Tratamento de Exceções no Desenvolvimento de Sistemas Confiáveis Baseados em Componentes

Gisele Rodrigues de Mesquita Ferreira

Dissertação de Mestrado

Instituto de Computação
Universidade Estadual de Campinas

Tratamento de Exceções no Desenvolvimento de Sistemas Confiáveis Baseados em Componentes

Gisele Rodrigues de Mesquita Ferreira

Dezembro de 2001

Banca Examinadora:

- Profa. Dra. Cecília Mary Fischer Rubira (Orientadora)
- Prof. Dr. Rogério de Lemos
  Computer Science Department - University of Kent at Canterbury
- Profa. Dra. Eliane Martins
  Instituto de Computação – UNICAMP
- Prof. Dr. Luiz Eduardo Buzato (Suplente)
  Instituto de Computação – UNICAMP

FICHA CATALOGRÁFICA ELABORADA PELA BIBLIOTECA DO IMECC DA UNICAMP

Ferreira, Gisele Rodrigues de Mesquita

F413t Tratamento de exceções no desenvolvimento de sistemas confiáveis baseados em componentes / Gisele Rodrigues de Mesquita Ferreira -- Campinas, [S.P. :s.n.], 2001.

Orientador : Cecília Mary Fischer Rubira

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

1.Tolerância a falhas (Computação). 2.Software desenvolvimento. 3.Projetos de sistemas. 4.Engenharia de software. I. Rubira, Cecília Mary Fischer. II. Universidade Estadual de Campinas. Instituto de Computação. III. Título.

Tratamento de Exceções no Desenvolvimento de Sistemas Confiáveis Baseados em Componentes

Este exemplar corresponde à redação final da Dissertação devidamente corrigida e defendida por Gisele Rodrigues de Mesquita Ferreira e aprovada pela banca examinadora.

Campinas 17 de Dezembro de 2001

Cecília Mary Fischer Rubira

Profa. Dra. Cecília Mary Fischer Rubira (Orientadora)

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

TERMO DE APROVAÇÃO

Tese defendida e aprovada em 17 de dezembro de 2001, pela Banca Examinadora composta pelos Professores Doutores:

Prof. Dr. Rogério de Lemos
UNIVERSITY OF KENT

Profa. Dra. Eliane Martins
IC - UNICAMP

Profa. Dra. Cecília Mary Fischer Rubira
IC - UNICAMP

Resumo

A adoção de uma metodologia adaptada à realidade de uma organização produtora de software é um fator decisivo para a geração de sistemas de alta qualidade que atinjam as necessidades dos clientes dentro de cronogramas e orçamentos previsíveis. Este trabalho apresenta uma metodologia para construção de sistemas tolerantes a falhas que faz uso de técnicas de tratamento de exceções para lidar com o comportamento excepcional do sistema e manter a confiabilidade e disponibilidade dos serviços. Tal metodologia mantém a preocupação com as situações excepcionais e seus tratadores desde a especificação dos requisitos do sistema, se estendendo pelas atividades de projeto e implementação. Esta metodologia é chamada MDCE, um acrônimo para Metodologia para Definição do Comportamento Excepcional de um sistema confiável. A metodologia MDCE traz diretrizes e guias importantes de serem observados pelos analistas em cada fase de projeto de sistemas tolerantes a falhas. A notação adotada pela MDCE foi a UML estendida com novos estereótipos com semânticas e restrições específicas de sistemas tolerantes a falhas. Além disto, este trabalho apresenta como usar os próprios diagramas da UML para representar o comportamento excepcional de um sistema.

MDCE é uma metodologia genérica que pode ser aplicada a modernos processos de desenvolvimento. Em particular, nesta dissertação aplicamos a metodologia MDCE ao Processo Catalysis e mostramos como nossa abordagem foi aplicada em um estudo de caso do Sistema de Mineração.

Abstract

The adoption of a methodology for software development organization is a decisive factor for the production of high quality systems that meet the client's needs and for the achievement of a predictable schedule and budget. This work presents a methodology for fault-tolerant software development by means of exception handling techniques to deal with exceptional behavior in order to keep the availability and reliability of the system's services. This methodology considers the treatment of exceptional situations and their handlers since the outset during the requirements specification phase and through design and implementation phases. This methodology is called MDCE, Methodology of Exceptional Behavior Description, and encompasses directives and guidelines that practitioners should evaluate in each development phase of a fault-tolerant system. The notation adopted by MDCE is UML extended with new stereotypes for modeling fault-tolerant systems. This work also shows how to use UML dynamic diagrams to represent exceptional behavior.

MDCE is a generic methodology that can be adopted by modern development processes. In particularly, we have applied MDCE to the Catalysis Process and shown how our approach works for the Mining System case study.

Agradecimentos

Primeiramente agradeço a Deus por permitir que eu chegasse até aqui. Tantos foram os obstáculos, mas cheguei. O maior dos obstáculos também foi o mais bonito. Hoje tenho em mãos não só esta dissertação mas também a minha linda e querida Ana Júlia.

À minha orientadora também tenho muito que agradecer. Obrigada Cecília pelos ensinamentos, pela objetividade e coerência com que conduziu este trabalho. Obrigada ainda pela amizade, confiança e consideração que teve por mim.

Agradeço à banca pelo cuidado e atenção que tiveram ao ler esta dissertação e também pelas correções e comentários sugeridos. Rogério de Lemos, obrigado pelos valiosos comentários e ensinamentos, pela paciência com meu inglês e inexperiência.

Aos meus familiares agradeço pelo apoio que sempre me deram. Especialmente agradeço à minha mãe por ajudar nos cuidados com meus filhos, à minha sogra pelo interesse em meu trabalho e ao meu pai por ter me mostrado a importância e valor dos estudos.

