto por Asterio et al. (Seção 2.4.3). Isso é necessário, uma vez que no desenvolvimento desses sistemas, deve-se supor a ocorrência de exceções não previstas na especificação. Para essas exceções não antecipadas, o desenvolvedor deve seguir as diretrizes de conversões excepcionais mostradas na Tabela 2.4 para definir exceções que representem esses novos pontos de saída excepcionais.

Além disso, o fato dos *data types* ficarem dentro do componente e a estrutura de pacotes fazer parte do nome da classe, ainda que dois *data types* possuam nome e estrutura semelhantes, é necessário definir as conversões entre eles, através da instanciação do tipo exato de cada componente. Finalmente, durante o refinamento de suas especificações, os conectores devem adequar o seu comportamento à rigorosidade do método em relação às exceções arquiteturais. Essa rigorosidade, que foi definida na Seção 3.6.1 (ativ. 5.1.4), diz respeito tanto às reconfigurações dos componentes do sistema, quanto à execução redundante de componentes.

Dessa forma, nos casos de reconfiguração, se houver mais de uma implementação disponível de algum componente, deve-se escolher a ordem de cada uma na reconfiguração. Além disso, a forma como os componentes poderão ser reconfigurados pode variar de acordo com a classificação da falha da exceção, definida na identificação de exceções, durante a definição dos requisitos do sistema (Seção 3.2.4). O método MDCE+ define dois tipos de reconfiguração: (i) permanente; e (ii) temporária.

Nas **reconfigurações permanentes**, as réplicas assumem o lugar do componente falho de forma definitiva, isto é, nas próximas requisições desse serviço, o conector chamará diretamente a última réplica que funcionou corretamente. Já no caso de uma **reconfiguração temporária**, as réplicas só desempenham o papel do componente principal durante uma execução específica. Dessa forma, nas próximas requisições, o conector confiará a execução do serviço ao componente principal, independentemente dele ter falhado na tentativa anterior. As Figuras 3.30 e 3.31 mostram respectivamente os diagramas de seqüência relativos aos dois tipos de reconfiguração.

Vale a pena ressaltar o fato de que mesmo que não exista mais de uma implementação do componente disponível, no caso de haver falhas com gravidade alta, o conector deve oferecer o tratamento arquitetural do sistema a partir da re-instanciação do componente seguida de uma nova tentativa de execução[27].

Outra observação importante, do ponto de vista do refinamento da especificação dos conectores é que no caso de sistemas com requisitos críticos de confiabilidade[28], a especificação dos conectores deve prever a implementação dos mecanismos de arbitragem, que

[27] do inglês *Re-try*
[28] do inglês *Reliability*

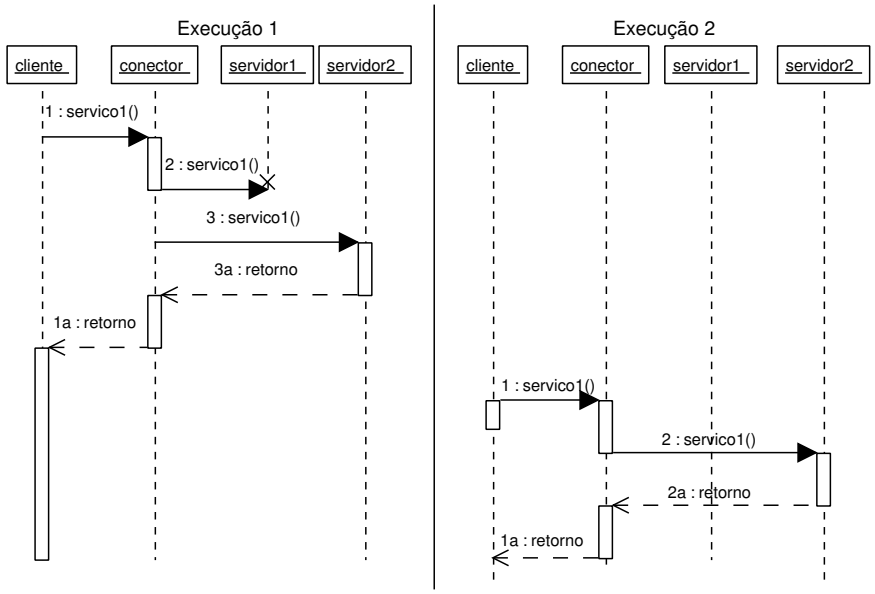

Figura 3.30: Simulação de uma Reconfiguração Permanente do Sistema

comparam os resultados das execuções replicadas e decidem qual será o valor final a ser retornado.

Após a finalização da especificação dos conectores do sistema, deve-se prosseguir com a sua implementação, conforme sugerido pelo COSMOS no seu modelo de conectores (Seção 2.2.2). De forma análoga à montagem dos componentes arquiteturais do sistema, após a implementação dos conectores arquiteturais, é necessário implementar o método responsável pela integração do sistema propriamente dito, a fim de ligar os componentes nos conectores, de acordo com as dependências de cada um. Isso também é feito através da execução das operações `setRequiredInterface(String nome, Object facade)`, das interfaces IManager de cada um dos componentes COSMOS.

Do ponto de vista da dependência entre os métodos executados pelo programa principal, primeiro devem ser montados os componentes arquiteturais do sistema, que foram especificados na Seção 3.8.1. Só então esses componentes podem ser integrados para constituir o sistema propriamente dito.

3.9 Materialização da Arquitetura Confiável Adotada

Em relação à arquitetura tolerante a falhas mostrada na Seção 2.5, sua importância está na definição clara dos papéis necessários em um sistema robusto e confiável[29]. Esta seção

[29] do inglês *Dependable*

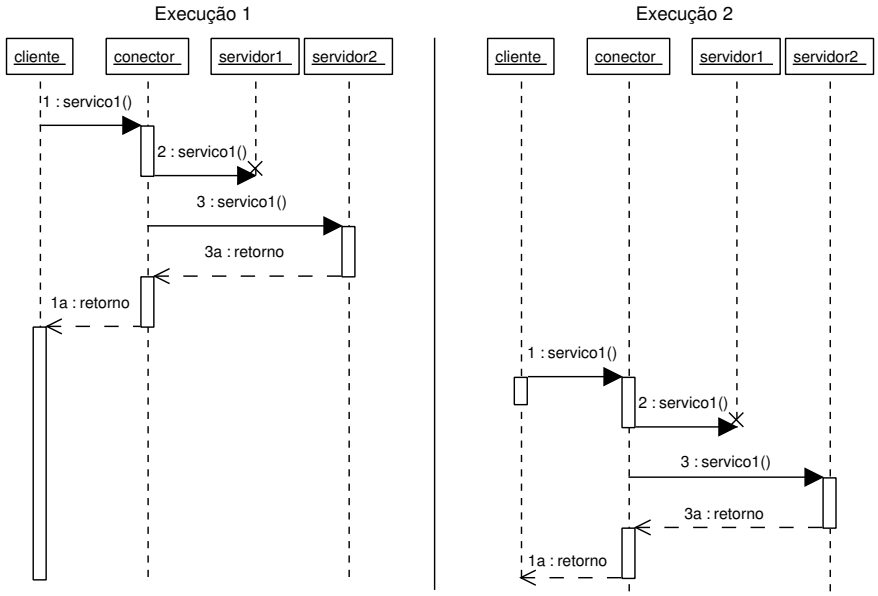

Figura 3.31: Simulação de uma Reconfiguração Temporária do Sistema

faz um mapeamento entre esses papéis e a implementação realizada com o MDCE+. Esse mapeamento, além de esclarecer como cada componente arquitetural é materializado, ele visa auxiliar na compreensão do método MDCE+, justificando a razão de existirem artefatos como colaborações, conectores e componentes excepcionais separados dos normais. A seguir, é feita uma explicação relativa a cada um dos componentes arquiteturais definidos no Capítulo 2.5.

3.9.1 Componente Exception

O papel do componente Exception é a instanciação e lançamento das exceções, além da atualização das suas informações contextuais. Por se tratar de componentes COSMOS, a instanciação das classes de exceções é realizada utilizando-se o padrão de projeto *Factory* [GHJV95]. A classe fábrica das exceções é a classe `ExceptionFactory`, mostrada na Figura 3.23.

Quanto ao lançamento da exceção, os próprios componentes (normais e excepcionais) desempenham esse papel. Isso acontece no momento em que eles detectam a violação de alguma assertiva ou a satisfação de alguma outra condição excepcional. No contexto da linguagem java, o lançamento é feito através da palavra reservada `throw`. Outras abordagens poderiam ser utilizadas, tais como a interceptação de mensagens para a adição de verificação, utilizando por exemplo reflexão computacional através de um protocolo de meta-objetos [OB98], ou até mesmo utilizando programação orientada a aspectos [K+97].

Finalmente, em relação às informações contextuais das exceções, ela é armazenada e atualizada na própria classe da exceção. Durante a sua instanciação, são adicionadas as informações relativas ao local do seu lançamento. Além disso, nas possíveis propagações entre os componentes, essas informações podem ser atualizadas. Exemplos dessas atualizações poderiam ser: (i) o número de propagações realizadas; ou (ii) mensagens de erros mais específicas ou mais fáceis de compreender por parte do usuário, desenvolvedor ou administrador do sistema. Algumas diretrizes à respeito da utilização de mensagens de erro estão disponíveis na versão detalhada do método [dSBR05].

3.9.2 Componente Handler

Quanto ao tratamento excepcional do sistema, além de oferecer os componentes para o tratamento local, a arquitetura deve possibilitar o tratamento arquitetural das exceções, sejam eles de reconfiguração ou que envolvam a colaboração de componentes na sua execução. Além disso, a arquitetura de software deve oferecer a estruturação necessária para separar explicitamente os comportamentos normal e excepcional. Essa separação visa principalmente aumentar o grau de reutilização dos componentes (normais e excepcionais) e melhorar a legibilidade do sistema, aumentando conseqüentemente a sua manutenibilidade.

No caso do método MDCE+, os tratadores são materializados através dos componentes excepcionais, *wrappers* e conectores do sistema. Dessa forma, o método abrange tanto os tratamentos internos (componentes excepcionais), quanto tratamentos arquiteturais que envolvam a colaboração ou até mesmo reconfiguração dos componentes do sistema (*wrappers* e conectores). Já a separação entre os dois tipos de comportamento (normal e excepcional) é garantido pela própria estruturação dos componentes arquiteturais do sistema, que seguem o modelo do componente tolerante a falhas ideal (Seção 2.3.7).

3.9.3 Componente ExceptionHandlingStrategy

O papel do componente ExceptionHandlingStrategy é desviar o fluxo de execução do componente no momento em que uma exceção é lançada. No contexto desse trabalho, o papel semântico desse componente é desempenhado pelo próprio mecanismo de tratamento de exceções da linguagem de programação utilizada. Para isso, os programadores devem definir regiões protegidas e associar tratadores às mesmas. Em Java, por exemplo, essas regiões são definidas através de blocos $try\{...\}$. A associação dos tratadores, por sua vez, é feita normalmente no final de cada região protegida, através das suas vinculações aos respectivos tipos de exceções. No contexto da linguagem Java, isso é feito através de cláusulas $catch(...)\{...\}$. Em cada uma dessas cláusulas são associados tratadores de

acordo com os tipos de exceções. Uma boa prática de programação é associar um tratador para cada tipo de exceção lançado na região protegida. Na ausência do tratador adequado, a exceção deve ser propagada para a região protegida de nível imediatamente superior, que executa o mesmo procedimento recursivamente.

O entrelaçamento de código decorrente da inserção do comportamento excepcional nas operações normais dos componentes se restringe a uma chamada do método relativa ao tratador, além da definição do região protegida. Uma proposta para a eliminação desse entrelaçamento é a utilização de tecnologias de combinação[30]. Essa composição pode ser feita tanto de forma dinâmica, como acontece com reflexão computacional [Mae87], quanto de forma estática, como no caso das técnicas de programação orientada a aspectos.

3.9.4 Componente ConcurrentExceptionHandlingAction

Como definido na Seção 3.9.2, o método MDCE+ estrutura a arquitetura do sistema de modo a oferecer os locais adequados para a implementação de tratadores de exceções que envolvem vários componentes do sistema de maneira colaborativa. Porém, para usufruir dessas vantagens é necessário especificar esses tratadores, cujo papel é representado pelo componente ConcurrentExceptionHandlingAction.

No contexto do método MDCE+, o comportamento colaborativo coordenado é tratado de maneira sistemática desde a definição dos requisitos (Seção 3.2.4). Essa especificação inicia-se com a identificação dos casos de uso coordenadores e dos respectivos participantes das colaborações. No decorrer da especificação, esses casos de uso são refinados progressivamente, de modo a especificar as suas particularidades, como por exemplo a especificação dos grafos de resolução de exceções. Finalmente, para aliar o comportamento colaborativo à simplicidade de implementação, essas funcionalidades e tratadores são implementados em um *wrapper*, que exerce o papel de coordenador e envolve os participantes da colaboração. Dessa forma, esses *wrappers* que coordenam as colaborações exercem o papel do componente ConcurrentExceptionHandlingAction.

3.10 Resumo

Este capítulo apresentou a principal contribuição deste trabalho, o método MDCE+. Criado para especificar o comportamento excepcional de sistemas baseados em componentes, o método MDCE+ é baseado em uma extensão do método MDCE. A seguir, são relatadas as principais atividades adicionadas pelo MDCE+ em cada fase do desenvolvimento:

[30] do inglês *weaving*

1. **Especificação e análise dos requisitos.** Além das atividades relativas à especificação do comportamento normal, foram adicionadas três atividades: (i) identificar as entidades críticas; (ii) classificar a falha (quanto a gravidade); e (iii) contextualizar a fase onde a exceção foi detectada.

2. **Definição dos aspectos gerenciais.** Essa fase não existia anteriormente. Além de definir as principais restrições para o desenvolvimento e reutilização de componentes, essa fase é responsável por definir as *releases* para proporcionar um desenvolvimento iterativo. Do ponto de vista do comportamento excepcional, foi definida uma atividade para identificar as funcionalidades críticas do sistema. Dessa forma, é possível definir heurísticas para direcionar melhor os gastos com tolerância a falhas.

3. **Projeto arquitetural.** Essa fase também não estava presente no método MDCE. Com a sua adição antes da fase de análise, o método MDCE+ visa antecipar ao máximo a atenção com a estrutura do sistema. O método apresentado para a especificação da arquitetura é baseado no método ATAM, apresentado na Seção 2.2.3.

4. **Análise do sistema.** Além da preocupação com o posicionamento dos componentes identificados na arquitetura, do ponto de vista do comportamento excepcional, não há alterações relevantes em relação ao método MDCE.

5. **Projeto do sistema.** A fase de projeto foi uma das fases que apresentaram as principais alterações em relação ao método MDCE. Devido ao maior foco na arquitetura do sistema, foram adicionadas sete novas atividades: (i) detalhar os tratadores de exceções; (ii) definir a rigorosidade do método; (iii) analisar o fluxo excepcional; (iv) revisar o modelo de falhas; (v) refinar colaborações (execução concorrente e colaborativa); (vi) refinar interfaces providas; e (vii) definir interfaces requeridas.

6. **Materialização dos componentes.** Por adotar um modelo de componentes, o método MDCE+ apresenta refinamentos importantes para essa fase. As principais características apresentadas pelo método MDCE+ são: (i) possibilitar a reutilização e a implementação de componentes; (ii) apresentar diretrizes para a reutilização de componentes COTS; (iii) adotar uma hierarquia de exceções; e (iv) adotar o modelo COSMOS para especificação interna dos componentes.

7. **Integração dos components do sistema.** O diferencial para a fase de montagem do sistema está ligada principalmente a três aspectos: (i) diferenciação entre os conceitos de componente arquitetural e componente de desenvolvimento; (ii) destaque dado aos conectores arquiteturais para a implementação dos requisitos não-funcionais; e (iii) diretrizes para a implementação de técnicas de reconfiguração dos componentes do sistema e execução redundante.

8. **Implantação (entrega ao cliente).** Diferentemente do método MDCE, o método MDCE+ adota a abordagem de desenvolvimento iterativo, baseado em entregas sucessivas de versões do produto ao cliente. Essas entregas são definidas após a especificação das funcionalidades do sistema e contam com a determinante do cliente.

Uma característica importante do MDCE+ é o fato das atividades de especificação terem sempre em mente a materialização dos quatro componentes arquiteturais existentes na arquitetura confiável apresentada na Seção 2.5.

Apesar do detalhamento dado à modelagem do comportamento excepcional, o método MDCE+ não é instanciável, isto é, não pode ser adotado isoladamente para o desenvolvimento de sistemas de software. Para torná-lo instanciável é necessário adaptá-lo a algum processo de desenvolvimento que complemente as lacunas relativas à especificação do comportamento normal. A seguir, o Capítulo 4 mostra o processo adaptado resultante da união entre o método MDCE+ e o processo de DBC *UML Components*.

Capítulo 4

Método MDCE+ Aplicado ao Processo *UML Components*

Neste capítulo, o método MDCE+ apresentado em detalhes no Capítulo 3 é aplicado ao processo *UML Components* [CD00]. Essa adaptação consiste na adição das atividades relativas ao tratamento excepcional e de gerenciamento, presentes no MDCE+ e ausentes no processo de desenvolvimento citado. O fato do MDCE+ possuir atividades de desenvolvimento semelhantes a um processo clássico facilita esse processo de adaptação. Como conseqüência disso, o processo resultante da combinação pode ser rapidamente absorvido/entendido por quem já utiliza o processo de desenvolvimento original (sem as atividades de tratamento excepcional).

O procedimento adotado para a aplicação do MDCE+ ao processo *UML Components* pode ser feito para qualquer outro processo de DBC. A opção pelo *UML Components* se deu por três motivos principais: (i) ser um processo de DBC que enfatiza a interação entre os componentes (com uma fase explícita para isso); (ii) ser simples e de fácil utilização prática; e (iii) ser utilizado pela empresa onde foi executado o estudo de caso apresentado no Capítulo 5. Vale a pena salientar que com a presença dessas características favoráveis (principalmente da primeira), as mudanças ao processo serão minimizadas.

No decorrer do capítulo, o processo resultante da junção do processo *UML Components* com o método MDCE+ será chamado de **processo adaptado**. Além das atividades relativas ao comportamento excepcional do sistema, com a adição das atividades do MDCE+ o processo adaptado apresenta outras alterações estruturais importantes, tais como um foco maior nos aspectos gerenciais; a adição de uma fase explícita para o projeto arquitetural do sistema; e a preocupação com a execução de operações concorrentes.

A Seção 4.1 mostra uma visão geral do processo *UML Components*. Em seguida, a Seção 4.2 mostra a visão geral do processo adaptado, onde são destacadas as novas fases adicionadas. Finalmente, as Seções 4.3 a 4.10 detalham as alterações do processo,

decorrentes da sistemática para especificar o comportamento excepcional. Como será visto, essas alterações abrangem tanto a adaptação das atividades e artefatos já existentes, quanto a adição de novas atividades e artefatos específicos.

4.1 O Processo *UML Components*

De uma maneira geral, o processo *UML Components* é um processo de desenvolvimento baseado em componentes voltado para o desenvolvimento de sistemas de informação. Por esse motivo, ele é um processo simples e de fácil utilização prática. Devido à sua simplicidade, o *UML Components* adota uma arquitetura pré-definida em quatro camadas. Duas dessas camadas são destacadas durante o desenvolvimento: camada de **sistema** e camada de **negócio**. A camada de sistema é responsável por agrupar os componentes que implementam as funcionalidades especificadas para o sistema. Porém, para que essas funcionalidades possam ser implementadas, esses componentes podem necessitar de algumas funcionalidades comuns ao domínio. Os componentes que implementam essas funcionalidades gerais são posicionados na camada de negócio. As quatro camadas da arquitetura e o papel de cada uma podem ser vistos na Figura 4.1.

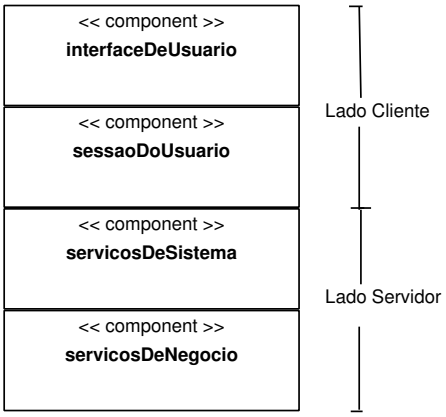

Figura 4.1: Arquitetura Adotada pelo *UML Components* [CD00]

Como mostrado na Figura 4.2, no processo *UML Components*, o desenvolvimento é dividido em seis fases: (i) especificação de requisitos[1]; (ii) especificação dos componentes[2]; (iii) provisionamento dos componentes[3]; (iv) montagem do sistema[4]; (v) testes[5]; e (vi)

[1]do inglês *requirements definition*
[2]do inglês *specification*
[3]do inglês *provisioning*
[4]do inglês *assembly*
[5]do inglês *test*

implantação. Esse processo é iterativo e enfatiza basicamente a fase de especificação dos componentes. Por essa razão essa fase é detalhada em três sub-fases: (i) identificação dos componentes (*component identification*); (ii) interação entre os componentes (*component interaction*); e (iii) especificação final (*component specification*).

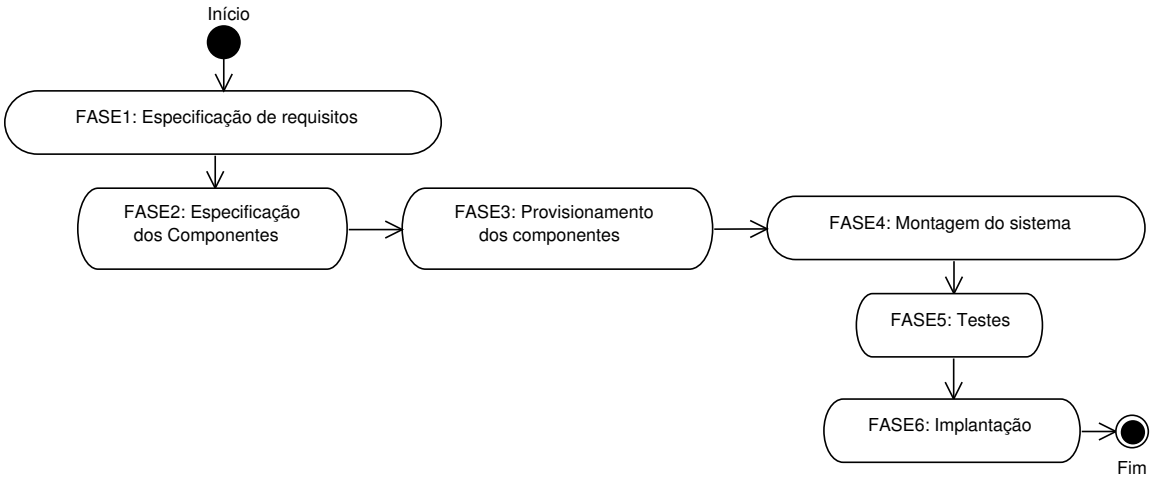

Figura 4.2: Fases do *UML Components* [CD00]

4.1.1 Especificação de Requisitos

Apesar de deixar bem claro o papel desta fase, inclusive descrevendo os artefatos que devem ser produzidos nela, o *UML Components* não detalha as atividades da sua execução. Na fase de especificação dos requisitos, as funcionalidades do sistema são representadas através de casos de uso. Além disso, é especificado o modelo conceitual do negócio[6], que representa as entidades básicas da lógica do mesmo. Dadas as suas importâncias, esses artefatos são consideradas as principais saídas desta fase.

O modelo conceitual do negócio representa o domínio do problema. Por essa razão ele necessita ser entendido e firmado entre os interessados no sistema. O propósito principal desse modelo é proporcionar um vocabulário comum entre os envolvidos no projeto. Além disso, esse modelo é a base para a identificação dos componentes da camada de negócio.

Os casos de uso são utilizados como notação para representar os requisitos funcionais do sistema. O modelo inicial de casos de uso pode ser construído com as principais funcionalidades do sistema. Após o início da especificação, esse modelo pode ser refinado para contemplar os demais requisitos funcionais.

[6]do inglês *business concept model*

4.1.2 Especificação dos Componentes

A fase de especificação dos componentes é a fase do desenvolvimento que o *UML Components* mais entra em detalhes. Como o próprio nome indica, essa fase é responsável por especificar os componentes do sistema. Essa especificação acontece tomando como entrada os artefatos produzidos na fase de especificação de requisitos: (i) modelo conceitual do negócio; e o (ii) modelo de casos de uso.

Os principais artefatos de saída da fase de especificação dos componentes são três: (i) especificação das interfaces (refinamento das assinaturas); (ii) especificação dos componentes (interfaces providas e requeridas); e (iii) a arquitetura do sistema, como conseqüência da definição das dependências entre os componentes. Como pode ser visto na Figura 4.3, para produzir esses artefatos, o *UML Components* divide essa fase em três estágios: (i) identificação dos componentes; (ii) interação entre os componentes; e (iii) especificação final. A seguir, é explicado o papel de cada um dos estágios de especificação.

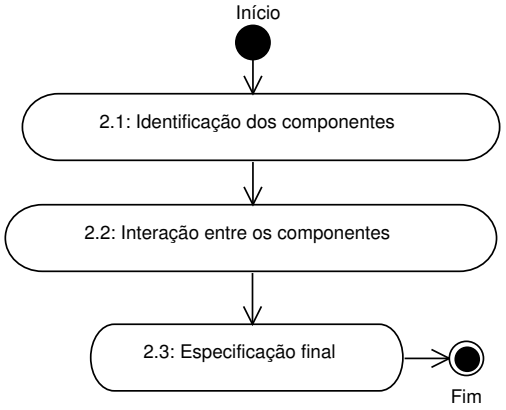

Figura 4.3: Especificação dos Componentes no *UML Components* [CD00]

1. O estágio de **identificação dos componentes** recebe como entrada o modelo conceitual do negócio e a especificação dos casos de uso do sistema, ambos provenientes da fase de especificação de requisitos. O objetivo básico dessa etapa da especificação é identificar um conjunto inicial de interfaces. Além disso, essas interfaces devem ser classificadas em duas categorias: (i) interfaces de sistema, relativas à funcionalidade do sistema; e (ii) interfaces de negócio, que conterão as operações básicas do negócio. Devido às características já citadas de cada grupo de interfaces, as interfaces de sistema são identificadas a partir dos casos de uso e as interfaces de negócio a partir do modelo conceitual fornecido.

   Em seguida, para cada uma dessas interfaces, é criado um componente. Devido à classificação da interface, o componente já pode ser posicionado na arquitetura,

nas camadas correspondentes à classificação da interface (sistema ou negócio). Em relação às interfaces de sistema, além da identificação da interface propriamente dita, neste estágio também são identificadas as primeiras operações. Apesar disso, são definidos apenas os seus nomes; a assinatura completa é refinada apenas durante o próximo estágio, de interação entre os componentes.

2. Durante a interação entre os componentes, a preocupação é voltada para o detalhamento das estruturas identificadas anteriormente. Por esse motivo, nesse estágio o projetista examina como cada operação do sistema será implementada. Em outras palavras, a preocupação é saber quais componentes do sistema deverão interagir para o serviço ser implementado. Para isso, o *UML Components* utiliza diagramas de colaboração UML. Através dessas colaborações, as operações requeridas do negócio são descobertas e as assinaturas das operações de sistema podem ser refinadas.

   Com a definição das dependências entre os componentes do sistema, a estrutura da arquitetura vai sendo definida. No final da execução do estágio de interação entre os componentes, a arquitetura do sistema já está definida.

3. Finalmente, o estágio de especificação final dos componentes é onde as especificações são refinadas e, se possível, representadas de uma maneira formal. Durante a especificação final, a arquitetura do sistema não deve variar. Dessa forma, o detalhamento da especificação só deve ser feito no momento em que a arquitetura do sistema já está considerada estável. A definição formal de assertivas é uma tarefa custosa, do ponto de vista de esforço de trabalho, e por isso deve-se evitar re-trabalho nessa tarefa.

Com a especificação dos componentes finalizada, é chegada a hora de providenciar cada um dos componentes para em seguida compor o sistema, testá-lo e entregá-lo ao cliente para utilização. Apesar da importância dessas fases, o *UML Components* não detalha a sua execução. As sub-seções seguintes descrevem o papel de cada uma dessas fases do desenvolvimento.

4.1.3 Provisionamento dos componentes

A etapa de provisionamento deve garantir a materialização dos componentes especificados. Essa materialização pode ocorrer de duas formas: (i) reutilização de componentes já existentes; e (ii) implementação de componentes novos. Normalmente, os componentes da camada de negócio são candidatos à reutilização, uma vez que representam características básicas para sistemas de um mesmo domínio. Já os componentes da camada de sistema

oferecem implementações específicas, peculiares às necessidades de um sistema em particular. Por esse motivo, apesar de também poderem ser reutilizados, as chances para isso são reduzidas.

Para cada uma das formas de materialização, há diferentes atividades a serem realizadas. Para o caso de reutilização de componentes, devido às possíveis inconsistências entre as interfaces especificadas e reutilizadas, pode ser necessário implementar adaptadores [Som01]. No caso dos componentes implementados, é necessário especificar as suas estruturas internas. Para isso, deve-se utilizar algum modelo de componentes que possibilite a sua especificação através das estruturas existentes nas linguagens de programação, tais como objetos e classes.

4.1.4 Montagem do sistema

A fase de montagem é responsável pela materialização da configuração da arquitetura do sistema. Sendo assim, ela consiste na integração dos componentes para construção da aplicação como um todo. Basicamente, existem duas atividades desempenhadas durante esta fase: (i) a construção dos conectores, que implementam um código de mapeamento entre os componentes do sistema; e (ii) a construção do programa principal, que deve conter o código de instanciação dos componentes, dos conectores, além da "ligação" entre eles, que é efetuada dinamicamente.

4.1.5 Testes

Teste é a atividade de executar um software com o objetivo de revelar falhas [Pre01]. Apesar de não comprovar a ausência de falhas, a sua execução aumenta consideravelmente a confiabilidade do sistema [Pre01]. Porém, para uma maior eficácia, os testes devem ser considerados desde as fases iniciais do desenvolvimento [Pre01, Som01]. Apesar da importância dos testes, tanto o processo *UML Components* quanto o método MDCE+ se restringem ao aspecto puramente de desenvolvimento, não contendo atividades sistemáticas de testes que tratem essa questão desde a especificação dos requisitos. Em parte isso é uma vantagem, uma vez que possibilita a utilização conjunta com processos de teste existentes.

De uma forma geral, o papel de um processo de testes é realizar verificações entre o sistema e os requisitos especificados. Dessa forma, os testes podem auxiliar na obtenção de respostas para as seguintes perguntas: (i) os requisitos estão corretos? (ii) os requisitos estão consistentes (não-contraditórios entre si)? (iii) o sistema atende minimanente a todos os requisitos? Como pode ser percebido, as respostas a essas questões são de fundamental importância para a garantia da qualidade do sistema.

4.1.6 Implantação

Após o seu desenvolvimento e uma garantia maior da satisfação dos seus requisitos, o sistema pode ser implantado para sua utilização. A conclusão da fase de implantação não significa que o ciclo de desenvolvimento do sistema foi finalizado. Pelo contrário, muito provavelmente serão necessários alguns ciclos de manutenções corretivas para que o sistema possa ser considerado estável [Pre01]. Além disso, mesmo com a estabilidade do sistema, nada impede que novas mudanças de requisitos sejam solicitadas por parte do cliente. As atividades de evolução da especificação do sistema, decorrentes dessas mudanças de requisitos, são conhecidas como manutenções perfectivas (ou evolutivas) [Som01, Pre01].

Nesse momento, a descrição do processo *UML Components* está finalizada. As seções seguintes apresentam cada uma das fases do processo adaptado, resultante da união entre o *UML Components* e o método MDCE+.

4.2 O Processo Adaptado

Em relação à adaptação propriamente dita, a proximidade semântica entre as fases e até mesmo as atividades do *UML Components* e do MDCE+ facilitou essa tarefa. Inclusive, do ponto de vista da classificação dos componentes, a separação oferecida pelo método MDCE+ entre os componentes relacionados com o domínio (componentes de infra-estrutura) e componentes relacionados com as funcionalidades (componentes da aplicação) equivalem respectivamente aos componentes de negócio e aos componentes de sistema do *UML Components*. Como citado no início do capítulo, essa semelhança entre os processos foi inclusive um dos motivos para a opção pelo *UML Components*. Mesmo com essa similaridade, todas as fases do *UML Components* foram afetadas. Além disso, como pode ser visto na Figura 4.4, duas novas fases foram adicionadas ao processo adaptado: (i) definição dos aspectos gerenciais; e (ii) projeto arquitetural.

O processo adaptado reúne características dos dois processos envolvidos. Dessa forma, além de ser um processo instanciável, ele alia a simplicidade do *UML Components*, à especificação sistemática do comportamento excepcional alcançada pelo MDCE+. Mas além de melhorar os aspectos de confiabilidade do sistema, a maior ênfase dada à arquitetura de software fazem com que o processo adaptado lide com aspectos de concorrência e comportamento colaborativo de uma maneira mais intuitiva para o desenvolvedor.

Com a adição de uma fase específica para o projeto arquitetural do sistema, foi possível generalizar a arquitetura adotada para o sistema. Porém, por priorizar a compatibilidade com o *UML Components*, algumas restrições foram mantidas. Essas restrições são mostradas na Seção 4.5.

As principais alterações em relação à especificação do comportamento normal do pro-

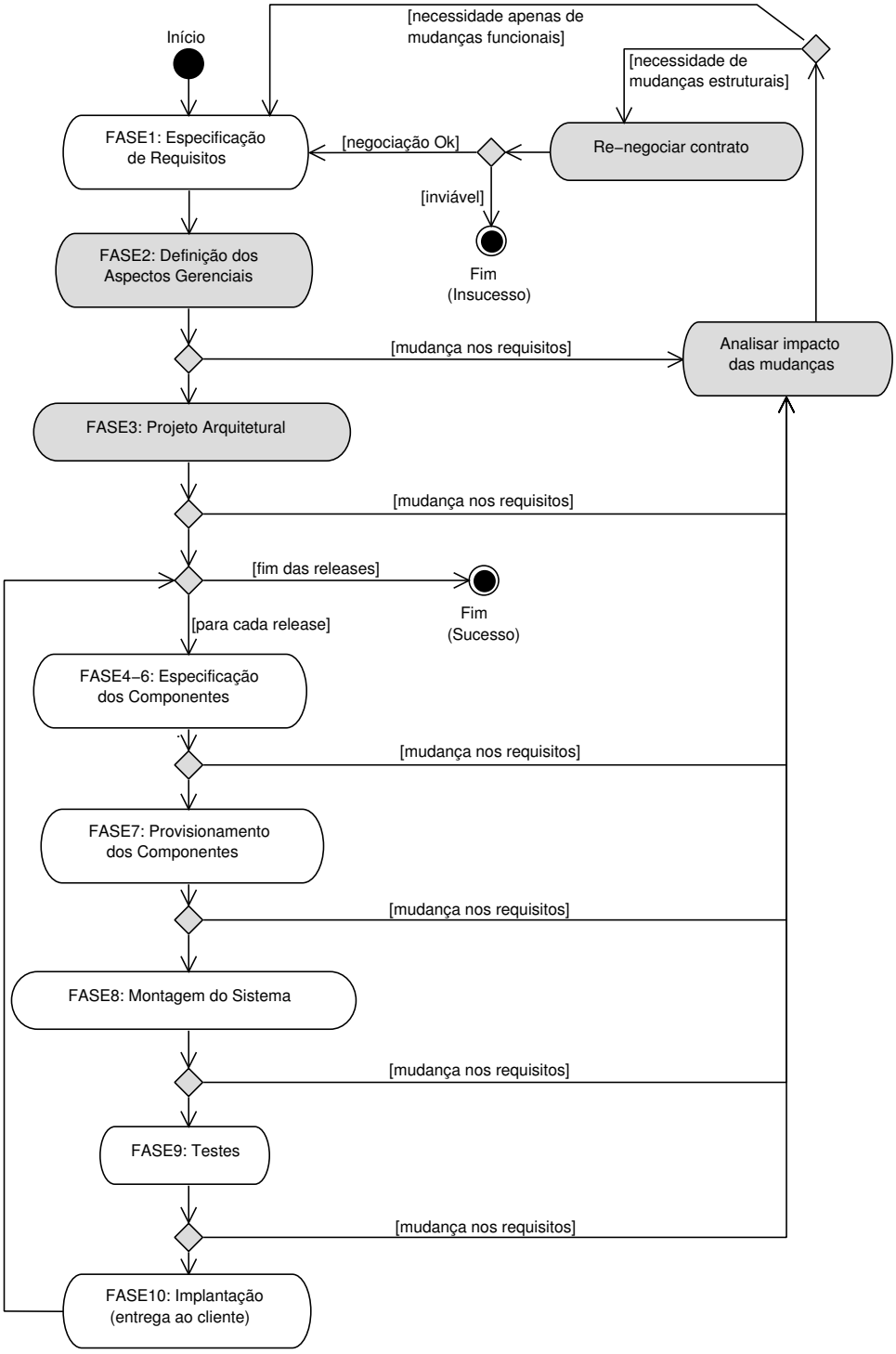

Figura 4.4: Fases do Processo Adaptado

cesso como um todo são mostradas a seguir:

1. Foram adicionadas diretrizes para a especificação e análise de requisitos, o que não está presente no *UML Components*. Esse processo assume como entrada os artefatos produzidos nessas fases, porém deixa a forma de construção a critério de cada equipe de desenvolvimento.

2. A maior flexibilidade dada à arquitetura do sistema, juntamente com a preocupação inicial com a estrutura do sistema proporciona uma redução no número de modificações durante o desenvolvimento. Essas modificações estruturais normalmente envolvem um re-trabalho grande e além de representar um aumento do custo, pode implicar na adição de um maior número de falhas e uma conseqüente redução da confiabilidade do sistema. Além disso, o foco dado à arquitetura do sistema também se reflete na existência de conectores explícitos, o que possibilita uma materialização mais efetiva da arquitetura do sistema.

3. O processo adaptado possui uma distinção entre os conceitos de componente arquitetural e componente de implementação. Os componentes arquiteturais do sistema agora são estruturados segundo o modelo do componente tolerante a falhas ideal, apresentado na Seção 2.3.7. Dessa forma, se faz uma separação explícita entre os comportamentos normal e excepcional mesmo no nível arquitetural do sistema, não apenas na implementação.

4. Outro fator importante do ponto de vista da abrangência do processo é a preocupação com a execução de atividades concorrentes. As atividades de controle da concorrência abrangem três facetas: (i) a execução coordenada de serviços (e tratadores excepcionais); (ii) a sincronização de chamadas por parte dos conectores; e (iii) a existência de funções de resolução de eventos concorrentes, como por exemplo os grafos de resolução excepcional (Seção 2.3.5).

5. Na fase de implementação, a fim de oferecer atividades para uma implementação concreta que utilize estruturas como por exemplo classes OO, o processo adaptado sugere a utilização de um modelo de componentes em particular. O modelo adotado é o modelo COSMOS, que foi apresentado na Seção 2.2.2. Além disso, o processo adaptado possibilita uma implementação, mesmo que simplificada, de colaborações. Essas estruturas permitem a execução concorrente e coordenada de alguns serviços.

6. Além das diretrizes para a implementação de novos componentes, o processo adaptado propõe novas atividades que sistematizam a reutilização de componentes prontos. Essa sistemática abrange aspectos de reutilização envolvendo componentes existentes na empresa e existentes em catálogos externos para aquisição.

7. Na fase de montagem, novas diretrizes foram propostas. No processo adaptado essa fase foi dividida em dois estágios: (i) montagem dos componentes arquiteturais; e (ii) montagem da configuração do sistema propriamente dita. Basicamente, as diretrizes adicionadas ao processo abrangem a especificação e a implementação dos componentes utilizando dois níveis de conectores: (i) internos aos componentes arquiteturais; e (ii) entre componentes arquiteturais. Além disso, o programa principal do sistema também é especificado, seguindo o processo de acoplamento de componentes proposto pelo modelo COSMOS.

As Seções 4.3 a 4.10 detalham cada uma das fases do processo adaptado, com o intuito de ressaltar as alterações relativas à especificação de ambos os comportamentos (normal e excepcional).

4.3 Fase 1: **Especificação e Análise dos Requisitos**

Vale a pena lembrar que para o processo *UML Components*, o objetivo da fase de especificação dos requisitos é produzir dois artefatos: (i) o modelo conceitual do negócio; e (ii) a especificação dos casos de uso. Devido à especificação e análise de requisitos não ser detalhada pelo *UML Components*, o processo adaptado deve proceder conforme as atividades definidas na Seção 3.2 do método MDCE+.

Dessa forma, a representação do domínio feita na Seção 3.2.2 deve seguir a notação proposta pelo *UML Components* para o modelo conceitual do negócio. Em outras palavras, o modelo deve ser representado como um diagrama de classes que relaciona as principais entidades do negócio. Apesar disso, é importante ressaltar que esse modelo representa o domínio do problema e está relacionado com a fase de requisitos, sendo portanto independente da implementação.

Em relação ao aspecto de confiabilidade adicionado, é feita uma análise das entidades do modelo conceitual. Essa análise visa identificar as entidades críticas do domínio. A Figura 4.5 mostra um exemplo de um modelo conceitual de negócio para um sistema de vendas. As entidades críticas estão destacadas com o estereótipo $<< critical >>$.

Por representar os requisitos funcionais do sistema, os casos de uso do sistema são identificados e especificados respectivamente, a partir das atividades apresentadas nas Seções 3.2.3 e 3.2.4. Além disso, foi necessário detalhar a representação desses artefatos. Esse detalhamento consistiu na adição de novas informações, necessárias para a descoberta de exceções e para a especificação do comportamento excepcional do sistema. A Tabela 4.1 apresenta a especificação do caso de uso Efetuar Venda, que representa uma funcionalidade de um sistema de vendas.

Após a especificação do comportamento normal do caso de uso, através das atividades

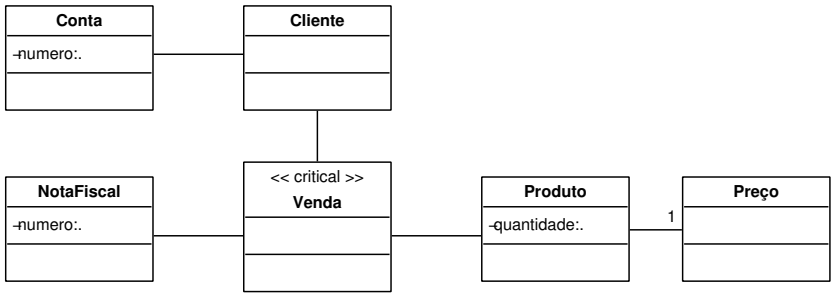

Figura 4.5: Modelo Conceitual de Negócio de um Sistema de Vendas [CD00]

Tabela 4.1: Caso de Uso Normal: Efetuar Venda

Efetuar Venda	
Descrição:	Realizar a venda de uma lista de produtos selecionados.
Participantes:	Cliente
Pré-condições:	1- Os códigos dos produtos são válidos; 2- Existe estoque suficiente.
Invariantes:	O produto está cadastrado no sistema.
Cenário primário:	1- Verificar se produto está cadastrado; 2- Debitar valor do estoque. 3- Obter dados do cliente. 4- Emitir nota fiscal.
Cenários alternativos:	Se o cliente não estiver cadastrado 3.1- Cadastrar cliente no sistema; 3.2- Return passo 3 do fluxo principal;
Pós-condições:	1- Cliente cadastrado no sistema; 2- Estoque do produto foi atualizado.

apresentadas na Seção 3.2.5, são especificadas as exceções, os cenários excepcionais e os casos de uso tratadores << *handler* >>. Baseado na violação das assertivas do caso de uso normal (Tabela 4.1), podem ser identificadas as exceções: `EstoqueInsuficiente-Exception` e `EstoqueBaixoException`. Conforme especificado pelo método MDCE+, essas exceções são agendadas neste momento, a fim de terem suas classes criadas na fase de identificação dos componentes (Seção 4.6). Por impossibilitar a execução da operação, a primeira foi classificada como uma falha muito alta e deu origem a um cenário de falha. Já a segunda, por não comprometer a execução bem sucedida da operação, foi classificada como uma falha muito baixa e deu origem a um cenário recuperável. A especificação desses cenários pode ser vista na Tabela 4.2. Em seguida, a Tabela 4.3 apresenta a especificação do comportamento do tratador do segundo cenário excepcional (Tratar Estoque Baixo).

Além dos aspectos relativos à confiabilidade, a junção com o método MDCE+ possibilitou ao processo adaptado a possibilidade de representar as funcionalidades que são executadas de forma concorrente e coordenada. Essa especificação se dá a partir da classificação dos casos de uso como coordenador ou participante, de acordo com o seu papel na colaboração.

Tabela 4.2: Cenários Excepcionais do Caso de Uso Efetuar Venda

Efetuar Venda	
Cenário Recuperável 1:	Caso o estoque seja suficiente, mas fique abaixo do limite mínimo e não exista nenhum pedido em aberto.
Condição Excepcional:	(estoque < mínimo) && (!∃ outros pedidos).
Ponto de identificação:	Passo 2 do fluxo básico.
Tratador:	Tratar Estoque Baixo.
Pós-condições:	Encaminhamento de solicitação de reposição de estoque.
Cenário Falho 1:	Caso o código de produto fornecido seja inválido.
Condição Excepcional:	código nulo OU vazio.
Ponto de identificação:	Passo 0 do fluxo básico.
Tratador:	Lançar exceção de defeito.
Pós-condições:	1- Operação não realizada. 2- O estado do sistema continua válido.
Cenário Falho 2:	Caso o produto não esteja cadastrado.
Condição Excepcional:	produto não cadastrado.
Ponto de identificação:	Passo 1 do fluxo básico.
Tratador:	Lançar exceção de defeito.
Pós-condições:	1- Operação não realizada. 2- O estado do sistema continua válido.
Cenário Falho 3:	Caso o estoque seja insuficiente.
Condição Excepcional:	quantidade da venda > estoque.
Ponto de identificação:	Passo 2 do fluxo básico.
Tratador:	Lançar exceção de defeito.
Pós-condições:	1- Operação não realizada. 2- O estado do sistema continua válido.

Tabela 4.3: Caso de Uso Excepcional: Tratar Estoque Baixo

Tratar Estoque Baixo	
Descrição:	Tratamento quando o estoque estiver muito baixo. Deve-se solicitar a reposição.
Participantes:	-
Pré-condições:	1- O código do produto é válido;
Invariantes:	O produto está cadastrado no sistema.
Cenário primário:	1- Repor estoque com a quantidade ideal do produto.
Pós-condições:	1- A solicitação de reposição foi encaminhada com sucesso;

4.4 Fase 2: **Definição dos Aspectos Gerenciais**

Devido ao fato do processo *UML Components* não tratar a definição das características gerenciais do projeto [CD00], essa fase foi adicionada ao processo. Apesar de não abranger todos os aspectos do gerenciamento de um software, as atividades definidas auxiliam principalmente a análise inicial dos riscos e a definição das iterações do desenvolvimento. As suas atividades são mostradas na Seção 3.3 do método MDCE+.

Em relação à análise dos riscos, a preocupação é voltada para a definição de restrições, sejam elas gerais do desenvolvimento ou relativas à reutilização de componentes. Além disso, são identificadas as funcionalidades consideradas críticas para o desenvolvimento. Essas funcionalidades serão o alvo principal de tolerância a falhas e a inviabilidade do oferecimento de alguma delas significa a necessidade de rever os requisitos, antes de con-

tinuar a execução do processo. Dada a sua importância para o negócio, o caso de uso Efetuar Venda foi considerado crítico.

As iterações do desenvolvimento são definidas baseado no princípio de desenvolvimento dirigido por característica[7], presentes em vários processos ágeis de desenvolvimento, como por exemplo *eXtreme Programming* (XP) [Coc03]. Dessa forma, após uma análise das precedências entre os requisitos funcionais especificados, o cliente juntamente com o engenheiro de requisitos e o gerente do projeto, definem as prioridades para as liberações das versões do sistema.

4.5 FASE 3: **Projeto Arquitetural**

Assim como a definição dos aspectos gerenciais do sistema, a fase de projeto arquitetural também foi adicionada ao processo adaptado. O objetivo dessa fase é auxiliar o arquiteto de software a escolher uma arquitetura adequada para o sistema.

Apesar do desenvolvimento ser iterativo, essa atividade é efetuada analisando-se todos os requisitos especificados para o sistema (funcionais e não-funcionais). Assim, o processo adaptado atinge um equilíbrio entre o dinamismo dos processos ágeis e o desenvolvimento sistemático dos processos tradicionais. Através dessa abordagem intermediária, o processo busca atender aos requisitos de confiabilidade do sistema, aliado a um desenvolvimento incremental.

A especificação estrutural do sistema antes do início da especificação dos seus componentes traz vários benefícios. Um deles é a identificação antecipada de restrições ligadas ao projeto do sistema. Essas restrições evitam um grande volume de re-trabalho durante o desenvolvimento, o que o torna menos custoso, tanto do ponto de vista do tempo, quanto financeiro. Porém, o sucesso do projeto arquitetural depende diretamente do histórico de sistemas anteriores da empresa, além da experiência do arquiteto. Essa dependência pessoal fica clara com a apresentação do método para escolha do estilo arquitetural, mostrado na Seção 3.4.1. Essas atividades conduzem a uma seleção baseada em descartes progressivos de arquiteturas candidatas. Dessa forma, a experiência e o *feeling* do arquiteto são imprescindíveis.

No contexto particular do processo *UML Components*, uma restrição que deve estar sempre presente na arquitetura adotada é a separação clara entre os componentes de sistema e os componentes de negócio. Por esse motivo, pelo menos essas duas camadas da arquitetura devem estar definidas. Além disso, existe uma outra restrição de comunicação: os componentes de negócio não podem depender dos componentes de sistema [CD00]. Apesar dessas restrições, a arquitetura que o sistema pode adotar no processo adaptado

[7] do inglês *feature driven development*

é bem mais flexível que a proposta pelo processo *UML Components* (Figura 4.1).

Para o sistema de vendas utilizado aqui como exemplo, foi adotada uma arquitetura semelhante à arquitetura proposta pelo *UML Components* (Figura 4.1).

4.6 Fase 4: **Identificação dos Componentes**

O papel do estágio de identificação dos componentes é equivalente ao desempenhado pela fase de análise do sistema do método MDCE+, que é mostrada na Seção 3.5. Porém, por enfatizar a especificação do comportamento excepcional de forma gradual e progressiva, o MDCE+ apresenta algumas atividades não contempladas pelo processo *UML Components*. Basicamente, essas atividades são referentes à especificação do comportamento excepcional do sistema, como mostrado na Seção 3.5.2.

A atividade adicionada refere-se ao processamento das exceções que já foram identificadas anteriormente. Essa identificação pode ter ocorrido tanto na especificação dos requisitos, quanto em iterações anteriores do desenvolvimento. O papel dessa atividade consiste de duas tarefas importantes: (i) criar as classes de exceções; e (ii) criar as interfaces dos componentes excepcionais. Por se preocuparem com a identificação de interfaces e entidades da especificação, a atividade de processamento das exceções segue a mesma linha de raciocínio das demais atividades existentes na etapa de identificação dos componentes. Porém dessa vez é analisado o aspecto excepcional do comportamento do sistema.

As classes de exceções devem armazenar as informações de contexto de cada uma das exceções, através da definição de atributos. Já as interfaces dos componentes excepcionais conterão as operações dos tratadores. Esses tratadores são representados pelos casos de uso excepcionais, que possuem o estereótipo $<< handler >>$.