Um agradecimento todo especial ao meu marido Lucas, todo o meu sustentáculo, revisor de meus artigos, pajem para nossos filhos e minha maior contribuição financeira. Foi você quem não me deixou desistir nas horas de desespero.

Às minhas sempre amigas de BH, Júnia, Fá e Pat, o meu muito obrigado. Aos amigos do LSD, Cândida, Gustavo, Islene, Valeska, Nelson, Delano, Gerson e Chris, agradeço pelo companheirismo e por terem tornado minha estadia em Campinas mais agradável.

Agradeço ainda à Fapesp pela contribuição financeira e a todos que de alguma forma contribuíram para o sucesso deste trabalho.

Pedrinho, meu filho querido, obrigado simplesmente por me proporcionar tantas alegrias.

Conteúdo

Lista de Tabelas

Lista de Figuras

Capítulo 1

Introdução

Sistemas de software são intrinsecamente complexos e esta complexidade ainda é agravada pelos novos requisitos impostos pelas aplicações modernas como confiabilidade, segurança e disponibilidade. Na medida em que se aumenta a complexidade dos sistemas e também o número de sistemas desenvolvidos simultaneamente (conseqüentemente, o número de pessoas a serem gerenciadas), se torna imprescindível a utilização de uma abordagem mais organizada e estruturada de trabalho. Somando-se a estas variáveis o constante avanço tecnológico, prazos de liberações dos produtos cada vez mais apertados e a necessidade de manutenção e melhorias dos sistemas de software, planejar, controlar e organizar a produção destes sistemas se torna ainda mais indispensável.

Novas metodologias, paradigmas e tecnologias para auxiliar o desenvolvimento de sistemas são alvos de pesquisa constante por parte dos engenheiros de software. Uma *metodologia* provê métodos de como se desenvolver um software e constitui-se das fases de definição do problema, desenvolvimento do sistema e manutenções corretivas [Pressman01].

Um *processo de desenvolvimento* de software é um conjunto de etapas, métodos, técnicas e práticas que empregam pessoas para o desenvolvimento e manutenção de um software e seus artefatos associados (planos, documentos, modelos, código, casos de testes, manuais, etc.). É composto de boas práticas de engenharia de software que conduzem o desenvolvimento, reduzindo os riscos e aumentando a confiabilidade [Jacobson+99]. Um processo é sustentado por vários conceitos e a *notação* é responsável por expressar estes conceitos. A *notação* é apenas uma linguagem que deve ser utilizada em conjunto com um *processo*. O uso de uma *notação* provê uma linguagem padronizada e organizada que facilita o entendimento entre usuários, gerentes e desenvolvedores.

Para facilitar o aprendizado e a utilização dos processos e notação, são utilizadas *ferramentas* que automatizam tarefas do ciclo de desenvolvimento de software [Pressman01]. Devido a complexidade dos sistemas computacionais modernos, torna-se quase que impossível desenvolve-los sem auxílio de uma ferramenta que reduza os esforços requeridos na confecção e manutenção destes sistemas.

1.1 Motivação

Existem circunstâncias que impedem um programa de prover os serviços especificados. Como se espera que estas circunstâncias ocorram raramente, programadores referem-se a elas como *exceções* [Cristian89]. Na prática, não é bem isto que se observa. *Situações excepcionais* são bastante freqüentes, pois o sistema é constantemente influenciado pelo meio aonde se encontra e por iterações mal sucedidas com seus usuários e hardware deteriorado [Cristian89]. Além disto, sistemas computacionais são objetos muito complexos, compostos por uma grande diversidade de subsistemas, tanto de software quantos de hardware, que interagem entre si. Conseqüentemente, várias *situações excepcionais* podem ocorrer e os problemas podem aparecer nas mais variadas formas que vão desde a indisponibilidade dos serviços até a realização incorreta dos mesmos, levando à corrupção de outros sistemas que até o momento funcionavam corretamente [LeeAnderson90].

Em sistemas computacionais, mais de dois terços do código é dedicado à detecção e tratamento das *exceções* [Cristian89]. Os projetistas, por considerarem que *exceções* são raras, fazem do código destinado ao seu tratamento a parte mais mal documentada, testada e legível do sistema. Entretanto, *exceções* devem ser tratadas com cuidado desde que o estado do programa pode estar inconsistente quando sua ocorrência for detectada. A continuação normal da execução do programa de um estado inconsistente pode levar a novas ocorrências excepcionais ou mesmo ao defeito[1] do sistema [Cristian89]. Observa-se que muitas das falhas existentes em sistemas computacionais estão na forma em como tratar as *situações excepcionais* e recuperação do sistema a um estado livre de erros.

Sistemas confiáveis são sistemas que mantêm seu funcionamento de acordo com sua especificação mesmo na presença de *situações excepcionais*

[1] do inglês *failure*

[LeeAnderson90]. O comportamento deste sistema, mediante a ocorrência de *situações excepcionais*, é chamado de *comportamento excepcional,* ou *comportamento anormal,* que define a forma como o sistema irá se comportar na tentativa de tratar as *situações excepcionais*.

Na tentativa de construir sistemas mais confiáveis existem duas técnicas complementares que podem ser adotadas: a *prevenção de falhas* e a *tolerância a falhas* [LeeAnderson90]. A técnica de *prevenção de falhas* se baseia na tentativa de prever e evitar as falhas. Entretanto, muitas vezes é impossível evitar uma falha, como por exemplo, falhas em componentes físicos deteriorados com o tempo. Desta forma, faz-se necessário o emprego de técnicas de *tolerância a falhas* que visam manter o sistema em pleno funcionamento mesmo na presença de falhas. Vale lembrar que até mesmo para que tolerância a falhas possa ser empregada é importante que as falhas sejam antecipadas e suas conseqüências identificadas para que medidas apropriadas de tolerância a falhas possam ser empregadas para detectar sua ocorrência e manter o correto funcionamento do sistema.