Com a identificação das interfaces dos tratadores, nesta fase serão identificados tanto os componentes normais do sistema, quanto os componentes excepcionais. Posteriormente, no estágio de interação entre os componentes (Seção 4.7), esses componentes serão agrupados para dar origem aos componentes arquiteturais do sistema. Vale a pena salientar que as interfaces normais e excepcionais que são derivadas dos casos de uso "coordenadores" darão origem a componentes classificados da mesma forma. O principal motivo dessa classificação diferenciada está na maneira como esses componentes são materializados. Isso é feito na fase de provisionamento dos componentes (Seção 4.9).

Devido à diferenciação feita entre os componentes arquiteturais e de implementação, no momento do posicionamento na arquitetura, somente os componentes normais são posicionados. Dessa forma, o posicionamento atribuído aos componentes normais será atribuído na verdade aos respectivos componentes arquiteturais, com comportamento normal e excepcional. Os componentes excepcionais especificados podem inclusive ser utilizados por mais de um componente normal do sistema.

A Figura 4.6(a) mostra as interfaces criadas durante a execução desse estágio da especificação. Em seguida, a Figura 4.6(b) mostra os componentes criados para oferecê-las.

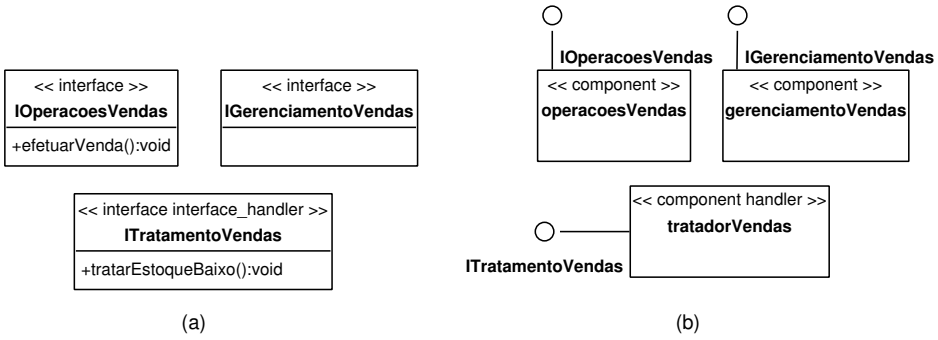

Figura 4.6: Interfaces Identificadas e Componentes Criados

4.7 FASE 5: **Interação Entre os Componentes**

Devido à similaridade entre os objetivos das duas atividades, a etapa de interação entre os componentes do processo adaptado pode se basear no *workflow* mostrado na Seção 3.6.1 do método MDCE+. Como pode ser visto aí, do ponto de vista da arquitetura do sistema, esse estágio desempenha as atividades necessárias para o seu refinamento. Isso se deve porque durante as interações entre os componentes é possível analisar o fluxo de informação que flui na arquitetura, seja ele normal ou excepcional. Através da análise desses fluxos, são identificadas as dependências entre os componentes do sistema. Além disso, com uma abordagem sistemática de análise do fluxo excepcional é possível identificar precocemente a existência de falhas de especificação. Exemplos comuns dessas falhas podem ser o esquecimento de especificar um tratador, ou a especificação de tratadores que nunca são utilizados.

Enquanto no processo *UML Components* as colaborações são representadas através de diagramas de colaboração UML, o método MDCE+ propõe a representação através de diagramas de atividades, que proporcionam uma maior representação semântica da seqüência de execução. Por esse motivo, o processo adaptado sugere a representação das interações através de diagramas de atividades, mas nada impede que essa tarefa continue sendo feita utilizando os outros diagramas dinâmicos da UML (de seqüência ou colaboração). As Figuras 4.7 e 4.8 mostram os diagramas de atividades referentes às colaborações das operações relativas aos casos de uso Efetuar Venda e Tratar Estoque Baixo.

Através da análise dessas interações, é possível refinar quatro aspectos importantes

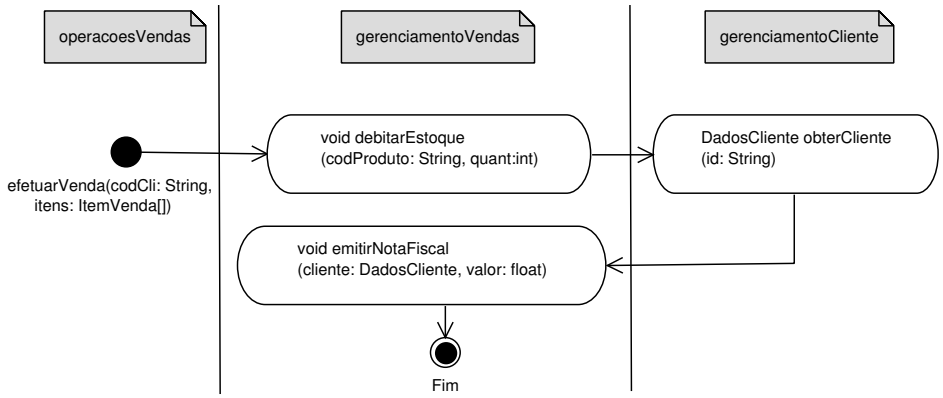

Figura 4.7: Colaboração da operação Efetuar Venda

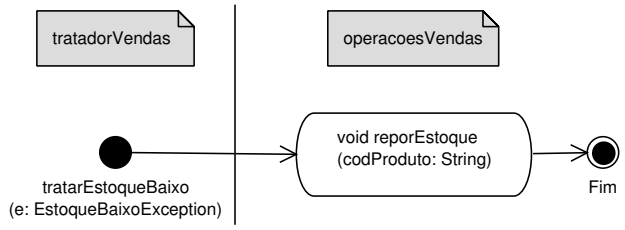

Figura 4.8: Colaboração da operação Tratar Estoque Baixo

da especificação do sistema: (i) as operações das interfaces da camada de negócio; (ii) o refinamento da assinatura das operações de sistema, que inclui a definição das estruturas de dados utilizadas; (iii) a conseqüente lista das dependências de cada operação do sistema; e (iv) a necessidade de se especificar novas funcionalidades requeridas, isto é, de se refinar a especificação dos requisitos.

A Figura 4.9(a) mostra as interfaces de negócio com as operações identificadas. A mesma figura (b) mostra as interfaces de sistema com as assinaturas das suas operações já refinadas e a lista de dependências de cada uma. Em relação ao refinamento dos requisitos, como pode ser visto na Figura 4.8, o tratamento excepcional especificado necessita executar um serviço do sistema que ainda não foi especificado. Por esse motivo, o processo deve iterar com o intuito de especificar o caso de uso Repor Estoque.

Após a definição das dependências entre as operações dos componentes, o método prossegue com a análise do fluxo excepcional entre os componentes. Após essa análise, já se pode estruturar os componentes arquiteturais do sistema, segundo o modelo do componente tolerante a falhas ideal apresentado na Seção 2.3.7. A Figura 4.10 mostra a estruturação do componente arquitetural operacoesVendasArq, que como definido no estágio anterior, pertence à camada de sistema. Na Figura 4.11, as dependências entre

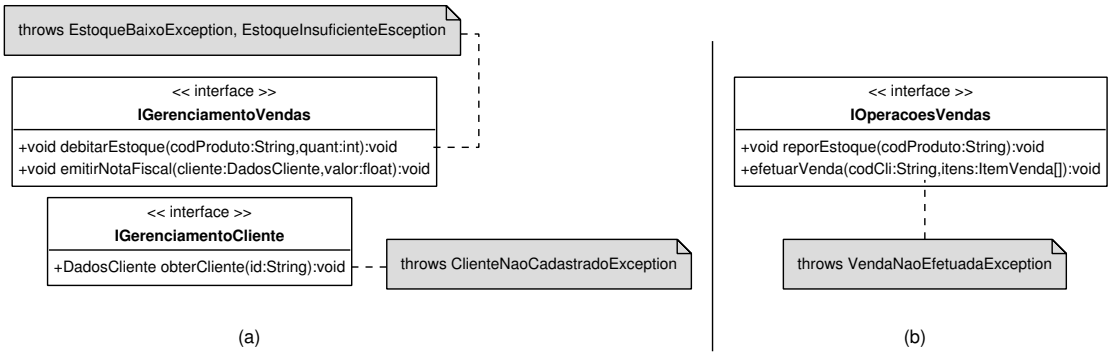

Figura 4.9: Atualização das Interfaces (Negócio e Sistema)

as operações das interfaces são utilizadas para refinar a representação da arquitetura do sistema. Note que só são representados os componentes arquiteturais.

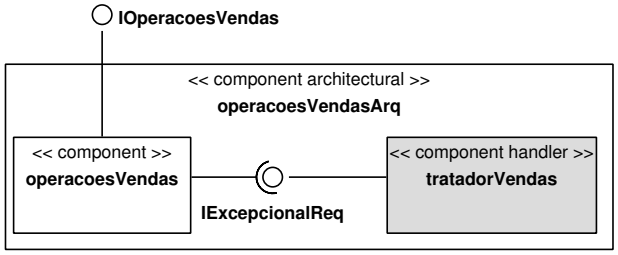

Figura 4.10: Estruturação do Componente Arquitetural Efetuar Venda

Com uma definição clara das dependências de cada componente, é possível revisar a lista de entidades críticas, de acordo com as funcionalidades críticas especificadas na Seção 4.4. Seguindo as atividades do método MDCE+, a Figura 4.12 mostra o modelo do negócio com as novas entidades críticas identificadas. Só a título de recordação, a identificação dessas se dá a partir dos componentes de negócio dos quais as funcionalidades críticas dependem.

Após o refinamento das entidades críticas do sistema, é possível escolher o nível de criticidade em que o processo deve se basear para continuar as atividades de especificação. Por ser um sistema *web* com requisitos de disponibilidade, suas funcionalidades devem estar disponíveis o maior tempo possível. Dessa forma, o sistema foi classificado como crítico e conforme indicado nas diretrizes da Seção 3.6.1 do método MDCE+, durante a definição dos componentes redundantes, será necessário ter uma réplica do componente de sistema relativo ao caso de uso Efetuar Venda. Essas réplicas representarão a execução do mesmo serviço em lojas distintas da empresa.

Antes de finalizar a execução da etapa de interação entre os componentes, é necessário

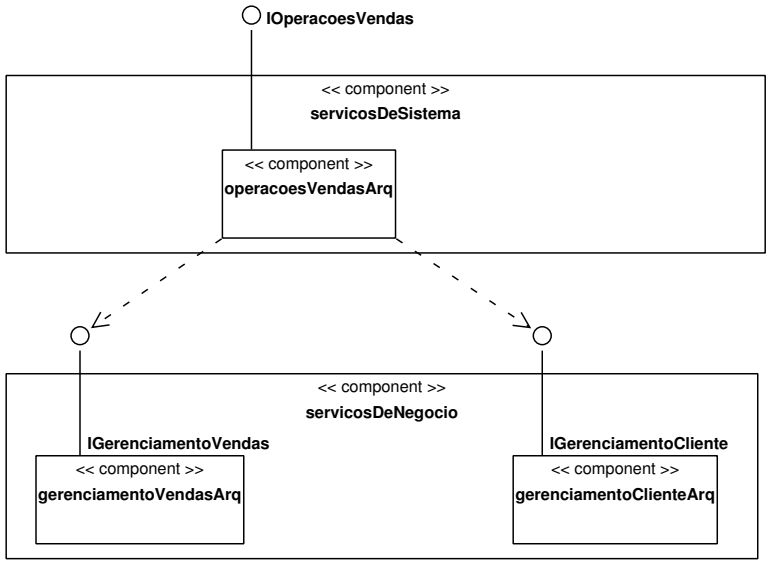

Figura 4.11: Arquitetura Refinada do Sistema

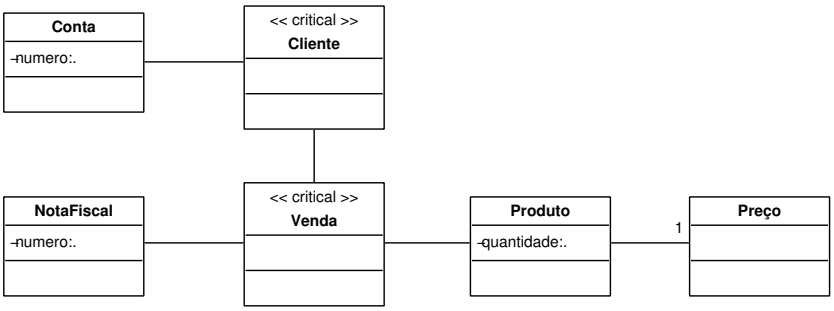

Figura 4.12: Novas Entidades Críticas Identificadas

identificar as interfaces requeridas de cada componente. Essa atividade, que não está presente no processo *UML Components*, tem uma interferência direta na criação dos conectores arquiteturais. Essa criação só acontecerá na fase de montagem mostrada na Seção 4.10. A Figura 4.13 mostra o componente arquitetural operacoesVendas com as suas duas interfaces requeridas. A interface `IReqSistema` é decorrente da necessidade do tratador excepcional executar serviços da camada de sistema (Figura 4.8). A interface `IReqNegocio` é conseqüência das operações requeridas pelo componente normal (Figura 4.7). Porém, como a operação requerida pelo tratador faz parte do mesmo componente normal, do ponto de vista externo ao componente arquitetural, essa dependência não existe.

Devido à simplicidade do exemplo, não foi necessário simplificar o modelo de falhas do sistema de vendas. Além disso, por não haver necessidade de execução concorrente,

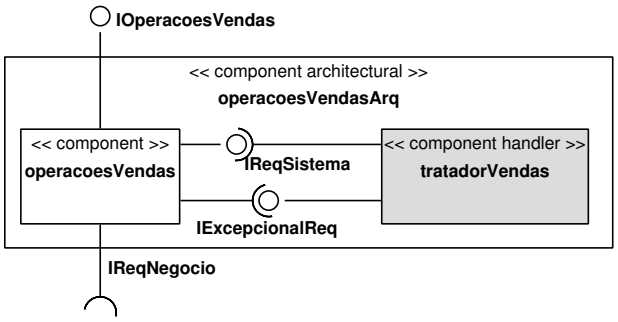

Figura 4.13: Interfaces Requeridas do Componente operacoesVendasArq

não foram especificadas colaborações para as funcionalidades do sistema.

4.8 FASE 6: **Especificação Final do Sistema**

A execução dessa fase não sofreu alterações em relação ao proposto pelo processo *UML Components*. A Tabela 4.4 mostra o refinamento das assertivas especificadas para o caso de uso Efetuar Venda. Essas assertivas foram formalizadas utilizando OCL.

Tabela 4.4: Formalização das Assertivas do Caso de Uso Efetuar Venda

Efetuar Venda	
Pré-condições:	1- income(itens: ItemVenda[]) pre: itens − > forAll (i: ItemVenda | i.codigo != null && i.codigo != " "); 2- income(itens: ItemVenda[]) pre: itens − > forAll (i: ItemVenda | obterEstoque(i.codigo) >= i.quantidade).
Invariantes:	income(itens: ItemVenda[]) inv: itens − > forAll (i: ItemVenda | estaCadastrado(i.codigo)).
Pós-condições:	1- income(codCli: String) post: estaCadastrado(codCli); 2- income(itens: ItemVenda[]) post: itens − > forAll (i: ItemVenda | @obterEstoque(i.codigo) >= !obterEstoque(i.codigo).

4.9 FASE 7: **Provisionamento dos Componentes**

Devido ao pouco detalhamento oferecido pelo *UML Components*, essa fase deve ser executada conforme mostrado na Seção 3.7 do método MDCE+.

Conforme sugerido pelo método, existem três formas de se materializar um componente: (i) reutilização de um componente já utilizado pela empresa; (ii) aquisição do componente a partir de um catálogo de terceiros; e (iii) implementação do componente. Mas antes de escolher a forma mais adequada para a materialização de cada componente do sistema, é necessário executar algumas atividades gerais, que são comuns às

três abordagens. Essas abordagens dizem respeito basicamente à materialização das partes excepcionais dos componentes e das estruturas de dados utilizadas na assinatura das operações (Seção 3.7.1).

Durante a criação das classes de exceções, essas classes devem ser posicionadas em uma hierarquia pré-definida pelo método, conforme mostrado na Figura 3.24. Essa hierarquia classifica as exceções a partir da natureza das falhas que elas representam. No contexto do sistema de vendas, as duas exceções `EstoqueInsuficienteException` e `EstoqueBaixoException` devem herdar da classe `Single`, da hierarquia de `DeclaredException`. Essa classificação é decorrente das exceções terem sido previstas na especificação do componente.

Em relação às formas de materialização dos componentes, para o caso específico do sistema de vendas do exemplo, será considerado que o componente de sistema componenteVendas deve ser implementado e que os componentes de negócio foram reutilizados de projetos anteriores da empresa.

No caso dos componentes reutilizados, deve-se proceder as atividades mostradas na Seção 3.7.3 para a estruturação dos possíveis adaptadores. Como fruto dessas atividades, a Figura 4.14 mostra a modelagem do componente de negócio gerenciamentoVendas. Esse componente foi estruturado segundo a proposta de Asterio et al. apresentada na Seção 2.4.2.

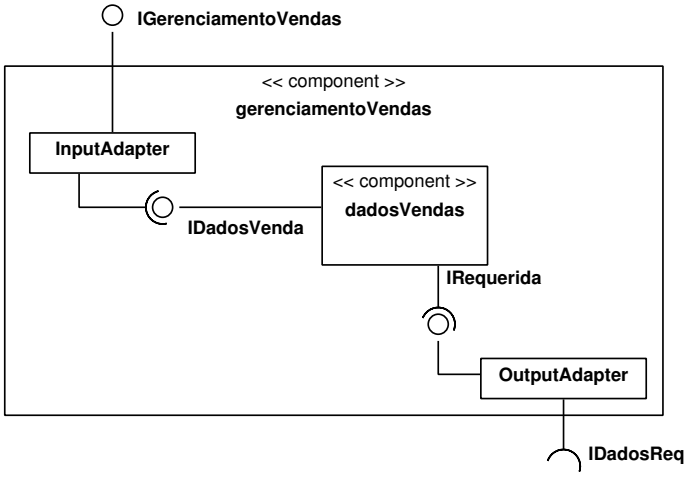

Figura 4.14: Modelagem do Componente gerenciamentoVendas

Já para a implementação do componente operacoesVenda, devem ser seguidas as diretrizes propostas na Seção 3.7.4 do método MDCE+. A Figura 4.15 mostra a modelagem do componente, que foi especificado utilizando o modelo COSMOS, apresentado na Seção 2.2.2. Apesar de ser um componente crítico que poderia possui mais de uma versão

projetada, isso não será necessário. Devido às características da criticidade do sistema, decidiu-se utilizar diferentes instâncias do mesmo componente. Dessa forma, julgou-se que a redundância unicamente da disposição física dos componentes proporciona a melhor relação custo x benefício.

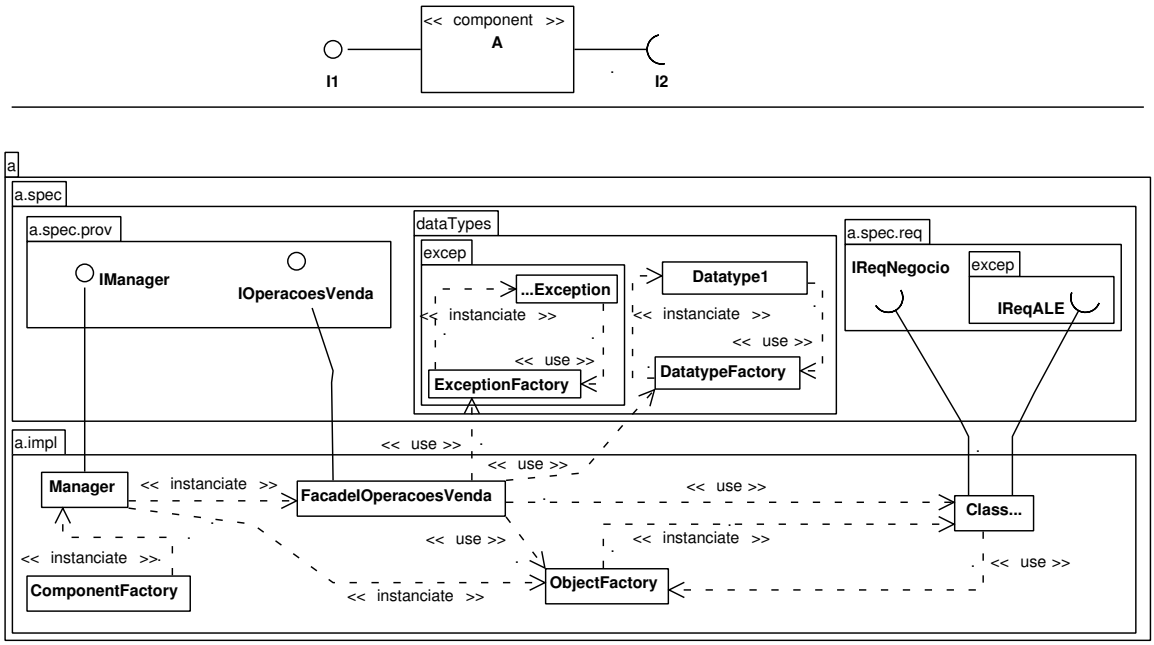

Figura 4.15: Modelagem do Componente operacoesVendas

4.10 FASE 8: Montagem do Sistema

Devido à diferenciação entre os conceitos de componente arquitetural e componente de implementação, a fase de montagem foi dividida em dois estágios: (i) montagem dos componentes arquiteturais, onde é realizada a ligação entre os componentes normais e excepcionais; e (ii) materialização da configuração do sistema, onde é realizada a ligação entre os componentes arquiteturais do sistema e a implementação do programa principal. Como a montagem do sistema não é detalhada pelo processo *UML Components*, o processo adaptado propõe que sejam seguidas as diretrizes apresentadas na Seção 3.8 do método MDCE+.

No contexto do exemplo do sistema de vendas, a Figura 4.16 mostra a materialização do componente arquitetural operacoesVendas. Devido às dimensões do exemplo, não foi necessário especificar os conectores internos de fronteira (BLE). Em relação aos conectores arquiteturais, antes da sua implementação é necessário refinar a sua especificação.

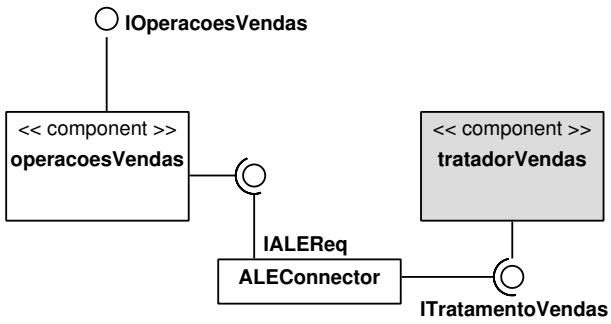

Figura 4.16: Materialização do Componente Arquitetural operacoesVendas

Tendo em vista os requisitos de disponibilidade do sistema especificado, é necessário especificar como os conectores arquiteturais farão o gerenciamento da reconfiguração. Essa especificação da reconfiguração consiste basicamente de dois passos: (i) definir a forma de reconfiguração, que pode ser simples ou permanente; e (ii) identificar as réplicas do componente. No contexto do componente operacoesVendas, que será replicado, foi definido que a reconfiguração seja permanente. Em relação às réplicas do componente principal, conforme definido na Seção 4.7, cada uma corresponderá a uma instância diferente do mesmo componente, localizada nas diferentes filiais da empresa. Dessa forma, quando a venda não for executada com sucesso em uma das lojas, o conector se encarregará de tentar executá-la em cada uma das filiais, e assim sucessivamente para toda a lista definida. Como mostrado na Figura 4.17, numa segunda execução da funcionalidade, o conector tenta a execução diretamente da última instância do componente que funcionou com sucesso.

Após o refinamento da sua especificação, os conectores arquiteturais podem ser implementados. A Figura 4.18 mostra a arquitetura do sistema após a implementação dos seus conectores arquiteturais. O estereótipo $<<$ *handler* $>>$ em um conector indica que ele implementa algum tipo de tratamento excepcional. No caso específico do conector `conSessaoSistema`, ele é responsável por realizar a reconfiguração da arquitetura do sistema, quando do lançamento de alguma exceção de defeito por um dos componentes `operacoesVendaArq`.

Com a materialização da arquitetura do sistema, ele está pronto para a execução. Sendo assim, o sistema deveria ser testado e em seguida implantado para utilização. Tendo em vista a ausência de sistemática para as fases de testes e implantação, tanto por parte do processo *UML Components*, quanto por parte do método MDCE+; essas fases não serão detalhadas.

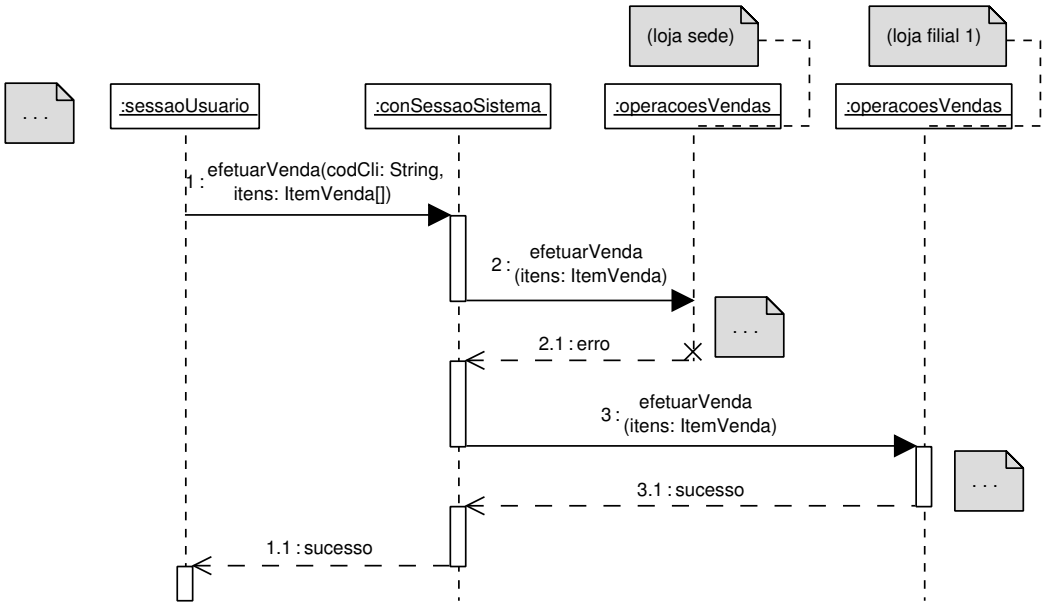

Figura 4.17: Simulação de Uma Reconfiguração Permanente do Sistema

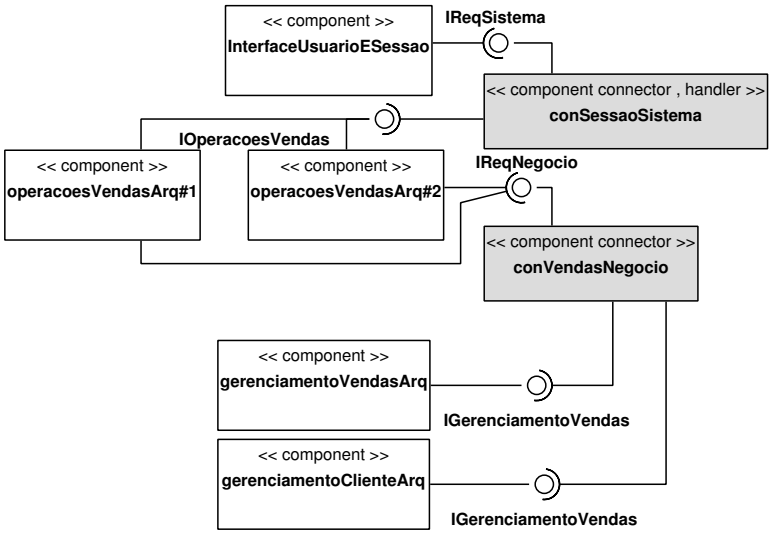

Figura 4.18: Arquitetura do Sistema Materializada com Conectores

4.11 Resumo

Este capítulo apresentou o processo resultante da integração feita entre o método MDCE+ e o processo *UML Components*. Uma característica dessa integração é a interferência nas atividades internas do *UML Components*, o que visa reduzir *overhead* do desenvolvimento. Essa redução é conseqüência do menor número de artefatos produzidos, quando comparado com uma execução seqüencial das duas abordagens. Dessa forma, artefatos que já eram utilizados para representar o comportamento normal, tais como os casos de uso e os diagramas de colaboração UML, foram refinados de modo a contemplarem também as particularidades do método MDCE+ para a representação do comportamento excepcional.

Por contemplar a modelagem dos comportamentos normal e excepcional, o processo resultante dessa adaptação é considerado instanciável, no sentido de poder ser utilizado para o desenvolvimento de sistemas. Apesar disso, um aspecto ainda a ser melhorado é a adição de mais atividades gerenciais, praticamente ausentes no processo *UML Components* e pouco presentes no método MDCE+.

Capítulo 5

Estudo de Caso: Sistema de Controle Bancário

Este capítulo mostra os principais passos da execução do estudo de caso desenvolvido para avaliar a aplicabilidade prática do método MDCE+. O conteúdo completo desse estudo de caso está disponível em um relatório técnico publicado no Instituto de Computação da Unicamp [dSBRMR05]. Além de avaliar a viabilidade do método, esse estudo de caso foi utilizado para integrar o método MDCE+ a um método de testes para componentes robustos [dSBRCF+05].

O estudo de caso consistiu na utilização do processo adaptado, que foi mostrado no Capítulo 4. Esse processo foi utilizado para desenvolver um sistema financeiro com requisitos de confiança no funcionamento relativos à disponibilidade.

O conteúdo desse capítulo está exposto conforme mostrado a seguir. A Seção 5.1 descreve o sistema que foi desenvolvido no estudo de caso. A Seção 5.2 apresenta alguns detalhes relevantes do estudo de caso, como por exemplo a infra-estrutura utilizada para a sua condução. A Seção 5.3 relata os critérios utilizados no planejamento do estudo de caso, que inclusive descreve as métricas utilizadas. A Seção 5.4 mostra um resumo da execução do estudo, apresentando os principais artefatos produzidos para a especificação e implementação de uma iteração do desenvolvimento. Finalmente, a Seção 5.5 apresenta algumas conclusões que foram tomadas a partir dos resultados desse estudo. Essa análise consiste de dois passos: (i) análise do produto final; e (ii) análise do método MDCE+.

5.1 Descrição do Problema

O sistema desenvolvido no estudo de caso pertence ao domínio de sistemas financeiros bancários. Foram especificados para ele as seis funcionalidades básicas mostradas a seguir:

1. **Requisição de talões de cheques.** O cliente pode se dirigir a uma agência para solicitar talões de cheque.

2. **Entrega de talões de cheques.** Essa funcionalidade consiste na retirada de talões de cheque solicitados previamente.

3. **Sustação de cheques.** O cliente pode cancelar um determinado número de cheques, em caso de perda, roubo, ou outro motivo específico.

4. **Captura de cheque para compensação.** A captura de cheque é feita durante um depósito em cheque. Esses cheques são capturados e processados para a compensação.

5. **Cancelamento do contrato da conta.** A qualquer momento, o cliente pode perder o crédito de sua conta. Dessa forma, o seu contrato deve poder ser cancelado.

6. **Cadastramento de limite adicional.** Dependendo da necessidade do cliente e das suas condições de crédito, o cliente pode receber um limite adicional que poderá ser utilizado por ele.

As funcionalidades cancelamento do contrato da conta e sustação de cheques, devem estar disponíveis o máximo de tempo possível. Esse atributo de qualidade imposto pelas regras de negócio tem o objetivo de evitar a utilização de um crédito que já não esteja disponível, quer seja por parte do cliente, isto é, utilização do seu limite de crédito; ou de terceiros, isto é, através do resgate de cheques cancelados. Por essa razão, assume-se que essas funcionalidades possuem requisitos críticos de disponibilidade.

Por apresentar uma preocupação maior com os atributos de qualidade do sistema, no contexto dessa dissertação, será tomado como exemplo a quinta funcionalidade citada (cancelamento do contrato da conta). Mas antes do detalhamento dos procedimentos de especificação e implementação, a Seção 5.2 descreve o ambiente de trabalho utilizado para a condução do estudo. Essa descrição do ambiente abrange tanto a descrição ferramental, quanto do perfil dos executores.

5.2 Descrição do Estudo de Caso

Como pode ser visto na Figura 5.1, o processo de avaliação do método MDCE+ foi dividido em três etapas: (i) preparação; (ii) execução; e (iii) análise dos resultados. A etapa de preparação, apresentada na Seção 5.3, foi executada pelo próprio autor da dissertação. Essa etapa consistiu de uma revisão bibliográfica com o intuito de definir o meio mais adequado para avaliar o método MDCE+, incluindo a elaboração do cronograma de execução.

Após a etapa de preparação, ficou constatado que o meio de avaliação mais indicado para as características do MDCE+ era a condução de um estudo de caso.

Com o intuito de verificar a robustez dos produtos desenvolvidos com o método MDCE+, uma pré-condição para a escolha do sistema a desenvolver foi a necessidade de que ele possuísse algumas características peculiares:

1. **Possuir requisitos de confiança no funcionamento[1].** Mais especificamente, o sistema deveria necessitar de algum tipo de tratamento excepcional na arquitetura. Em outras palavras, o sistema deveria possuir componentes redundantes, quer seja para implementar técnicas de tolerância a falhas que aumente a confiabilidade[2], quer seja para aumentar a disponibilidade de algum serviço crítico.

2. **O sistema deve possuir tanto componentes novos, quanto componentes reutilizados.** Dessa forma, espera-se verificar a eficiência da detecção e tratamento de exceções entre os componentes com e sem tratadores de nível de aplicação (ALE), que foram apresentados na Seção 2.4.2, que trata da abordagem de tratamento intra-componente. Por tratar das exceções lançadas pelas classes de implementação, nos componentes reutilizados esses tratadores não são especificados.

3. **Devem haver componentes com modelos de falhas distintos.** Com isso, pretende-se avaliar a eficiência da abordagem inter-componentes, apresentada na Seção 2.4.3.

4. **O desenvolvimento deveria possuir mais de uma iteração.** Para permitir o ajuste do método no decorrer da execução.

5. **O tempo de execução deveria ser minimizado.** A utilização do tempo, principalmente dos executores externos do estudo de caso, deveria ser otimizada.

A execução do estudo de caso, apresentada na Seção 5.4, ocorreu em uma empresa situada na cidade de São Paulo, que tem mais de 16 anos de experiência em consultoria e desenvolvimento de software para o mercado financeiro, a Autbank Projetos e Consultoria Ltda. Foram envolvidos dois analistas da empresa, sendo um deles com experiência no desenvolvimento de sistemas utilizando o processo *UML Components*. Com o intuito de otimizar a utilização do tempo, a execução do estudo de caso foi dividida em três etapas: (i) especificação do sistema; (ii) implementação e geração dos casos de teste; e (iii) execução dos testes e coleta dos últimos dados.

Devido ao caráter pragmático das etapas seguintes a ela, a especificação do sistema foi considerada a etapa mais importante para garantir a alta qualidade dos resultados. Por

[1] **do inglês *dependability***

[2] do inglês *reliability*

essa razão, apesar das outras etapas terem sido executadas na Unicamp com a participação do autor da dissertação, a produção dos artefatos de especificação ficou ao encargo de duas funcionárias da empresa, que sem nenhum contato prévio com o método MDCE+, especificaram o sistema no próprio local de trabalho.

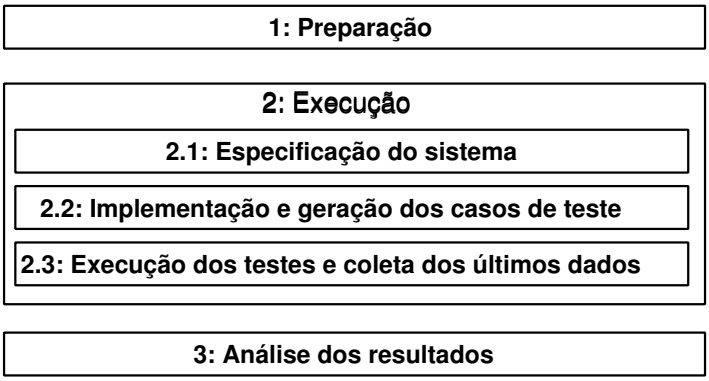

Figura 5.1: Etapas do Estudo de Caso

Uma característica importante da equipe de desenvolvimento é o fato de uma das pessoas envolvidas ser especialista no domínio do negócio, tendo mais de 10 anos de experiência. Esse fato facilitou a identificação de um número maior de exceções relativas às regras do negócio.

Com o intuito de proporcionar uma maior interação entre os integrantes da equipe de desenvolvimento, durante a fase de especificação, os modelos foram construídos manualmente, isto é, sem o auxílio de ferramentas CASE.

Finalizada a especificação, a próxima etapa foi a etapa de implementação dos componentes do sistema. Por ser considerada uma etapa de tradução dos modelos, ela foi desempenhada pelo próprio autor da dissertação.

Com o sistema implementado, chegou a hora de se aplicar os testes unitários, que foram especificados paralelamente ao desenvolvimento. Os detalhes dos testes não serão tratados nessa dissertação, mas estão disponíveis no documento completo do estudo de caso [dSBRMR05].

5.3 Preparação do Estudo de Caso

O principal objetivo do estudo de caso foi avaliar o método de desenvolvimento MDCE+, tanto com um enfoque quantitativo, quanto qualitativo. Do ponto de vista prático, a preparação da execução do estudo de caso foi dividida em duas partes: (i) condução da avaliação e métricas utilizadas; e (iii) cronograma de desenvolvimento.

5.3.1 Condução da Avaliação e Métricas Utilizadas

No tocante à condução do estudo de caso, inicialmente foram identificados os objetivos desejados. Após essa definição, foi escolhido o sistema a ser especificado, assim como os critérios de medição e as métricas utilizadas. Esses critérios foram definidos com base nos objetivos. Além disso, foram pesquisadas algumas diretrizes que auxiliam a condução de estudos de caso. Essas diretrizes oferecem instruções do que fazer (e evitar), a fim de minimizar interferências externas e melhorar a qualidade dos resultados finais obtidos.

Os principais objetivos pretendidos com a execução do estudo de caso são referentes a aspectos de tolerância a falhas, por exemplo, a qualidade das exceções, que consiste na medição do grau de independência entre os tipos das exceções e a plataforma de desenvolvimento utilizada. Outro critério importante é a qualidade do tratamento, isto é, a porcentagem de funcionalidades críticas recuperadas de situações excepcionais, de forma transparente ao cliente. Além dos critérios relativos a aspectos de tolerância a falhas, outros aspectos gerais também foram analisados, tais como a facilidade de manutenção e a melhoria da testabilidade decorrentes da utilização do método MDCE+.

Os resultados do estudo de caso foram obtidos de duas formas: (i) comparação de projetos similares existentes na própria empresa e (ii) comparação entre componentes do próprio projeto, desenvolvidos com variações (versões "relaxadas") do método avaliado. O plano de execução do estudo de caso foi elaborado de maneira a conter as atividades necessárias para que a avaliação execute normalmente [SK96, KJ97a, ZWB03]. Antes da execução de cada fase do processo avaliado, foi necessário dedicar uma parte do tempo para atividades de treinamento teórico e prático. Esse treinamento foi necessário, uma vez que o estudo de caso foi conduzido por pessoas sem familiaridade com o processo avaliado. Essa falta de contato prévio foi considerado uma vantagem para a execução do estudo de caso, uma vez que evita interferências humanas na qualidade dos seus resultados [KP98, SK96, KJ97a, ZWB03].

Outras atividades preventivas executadas durante a condução do estudo de caso foram inspeções, cujos objetivos principais eram evitar dúvidas de interpretação e relaxamento na execução das atividades do processo [SK96, KJ97a, ZWB03]. Além disso, as dificuldades de entendimento e todas as modificações do método efetuadas no decorrer da execução foram devidamente documentadas.

Durante a execução do estudo de caso e a análise dos resultados dos testes, foram obtidas algumas medidas, como por exemplo o número de exceções descobertas e o tempo gasto no desenvolvimento, que foram medidos separadamente por cada fase, entre outras. Além desses valores absolutos, foram feitas avaliações qualitativas tanto em relação ao processo de desenvolvimento, quanto ao sistema final produzido.

5.3.2 Cronograma de Desenvolvimento

Nesta seção, são apresentadas as principais atividades executadas para o desenvolvimento do estudo de caso, juntamente com o seu cronograma de execução. A Tabela 5.1 mostra a distribuição dessas atividades durante os 16 dias de duração do estudo de caso.

Tabela 5.1: Atividades para a Execução do Estudo de Caso

Atividade		Dias na empresa				Dias na universidade			
Id	Descrição	1/2	3/4	5/6	7/8	9/10	11/12	13/14	15/16
1	Treinamento	•							
2	Especificação de requisitos	•							
3	Especificação dos componentes	•	•	•	•				
4	Execução da fase de provisionamento				•	•	•	•	•
5	Execução da fase de montagem								•
6	Coleta das métricas qualitativas relativas ao método utilizado				•				
7	Coleta das métricas quantitativas	•		•	•				•
8	Análise dos resultados								•

5.4 Execução do Estudo de Caso

A seguir, as Seções 5.4.1 a 5.4.8 mostram os principais artefatos gerados em cada uma das fases da execução do processo adaptado. Como mostrado no Capítulo 4, o processo é constituído de seis fases, sendo a fase de especificação dividida em três estágios.

5.4.1 Especificação de Requisitos

Primeiramente foi definido o **modelo conceitual do negócio**, que representa a estruturação das entidades básicas do domínio. Após a especificação desse modelo, foram identificadas as entidades críticas, isto é, as entidades consideradas essenciais para a execução das principais atividades do negócio. Esse modelo, com a respectiva representação das entidades críticas, está representado na Figura 5.2.

Além da especificação desse modelo, os requisitos do sistema foram analisados e especificados. Em seguida, os requisitos foram modelados através de casos de uso UML, conforme o modelo sugerido pelo método MDCE+ na Seção 3.2.4. Após essa especificação, foram identificadas exceções, baseado nas violações das assertivas especificadas. A Figura 5.3 mostra os **casos de uso identificados** a partir das funcionalidades espe-

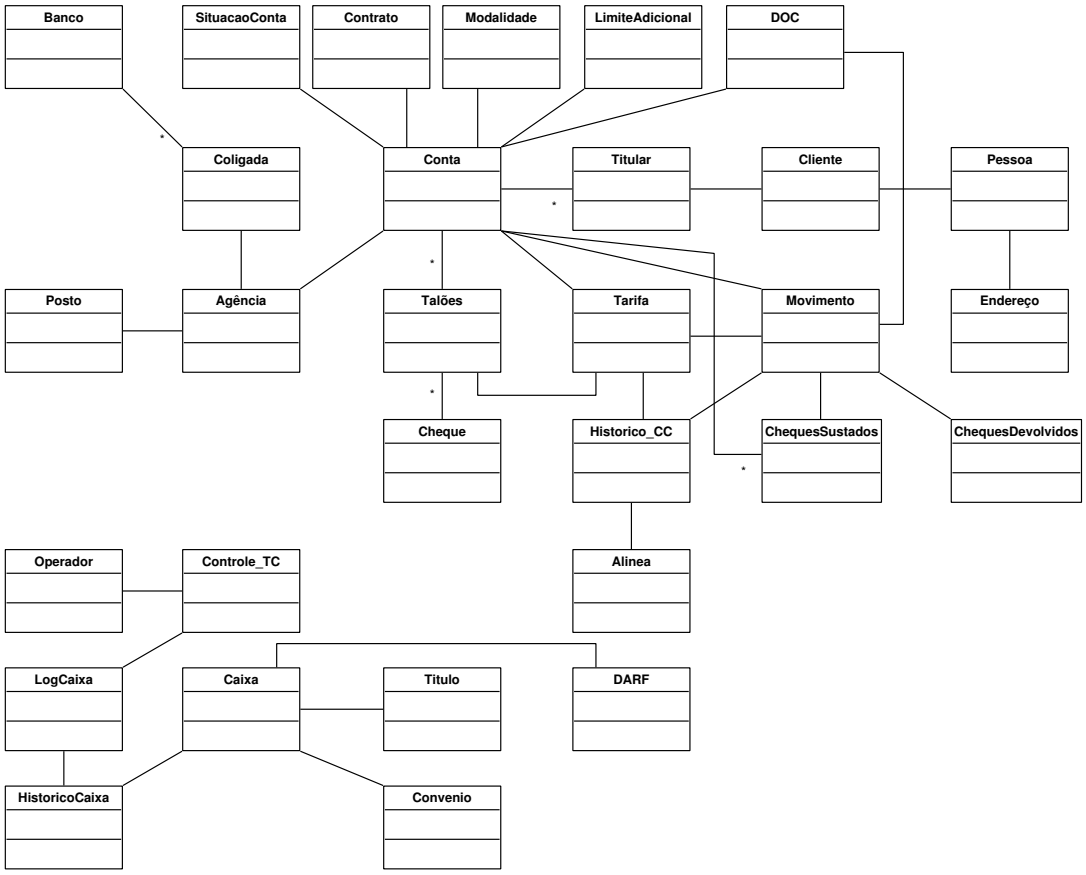

Figura 5.2: Modelo Conceitual do Negócio

radas para o sistema. Em seguida, a Tabela 5.2 mostra a **especificação detalhada do caso de uso** Cancelar Contrato da Conta.

Baseado na análise das assertivas especificadas na Tabela 5.2, foram detectadas 13 exceções: (i) `TEAgenciaNaoEstaAbertaException`; (ii) `TEAgenciaInvalidaException`; (iii) `TEAgenciaNaoCadastradaException`; (iv) `TEContaInvalidaException`; (v) `TEContaNaoCadastradaException`; (vi) `TEUsuarioInvalidoException`; (vii) `TETransacaoNaoFoiConcluidaException`; (viii) `TEModalidadeNaoCadastrada-Exception`; (ix) `TETaxaDeveSerMaiorZeroException`; (x) `TEModalidadeInvalida-Exception`; (xi) `TEModalidadeIgualModalidadeContaException`; (xii) `TETipoPessoa-NaoCorrespondeTipoPessoaContaException`; e (xiii) `TEContratoINEstaCancelado-Exception`; Porém, devido à ausência de tratamento para a maioria dessas exceções (a maioria apenas sinaliza o erro), a Tabela 5.3 mostra a especificação de apenas uma delas, a exceção `TEAgenciaNaoEstaAbertaException`. Devido ao fato de possuírem um tratamento semelhante, todas as exceções arquiteturais decorrentes de alguma falha na execução (exceções de defeito) são aninhadas em uma exceção genérica (`TETransacaoNao-`

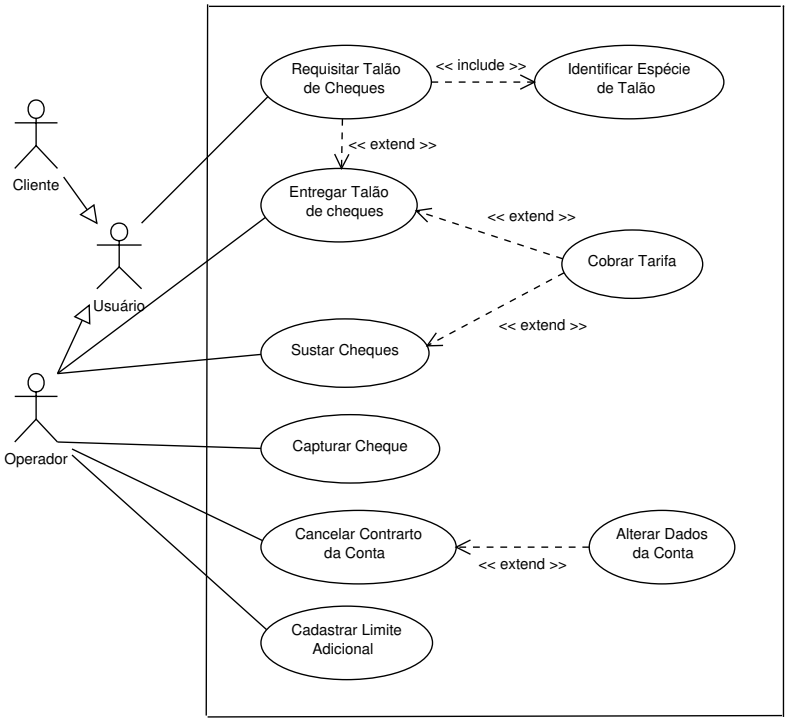

Figura 5.3: Diagrama de Casos de Uso

`FoiConcluidaException`). No caso da funcionalidade crítica Cancelar Contrato da Conta, essa exceção será tratada no nível arquitetural do sistema.

O fato de não terem sido especificados tratadores para essas exceções mostra a pouca preocupação com um tratamento excepcional de forma elaborada. Segundo os desenvolvedores, a maioria das exceções especificadas nos sistemas da empresa tem o intuito de documentar os tipos esperados de falhas. Dessa forma, o tratamento realizado é o registro das exceções ocorridas através de abordagens de *logging*. Por se tratar de um requisito não-funcional implementável na arquitetura, o registro das falhas através de *logging* só será especificado nos conectores arquiteturais, mais especificamente na fase de montagem do sistema (Seção 5.4.8).

5.4.2 Definição dos Aspectos Gerenciais

Durante a definição dos aspectos gerenciais, sentiu-se a necessidade de se conhecer a política da empresa em relação ao desenvolvimento de sistemas. Essa etapa consistiu de quatro etapas complementares: (i) definição das restrições do desenvolvimento; (ii) definição das restrições de reutilização; (iii) identificação das funcionalidades críticas; e (iv) definição das *releases* para as iterações.

Tabela 5.2: Especificação do Caso de Uso Cancelar Contrato da Conta

Cancelar Contrato da Conta	
Descrição:	Efetua o cancelamento do contrato da conta, mas se a conta foi cadastrada no dia, efetua-se apenas uma alteração da modalidade e das taxas dessa nova modalidade.
Participantes:	Operador
Pré-condições:	1- O nome do cliente deve ser != NULL E != " "; 2- A agência da conta deve ser != NULL E != " " E tem que estar cadastrada; 3- A conta deve ser != NULL E != " " E tem que estar cadastrada; 4- O código da modalidade ori. deve ser != NULL E != " " E tem que estar cadastrado; 5- O tipo de pessoa da modalidade ori. tem que corresponder ao mesmo tipo de pessoa do tipo de depósito; 6- O contrato não pode estar cancelado;
Invariantes:	1- A agência da conta deve estar aberta.
Cenário primário:	1- Recuperar a agência da conta; 2- Recuperar a conta; 3- Recuperar a modalidade Ori; 4- Recuperar tipo de depósito; 5- Se NÃO cadastrado no dia, então 5.1- Cancelar contrato;
Cenários alternativos:	Se o cadastrado foi feito no mesmo dia 5.1- << *extend* >> Alterar dados da conta;
Pós-condições:	1- Se o cadastrado foi feito no dia, alguns campos da conta serão atualizados; Senão, deverá ser alterado o estado do contrato para cancelado.

Tabela 5.3: Um Cenário Excepcional do Caso de Uso Cancelar Contrato da Conta

Cancelar Contrato da Conta	
Cenário Falho 1:	Caso a agência não esteja aberta, a execução não pode continuar e uma exceção é lançada.
Sinal:	O status da agência informada ser diferente de aberto.
Ponto de identificação:	Após o passo 1 do cenário principal.
Tratador:	Aninhar a exceção `TEAgenciaNaoEstaAbertaException` em uma exceção `TETransacaoNaoFoiConcluidaException` (*nesting*).
Pós-condições:	Foi propagada a exceção `TETransacaoNaoFoiConcluidaException`, com a exceção `TEAgenciaNaoEstaAbertaException` aninhada a ela. O estado do sistema continua o mesmo.