Falhas de projeto em sistemas de software são dificilmente identificadas, pois estes sistemas são, em sua maioria, muito complexos, com um vasto conjunto de possibilidades, tornando os testes exaustivos impraticáveis. Além disto, a inclusão dos requisitos de tolerância a falhas nos componentes de software eleva ainda mais a complexidade do sistema resultante. Desta forma, uma abordagem bem estruturada que auxilie a confecção de projetos bem especificados e a inclusão de medidas de tolerância a falhas é um pré-requisito essencial para sucesso do sistema. Por outro lado, uma abordagem não estruturada pode facilmente reduzir ainda mais a confiabilidade dos sistemas introduzindo mais falhas do que as previstas [LeeAnderson90].

Entretanto, por falta de uma metodologia e ferramentas adequadas para construção de sistemas confiáveis, é uma prática comum entre os desenvolvedores adiarem a preocupação com as *situações excepcionais* apenas para a fase de projeto, executando-a de forma "ad hoc", guiados apenas por sua intuição. Acreditamos que melhores resultados podem ser alcançados se as *situações excepcionais* de cada componente forem consideradas desde as fases iniciais de análise e projeto e ainda que os erros seriam identificados mais cedo, sem a perda do contexto, podendo tratá-lo de forma eficiente. Além disto, as informações

obtidas em uma fase indicarão restrições para as etapas seguintes, evitando retrocessos e os esforços de garantia da confiabilidade estarão distribuídos ao longo do processo, gerando um sistema confiável de melhor qualidade.

1.2 Objetivos

Uma forma para tratar as *situações excepcionais* é através do emprego de técnicas de *tratamento de exceções*. Após a identificação de uma falha, uma *exceção* é levantada indicando a presença de um erro no sistema. A partir deste momento, um *tratador* para a *exceção* é encontrado pelo próprio mecanismo de tratamento de exceções da linguagem de programação. O *tratador* é responsável por tentar tratar a falha e retornar o sistema para a um estado livre de erros.

Tratamento de exceções é uma técnica efetiva para incorporar tolerância a falhas em sistemas computacionais. A maioria das linguagens de programação modernas, tais como C++, Java, Ada, Eiffel e Smaltalk, incluem mecanismos de tratamento de exceções como uma de suas características. Mecanismos de tratamento de exceções de linguagens de programação possuem suas características próprias, mas, essencialmente, eles representam erros como exceções e possibilitam sua identificação e escolha de um tratador adequado para tratá-las. Apesar dos vários mecanismos para tratamento de exceções já propostos, ainda existem problemas em aplicá-los na prática, tal como a falta de legibilidade de um código que mantém a implementação da funcionalidade amalgamado ao tratamento das situações excepcionais. Além disto, os mecanismos de tratamento de exceções provêem certas flexibilidades que dificultam o uso correto de exceções tais como propagação automática e não obrigatoriedade de declaração das exceções nas interfaces públicas dos métodos.

As aplicações modernas, em geral, requerem técnicas de tolerância a falhas, seja para manterem a disponibilidade, como os sistemas Web ou aplicações bancárias, ou para manterem a confiabilidade, como os sistemas críticos embarcados ou médicos. O objetivo principal é a definição de uma metodologia que auxilie a construção de sistemas que requeiram o mínimo de tolerância a falhas para manterem seu funcionamento e disponibilidade mesmo na presença de situações excepcionais. Tal metodologia deve proporcionar uma abordagem sistemática para a especificação do *comportamento excepcional* de um sistema de software durante todas as suas fases de desenvolvimento, a começar pela

especificação de requisitos, passando pelo projeto e culminando na implementação. Os resultados da especificação do comportamento excepcional do sistema durante estas etapas podem servir de base para a fase de testes. Durante este trabalho, chamaremos esta metodologia de MDCE, um acrônimo para *Metodologia para Definição do Comportamento Excepcional* de um sistema confiável.

Tal metodologia deve ser genérica o suficiente para que possa ser adaptada a alguns processos de desenvolvimento. No presente trabalho, realizaremos sua aplicação ao Processo Catalysis [Desmond98], um processo de desenvolvimento de sistemas orientados a objetos e baseado em componentes. A escolha do Processo Catalysis foi fundamentada pela sua noção de *colaboração* como unidade básica de desenvolvimento de um sistema. Consideramos que um componente sozinho não é capaz de prover os recursos necessários para identificação ou tratamento de uma falha de forma efetiva. Sendo assim é necessário considerar a *colaboração* entre componentes do sistema para capturar o comportamento interativo entre eles.

Quanto à notação, faremos uso da UML (Unified Modeling Language), definida em [Rumbaugh+99] e [Booch+99]. A UML foi padronizada pela OMG em 1997 se tornando um padrão para modelagem de sistemas orientados a objetos. A partir daí tornou-se quase que obrigatória para que as diversas organizações tenham uma linguagem comum de comunicação de suas experiências.

Além disto, faz parte deste trabalho a utilização de ferramentas de apoio para o desenvolvimento de sistemas confiáveis. Várias ferramentas estão disponíveis no mercado, entre elas as ferramentas da empresa Rational Software[2], como por exemplo, a ferramenta Rose [Rose] que oferece apoio a construção de todos os modelos de modelagem da UML e geração automática de código. Já dispondo de uma ferramenta de modelagem tão poderosa, nossa proposta se resume a estendê-la para que seja possível representar os diagramas propostos pela metodologia MDCE. Desejamos ainda poder automatizar a construção de algumas partes dos modelos propostos, além da geração de *scripts* que verifiquem a consistência do modelo gerado segundo a metodologia MDCE. Por fim, a factibilidade da metodologia MDCE proposta será verificada através de um estudo de caso: o Sistema de Mineração apresentado em [Sloman+87].

[2] http://www.rational.com

1.3 Organização deste documento

Este documento foi dividido em sete capítulos organizados da seguinte forma:

- **Capítulo 2 - Fundamentos Teóricos**: Para embasar este trabalho, o segundo capítulo apresenta fundamentos teóricos das duas áreas que o compõem: Tolerância a Falhas e Engenharia de Software. Em Tolerância a Falhas foram apresentadas a terminologia utilizada no decorrer do trabalho e suas fases de implantação. Já em Engenharia de Software são apresentados os aspectos que precisam estar presentes para que um projeto seja bem sucedido: processo, notação, arquitetura e ferramentas.

- **Capítulo 3 - MDCE - Uma Metodologia para Definição do Comportamento Excepcional de Sistemas Confiáveis**: Neste capítulo são apresentados aspectos importantes a serem considerados na modelagem de sistemas confiáveis e na definição de uma metodologia e sua posterior aplicação em uma organização. Além disto, apresentamos uma visão geral do que caracteriza cada etapa da metodologia MDCE.

- **Capítulo 4 - Metodologia MDCE Aplicada ao Catalysis** : Neste capítulo a metodologia MDCE apresentada no Capítulo 3 foi aplicada ao processo de desenvolvimento Catalysis. Algumas etapas de desenvolvimento que são específicas do processo Catalysis foram estendidas de forma a manter a representação e identificação do comportamento excepcional.

- **Capítulo 5 - Propostas de Extensões** da UML : Este capítulo é composto de extensões propostas para a linguagem UML para visualizar, especificar, construir e documentar o comportamento excepcional de um sistema confiável.

- **Capítulo 6 - Estudo de Caso** : Neste capítulo, aplicamos o processo Catalysis, apresentado no Capítulo 4, no estudo de caso do Sistema de Mineração [Sloman+87].

- **Capítulo 7 – Conclusões e Trabalhos Futuros** : Este capítulo sintetiza as contribuições deste trabalho e apresenta as conclusões. Ele ainda discute as possíveis melhorias da metodologia proposta e sugere trabalhos futuros.

- **Apêndice A - Automatização de Tarefas Usando a Ferramenta Rose** : Este apêndice apresenta um modelo e exemplo de aplicação de um *script* que automatiza a confecção de um modelo de classes segundo especificado pela metodologia MDCE.

- **Apêndice B - Simulador do Sistema de Mineração** : Este apêndice apresenta o projeto de um simulador para o estudo de caso do Sistema de Mineração.

Capítulo 2

Fundamentos Teóricos

Este capítulo estabelece a terminologia e define os conceitos utilizados neste trabalho. Na seção 2.1 apresentamos a definição de um *sistema tolerante a falhas* e os conceitos e técnicas referentes à implantação de tolerância a falhas em sistemas computacionais. Na seção 2.2 apresentamos os princípios de engenharia de software que foram empregados neste trabalho, tais como, componentes, processo, ferramentas e arquitetura.

2.1 Princípios de Tolerância a Falhas

A partir do momento que os computadores se tornaram uma ferramenta indispensável na sociedade moderna, um princípio fundamental surgiu: os benefícios trazidos pelos sistemas computacionais são tantos quantos os prejuízos que estes nos proporcionam quando deixam de funcionar ou quando funcionam incorretamente [Avizienes97]. *Sistemas confiáveis*[3] são sistemas que não irão desviar das intenções de seus projetistas diante de falhas de projeto, físicas ou interação com os usuários, ou ainda permitir que vírus ou ataques maliciosos afetem seus serviços essenciais. *Sistemas disponíveis*[4] são sistemas que se mantém disponíveis mesmo diante de falhas de projeto, físicas ou humanas. Tolerância a falhas é a melhor forma de manter a *confiabilidade* e *disponibilidade* destes sistemas pois muitas vezes é impossível evitar que falhas aconteçam, como por exemplo, falhas decorrentes da deterioração de hardware ou falhas de projeto.

2.1.1 Definições: Falhas, Erros e Defeitos

As possíveis causas de falhas são fenômenos naturais de origem interna ou

[3] do inglês: *reliability*
[4] do inglês: *availability*

externa e ações humanas acidentais ou intencionadas. Em qualquer uma das situações, a existência de *falhas*[5] em um componente leva a manifestação de *erros*[6] no sistema que se não forem corretamente tratados podem resultar em um *defeito*[7] do sistema. Veja Figura 1. Um defeito é definido como um desvio da especificação que não podem ser tolerados e sim devem ser evitados.

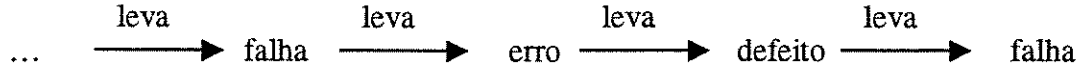

Figura 1 – Falhas, Erros e Defeitos

Para ilustrar, considere um chip de memória, que apresenta uma falha do tipo fixo-em-zero (stuck-at-zero) em um de seus bits. Esta falha pode provocar uma interpretação errada da informação armazenada em uma estrutura de dados e como resultado o sistema pode negar autorização de embarque para todos os passageiros de um vôo.

É interessante observar que uma falha pode resultar em um, mais de um ou nenhum erro. Aquela porção da memória pode nunca ser usada e a falha pode nunca vir a se manifestar sob a forma de erro. Um erro não necessariamente conduz a um defeito. No exemplo, a informação de vôo lotado poderia eventualmente ser obtida a partir de outros dados redundantes da estrutura. Além disto, várias falhas podem resultar em um mesmo erro. Isto faz com que a atividade de identificar a falha que ocasionou um erro seja uma tarefa difícil. Por exemplo, negar autorização de embarque para todos os passageiros de um vôo pode ser devido ao bit que permaneceu em zero do exemplo citado ou devido à interrupção do canal de comunicação da estação do aeroporto com o servidor de banco de dados onde as informações estão armazenadas. Conseqüentemente, a identificação do erro não resulta naturalmente na identificação da falha responsável. Desta forma, faz-se necessário uma análise rigorosa do contexto de manifestação do erro para que se possa inferir as causas e poder tratá-las.