As **restrições de desenvolvimento** identificadas são listadas a seguir:

1. Restrições estruturais. A empresa possui alguns sistemas utilitários que podem ser utilizados para oferecer serviços básicos, como a implementação da comunicação entre os vários sistemas da empresa. Essa restrição será levada em consideração no projeto arquitetural do sistema.

2. Restrição da Linguagem de Programação. O sistema deve ser implementado utilizando a linguagem de programação Java.

3. Restrições gerais. Para nomear os diversos artefatos produzidos, deve-se utilizar uma nomenclatura padronizada da empresa.

Além dessas restrições, foram identificadas duas **restrições de reutilização**:

1. Restrições de Tempo de Desenvolvimento. O desenvolvimento deve priorizar a agilização do desenvolvimento. Em outras palavras, a reutilização deve ser maximizada.

2. Só reutilizar componentes da própria empresa. Durante as atividades de reutilização, os componentes desenvolvidos por terceiros não devem ser levados em consideração.

Em relação às **funcionalidades críticas identificadas**, por apresentar requisitos críticos de disponibilidade, os casos de uso Cancelar Contrato da Conta, Alterar Dados da Conta e Sustar Cheques foram considerados críticos.

Finalmente, a última etapa dessa fase é a definição das *releases* para as iterações. Para tornar o desenvolvimento mais dinâmico, decidiu-se iterar uma funcionalidade de cada vez. Sendo assim, como mostrado na Tabela 5.4, foram definidas seis iterações.

Tabela 5.4: Iterações Definidas

Ordem	Funcionalidade	Casos de Uso
1^a	Requisição de talão de cheques	Requisitar Talão de Cheques Identificar Espécie de Talão
2^a	Entrega de talão de cheques	Entregar Talão de Cheques Requisitar Talão de Cheques Cobrar Tarifa
3^a	Sustação de cheques	Sustar Cheques Cobrar Tarifa
4^a	Captura de cheque para compensação	Capturar Cheque
5^a	Cancelamento do Contrato da Conta	Cancelar Contrato da Conta Alterar Dados da Conta
6^a	Cadastramento de limite adicional	Cadastrar Limite Adicional

5.4.3 Projeto Arquitetural

Nesta fase foi escolhida a arquitetura adotada para o sistema. Além das restrições impostas pelo *UML Components*, a arquitetura adotada para o estudo de caso segue restrições impostas pelo ambiente da empresa, tais como componentes utilitários e a maneira de acesso aos dados. Essas restrições foram definidas na Seção 5.4.2. A arquitetura definida pode ser vista na Figura 5.4.

A camada `bd` da Figura 5.4 contém os componentes que oferecem as operações de acesso à base de dados. Os componentes da camada `utilitária` oferecem alguns serviços gerais, isto é, que são úteis em outros domínios de desenvolvimento, por exemplo, validadores de campos de formulário. Finalmente, a camada `framework` contém os componentes que oferecem outros serviços gerais, porém de maior granularidade, por exemplo, serviços de comunicação com alguns sistemas legados existentes. As camadas de `sistema` e de `negócio` possuem as mesmas características sugeridas pelo processo *UML Components* visto na Seção 4.1.

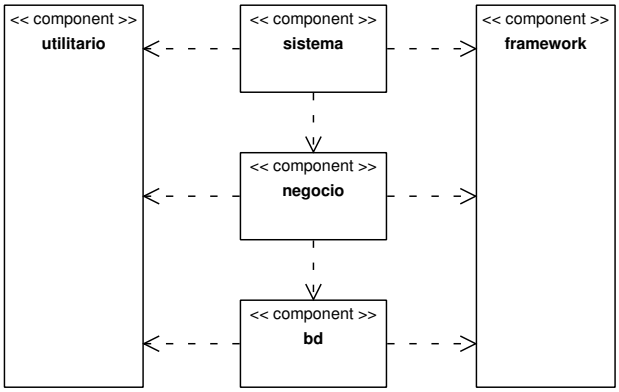

Figura 5.4: Arquitetura do Sistema

5.4.4 Identificação dos Componentes

Como visto na Seção 4.6, o objetivo desse estágio da especificação é identificar os componentes a partir das suas interfaces providas. Sendo assim, os principais artefatos produzidos são: (i) modelo de tipos de negócio; (ii) interfaces providas (normais e excepcionais); (iii) classes das exceções; e (iv) componentes identificados.

O **modelo de tipos de negócio** restringe o modelo conceitual (Figura 5.2) de acordo com os limites do sistema [CD00]. A Figura 5.5 mostra esse modelo, com as entidades principais destacadas com o estereótipo << *core* >>, segundo a notação do processo *UML Components*.

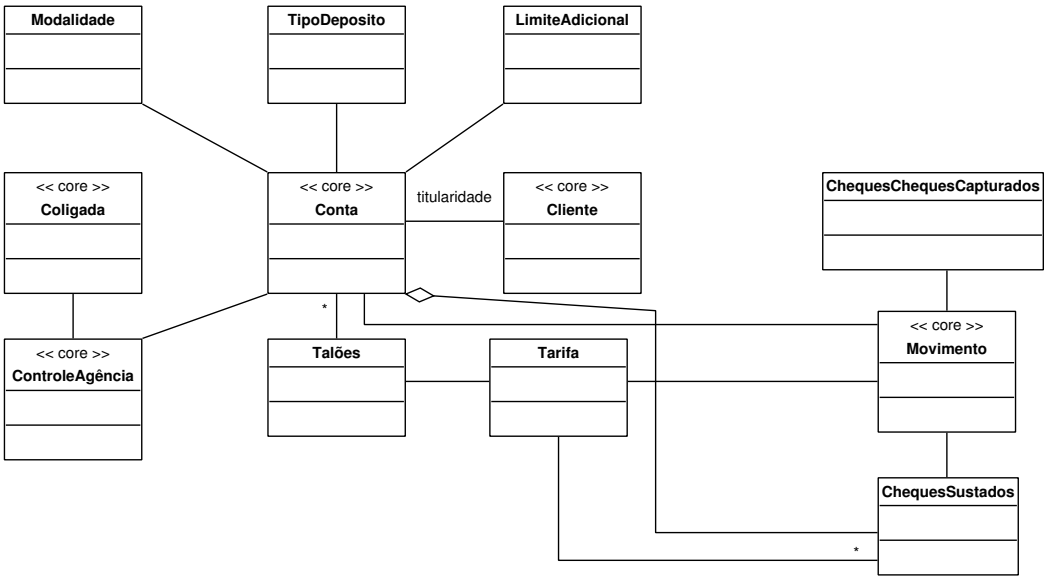

Figura 5.5: Modelo de Tipos de Negócio

Como visto, as interfaces normais são classificadas em duas categorias: interfaces de sistema e interface de negócio. As **interfaces identificadas** são mostradas na Figura 5.6. A sua classificação entre sistema, negócio ou excepcional é representada pelo pacote na qual elas se inserem.

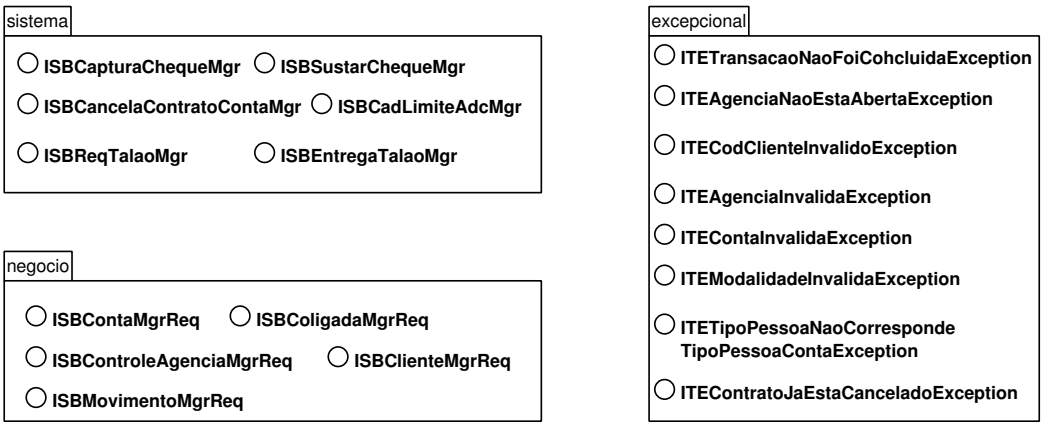

Figura 5.6: Interfaces Identificadas

Em relação ao processamento das exceções, além das interfaces e operações identificadas para os tratadores, as classes das exceções também devem ser especificadas. A Figura 5.7 mostra as **classes que representam as exceções** identificadas.

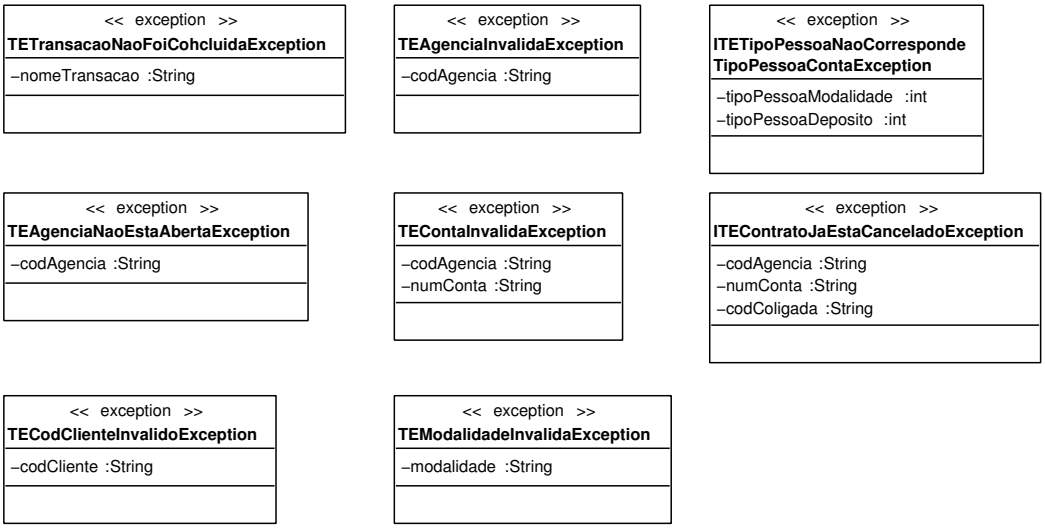

Figura 5.7: Exceções Identificadas

Com as interfaces normais e excepcionais identificadas, o próximo passo é a criação dos componentes do sistema. O procedimento de criação é semelhante tanto os componentes

normais (de sistema e de negócio), quanto para os componentes excepcionais (tratadores de exceções). A criação dos componentes consiste basicamente no agrupamento das interfaces coesas de uma mesma camada arquitetural, em componentes. A Figura 5.8 mostra os **componentes identificados** nessa etapa.

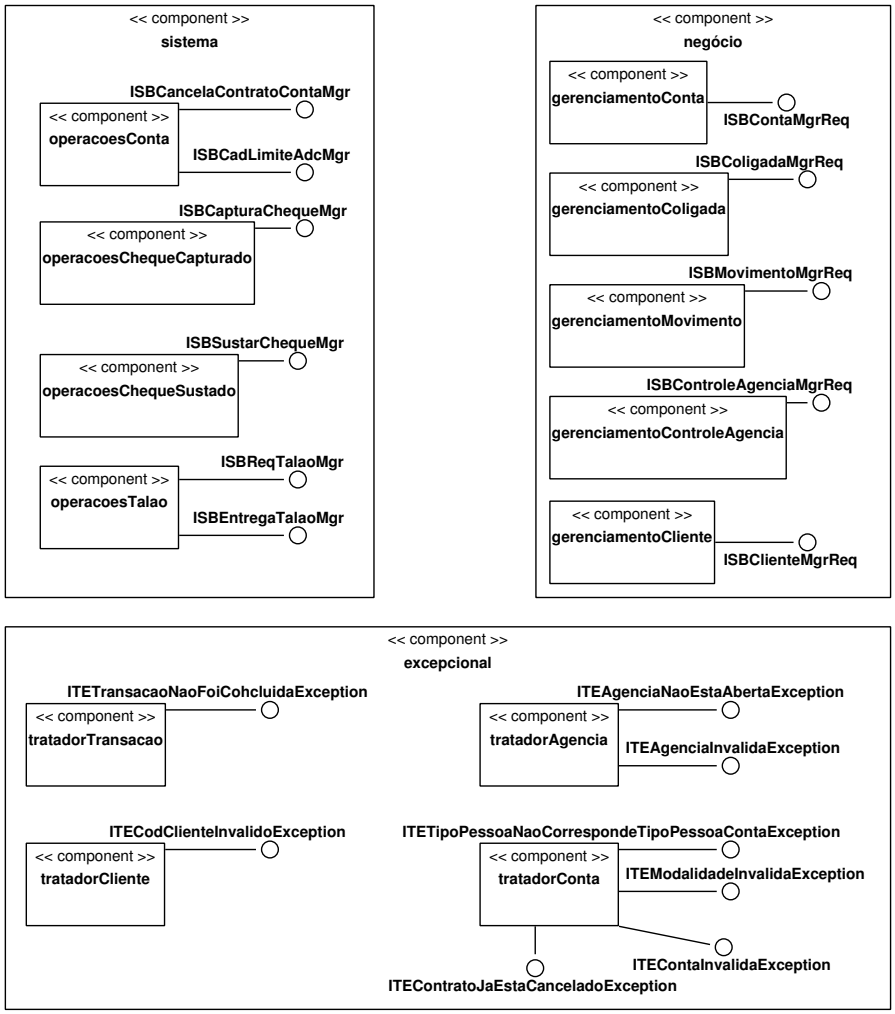

Figura 5.8: Componentes Identificados

5.4.5 Interação entre os Componentes

Como visto na Seção 4.7, o objetivo deste estágio da especificação é analisar o fluxo de informação que flui entre os componentes do sistema, sejam eles normais ou excepcionais. Através da análise desses fluxos, são identificadas as operações dos componentes da camada de negócio. Além disso, as assinaturas das operações dos componentes da camada de sistema são refinadas. De forma complementar, são especificadas as dependências entre os componentes do sistema. Através dessas dependências, são produzidos dois artefatos importantes da fase de especificação: (i) os componentes arquiteturais, através das dependências entre os componentes normais e excepcionais; e (ii) a configuração da arquitetura do sistema, através das dependências entre os componentes normais.

Inicialmente, conforme proposto pelo método MDCE+, a interação entre os componentes é documentada através de diagramas de atividades UML. A Figura 5.9 mostra a **interação realizada a partir da simulação da operação** cancelarContrato(), relativa ao caso de uso Cancelar Contrato da Conta. Nessa figura, a detecção das exceções não está representada. A representação completa da interação pode ser vista na Figura 5.10.

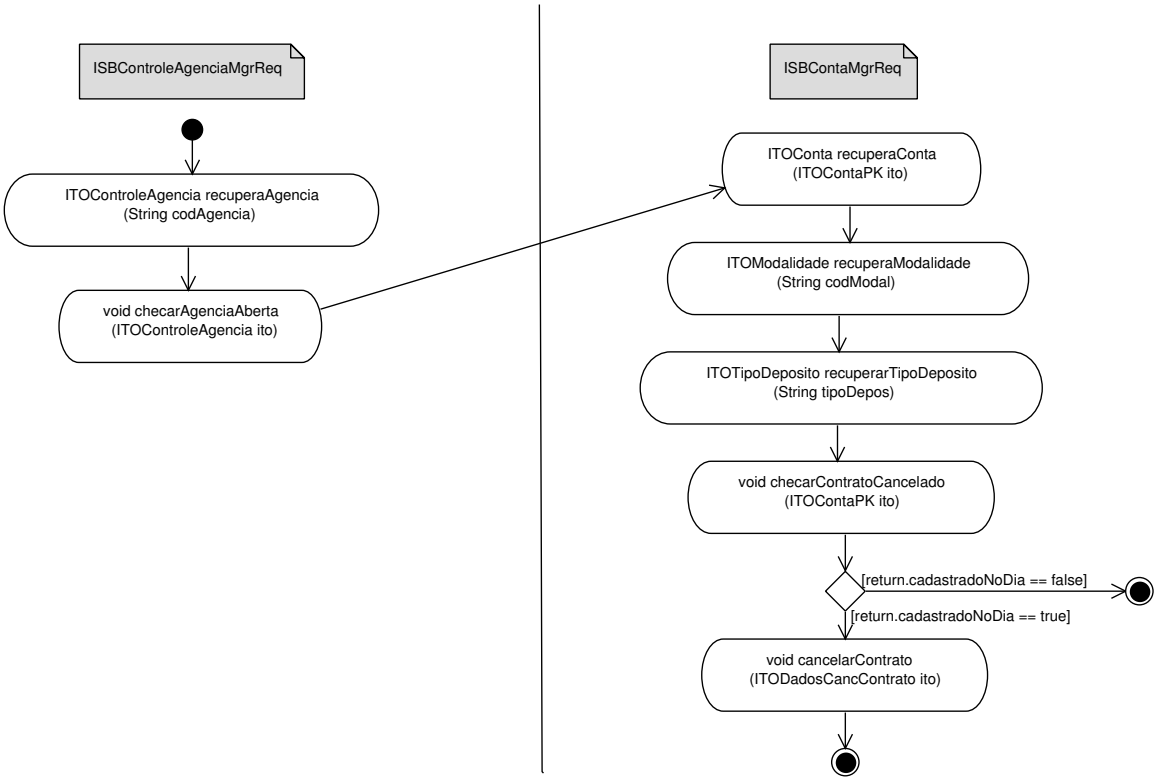

Figura 5.9: Interação da Operação cancelarContrato()

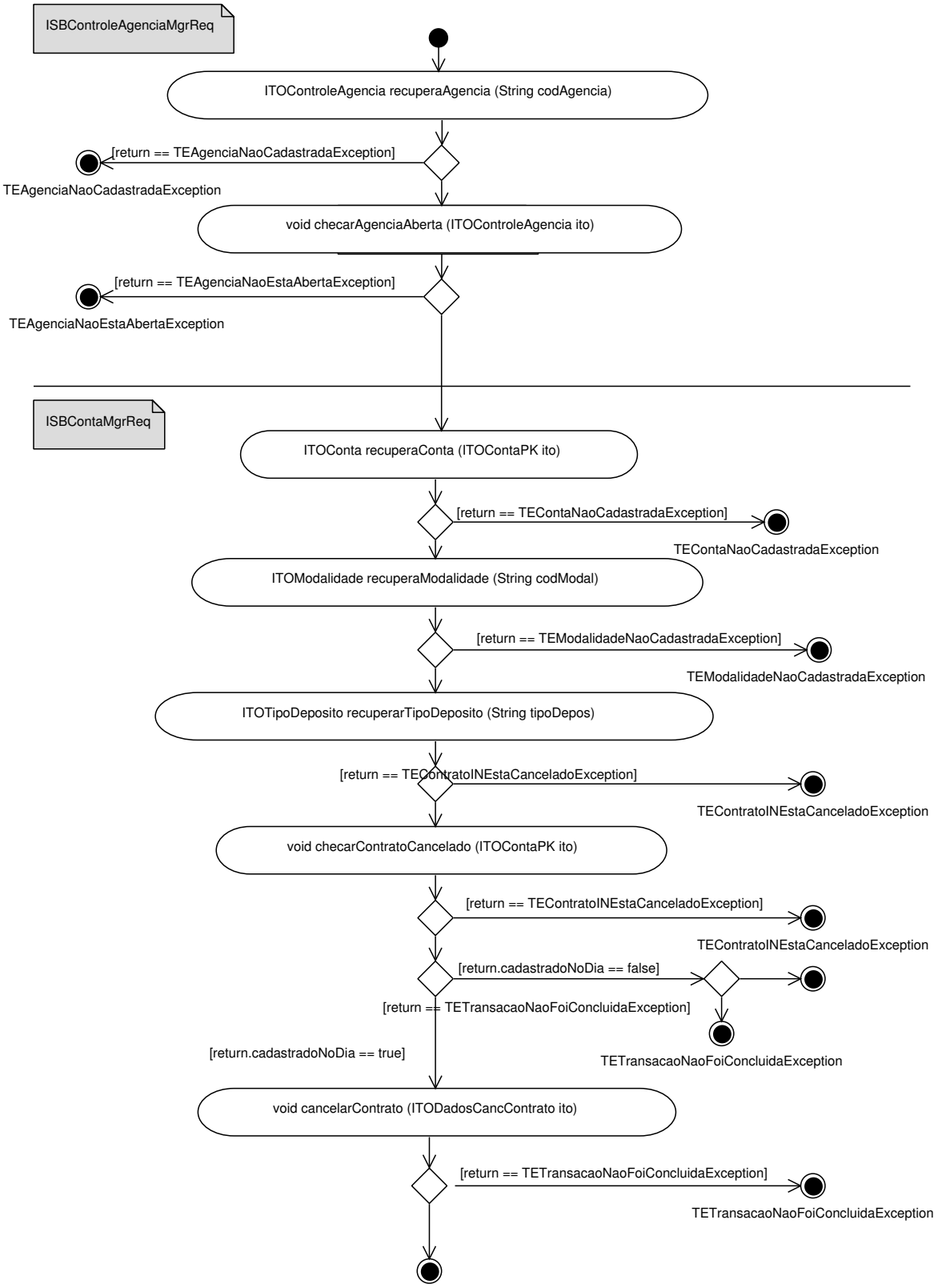

Figura 5.10: Interação da Operação cancelarContrato() com Detecção de Exceções

Além da representação do comportamento normal dos componentes, durante a interação entre os componentes devem ser modeladas as interações dos tratadores de exceções. Devido às exceções identificadas até o momento não possuírem um tratamento específico, até este momento, a representação dessas interações não foi necessária. A seguir, na fase de montagem do sistema (Seção 5.4.8), será representada a colaboração relativa ao tratador da exceção `TETransacaoNaoFoiConcluidaException`, que será identificado adiante.

Com a análise das interações especificadas nesta fase, foi possível refinar as interfaces do sistema. Esse refinamento consistiu tanto na descoberta das operações do negócio, quanto no refinamento das assinaturas das operações de sistema. As **interfaces refinadas** são mostradas na Figura 5.11. Durante esse refinamento, percebeu-se a necessidade de se criar alguns tipos estruturados de dados, que na terminologia do *UML Components* são conhecidos como *data types*. A Figura 5.12 mostra os **tipos de dados utilizados** nas operações envolvidas na interação.

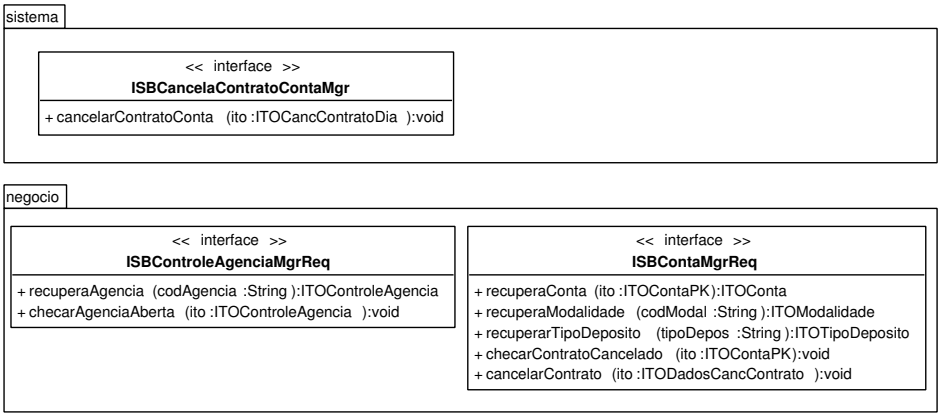

Figura 5.11: Interfaces Refinadas com a Colaboração

Com as dependências entre os componentes já identificadas, é chegado o momento de especificar os componentes arquiteturais e a arquitetura do sistema. O **componente arquitetural** operacoesContaArq é mostrado na Figura 5.13. Após isso, são definidas as interfaces requeridas de cada componente. Seguindo as diretrizes do método MDCE+, deve ser especificada uma interface para cada camada da arquitetura que o componente interage. A Figura 5.14 mostra as interfaces requeridas pelo componente operacoesContaArq. Nessa figura é possível perceber a contextualização do componente na arquitetura, através dos componentes que oferecem as suas interfaces requeridas. Essas ligações entre os componentes são representadas através de dependências UML.

Em relação à revisão do modelo de falhas, durante a análise das propagações das exceções, foram identificadas duas exceções novas: (i) `TETipoDepositoNaoCadastradoException` e (ii) `TETaxaInvalidaException`. Essas exceções são conseqüência da pro-

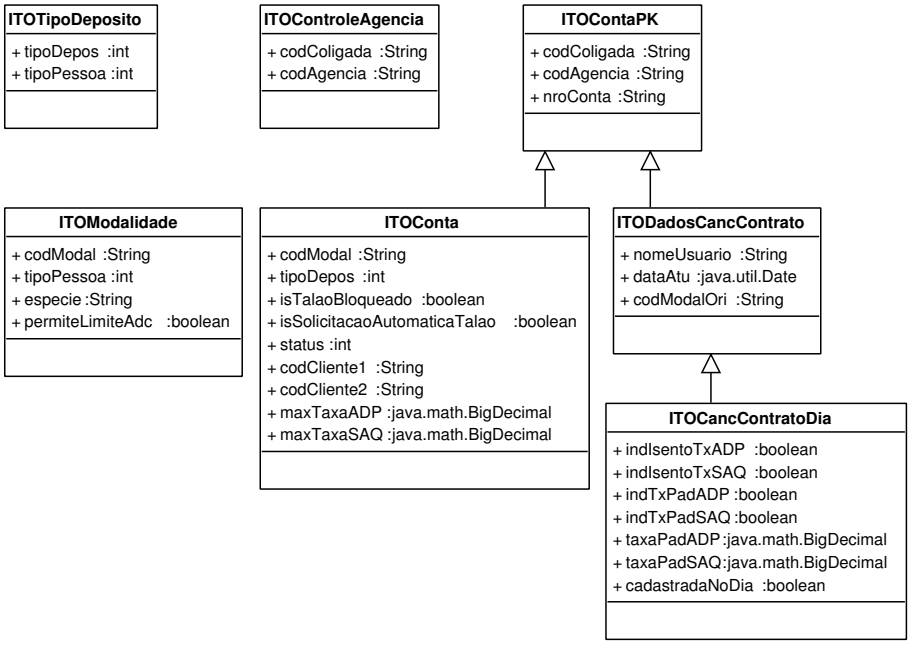

Figura 5.12: Tipos de Dados Necessários

pagação de outras situações excepcionais que podem ocorrer durante o acesso aos dados, que é feito pelos componentes da camada de persistência (Figura 5.4).