Inicialmente, o conceito de falhas estava relacionado apenas às falhas de origem física, como a falha do bit que se fixou em zero no exemplo. Entretanto, dado o aumento da complexidade dos sistemas de software, *falhas de projetos* se

[5] do inglês: fault
[6] do inglês: error
[7] do inglês: failure

tornaram quase que inevitáveis. Falhas de projeto são problemas de especificação, projeto ou implementação do sistema. Além destas falhas, existem as *falhas de interface* que são decorrentes de uma operação incorreta realizada por um usuário do sistema. Existem ainda aquelas falhas decorrentes de ações maliciosas que objetivam alterar ou interromper o serviço prestado, as *falhas intencionais*. Para definir uma falha considera-se ainda a sua duração (persistência ou transiente) e extensão (interna a um módulo ou externa).

2.1.2 Componente Tolerante a Falhas Ideal

Segundo Lee e Anderson [LeeAnderson90], um sistema computacional é definido em termos de componentes que interagem entre si para fornecer a funcionalidade desejada. Esta definição é recursiva, ou seja, um componente pode ser um sistema, formado por diversos componentes. O termo componente é empregado em um sentido genérico, isto é, um modelo abstrato que se refere tanto a componentes de software quanto componentes de hardware. Componentes recebem requisições de serviços e produzem respostas a estas requisições. Com o intuito de produzir respostas a uma dada requisição, um componente pode solicitar a seus sub-componentes um determinado serviço e assim sucessivamente.

As respostas de um componente ideal podem ser de dois tipos: normais e excepcionais (ou anormais). Respostas normais são produzidas quando o componente realiza o serviço requisitado de forma satisfatória. Respostas anormais ou excepcionais são aquelas produzidas quando alguma situação excepcional ocorreu durante a execução do serviço que não pôde ser realizado com sucesso. As respostas excepcionais de um componente são denominadas *exceções*.

Como conseqüência do particionamento das respostas em duas categorias, a atividade de um componente também pode ser particionada em atividade normal que implementa os serviços normais do componente e a atividade excepcional (ou anormal) que implementa as medidas de tolerância as falhas que causaram a manifestação das exceções. Esta separação entre atividade normal e excepcional leva-nos à definição do componente ideal tolerante a falhas apresentado na Figura 2.

Exceções podem ser levantadas nos componentes do sistema devido a problemas internos *(exceções internas)*, pela incapacidade de fornecer um serviço específico *(exceções externas)* ou em alguma requisição incorreta ou inexistente de

serviços *(exceções de interface).* No modelo do componente ideal, após o levantamento de uma exceção, um mecanismo de tratamento de exceções é responsável pela interrupção do processamento normal do componente. Em seguida, o mecanismo realiza a busca de um tratador adequado para tratar a exceção. A busca pelo tratador pode resultar em dois cenários diferentes: (i) o tratador adequado para a exceção é encontrado e o componente será capaz de tratá-la localmente encapsulando a falha em seu interior; (ii) o componente não estará apto a tratar a falha localmente, ou por falta de um tratador adequado ou por um mal projeto deste tratador, e então uma exceção de defeito será propagada para um nível superior da hierarquia de chamadas.

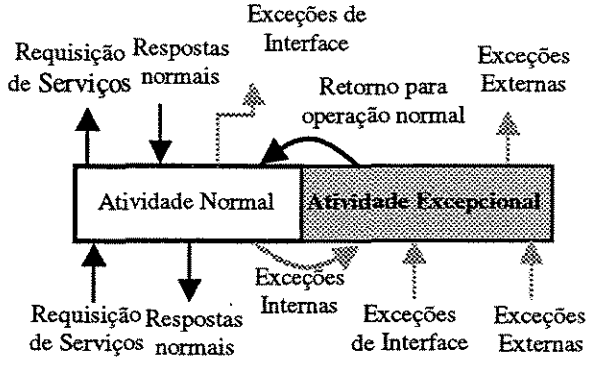

Figura 2 – Componente ideal tolerante a falhas

2.1.3 Etapas de Implantação de Tolerância a Falhas

O objetivo das técnicas de tolerância a falhas é prevenir erros e falhas que possam levar o sistema a um defeito. Portanto estratégias para lidarem com erros e com as falhas são ambas importantes. Todas as técnicas de tolerância a falhas se baseiam na distribuição e na redundância de elementos [LeeAnderson90].

A redundância dentro de um sistema pode ser de componentes (e conseqüentemente todas suas interações), de projeto e temporal. Entretanto, muitos fatores devem ser levados em conta ao se empregar redundância, seja de qual tipo for. Um dos fatores é o custo monetário dado ao investimento adicional na compra de equipamentos e mesmo de projetos de software. Além disto, as perdas em termos de desempenho, sobrecarga e espaço que a redundância pode acarretar ao sistema podem ser altas. Tais custos devem ser comparados às conseqüências causadas por problemas no funcionamento do sistema ou mesmo pela indisponibilidade dos serviços prestados.