Além disso, devido ao requisito de disponibilidade da operação cancelarContrato(), foi especificado um novo tratador para a exceção `TETransacaoNaoFoiConcluidaException`. Como pode ser visto na especificação desse tratador, apresentada na Tabela 5.5, esse tratador realiza uma reconfiguração entre os componentes do sistema. Por se tratar de um tratamento realizado na arquitetura de software, esse tratador não faz parte de nenhum componente excepcional e será implementado no próprio conector arquitetural, durante a montagem do sistema (Seção 5.4.8).

Tabela 5.5: Especificação do Caso de Uso Tratar Erro de Cancelamento de Contrato

Tratar Erro de Cancelamento de Contrato	
Descrição:	Ocorre uma tentativa de reconfigurar o sistema para utilizar um componente redundante. Em seguida, tenta-se executar a funcionalidade novamente.
Participantes:	Operador.
Pré-condições:	Nenhuma.
Invariantes:	Nenhuma.
Cenário primário:	1- Reconfigurar os componentes do sistema; 2- Tentar a execução novamente; 3- Se *não* conseguiu E não tem como reconfigurar; 3.1- Propagar exceção `TETransacaoNaoFoiConcluidaException`.
Pós-condições:	O serviço foi executado normalmente, isto é, de forma transparente para o usuário.

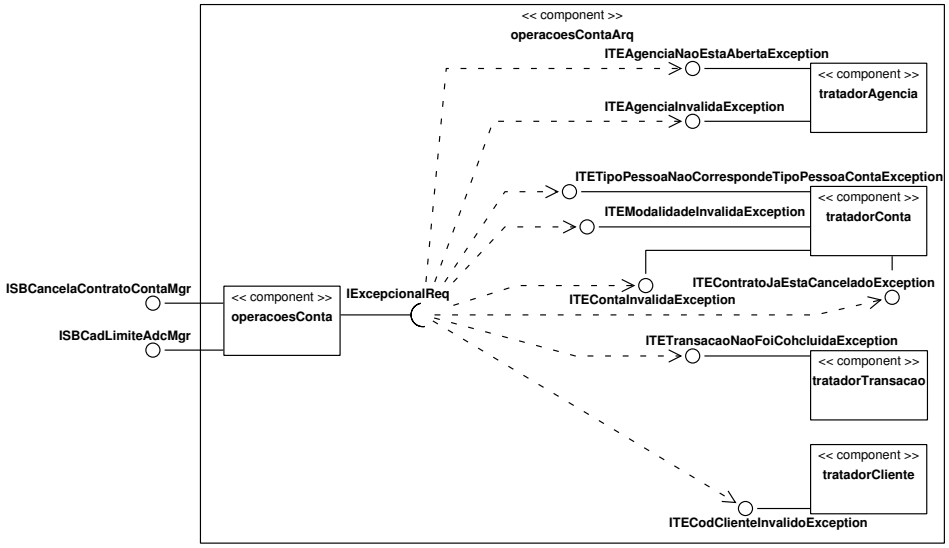

Figura 5.13: Componente Arquitetural operacoesContaArq

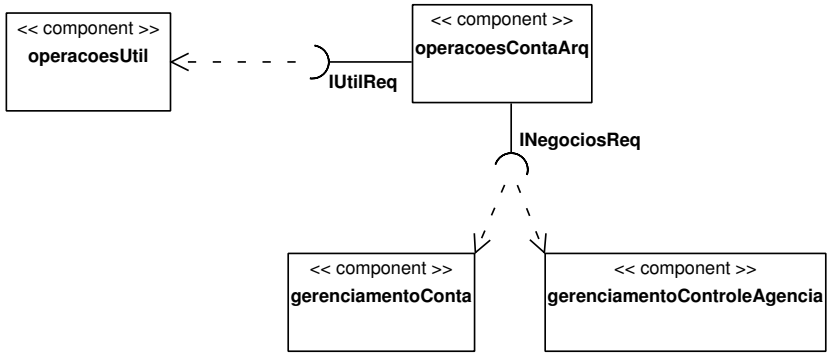

Figura 5.14: Interfaces Requeridas do Componente operacoesContaArq

Após a análise das propagações das exceções, o próximo passo para o refinamento do modelo é a classificação das exceções em dois grupos: (i) **exceções arquiteturais** e (ii) **exceções não-arquiteturais**. A Tabela 5.6, apresenta a lista completa das exceções com as respectivas classificações em um desses dois grupos. Como visto nessa tabela, apesar do grande número de exceções identificadas no decorrer da especificação, do ponto de vista da arquitetura do sistema, existe um número relativamente reduzido de exceções. Isso se deve ao fato da maioria das exceções não ser tratada em nenhum contexto, sendo utilizadas somente para documentar o modelo de falhas e registrar os erros ocorridos através do *logging* de suas informações contextuais.

Tabela 5.6: Classificação das Exceções de Acordo com a Propagação

Exceção	Classificação
TEAgenciaNaoEstaAbertaException	não-arquitetural
TEAgenciaInvalidaException	**arquitetural**
TEAgenciaNaoCadastradaException	não-arquitetural
TEContaInvalidaException	**arquitetural**
TEContaNaoCadastradaException	não-arquitetural
TEUsuarioInvalidoException	**arquitetural**
TETransacaoNaoFoiConcluidaException	**arquitetural**
TEModalidadeNaoCadastradaException	não-arquitetural
TETaxaDeveSerMaiorZeroException	não-arquitetural
TEModalidadeInvalidaException	**arquitetural**
TEModalidadeIgualModalidadeContaException	não-arquitetural
TETipoPessoaNaoCorrespondeTipoPessoaContaException	não-arquitetural
TEContratoINEstaCanceladoException	não-arquitetural
TETipoDepositoNaoCadastradoException	não-arquitetural
TETaxaInvalidaException	**arquitetural**

5.4.6 Especificação Final dos Componentes

Basicamente, o papel da fase de especificação final é refinar a especificação das assertivas. Esse refinamento é feito através da especificação formal das pré-, pós-condições e invariantes. Além dessa etapa opcional de formalização das assertivas, para cada interface normal existente é definido um modelo, denominado modelo de informação da interface[3]. Este modelo consiste da identificação das entidades do modelo de tipos do negócio[4] que sejam relevantes para a interface. Além dessas entidades, são destacados os tipos de dados utilizados. A Figura 5.15 mostra o modelo de informação da interface `ICancelamentoContratoMgt`

5.4.7 Provisionamento dos Componentes

Com os componentes já especificados, é chegado o momento de materializá-los. Antes de iniciar a estruturação interna do componente, é necessário decidir para cada componente os que serão reutilizados e os que serão implementados. No caso específico deste estudo de caso, os componentes que implementam as funcionalidades específicas dos casos de uso (componentes de sistema) foram implementados. Enquanto isso, todas as operações de infra-estrutura requeridas pelo sistema (componentes de negócio) foram reutilizadas de componentes existentes.

Quanto aos componentes reutilizados, foi necessário realizar um mapeamento entre as suas operações e as operações requeridas especificadas. Para isso, foram implementados *wrappers*, cuja função é mascarar a assinatura real das operações reutilizadas, a fim de tornar a reutilização transparente para o componente cliente. Do ponto de vista

[3]do inglês *interface information model*
[4]do inglês *business type model*

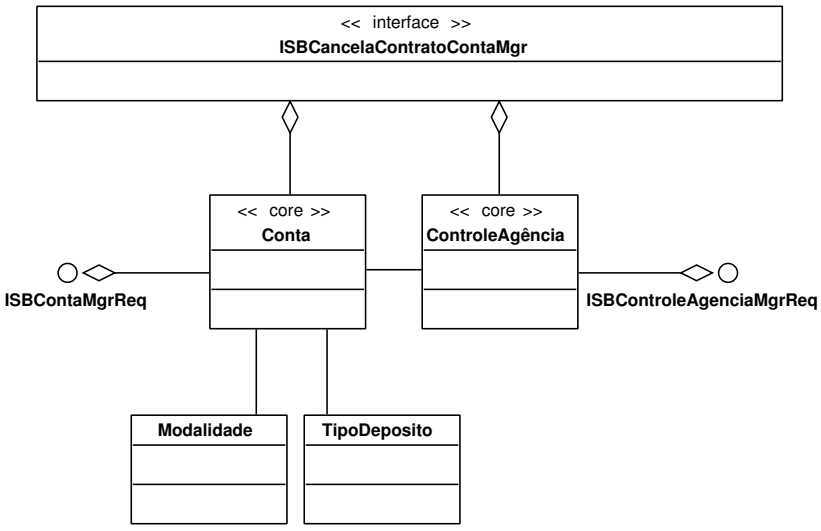

Figura 5.15: Modelo de Informação da Interface `ICancelamentoContratoMgt`

dos aspectos de confiabilidade, foram especificados adaptadores de fronteira (BLE). A Figura 5.16 mostra a estruturação do componente gerenciamentoContas, constituído a partir da reutilização do componente dadosConta.

No caso dos componentes implementados, foi necessário especificar a sua estrutura interna de acordo com alguma tecnologia de desenvolvimento disponível. O modelo de materialização utilizado para os componentes foi o modelo COSMOS. A Figura 5.17 mostra a estruturação interna do componente operacoesConta. Como pode ser visto nessa figura, a interface excepcional requerida desse componente foi classificada de acordo com os três níveis de tratamento sugeridos na Seção 2.4: (i) tratador de fronteira `ProvidedBLE`; (ii) tratador de aplicação `ALE`; e (iii) tratador de fronteira `RequiredBLE`.

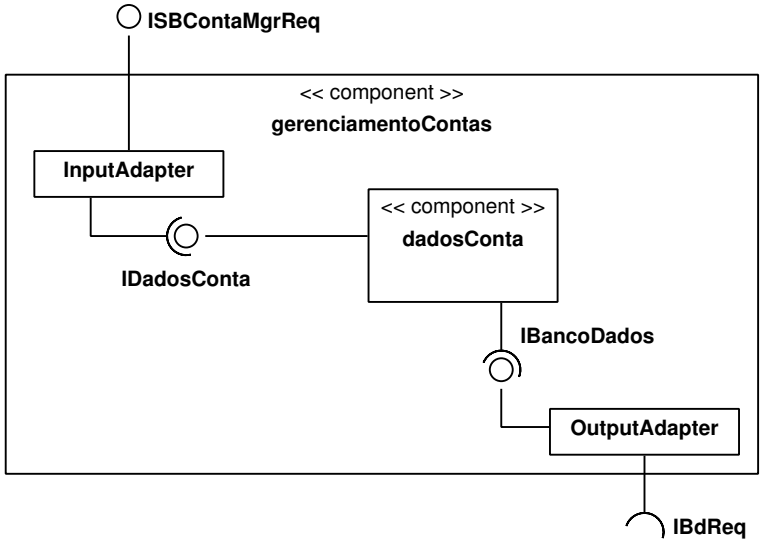

Figura 5.16: Estruturação Interna do Componente gerenciamentoContas

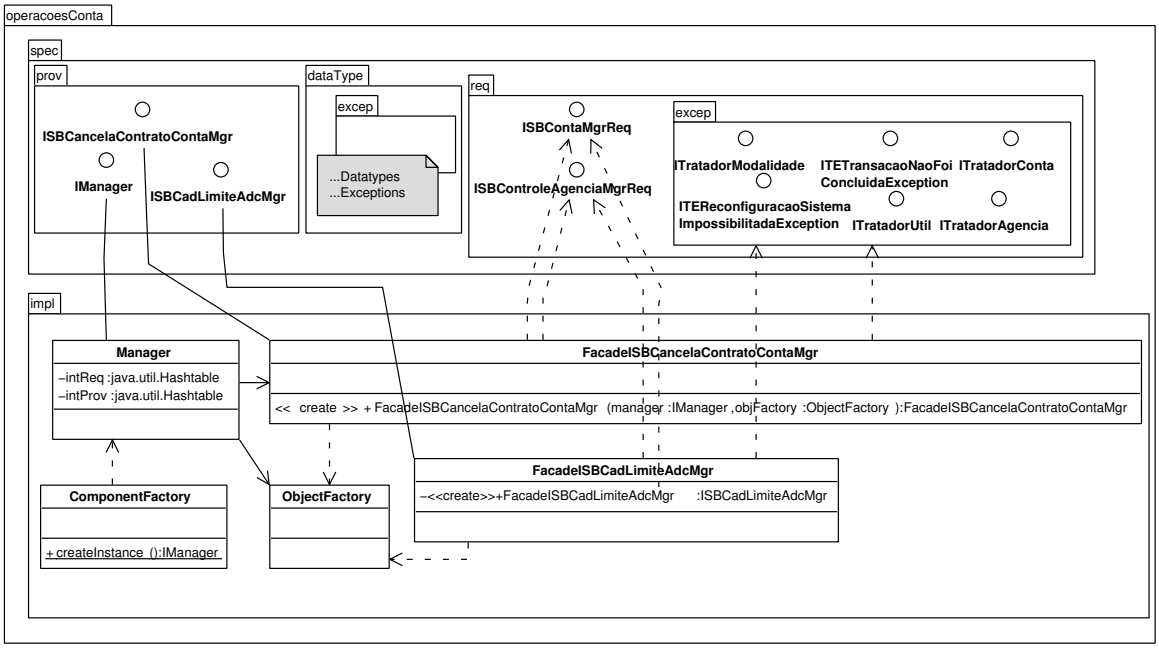

Figura 5.17: Estruturação Interna do Componente operacoesConta

5.4.8 Montagem do Sistema

Devido ao foco na conexão entre os componentes do sistema, a fase de montagem consiste basicamente da materialização da arquitetura do sistema. Em outras palavras, consiste na implementação dos conectores e do programa principal. No método MDCE+ existem dois tipos de conectores: (i) **internos**, que integram as partes normal e excepcional do componente tolerante a falhas ideal; e (ii) **arquiteturais**, que conectam dois ou mais componentes arquiteturais, além de implementarem os tratadores de exceções na arquitetura do sistema.

Por intermediarem a comunicação entre componentes arquiteturais, as especificações dos conectores arquiteturais devem ser refinadas, conforme mostrado na Seção 3.8.2. Esse refinamento consiste na especificação dos tratadores de exceções que envolvem mais de um componente do sistema e dos requisitos não-funcionais que possam ser materializados na arquitetura. No contexto desse estudo de caso, o conector conOperacoesContaSessao, mostrado na Figura 5.20, além de materializar a comunicação entre o componente operacoesContaArq e o componente interfaceESessaoArq, esse conector deve implementar o serviço de *logging* desejado para o sistema.

Além disso, do ponto de vista do tratamento de exceções, o conector conOperacoesContaSessao também deve oferecer o tratamento para a exceção `TETransacaoNaoFoiConcluidaException`, que pode ser lançada pela operação cancelarContrato() do componente operacoesConta. Esse tratador foi especificado na fase de interação entre os componentes e é mostrado na Tabela 5.5. A Figura 5.18 mostra a seqüência de execução do tratador.

Após a execução dessa fase, a arquitetura do sistema é especificada. A Figura 5.19 mostra o componente arquitetural operacoesContaArq. Os conectores internos e arquiteturais foram implementados utilizando o COSMOS. Na Figura 5.19, os conectores internos ligam o componente normal operacoesConta e os respectivos componentes excepcionais, materializando o componente arquitetural, que segue as diretrizes de um componente ideal.

A Figura 5.20 mostra a integração do componente operacoesContaArq com o componente da camada superior, que utiliza os seus serviços. Perceba que essa ligação é intermediada por um conector arquitetural. Os tratadores implementados nesse conector desempenham funções de reconfiguração do sistema, uma vez que caso a operação `cancelarContrato(...)` não seja executada com sucesso, o conector tenta executar o serviço por um componente redundante. A Figura 5.21 mostra o diagrama de seqüência que simula a execução dessa reconfiguração. Nesse caso, optou-se por adotar uma configuração temporária.

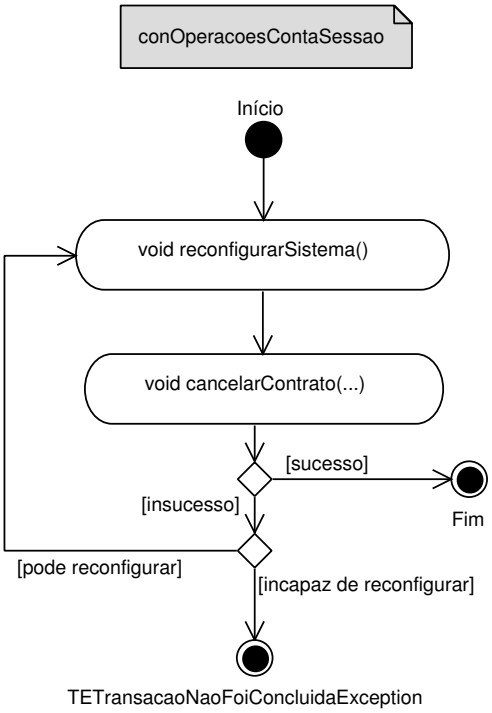

Figura 5.18: Comportamento do Tratador de `TETransacaoNaoFoiConcluidaException`

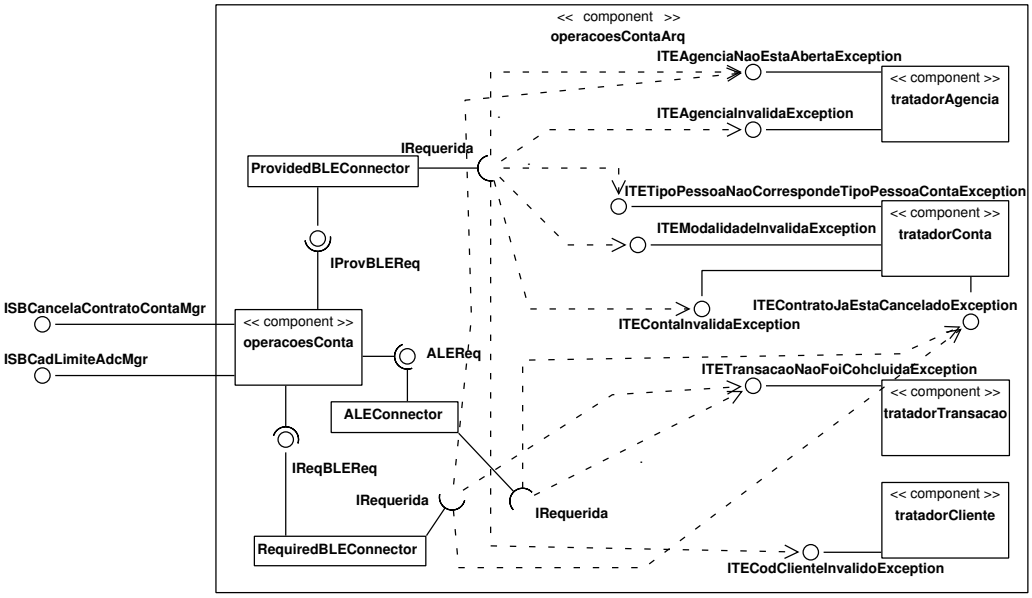

Figura 5.19: Estruturação do Componente Arquitetural operacoesContaArq

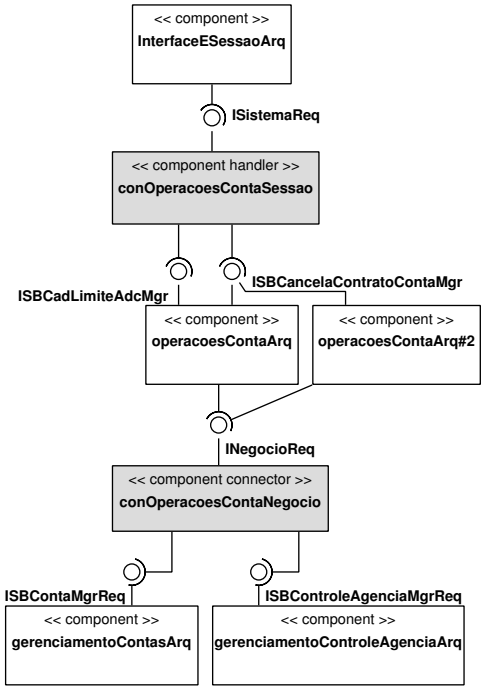

Figura 5.20: Parte da Arquitetura do Sistema

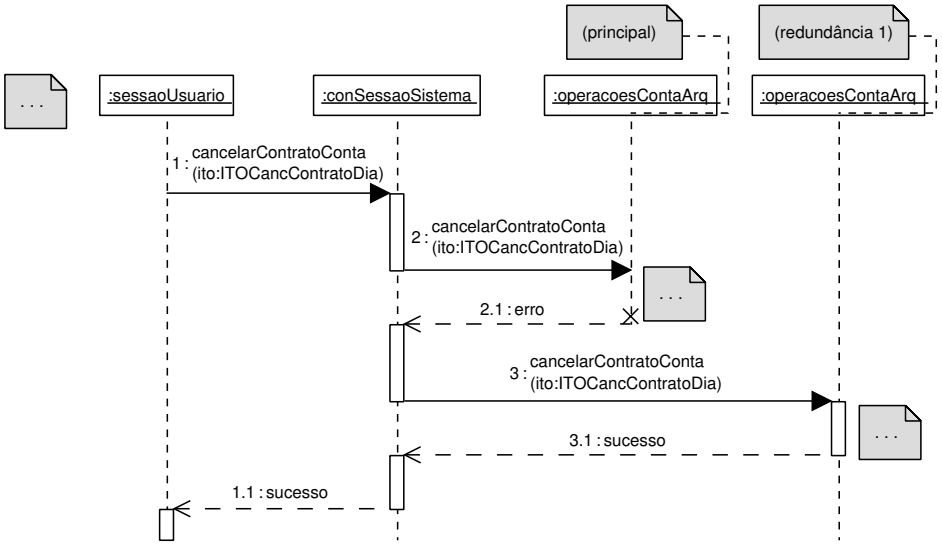

Figura 5.21: Simulação da Reconfiguração do Serviço cancelarContrato(...)

5.5 Análise dos Resultados

O tempo gasto na fase de especificação dos casos de uso referentes às seis funcionalidades foi de 64 pessoa/hora. E como até a data da execução do estudo não tínhamos disponível uma ferramenta de geração automática de código voltada ao COSMOS, o tempo gasto na fase de provisionamento dos componentes também foi de 64 pessoas-hora. A fase de provisionamento consistiu da implementação dos componentes da camada de sistema e da construção dos *wrappers* para os componentes reutilizados. Sendo assim, o tempo útil total gasto na execução do estudo de caso foi de 128 pessoa/hora, ou aproximadamente 0,75 pessoas-mês, distribuídas conforme mostrado na Figura 5.22.

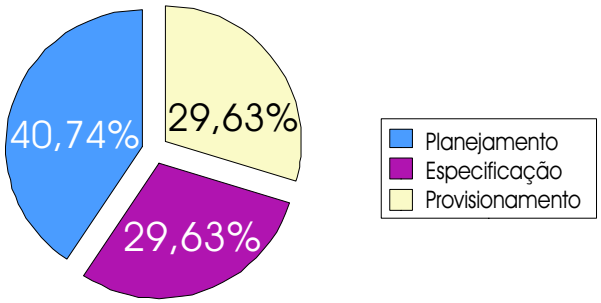

Figura 5.22: Distribuição do Custo Total Durante as Fases do Desenvolvimento

Devido à falta de conhecimento do método MDCE+, antes de cada fase do desenvolvimento, foi necessário uma seção de treinamento com os executores do estudo de caso. Este treinamento consumiu aproximadamente 10% do tempo gasto na fase de especificação de requisitos e 7% do tempo gasto na fase de especificação. Em um segundo estudo de caso com a mesma equipe, esse tempo poderia ser reduzido ou até mesmo desnecessário.

A Figura 5.23 mostra a proporção de cada etapa da fase de especificação dos componentes, em relação ao tempo de duração. Vale salientar que essa fase utilizou aproximadamente 29,63% do tempo total do estudo de caso (Figura 5.22).

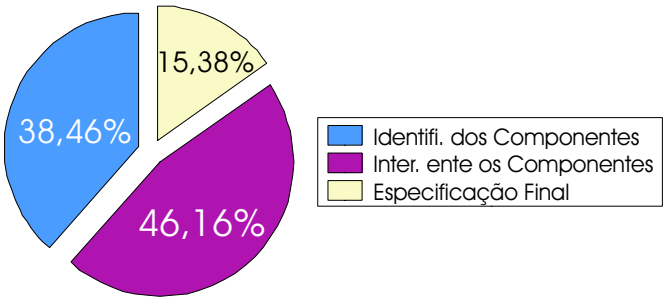

Figura 5.23: Distribuição do Custo da Fase nos Estágios da Especificação

Após a execução do estudo de caso, o processo adaptado foi avaliado de acordo com alguns critérios de qualidade. Essa análise foi feita a partir da coleta da opinião dos dois executores através de questionários anônimos. Esses questionários, que estão disponíveis no Apêndice A, apresentam uma maioria de questões objetivas e algumas sugestões e comentários abertos. Inicialmente, foi feita uma avaliação de cada fase do desenvolvimento separadamente, utilizando o questionário apresentado na Seção A.1 do Apêndice A; em seguida, o processo adaptado foi avaliado de uma maneira geral, utilizando o questionário mostrado na Seção A.2 do mesmo apêndice. As respostas das questões mostram as opiniões em uma escala de 1 (pior) a 5 (melhor). O resultado final dessa avaliação pode ser visto nas Tabelas 5.7 e 5.8.

As Tabelas 5.7 e 5.8 apresentam a média entre as respostas dos dois executores do estudo de caso. Analisando essas Tabelas, os principais benefícios da adoção do método MDCE+ foram os seguintes:

1. o método foi considerado satisfatoriamente fácil de usar. Apesar dele ser considerado intuitivo, a fase de especificação dos componentes apresentou um grande número de atividades, o que aumentou a sua complexidade e reduziu a média do processo.

2. A qualidade das exceções foi considerada melhor. As exceções descobertas utilizando o processo adaptado apresentou uma alta qualidade. Essa classificação se baseou na relação entre as exceções e a lógica do negócio, o que proporciona uma maior independência entre o sistema e a linguagem de programação adotada.

3. Refinamento do tipo do erro. O aumento da qualidade das exceções, aliado ao maior número de exceções detectadas (aproximadamente 20%), aumentou o detalhamento dos erros e segundo a opinião dos especialistas, deve facilitar a detecção e correção dos *bugs* nas atividades de manutenção.

4. Uma boa relação custo X benefício. Apesar do tempo de aprendizagem ser considerado alto, considerou-se que os benefícios apresentados anteriormente foram satisfatórios, principalmente no contexto de sistemas críticos.

5.5.1 Avaliação do Produto

Além da análise feita ao processo adaptado, que será apresentada na Seção 5.5.2, no decorrer do estudo de caso, também foi avaliada a qualidade do produto final. Esta análise se baseou principalmente nos quatro critérios mostrados a seguir:

1. Opiniões de especialistas em relação ao custo adicional e aos benefícios para a manutenibilidade do sistema;

Tabela 5.7: Análise das Fases da Especificação (valores de 1 a 5)

Critério Avaliado	Especificação de Requisitos	Especificação do Componente
- Entendimento	Fácil (4)	Médio/Fácil (3,7)
- Seqüência lógica	Intuitivo (4)	Intuitivo (4)
- Possibilidade de mudança de requisitos durante a especificação	Quase sempre (3,5)	Quase sempre (3,7)
- Qualidade das exceções	Diretamente relacionada ao negócio (4)	Diretamente relacionada ao negócio (4)
- Número de exceções descobertas	Alto (4,5)	Alto (4,7)
- Documentação do sistema	Bom (4)	Bom (4)
- Nível de conhecimento necessário	Principalmente do negócio (4)	Principalmente do negócio (4)

Tabela 5.8: Análise Geral do Processo Adaptado (valores de 1 a 5)

Critério Avaliado	Avaliação Geral
- Entendimento	Médio/Fácil (3,8)
- Seqüência lógica	Intuitivo (4)
- Possibilidade de mudança de requisitos durante a especificação	Quase sempre (3,6)
- Qualidade das exceções	Diretamente relacionada ao negócio (4)
- Número de exceções descobertas	Alto (4,6)
- Documentação do sistema	Bom (4)
- Nível de conhecimento necessário	Principalmente do negócio (4)
- Número de documentos produzidos (artefatos obrigatórios e opcionais)	Médio (3)
- Número de documentos produzidos (apenas artefatos obrigatórios)	Médio/Baixo (3,5)
- *Overhead* de tempo	Médio/Pouco (3,5)
- Custo x benefício (sistemas não críticos)	Um pouco caro / Barato (3,5)
- Custo x benefício (sistemas críticos)	Barato, vale à pena prevenir (4)

2. A qualidade das exceções, isto é, quão próximo do domínio elas estão, e conseqüentemente, quão independente da linguagem de programação adotada;

3. O número de exceções relevantes (que representam erros graves) que foram identificadas;

4. Especificação das atividades de tratamento e mascaramento de falhas (possibilidade de recuperar de estados errôneos ou de serviços indisponíveis).

Apesar de utilizar mais de 20% do tempo com atividades de especificação do sistema, o *overhead* total do desenvolvimento foi cosiderado baixo. De acordo com os executores do estudo de caso, isso se deve à melhoria da documentação do sistema referente aos comportamentos normal e excepcional. Esse detalhamento na documentação agilizou a implementação do sistema, tendo em vista a menor necessidade de interação entre as equipes de especificação e implementação.

Com a utilização da abordagem sistemática do processo adaptado, foram descobertas aproximadamente 20% a mais de exceções. Esse resultado foi obtido ao comparar com outro sistema da empresa, desenvolvido utilizando o processo *UML Components*, a partir da

mesma especificação de requisitos. Esse acréscimo no número de exceções representa um maior refinamento dos tipos de falhas. Aliado ao crescimento da qualidade das exceções e à maior qualidade das informações contextuais, esse acréscimo deve facilitar as atividades de manutenção relacionadas à identificação e correção de *bugs*. Apesar de esperado, esse benefício não foi quantificado adequadamente. Para isso, deve-se fazer experimentos maiores com abrangência em todo o ciclo de desenvolvimento, incluindo atividades de manutenções corretivas e perfectivas.

A melhoria da qualidade das exceções foi decorrente da separação das exceções de acordo com a proximidade da lógica do negócio. Conseqüentemente, com a utilização de uma hierarquia de classificação de exceções própria, a maioria das exceções são independentes da linguagem de programação adotada. Essa melhoria facilita, por exemplo, a mudança de plataforma de desenvolvimento ou até mesmo o desenvolvimento em linhas de produção de software[5]. A preocupação constante com a contextualização das informações também contribuiu para a melhoria da qualidade do tratamento, uma vez que foi possível disponibilizar informações mais detalhadas para a elaboração de relatórios de falhas.

A facilidade para a especificação de exceções arquiteturais foi outro benefício percebido. Diferentemente das exceções de outros sistemas equivalentes, que apenas sinaliza a impossibilidade de execução, o sistema desenvolvido no estudo de caso apresenta um tratamento diferenciado para os serviços considerados críticos.

Devido ao aspecto não crítico de alguns casos de uso, como por exemplo Requisitar Talão de Cheques, o estudo de caso também foi importante para avaliar como o método poderia se adequar ao desenvolvimento de sistemas de software sem requisitos críticos de confiança no funcionamento. Essa adequação aconteceu principalmente na sub-fase de interação entre os componentes, durante o refinamento do modelo de falhas. Foram especificadas exceções genéricas, que foram utilizadas para substituir as mais específicas, satisfazendo as decisões de projeto.

5.5.2 Avaliação do Processo Adaptado

Com a utilização do processo em apenas um estudo de caso, só foi possível fazer uma análise preliminar, de acordo com os oito critérios de qualidade de processos sugeridos por Sommerville [Som01]: (i) facilidade de entendimento; (ii) visibilidade; (iii) suporte ferramental; (iv) nível de aceitação; (v) confiabilidade[6]; (vi) robustez; (vii) manutenibilidade; e (viii) agilidade. A descrição de cada um dos critérios pode ser vista na Tabela 5.9.

A análise desses critérios foi feita em conjunto, tanto pelo autor, quanto pelos executores do estudo de caso. A seguir, são apresentados os resultados dessa análise, onde cada

[5]do inglês *software product lines*
[6]do inglês *reliability*

Tabela 5.9: Critérios de Avaliação do Método MDCE+

Critério Avaliado	Questões a serem Respondidas
Facilidade de Entendimento	O objetivo que o processo se destina é claro? O que se deve fazer em cada atividade do processo é facilmente compreendido?
Visibilidade	O objetivo de cada atividade do processo e os artefatos produzidos são definidos claramente?
Suporte Ferramental	Existe suporte de ferramentas CASE para auxiliar na construção dos artefatos propostos pelo processo?
Nível de Aceitação	Qual o nível de aceitação do processo pela equipe de desenvolvimento? Houve resistência para a utilização?
Confiabilidade	O processo é flexível ao ponto de que erros percebidos do processo possam ser ajustados antes de ocasionarem problemas maiores ao produto?
Robustez	O processo prevê a existência de problemas inesperados e possui atividades gerais para tratá-las?
Manutenibilidade	Qual a facilidade para que o processo evolua no intuito de abranger mudanças organizacionais na empresa, ou até mesmo para a melhoria do processo?
Agilidade	Quão rápido o processo é capaz de proceder o desenvolvimento do produto final?

um dos critérios analisados foi classificado numa escala de **baixo** ***(≥0 e ≤1,75)***, **médio** ***(>1,75 e ≤3,5)***, ou **alto** ***(>3,5 e ≤5)***, seguido de uma justificativa:

1. **Facilidade de Entendimento: média.** Mesmo com o grande número de atividades, a definição clara do papel de cada uma delas facilitou o entendimento.

2. **Visibilidade: alta.** Os resultados produzidos em todas as fases são definidos claramente através dos documentos produzidos.

3. **Suporte Ferramental: médio.** Todos os modelos necessários podem ser construídos em ferramentas CASE atuais, porém, mais facilidades poderiam ser adicionadas a essas ferramentas, como por exemplo a geração automática de código para os componentes COSMOS. Atualmente, está sendo desenvolvido o ambiente Bellatrix [TR05], que irá abranger todas as fases do método, dando suporte tanto à produção de modelos, quanto à orientação das atividades do método e à geração automática de código-fonte.

4. **Nível de Aceitação: alto.** A identificação de exceções foi beneficiada pelo nível de conhecimento elevado da lógica do negócio, por parte de uma das executoras. Como conseqüência dessa familiaridade, a execução das atividades do processo se tornou mais intuitiva, o que facilitou a sua aceitação.

5. **Confiabilidade: alta.** O progresso das fases do método é documentado pelo desenvolvimento de artefatos de entrada/saída. O fato desses artefatos serem usados

em várias fases facilita a antecipação da identificação de erros, reduzindo os custos das correções.

6. **Robustez: média.** Apesar de não existirem muitas diretrizes para gerenciamento do projeto, o método MDCE+ apresenta uma atividade pra analisar o impacto de mudanças repentinas de requisitos. Porém, essas atividades não apresentam o detalhamento necessário para resolver outros problemas inesperados que também possam comprometer a viabilidade do sistema. Vale à pena salientar que esse critério se refere à robustez do processo em relação à ocorrência de fatores imprevistos. Dessa forma, a robustez do produto final não é afetada.

7. **Manutenibilidade: média.** Com a separação clara das atividades de acordo com os respectivos papéis desempenhados, a inclusão de novas fases ou a modificação de fases anteriores fica facilitada. A remoção, porém, não é tão simples, uma vez que isso poderia provocar "efeitos colaterais" decorrentes da dependência que uma atividade tem dos artefatos produzidos pelas suas antecessoras.

8. **Agilidade: média.** A prioridade do método é a confiabilidade do produto final e o detalhamento da documentação para auxiliar os testes. Por isso, sem o auxílio de ferramentas CASE apropriadas, os documentos produzidos reduzem a produtividade do desenvolvimento. Porém, apesar da baixa produtividade para a construção de cada um dos artefatos, quando comparado a outros processos de desenvolvimento, como por exemplo o *UML Components*, o número de artefatos produzidos não é considerado elevado.

5.6 Resumo

Este capítulo apresentou o procedimento de avaliação do método MDCE+. Esse procedimento consistiu da execução de um estudo de caso com a finalidade de verificar a eficácia do método no desenvolvimento de sistemas com requisitos críticos de confiança no funcionamento. A preferência por um sistema com esses requisitos visou destacar o papel da arquitetura para o tratamento efetivo das situações excepcionais. A execução do estudo de caso consistiu da utilização do processo adaptado apresentado no Capítulo 4, para a modelagem e implementação de um sistema financeiro. De uma maneira geral, a expectativa em relação ao método MDCE+ foi atendida. Porém, devido à ausência de ferramentas CASE que orientem a esse processo específico, preferiu-se construir os artefatos de forma manual.

Outra restrição percebida para o método MDCE+ foi a ausência de atividades para tratamento de situações inesperadas durante o desenvolvimento, tais como o controle dos

gastos do projeto e a verificação dos prazos.

Alguns artefatos produzidos durante a execução do estudo de caso são apresentados na Seção 5.4. Em seguida, a Seção 5.5 apresenta alguns resultados frutos das análises do produto final e do processo em si.

Uma informação importante é o fato desse estudo de caso também ter sido utilizado para realizar a integração entre o método MDCE+ a um método de testes para o comportamento excepcional de componentes [dSBRCF+05]. Uma versão mais detalhada com todos os artefatos produzidos no estudo de caso estão disponíveis em outro lugar [dSBRMR05].

Capítulo 6

Conclusões

Este trabalho apresentou o MDCE+, um método para auxiliar a modelagem do comportamento excepcional de sistemas baseados em componentes. O método MDCE+ melhora a confiança no funcionamento do sistema, oferecendo uma maneira sistemática para modelar as exceções e os seus respectivos tratadores, distribuindo essas atividades em todas as fases do desenvolvimento do software. Essa maneira estruturada de detectar e tratar exceções no contexto da ocorrência de falhas é particularmente relevante para sistemas que apresentam requisitos de confiabilidade extrema.

Em relação às quatro dimensões da confiabilidade, vistas na Seção 2.3.2, o método MDCE+ atua diretamente em três delas: (i) prevenção de falhas, por se tratar de um método sistemático que estrutura o raciocínio e evita a inserção de erros; (ii) tolerância a falhas, através da estruturação do comportamento excepcional do sistema; e (iii) avaliação de falhas, através da revisão do modelo de falhas e da análise do fluxo excepcional, que pode ser auxiliado por ferramentas CASE [FdSBR05a]. Apesar de não atuar na dimensão da remoção de falhas, essa deficiência foi suprida com a adaptação feita a um método de testes de componentes críticos [dSBRCF+05].

Duas características importantes do método MDCE+, que reforçam o seu aspecto robusto, são: (i) o fato dele combinar as abordagens descendente e ascendente para o desenvolvimento de sistemas confiáveis; e (ii) o fato dele ser centrado na arquitetura. O foco na arquitetura de software contribui para uma melhor definição e análise do fluxo de exceções entre os componentes do sistema. Com essa análise estrutural, o método auxilia inclusive no desenvolvimento de sistemas concorrentes que possuem componentes executados de forma coordenada. Isso é feito através do conceito de colaborações [XRR+99, dLR99].

Além de aumentar a qualidade do tratamento e possibilitar a coordenação de componentes concorrentes, analisando o fluxo da arquitetura é possível detectar falhas de projeto antecipadamente. Ainda durante a análise das propagações na arquitetura de software, o método MDCE+ apresenta diretrizes úteis para a implementação de sistemas com re-

quisitos críticos de confiabilidade e disponibilidade, através da utilização sistemática de componentes redundantes (tolerância a falhas).

Em relação aos aspectos relativos à manutenibilidade, o método MDCE+ oferece uma separação clara entre os conceitos de componente arquitetural e componente de implementação. Isso possibilita uma maior abstração de modelagem, o que facilita a compreensão, o desenvolvimento e a manutenção de sistemas complexos [Som01, Pre01]. Os componentes arquiteturais são estruturados segundo o modelo do componente tolerante a falhas ideal, apresentado na Seção 2.3.7. Dessa forma, existe uma separação explícita entre os comportamentos normal e excepcional de cada componente. Com isso, os sistemas desenvolvidos são compreendidos mais facilmente e conseqüentemente, sua manutenção também fica facilitada. Além disso, com o controle da complexidade, espera-se que seja inserido um menor número de falhas durante o seu desenvolvimento (prevenção de falhas).

Para complementar o método MDCE+ em relação às atividades de especificação do comportamento normal do sistema, esse método foi adaptado ao processo *UML Components*. Essa adaptação foi apresentada no Capítulo 4. Porém, apesar da importância dos testes para a melhoria da qualidade do software (remoção de falhas), tanto o processo *UML Components* quanto o método MDCE+ se restringem ao aspecto puramente de construção. Dessa forma, eles não apresentam atividades sistemáticas relativas a testes, que tratem essa questão desde a especificação dos requisitos.

Motivado por essa deficiência, o método MDCE+ foi adaptado a um método de testes para o comportamento excepcional de componentes [dSBRCF+05, dSBRMR05]. A principal característica do método unificado de desenvolvimento e testes é o fato das atividades de desenvolvimento e testes serem executadas em paralelo. Esse paralelismo reduz o *overhead* da adoção do método, o que favorece a sua utilização para o desenvolvimento de sistemas reais.

Durante a adaptação entre o MDCE+ e o método de testes, foi desenvolvido um estudo de caso. Esse estudo de caso, que foi apresentado no Capítulo 5, também foi utilizado para aferir as qualidades do método resultante em relação aos aspectos de confiabilidade. Dessa forma, foi feita uma avaliação em dois pontos de vista relacionados, porém distintos: (i) a qualidade do produto construído (Seção 5.5.1); e (ii) a qualidade do método em si (Seção 5.5.2).

A seguir, a Seção 6.1 apresenta as principais contribuições desse trabalho. Finalmente, a Seção 6.2 aponta alguns direcionamentos futuros para pesquisa.

6.1 Contribuições

As contribuições deste trabalho se situam principalmente no campo da engenharia de software, em particular nas áreas de processos de desenvolvimento, tratamento de exceções,

desenvolvimento baseado em componentes, arquitetura de software e desenvolvimento de sistemas tolerantes a falhas. A seguir, são listadas as principais contribuições deste trabalho, seguidas de uma breve descrição:

1. **O método MDCE+.** A principal contribuição desse trabalho foi a especificação do método MDCE+, apresentado no Capítulo 3. Durante a condução da pesquisa, foram percebidas algumas outras contribuições secundárias, relacionadas ao método MDCE+:

   (a) **Estudo dos principais mecanismos e técnicas para tratamento de exceções.** Durante a definição das atividades do método MDCE+, foi necessário estudar alguns dos principais mecanismos de tratamento de exceções das linguagens de programação, tais como Java, C++ e C#. De forma complementar, foram estudadas algumas técnicas para otimizar as suas utilizações. O produto desse estudo está disponível na versão detalhada do método MDCE+ [dSBR05], na forma de diretrizes para a estruturação de exceções nas linguagens de programação.

   (b) **Utilização de uma hierarquia de exceções específica.** Com a utilização da hierarquia de exceções mostrada na Seção 3.7.1, a estruturação do comportamento excepcional ficou facilitada. Isso se deve principalmente á classificação dos erros de acordo com a sua natureza. Além disso, essa hierarquia proporciona uma representação uniforme para as exceções simples, que são lançadas por um componente, e compostas, que são resultantes do lançamento concorrente de mais de uma exceção.

   (c) **Adoção do modelo COSMOS [JGR03].** O modelo COSMOS, mostrado na Seção 2.2.2, é um modelo de componentes que possibilita um melhor mapeamento entre a arquitetura e o código. Sendo assim, a sua adoção possibilitou uma maior clareza no código implementado e uma conseqüente melhoria na manutenção.

   (d) **Adoção de uma arquitetura voltada para sistemas confiáveis.** Com a adoção da arquitetura apresentada na Seção 2.5, foi possível explicitar a estruturação dos papéis necessários para que o tratamento de exceções possa ser feito de forma eficiente.

   (e) **O *framework* Aereal [FdSBR05a].** A criação de um *framework* para análise do fluxo de exceções na arquitetura possibilitou a automação necessária para verificar a existência de falhas de projeto durante a especificação do sistema.

2. **Adaptação do método MDCE+ ao processo *UML Components*.** O processo resultante dessa adaptação é um processo de desenvolvimento instanciável, isto é, que abrange a modelagem e a implementação dos comportamentos normal e excepcional de um sistema.

   (a) **Especificação detalhada das fases do *UML Components*.** Antes da adaptação ao processo *UML Components*, esse processo foi representado através de diagramas de atividades UML. O principal objetivo dessa representação foi proporcionar uma forma rápida e eficiente de consulta e acompanhamento das suas atividades.

3. **Integração com um método de testes.** Apesar de tratar todas as fases do desenvolvimento do software, o processo resultante da adaptação do método MDCE+ ao processo *UML Components* apresenta deficiências em relação aos aspectos de testes. Devido à importância dos testes para o desenvolvimento de sistemas confiáveis, foram adicionadas atividades de testes ao processo *UML Components* modificado [dSBRCF$^+$05, dSBRMR05]. Apesar de ser um processo integrado, as atividades de teste pertencem ao contexto de um outro projeto de pesquisa [Roc03].

   (a) **Desenvolvimento de um estudo de caso.** Para facilitar esse processo de adaptação, optou-se por fazê-lo durante a execução de um estudo de caso real. Mais detalhes a respeito do estudo de caso são mostrados no Capítulo 5. Os detalhes relativos ao desenvolvimento e testes em paralelo estão disponíveis em outros documentos [dSBRCF$^+$05, dSBRMR05].

4. **Um processo de desenvolvimento com ênfase na maximização do reuso.** Tendo em vista a exigência do mercado de software em reduzir o tempo de desenvolvimento, foi especificado um processo de desenvolvimento simplificação que visa maximizar o número de componentes reutilizados [dSBBdCGR05]. Esse processo foi desenvolvido no contexto do projeto de pesquisa MCT/FINEP/Ação Transversal - Biblioteca de Componentes - 05/2004.

6.2 Trabalhos Futuros

Com a conclusão desse trabalho, foram identificados alguns direcionamentos para pesquisas futuras. Esses direcionamentos visam principalmente o aperfeiçoamento do método MDCE+, no sentido de suprir algumas de suas limitações.

Os trabalhos futuros mais imediatos estão relacionados com a evolução do processo segundo os critérios sugeridos por Sommerville [Som01]. Dessa forma, baseado na avaliação do método apresentada na Seção 5.5.2, deve-se pesquisar meios de aperfeiçoar os

aspectos qualificados como baixo ou **médio**. Em especial, deve-se dar uma maior atenção aos aspectos relativos a: (i) suporte ferramental; (ii) robustez; e (iii) agilidade; A seguir, são apontadas algumas sugestões de melhoria:

1. **Suporte ferramental.** Um trabalho futuro mais imediato é a construção de uma ferramenta CASE, que seja orientada ao método MDCE+. Essa ferramenta deve auxiliar na construção dos diversos artefatos produzidos pelo método. Uma proposta de especificação dessa ferramenta está sendo feita no contexto de um outro projeto de pesquisa [TFdCGR04, TR05].

2. **Robustez.** Uma abordagem possível para aumentar a robustez do processo é adicionar mais atividades de gerenciamento e controle de riscos. Dessa forma, na ocorrência de alguns determinados eventos, a modelagem poderia ser suspensa e alguma medida preventiva/paliativa poderia ser indicada [Som01].

3. **Agilidade.** Uma possibilidade para agilizar o desenvolvimento dos sistemas seria estudar alguns processos ágeis e tentar introduzir algumas de suas características no MDCE+. Porém, deve-se ter sempre em mente que não se pode comprometer a confiabilidade do produto final produzido, uma vez que esse é o objetivo principal do método MDCE+.

Todas essas melhorias no processo poderiam ser feitas gradativamente, através da aplicação de técnicas de aperfeiçoamento baseada em estudos de caso [PC04]. Dessa forma, um outro trabalho futuro certamente é a aplicação do método em outros estudos de caso, o que também seria útil para avaliar o método de uma maneira mais criteriosa, embasada em resultados estatísticos.

Além dessas restrições identificadas para o processo em si, do ponto de vista dos aspectos de tolerância a falhas, a abordagem de implementação de colaborações apresenta uma deficiência grave. Devido ao controle centralizado que é exercido pelo coordenador da colaboração, isso acaba acarretando em um ponto único de falha. Apesar dessa deficiência poder ser amenizada com a utilização de um repositório[1] de coordenadores equivalentes, um outro trabalho futuro poderia ser estudar uma nova estruturação para as colaborações.

Um outro direcionamento de trabalho futuro poderia ser o estudo de como as técnicas de reflexão computacional [Mae87, OB98] e programação orientada a aspectos [K+97] poderiam ser utilizados para reduzir o entrelaçamento de código decorrente da adição dos blocos de proteção, que em Java correspondem aos blocos `try{...} catch(){...}` .

O projeto Bellatrix [TFdCGR04] é um projeto em andamento no Instituto de Computação da Unicamp, que se propõe a desenvolver um ambiente constituído de uma série

[1] do inglês *pool*

de *plugins* para a plataforma Eclipse [Pro]. Esse ambiente é voltado para o desenvolvimento baseado em componentes e guia o desenvolvedor seguindo as atividades propostas pelo processo *UML Components*. Dessa forma, com o intuito de tornar o ambiente Bellatrix orientado ao método MDCE+, um outro trabalho futuro poderia ser a construção de um *wizard* para guiar o desenvolvedor na detecção das exceções e na especificação dos seus tratadores, através de um *workflow* de atividades e diretrizes propostas pelo método MDCE+.

6.3 Publicações

Durante a condução desse trabalho, foram produzidas quatro publicações, sendo três em congressos internacionais [dSBRCF+05, FdSBR05b, FdSBR05a] e uma em congresso nacional [dSBFR04]. A seguir, é mostrado um breve resumo de cada uma delas.

1. **A Method for Modeling and Testing Exceptions in Component-Based Software Development, LADC'05 - II Latin-American Symposium on Dependable Computing, 2005 [dSBRCF+05].** *(Patrick H. da S. Brito, Camila R. Rocha, Fernando Castor Filho, Eliane Martins e Cecília M. F. Rubira)*: Este trabalho apresenta uma versão quase final do método MDCE+. Além desse refinamento em relação ao artigo publicado no WDBC [dSBFR04], um outro diferencial desse artigo é a integração desse método a um processo de testes para o comportamento excepcional de componentes. A integração entre o método MDCE+ e o processo de testes aconteceu a partir da execução de um estudo de caso, que é apresentado em um dos relatórios técnicos a seguir [dSBRMR05].

2. **Modeling and Analysis of Architectural Exceptions, REFTS'05 - Workshop on Rigorous Engineering of Fault Tolerant Systems, 2005 [FdSBR05b].** *(Fernando Castor Filho, Patrick H. da S. Brito e Cecília M. F. Rubira)*: Este trabalho é complementar ao apresentado no WADS'05 [FdSBR05a]. O foco desse artigo foi a definição de um modelo para a definição de mecanismos de tratamento de exceções (MTE). O modelo apresentado é flexível o bastante para definir os MTEs de várias linguagens de programação existentes, tais como Java, C# e Ada. Além disso, no contexto do *framework* Aereal, este artigo mostra que, ao invés de estender ou adotar algum MTE em particular, ele define seu próprio mecanismo de tratamento de exceções e possibilita aos desenvolvedores estendê-lo ou definir um outro de acordo com as suas necessidades.