A implantação das técnicas de tolerância a falha de modo que erros e falhas possam ser detectados e recuperados eficientemente consistem em quatro fases segundo [LeeAnderson90]:

1. **Detectar o erro:** Os mecanismos empregados na detecção de erros têm como objetivo o identificar o erro causado pela falha e levantar exceções sinalizando sua presença. O tratamento da exceção levantadas nesta fase é responsabilidade das fases subseqüentes.
2. **Confinar e avaliar o dano:** Quando um erro é detectado, muitos outros componentes já podem ter sido influenciados por ele. Isto acontece porque existe um atraso entre a manifestação da falha e a detecção de suas conseqüências errôneas. Durante este período, muita informação incorreta pode ter sido disseminada entre os elementos do sistema, levando a ocorrência de outros erros que ainda podem não ter sido detectados. Desta forma, antes que tente recuperar o erro, é necessário avaliar a extensão do dano causado ao sistema.
3. **Recuperar o erro:** Após a identificação do erro e verificação do estrago por ele causado, é necessário eliminar os erros e levar o sistema para um estado livre de erros. A recuperação do erro é responsabilidade do tratador (handler) da exceção sinalizada.
4. **Tratar a falha:** Embora a técnica de recuperação de erros tenha retornado o sistema para um estado livre de erros, ainda são necessárias novas técnicas que permitem que o sistema continue funcionando. O importante é garantir que a mesma falha não se repetirá imediatamente, retornando o sistema para um estado instável. Manifestações freqüentes de uma mesma falha podem forçar o sistema a falhar apesar de todos os esforços das técnicas de tolerância as falhas empregadas, seja porque as conseqüências da falha vão se agravando ou porque o sistema está ocupado em restaurar estados estáveis não tendo condições de fornecer o serviço de sua responsabilidade.

O restante deste capítulo será dedicado a apresentação dos termos e tecnologias relacionados à Engenharia de Software.

2.2 Princípios de Engenharia de Software

Esta seção se dedica a apresentar o estado da arte em termos de tecnologia de Engenharia de Software para a construção de sistemas flexíveis, reusáveis e

robustos. Na seção 2.2.1 apresentamos a definição de *Desenvolvimento de Sistemas Baseado em Componentes* que garantem um desenvolvimento mais rápido, confiável e barato. Já a seção 2.2.2 apresenta a definição de *Processo de Desenvolvimento*, método básico para obter sucesso na construção de qualquer sistema computacional. Na seção 2.2.3 apresentamos o conceito de *Arquitetura de Software* e como suas propriedades podem influenciar o sistema resultante. Por fim, a seção 2.2.4 introduz os conceitos, vantagens e exemplos de ferramentas de auxílio automatizado para a construção de sistemas de software chamadas de *Ferramentas CASE.*

2.2.1 Desenvolvimento Baseado em Componentes

O desenvolvimento de software baseado em componentes (CBSE – *Component-Based Software Engineering*) permite que uma aplicação seja construída a partir de componentes de software que já foram previamente especificados, construídos e testados. Esta é uma abordagem que vem ganhando muita atenção na comunidade de engenharia de software [Szyperski98]. A disponibilidade de mecanismos de interconexão de componentes, como OMG CORBA, Microsoft COM/DCOM e Sun JavaBeans, é um dos principais estímulos ao desenvolvimento baseado em componentes. O desenvolvimento de aplicações torna-se um processo de seleção, adaptação e composição de componentes.

Existem muitas definições de componentes de software, mas de forma bem genérica e informal, um componente é um pacote desenvolvido e testado separadamente e distribuído como uma unidade que pode ser composta a outros componentes para construir algo com maior funcionalidade [Szyperski98]. Componentes de software possuem duas representações: uma representação *lógica* e uma *binária* que também pode ser chamada de *física.* Um *componente lógico* é uma representação, em tempo de projeto, de um pacote que deve estar bem separado do seu ambiente e de outros pacotes, encapsulando suas características e mantendo suas funcionalidades auto-contidas [Szyperski98]. Para isto, ele deve possuir uma clara especificação das funcionalidades oferecidas e requeridas disponibilizadas através de sua interface. Um *componente lógico* pode ou não resultar em *componentes binários,* isto é, pode ou não estar representado sob a forma de linguagem de máquina, ou de uma representação intermediária, que de alguma forma possa ser executada em um computador, como acontece com os

byte-codes da linguagem Java.

Em uma aplicação baseada em componentes deve ser possível substituir um componente por outro, com uma especificação equivalente, sem afetar a conexão e a funcionalidade disponível. Isto só é possível porque a conexão de componentes se dá exclusivamente pelas suas interfaces. Se houver qualquer dependência de implementação, a possibilidade de substituição de componentes é perdida.

Neste trabalho, nos concentramos em projetar componentes de software que sejam *flexíveis*, isto é, componentes que possam ser facilmente *estendidos* e *reutilizados* [Richter99]. Novas características podem ser acrescentadas a um projeto *extensível* de componente sem muita dificuldade, afetando o menor número de elementos possíveis. Um projeto de um componente *reutilizável* é tal que possa ser aproveitado inteiramente ou em partes em uma nova aplicação. A forma como um componente é especificado influi na flexibilidade do projeto o que pode decidir o grau de reuso e modificabilidade do mesmo. Por exemplo, um componente de software, para ser reutilizado de forma efetiva, deve ser confiável, isto é, deve manter medidas de tolerância a falhas na tentativa de tratar as falhas localmente ou sinalizá-las sob a forma de *exceções*.

Podemos dizer então que a especificação de um componente, isto é, sua representação lógica, é mais importante do que a forma como esta especificação foi implementada, ou seja, do que sua representação binária. Desta forma, neste trabalho nos concentramos na construção de *componentes lógicos* que sejam flexíveis e robustos. A partir deste momento, quando nos referirmos a *componentes* estamos tratando de *componentes lógicos*.

Associa-se a reutilização de componentes com a idéia de que basta integrá-los a uma aplicação e reutilizá-los da forma como foram implementados. Porém, normalmente o componente deve ser *adaptado* a fim de adequá-lo ao contexto da aplicação. Neste trabalho, estamos preocupados em construir sistemas confiáveis onde o problema de adaptação certamente está presente. A grande questão é como fazer uso de componentes de software não confiáveis para a construção de sistemas que devam ser confiáveis. Para que um componente não confiável possa ser usado na construção de sistemas confiáveis é necessária a utilização de técnicas de adaptação de componentes para inclusão de mecanismos de tolerância a falhas. O trabalho [Weiss01] apresenta um modelo para a adaptação de componentes de software, que não requer a manipulação do código fonte dos

componentes, e nem que estes componentes sejam construídos seguindo estruturas pré-determinadas. O modelo utiliza o conceito de reflexão computacional, a fim de realizar alterações tanto na estrutura do componente quanto no seu comportamento de forma transparente.

Como dissemos anteriormente, a especificação de um componente é mais importante do que a forma como esta especificação será implementada. Desta forma, um processo de desenvolvimento que garanta uma especificação completa e flexível dos componentes é fundamental para o sucesso da construção do sistema como um todo. Veja na seção seguinte em que consiste e qual é a importância de um processo de desenvolvimento de software.

2.2.2 Processo de Desenvolvimento de Software

Os processos de desenvolvimento de software têm ganhado considerável atenção nas últimas décadas devido ao crescimento exponencial da complexidade, tamanho e custo dos sistemas atuais. Um *processo de desenvolvimento* de software é um conjunto de etapas, métodos, técnicas e práticas que empregam pessoas para o desenvolvimento e manutenção de um software e seus artefatos associados (planos, documentos, modelos, código, casos de testes, manuais, etc.). É composto de boas práticas de engenharia de software que conduzem o desenvolvimento, reduzindo os riscos e aumentando a confiabilidade [Jacobson+99].

Um desenvolvimento de software segue dois processos distintos ao mesmo tempo: um *processo gerencial* que esquematiza atividades, planeja liberações, aloca recursos e monitora progressos e um *processo de desenvolvimento* que especifica e implementa o sistema dado seus requisitos [Cheesman+01]. Na literatura encontramos alguns exemplos de abordagens que incluem um dos processos ou ambos. Por exemplo, o Catalysis [Desmond98] e o UML Components [Cheesman+01] são principalmente *processos de desenvolvimento* e o Unified Process (UP) [Jacobson+99] cobre tanto o *processo gerencial* quanto o *processo de desenvolvimento*.

Neste trabalho estamos concentrados em *processos de desenvolvimento* e para facilitar a leitura, quando nos referirmos a um *processo* estamos tratado de um *processo de desenvolvimento*.

Embora existam muitos processos de desenvolvimento de software, algumas atividades fundamentais estão presentes em todos eles. São elas: levantamento de

requisitos, projeto, implementação e testes, como mostra a Figura 3. A fase de *levantamento de requisitos* se focaliza em identificar os serviços que devem ser automatizados, as informações que devem ser processadas, além de questões como desempenho, confiabilidade, disponibilidade e segurança. O modelo de casos de uso é uma ferramenta efetiva e amplamente empregada para o levantamento dos requisitos. Durante a fase de *projeto* os desenvolvedores se concentram em definir como os dados serão estruturados, como a funcionalidade será implementada em uma arquitetura de software e em como as interfaces serão caracterizadas. Durante a *implementação*, o projeto é transportado para uma linguagem de implementação em forma de código fonte e durante a etapa de *testes* o sistema é verificado para certificar-se de que os requisitos especificados estão todos implementados corretamente.

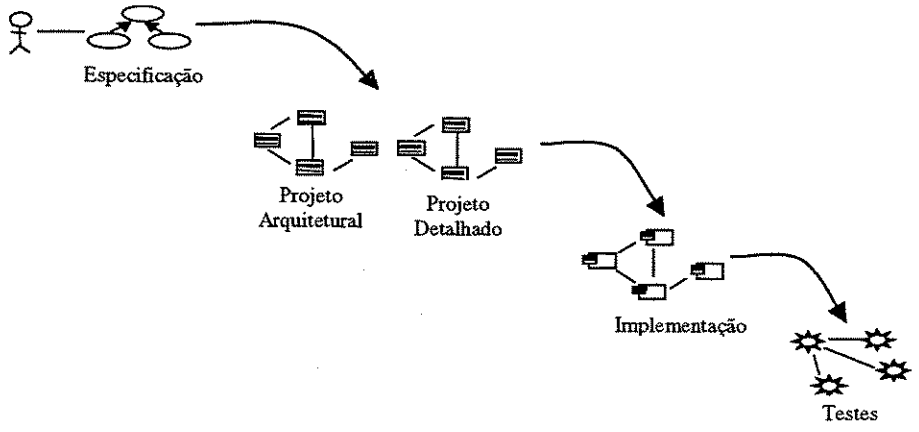

Figura 3 – Fases do ciclo de vida de um sistema

Os requisitos de um software são as características, propriedades e comportamentos desejáveis para aquele produto. Costuma-se dividir os requisitos em:

- requisitos *funcionais*, que representam os comportamentos que um programa ou sistema deve apresentar diante de certas ações de seus usuários;
- requisitos *não funcionais*, que quantificam determinados aspectos do comportamento.

Por exemplo, em um terminal de caixa automático, os tipos de transações bancárias suportadas são características funcionais. A facilidade de uso, o tempo de resposta e o tempo médio entre falhas são características não funcionais.

A arquitetura de um sistema de software define a forma como seus elementos

estão organizados e interagem entre si. Algumas propriedades arquiteturais influenciam, direcionam e restringem todas as fases do ciclo de vida do software. São por estes e outros motivos que o projeto arquitetural é tão importante para a construção de um sistema que realmente atenda as necessidades do cliente. Veja na seção 2.2.3 em que consiste a arquitetura de software e como alguns estilos arquiteturais podem ser usados para propiciar a construção de sistemas que vêem de encontro às expectativas e restrições do cliente.