3. **A Framework for Analyzing Exception Flow in Software Architectures, WADS'05 - ICSE 2005 Workshop on Architecting Dependable Systems,**

2005 [FdSBR05a]. *(Fernando Castor Filho, Patrick H. da S. Brito e Cecília M. F. Rubira)*: Este trabalho apresenta o *framework* Aereal. Esse *framework* é voltado para a análise do fluxo de exceções entre os componentes arquiteturais. Ele utiliza como entrada a definição dos fluxos de informações normais e excepcionais, que são feitos na linguagem de descrição de arquitetura (ADL[2]) ACME [GMW00]. A partir desses fluxos separados, o *framework* se encarrega de combiná-los e de converter a especificação resultante para a ADL Alloy [Jac02], o que possibilita a verificação automática das propriedades utilizando a ferramenta Alloy Analyzer [JSS00].

4. **Um Método para Modelagem de Exceções em Desenvolvimento Baseado em Componentes, WDBC'04 - IV Workshop de Desenvolvimento Baseado em Componentes, 2004 [dSBFR04].** *(Patrick H. da S. Brito, Fernando Castor Filho e Cecília M. F. Rubira)*: Este trabalho apresentou a primeira versão do método MDCE+. Apesar de na seqüência dos trabalhos ele ter evoluído, a maioria dos conceitos apresentados nesse artigo permaneceram constantes até a escrita dessa dissertação.

[2]do inglês *Architecture Description Language*

Apêndice A

Questionários de Avaliação do Estudo de Caso

A.1 Questionário das Fases

1. **Qual a sua opinião sobre a dificuldade de entendimento desta fase (complexidade)?**

   ( ) Impraticável ( ) Muito Difícil ( ) Difícil ( ) Fácil ( ) Muito Fácil

2. **Qual a sua opinião a respeito da seqüência lógica das atividades (compreensão)?**

   ( ) Coincidente ( ) Maioria coincidente ( ) Minoria coincidente
   ( ) Pouco intuitiva ( ) Intuitiva

3. **Quando você acha que o desenvolvimento iterativo possibilitou "voltar atrás" durante o desenvolvimento?**

   ( ) Nunca ( ) Raramente ( ) Satisfatório ( ) Quase sempre
   ( ) Sempre que precisei

4. **Qual a sua opinião sobre a qualidade das exceções descobertas (independência de tecnologia)?**

   ( ) Todas são inerentes à linguagem ( ) A maioria inirente à linguagem
   ( ) Balanceado ( ) A maioria é relativa à lógica do negócio
   ( ) Todas são relativas à lógica do negócio

5. **Qual a sua opinião sobre a facilidade proporcionada pela estruturação do raciocínio (se ajuda a lembrar de alguma exceção que normalmente seria esquecida)?**

   ( ) Não facilitou em nada ( ) Facilitou muito pouco
   ( ) Facilitou na minoria das exceções ( ) Facilitou na maioria das exceções
   ( ) Facilitou em todas as exceções

6. **Qual a sua opinião sobre a melhoria da qualidade da documentação do sistema, incluindo o comportamento excepcional?**

   ( ) Piorou bastante ( ) Piorou um pouco ( ) A mesma
   ( ) Melhorou um pouco ( ) Melhorou bastante

7. **Qual a sua opinião a respeito do conhecimento tecnológico exigido para a utilização do método (obrigatoriedade de se conhecer uma linguagem de programação)?**

   ( ) Essencial, sem conhecer a linguagem é inviável
   ( ) Muito determinante, sem esse conhecimento fica prejudicado
   ( ) Determinante, ajuda bastante mas pode-se ficar sem
   ( ) Pouco determinante, se conhecer é melhor mas não influencia diretamente
   ( ) Irrelevante, sem importância alguma

8. **Qual a sua opinião a respeito do número de documentos produzidos pelo método ("burocracia")?**

   (a) **Levando-se em conta os artefatos facultativos e obrigatórios:**
   ( ) Impraticável ( ) Muito grande ( ) Médio ( ) Pouco ( ) Muito pouco

   (b) **Levando-se em conta apenas os artefatos obrigatórios:**
   ( ) Impraticável ( ) Muito grande ( ) Médio ( ) Pouco ( ) Muito pouco

9. **Qual a sua opinião a respeito "overhead" de tempo decorrente da utilização do método MDCE+?**

   ( ) Muito alto ( ) Alto ( ) Médio ( ) Pouco ( ) Muito baixo

10. **Qual a sua opinião a respeito da relação custo-benefício?**

    (a) **Para sistemas SEM requisitos críticos de confiabilidade:**
    ( ) Benefício irrisório "comparado com o custo" ( ) Muito caro
    ( ) Difícil ( ) Fácil ( ) Custo irrisório "comparado com o benefício"

    (b) **Para sistemas COM requisitos críticos de confiabilidade:**
    ( ) Benefício irrisório "comparado com o custo" ( ) Muito caro
    ( ) Difícil ( ) Fácil ( ) Custo irrisório "comparado com o benefício"

11. **Comentários / Sugestões:**
    "Questão aberta."

A.2 Questionário Geral

1. **Qual a sua opinião sobre a dificuldade de entendimento desta fase (complexidade)?**
   ( ) Impraticável ( ) Muito Difícil ( ) Difícil ( ) Fácil ( ) Muito Fácil

2. **Qual a sua opinião a respeito da seqüência lógica das atividades (compreensão)?**
   ( ) Coincidente ( ) Maioria coincidente ( ) Minoria coincidente
   ( ) Pouco intuitiva ( ) Intuitiva

3. **Quando você acha que o desenvolvimento iterativo possibilitou "voltar atrás" durante o desenvolvimento?**
   ( ) Nunca ( ) Raramente ( ) Satisfatório ( ) Quase sempre
   ( ) Sempre que precisei

4. **Qual a sua opinião sobre a qualidade das exceções descobertas (independência de tecnologia)?**
   ( ) Todas são inerentes à linguagem ( ) A maioria inirente à linguagem
   ( ) Balanceado ( ) A maioria é relativa à lógica do negócio
   ( ) Todas são relativas à lógica do negócio

5. **Qual a sua opinião sobre a facilidade proporcionada pela estruturação do raciocínio (se ajuda a lembrar de alguma exceção que normalmente seria esquecida)?**

   ( ) Não facilitou em nada ( ) Facilitou muito pouco
   ( ) Facilitou na minoria das exceções ( ) Facilitou na maioria das exceções
   ( ) Facilitou em todas as exceções

6. **Qual a sua opinião sobre a melhoria da qualidade da documentação do sistema, incluindo o comportamento excepcional?**

   ( ) Piorou bastante ( ) Piorou um pouco ( ) A mesma
   ( ) Melhorou um pouco ( ) Melhorou bastante

7. **Qual a sua opinião a respeito do conhecimento tecnológico exigido para a utilização do método (obrigatoriedade de se conhecer uma linguagem de programação)?**

   ( ) Essencial, sem conhecer a linguagem é inviável
   ( ) Muito determinante, sem esse conhecimento fica prejudicado
   ( ) Determinante, ajuda bastante mas pode-se ficar sem
   ( ) Pouco determinante, se conhecer é melhor mas não influencia diretamente
   ( ) Irrelevante, sem importância alguma

8. **Comentários / Sugestões:**

   "Questão aberta."

Referências Bibliográficas

[ABM00] Colin Atkinson, Joachim Bayer, and Dirk Muthig. Component-based product line development: The kobrA approach. In P. Donohoe, editor, *Proceedings of theFirstSoftware Product Line Conference*, pages 289–309, 2000.

[AFC01] Carina Alves, João Bosco Pinto Filho, and Jaelson Castro. Analysing the tradeoffs among requirements, architectures and cots components. In *WER*, pages 20–31, 2001.

[AL90] T. Anderson and P. A. Lee. *Fault Tolerance: Principles and Practice*. Springer-Verlag, 2nd edition, 1990.

[Avi97] Algirdas Avizienis. Towards systematic design of fault-tolerant systems. *IEEE Computer*, 30(4):51–58, April 1997.

[Bar05] Mario R. Barbacci. Sei architecture analysis techniques and when to use them. Technical Report CMU/SEI-2002-TN-005, SEI, Pittsburgh, PA, USA, 2005.

[Bec99] Kent Beck. *Extreme Programming Explained: Embrace Change*. Addison-Wesley Professional, October 1999.

[Bin99] Robert V. Binder. *Testing object-oriented systems: models, patterns, and tools*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.

[BKB00] Len Bass, Mark Klein, and F. Bachmann. Quality attribute design primitives. Technical report CMU/SEI-2000-TR-017, Pittsburgh, PA, USA, 2000.

[Boe81] Barry W. Boehm. *Software Engineering Economics*. Prentice Hall, 1981.

[BRJ99] Grady Booch, James Rumbaugh, and Ivar Jacobson. *The Unified Modeling Language user guide*. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, 1999.

[CD00] John Chessman and John Daniels. *UML Components*. Addison-Wesley, 2000.

[CdSC01] Ariadne M. B. Rizzoni Carvalho and Thelma C. dos Santos Chiossi. *Introdução à Engenharia de Software*. Editora da Unicamp, 1st edition, 2001.

[CK03] Paul Clements and Rick Kazman. *Software Architecture in Practices*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2003.

[Coc03] Alistair Cockburn. *Writting Effective Use Cases*. Addison Wesley, 2003.

[CR86] Roy H. Campbell and Brian Randell. Error recovery in asynchronous systems. *IEEE Trans. Software Eng.*, 12(8):811–826, 1986.

[Cri89] Flaviu Cristian. Exception handling. In T. Anderson, editor, *Dependability of Resilient Computers*, pages 68–97. Blackwell Scientific Publications, 1989.

[CWMY02] Feng Chen, Qianxiang Wang, Hong Mei, and Fuqing Yang. An architecture-based approach for component-oriented development. *26th Annual International Computer Software and Applications Conference*, August 2002.

[dCG04] Paulo Asterio de Castro Guerra. *Uma Abordagem Arquitetural para Tolerância a Falhas em Sistemas de Software Baseados em Componentes*. PhD thesis, IC, Unicamp, June 2004.

[dCGFPR04] Paulo Asterio de C. Guerra, Fernando Castor Filho, Vinicius Asta Pagano, and Cecília M. F. Rubira. Structuring exception handling for dependable component-based software systems. In *EUROMICRO*, pages 575–582, 2004.

[Del98] Chrysanthos Dellarocas. Toward exception handling infrastructures for component-based software. In *Proceedings of the International Workshop on Component-based Software Engineering, 20th// International Conference on Software Engineering (ICSE)*, Kyoto, Japan, April 1998.

[dLR99] Rogério de Lemos and A. Romanovsky. Exception handling in a cooperative object-oriented approach. In *Proc. of the 2nd IEEE ISORC'99*, May 1999.

[dLR00] Rogério de Lemos and A. Romanovsky. Exception handling in the software lifecycle. In *Int. Journal of Computer Science and Engineering (Special Issue on Object-Oriented Real-Time Distributed Systems)*, 2000.

[dSBBdCGR05] Patrick Henrique da Silva Brito, Maria Antonia Martins Barbosa, Paulo Asterio de Castro Guerra, and Cecília Mary Fischer Rubira. Um processo para o desenvolvimento baseado em componentes com reuso de componentes. Relatório Técnico (a publicar), Instituto de Computação, 2005.

[dSBFR04] Patrick Henrique da Silva Brito, Fernando Castor Filho, and Cecília Mary Fischer Rubira. Um método para modelagem de exceções em desenvolvimento baseado em componentes. In *IV WDBC*, 2004.

[dSBR05] Patrick Henrique da Silva Brito and Cecília Mary Fischer Rubira. Um método para modelagem de exceções em desenvolvimento baseado em componentes. Relatório Técnico (a publicar), Instituto de Computação, 2005.

[dSBRCF+05] Patrick Henrique da Silva Brito, Camila Ribeiro Rocha, Fernando Castor-Filho, Eliane Martins, and Cecília Mary Fischer Rubira. A method for modeling and testing exceptions in component-based software development (to appear). *2nd Latin-American Symposium on Dependable Computing*, October 2005.

[dSBRMR05] Patrick Henrique da Silva Brito, Camila Ribeiro Rocha, Eliane Martins, and Cecília Mary Fischer Rubira. Um método integrado para modelagem e testes de exceções no cbd: Um estudo de caso. Relatório Técnico (a publicar), Instituto de Computação, 2005.

[DW99] Desmond D'Souza and Alam Cameron Wills. *Objects, Components, and Frameworks with UML The Catalysis Approach.* Addison-Wesley, 2nd edition, 1999.

[EFLR99] Andy Evans, Robert France, Kevin Lano, and Bernhard Rumpe. The UML as a formal modeling notation. In Jean Bézivin and Pierre-Alain Muller, editors, *The Unified Modeling Language, UML'98 - Beyond the*

Notation. First International Workshop, Mulhouse, France, June 1998, Selected Papers, volume 1618, pages 336–348. Springer, 1999.

[FdCGR03] Fernando Castor Filho, Paulo Asterio de C. Guerra, and Cecília M. F. Rubira. An architectural-level exception-handling system for component-based applications. In *LADC*, pages 321–340, 2003.

[FdSBR05a] Fernando Castor Filho, Patrick Henrique da Silva Brito, and Cecília M. F. Rubira. A framework for analyzing exception flow in software architectures. In *ICSE'2005 Workshop on Architecting Dependable Systems (WADS)*, St. Louis, MO, USA, May 2005.

[FdSBR05b] Fernando Castor Filho, Patrick Henrique da Silva Brito, and Cecília M. F. Rubira. Modeling and analysis of architectural exceptions. In *Workshop on Rigorous Engineering of Fault Tolerant Systems (REFTS'05)*, Newcastle, UK, July 2005.

[Fei96] L. Feijs. Architecture visualisation and analysis: Motivation and example. In *Intl. Workshop on Development and Evolution of Software Architectures for Product Families*, 1996.

[Fer01] Gisele R. M. Ferreira. Tratamento de exceções no desenvolvimento de sistemas confiáveis baseados em componentes. Master's thesis, IC, Unicamp, December 2001.

[FH95] Robert B. France and Thomas B. Horton. Applying domain analysis and modeling: an industrial experience. In *SSR '95: Proceedings of the 1995 Symposium on Software reusability*, pages 206–214, New York, NY, USA, 1995. ACM Press.

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlisside. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, Reading, Massachusetts, 1995.

[Gil03] Howard Gilbert. Exception handling in java and c#. http://www.yale.edu/pclt/exceptions.htm, May 2003. Last access in 2005.

[GMW00] David Garlan, Robert T. Monroe, and David Wile. Acme: Architectural description of component-based systems. In Gary T. Leavens and Murali Sitaraman, editors, *Foundations of Component-Based Systems*, pages 47–68. Cambridge University Press, 2000.

[Goo75] John B. Goodenough. Exceptional handling: Issues and a proposed notation. *CACM*, 18(12), 1975.

[Gär99] Felix C. Gärtner. Fundamentals of fault-tolerant distributed computing in asynchronous environments. *ACM Comput. Surv.*, 31(1):1–26, 1999.

[GR01] Alessandro F. Garcia and Cecília M. F. Rubira. An architectural-based reflective approach to incorporating exception handling into dependable software. pages 189–206, 2001.

[GRdL03] Paulo A. C. Guerra, Alexander Romanovsky, and Rogério de Lemos. A fault-tolerant software architecture for cots-based software systems. *European Software Engineering Conference - Foundations of Software Engineering (ESEC/FSE'03)*, September 2003.

[GRRX01] Alessandro F. Garcia, Cecília M. F. Rubira, Alexander B. Romanovsky, and Jie Xu. A comparative study of exception handling mechanisms for building dependable object-oriented software. *Journal of Systems and Software*, 59(2):197–222, 2001.

[Har87] David Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8(3):231–274, June 1987.

[HHK+99] Thomas B. Hilburn, Iraj Hirmanpour, Soheil Khajenoori, Richard Turner, and Abir Qasem. A software engineering body of knowledge version 1.0. Technical report CMU/SEI-99-TR-004, Pittsburgh, PA, USA, 1999.

[IB01] Valérie Issarny and Jean-Pierre Banâtre. Architecture-based exception handling. *Hawaii International Conference on System Sciences (HICSS'01)*, 2001.

[Jac02] Daniel Jackson. Alloy: a lightweight object modelling notation. *ACM Trans. Softw. Eng. Methodol.*, 11(2):256–290, 2002.

[Jal94] Pankaj Jalote. *Fault tolerance in distributed systems*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1994.

[JBR99] Ivar Jacobson, Grady Booch, and James Rumbaugh. *The unified software development process*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.

[JCJv92] I. Jacobson, M. Christerson, P. Jonsonn, and G. Övergaard. *Object-Oriented Software Engineering: A Use Case Driven Approach*. Addison-Wesley, 1992.

[JGR03] Moacir C. Silva Jr., Paulo A. C. Guerra, and Cecília Rubira. A java model for evolving software systems. *IEEE International Conference on Automated Software Engineering(ASE'03)*, October 2003.

[JM97] Jean-Marc Jézéquel and Bertrand Meyer. Design by contract: The lessons of ariane. *Computer*, 30(1):129–130, 1997.

[JP89] Ingrid Jansch-Pôrto. Fundamentos de tolerância a falhas. *RITA*, 1(3):63–99, 1989.

[JSS00] Daniel Jackson, Ian Schechter, and Hya Shlyahter. Alcoa: the alloy constraint analyzer. In *ICSE '00: Proceedings of the 22nd international conference on Software engineering*, pages 730–733, New York, NY, USA, 2000. ACM Press.

[K+97] Gregor Kiczales et al. Aspect-oriented programming. In *Proc. of ECOOP'97*. LNCS 1241, 1997.

[KCH+90] K. Kang, S. Cohen, J. Hess, W. Nowak, and S. Peterson. Feature-oriented domain analysis (FODA) feasibility study. Technical report CMU/SEI-90-TR-021, Pittsburgh, PA, USA, 1990.

[KJ97a] Barbara Ann Kitchenham and Lindsay Jones. Evaluating software engineering methods and tool. part 5: The influence of human factors. *ACM Sigsoft*, 22(1):13–15, January 1997.

[KJ97b] Ron Kohavi and George H. John. Wrappers for feature subset selection. *Artificial Intelligence*, 97(1-2):273–324, 1997.

[KK99] Mark Klein and Rick Kazman. Attribute-based architectural styles. Technical report CMU/SEI-99-TR-022, Pittsburgh, PA, USA, 1999.

[KKC00] Rick Kazman, Mark Klein, and Paul Clements. Atam: Method for architecture evaluation. Technical report CMU/SEI-2000-TR-004, Pittsburgh, PA, USA, August 2000.

[KP98] Barbara Ann Kitchenham and Lesley M. Pickard. Evaluating software engineering methods and tool. part 10: Designing and running a quantitative case study. *ACM Sigsoft*, 23(3):20–22, May 1998.

[Lap85] J. C. Laprie. Dependable computing and fault tolerance: concepts and terminology. pages 2–11, June 1985.

[LR01] Chris Leer and David S. Rosenblum. Wren—an environment for component-based development. In *ESEC/FSE-9: Proceedings of the 8th European software engineering conference held jointly with 9th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 207–217, New York, NY, USA, 2001. ACM Press.

[LS98] Jun Lang and David B. Stewart. A study of the applicability of existing exception-handling techniques to component-based real-time software technology. *ACM Trans. Program. Lang. Syst.*, 20(2):274–301, 1998.

[Mae87] Pattie Maes. Concepts and experiments in computational reflection. In *OOPSLA '87: Conference proceedings on Object-oriented programming systems, languages and applications*, pages 147–155, New York, NY, USA, 1987. ACM Press.

[McI76] M. Douglas McIlroy. Mass-produced software components. In J. N. Buxton Peter Naur, Brian Randell, editor, *Software Engineering: Concepts and Techniques*, pages 88–94. Petrocelli/Charter, 1976.

[MH99] Richard Monson-Haefel. *Enterprise JavaBeans*. O'Reilly & Associates, Inc., Sebastopol, CA, USA, 1999.

[MKMG97] Robert T. Monroe, Andrew Kompanek, Ralph Melton, and David Garlan. Architectural styles, design patterns, and objects. *IEEE Softw.*, 14(1):43–52, 1997.

[MM00] Raphael Marvie and Philippe Merle. Vers un modèle de composants pour CESURE - le CORBA Component Model. Technical Report 3, Projet RNRT 98 CESURE, Novembre 2000. http://www.gemplus.fr/cesure/.

[Nor04] Robert L. Nord, editor. *Software Product Lines, Third International Conference, SPLC 2004, Boston, MA, USA, August 30-September 2, 2004, Proceedings*, volume 3154 of *Lecture Notes in Computer Science*. Springer, 2004.

[OB98] Alexandre Oliva and Luiz Eduardo Buzato. Composition of meta-objects in Guaraná. Technical Report IC-98-33, September 1998.

[Pag04] Vinicius Asta Pagano. Uma abordagem arquitetural com tratamento de exceções para sistemas de software baseados em componentes. Master's thesis, IC, Unicamp, Agosto 2004.

[PC04] Daniel J. Paulish and Anita D. Carleton. Case studies of software-process-improvement measurement. *IEEE Computer*, 27(9):50–57, September 2004.

[PF01] Steve R. Palmer and Mac Felsing. *A Practical Guide to Feature-Driven Development*. Pearson Education, 2001.

[Pfl01] Shari Lawrence Pfleeger. *Software Engineering: Theory and Practice*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2001.

[Pra96] D. K. Pradhan. *Fault-Tolerant System Design*. Prentice Hall, Upper Saddle River, NJ, USA, 1996.

[Pre01] Roger S. Pressman. *Software Engineering: a Practitioner's Approach*. McGraw-Hill, 5th edition, 2001.

[Pro] The Eclipse Project. Eclipse project. http://www.eclipse.org/. Last access in 2004.

[Ran04] Brian Randell. Dependable pervasive systems. *23rd IEEE International Symposium on Reliable Distributed Systems (SRDS'04)*, October 2004.

[RdLeFCF04] C. Rubira, R. de Lemos, and G. Ferreira e F. Castor Filho. Exception handling in the development of dependable component-based systems. In *Software Practice and Experience*, 2004.

[RJB99] James Rumbaugh, Ivar Jacobson, and Grady Booch, editors. *The Unified Modeling Language reference manual*. Addison-Wesley Longman Ltd., Essex, UK, UK, 1999.

[Roc03] Camila Ribeiro Rocha. Uma metodologia para a melhoria da testabilidade de componentes tolerantes a falhas. Proposta de dissertação - IC - Unicamp, 2003.

[RS03] Darrel Reimer and Harini Srinivasan. Analysing exception usage in large java applications. In *Proc. of ECOOP Workshop on Exception Handling in Object-Oriented Systems(EHOOS'2003)*, 2003.

[RX95] Braian Randel and Jie Xu. The evolution of the recovery block concept. *Software Fault Tolerance*, pages 1–20, 1995.

[SDUV03] F. Souchon, C. Dony, C. Urtado, and S. Vauttier. A proposition for exception handling in multi-agent systems. In *Proc. of the SELMAS'03 International Worskshop*, 2003.

[Ses98] Roger Sessions. *COM and DCOM: Microsoft's vision for distributed objects*. John Wiley & Sons, Inc., New York, NY, USA, 1998.

[SG96] Mary Shaw and David Garlan. *Software Architecture: Perspectives on an Emerging Discipline*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1996.

[SK96] Chris Sadler and Barbara Ann Kitchenham. Evaluating software engineering methods and tool. part 4: The influence of human factors. *ACM Sigsoft*, 21(5):11–13, September 1996.

[Som01] Ian Sommerville. *Software Engineering*. Addison-Wesley, 6th edition, 2001.

[SUVD03] F. Souchon, C. Urtado, S. Vauttier, and C. Dony. Exception handling in component-based systems: a first study. In *Proc. of the Exception Handling in Object Oriented Systems: towards Emerging Application Areas and New Programming Paradigms Workshop (at ECOOP'03 international conference)*, 2003.

[SW98] Geri Schneider and Jason P. Winters. *Applying Use Cases: A Practical Guide*. Addison-Wesley, 1st edition, 1998.

[Szy98] C. Szyperski. *Component Software: Beyond Object-Oriented Programming*. Addison-Wesley, 1998.

[Szy02] Clemens Szyperski. *Component Software: Beyond Object-Oriented Programming*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002.

[TFdCGR04] Rodrigo Teruo Tomita, Fernando Castor Filho, Paulo Asterio de C. Guerra, and Cecília Mary Fischer Rubira. Bellatrix: Um ambiente para suporte arquitetural ao desenvolvimento baseado em componentes. In *IV WDBC*, 2004.

[TR05] Rodrigo Teruo Tomita and Cecília Mary Fischer Rubira. Requisitos para um ambiente de desenvolvimento baseado em componentes e centrado na arquitetura de software. Relatório Técnico (a publicar), Instituto de Computação, 2005.

[WBGK] B. L. Williams, S. Brown, M. Greenberg, and M. A. Kahn. Risk perception in context: The savannah river site stakeholder study. *Risk Analysis*, 19(6):1019+.

[WCO] WCOP'96. International workshop on component-oriented programming. http://sky.fit.qut.edu.au/ szypersk/WCOP96/. Last access in 2005.

[XRR+99] J. Xu, B. Randell, A. Romanovsky, R. J. Stroud, A. F. Zorzo, E. Canver, and F. von Henke. Rigorous development of a safety-critical system based on coordinated atomic actions. In *FTCS '99: Proceedings of the Twenty-Ninth Annual International Symposium on Fault-Tolerant Computing*, page 68, Washington, DC, USA, 1999. IEEE Computer Society.

[XRR00] Jie Xu, Alexander Romanovsky, and Brian Randell. Concurrent exception handling and resolution in distributed object systems. *IEEE Trans. Parallel Distrib. Syst.*, 11(10):1019–1032, 2000.

[YSYY03] Zhang You-Sheng and He Yu-Yun. Architecture-based software process model. *Software Engineering Notes*, 28(2), March 2003.

[ZWB03] M. Zelkowitz, D. Wallace, and D. Binkley. Experimental validation of new software technology, 2003.