Dado o tamanho e complexidade dos sistemas computacionais modernos, nenhum processo de desenvolvimento pode ser aplicado sem a utilização de ferramentas de apoio informatizado a construção e manutenção dos artefatos envolvidos. Estas ferramentas de apoio informatizado ao processo são chamadas de Ferramentas CASE e estão apresentadas na seção 2.2.4.

2.2.3 Arquitetura de Software

A arquitetura de software define o sistema em termos de componentes, a interação entre eles e os atributos e funcionalidades de cada componente através de um alto nível de abstração[Sommerville01]. Uma definição da arquitetura nos dá uma clara perspectiva de todo o sistema e do controle necessário para seu desenvolvimento. Isto é conseguido através de várias abstrações e visões do sistema sobre as perspectivas dos diferentes colaboradores, isto é, desenvolvedores, analistas, projetistas, clientes, usuários, entre outros.

Uma propriedade arquitetural representa uma decisão de projeto relacionada a algum requisito *não-funcional* [Sommerville01]. Exemplos de propriedades arquiteturais são:

- **Modificabilidade**: característica que define a capacidade do sistema de se adaptar a alterações de requisitos ou mesmo a inclusão de novos requisitos;
- **Reusabilidade**: característica que define o grau de reuso de um componente, isto é, define quão genérico, independente da aplicação, é este componente para que possa ser reutilizado em diversas aplicações;
- **Desempenho**: tempo de resposta ou tempo de processamento de uma requisição que deve ser compatível com a realidade e necessidade do cliente;
- **Tolerância a Falhas**: capacidade do sistema de reagir e recuperar-se diante de situações excepcionais;

As propriedades arquiteturais são derivadas dos requisitos do sistema e

influenciam, direcionam e restringem todas as fases do ciclo de vida do software. *Modificabilidade*, por exemplo, depende fortemente de como o sistema foi modularizado, pois isto reflete as estratégias de encapsulamento do sistema. *Reusabilidade* de componentes depende do nível de acoplamento dos componentes do sistema. *Desempenho* depende da complexidade de comunicação entre os componentes e especialmente da distribuição física destes componentes. *Tolerância a falhas*, por sua vez, só é possível através da aplicação de técnicas de redundância de software e/ou hardware e tratamento de exceções.

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 [Monroe+97] [Shaw+96]. Um *estilo arquitetural* caracteriza uma família de sistemas que são relacionadas pelo compartilhamento de propriedades estruturais e comportamentais (semântica). Um estilo de arquitetura provê um vocabulário de elementos da arquitetura, também chamados de componentes arquiteturais, e seus conectores. Além disto, apresenta ainda regras e restrições sobre a combinação dos componentes arquiteturais.

Alguns estilos descrevem propriedades arquiteturais que são independentes de domínio, como por exemplo os estilos *Pipe and Filter*, *n-Camadas* e *Cliente-Servidor*. Porém existem estilos que são empregados quando se necessita de alguma propriedade que é específica de um domínio de aplicações, como por exemplo tolerância a falhas. Para estes domínios podemos empregar um estilo arquitetural baseado no Componente Ideal Tolerante a Falhas apresentado na seção 2.1.2 - Figura 2. Neste estilo arquitetural, um componente arquitetural possui uma separação clara entre o seu comportamento normal e excepcional e um modelo de interação bem definido que inclui as respostas normais e excepcionais que podem ser retornadas pelos seus serviços. Além disto, este estilo arquitetural define uma hierarquia em camadas onde um componente só pode requisitar serviços de um sub-componente, ou seja, de um componente hierarquicamente abaixo dele. Veja Figura 4.

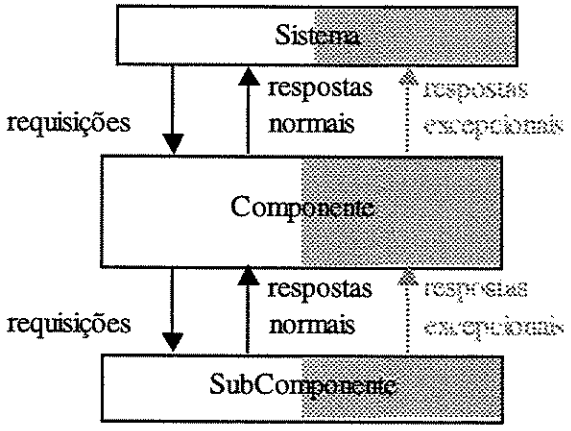

Figura 4 – Estilo arquitetural para sistemas confiáveis

Um ambiente distribuído também é propício para construção de sistemas tolerantes a falhas pois possibilita manter as cópias distribuídas e isoladas geograficamente para colaborarem na manutenção da disponibilidade e confiabilidade do sistema.

2.2.4 Ferramentas CASE

Do inglês *Computer Aided Software Engineering*, o termo CASE é utilizado para designar ferramentas de auxílio automatizado à Engenharia de Software. Estas ferramentas são empregadas de forma a reduzir significativamente o custo de produção e tempo de desenvolvimento dos sistemas, além de possibilitar um aumento na qualidade e produtividade dos sistemas.

Apesar da idéia de utilizar o computador para auxiliar nas tarefas de engenharia de software não ser nova, sua aplicabilidade não era ampla devido a inexistência de padrões entre os métodos de engenharia de software. Cada ferramenta CASE oferecia, até bem pouco tempo, uma metodologia própria com uma notação particular. O surgimento de uma notação unificada UML trouxe um novo impulso a este tipo de ferramenta.

Ferramentas CASE representam um papel fundamental na solução para os problemas de desenvolvimento e manutenção de sistemas. Se usadas adequadamente, estas ferramentas proporcionarão os seguintes benefícios [Sommerville01]:

- Aumento da qualidade dos produtos finais uma vez que diminuem a probabilidade de erros através de verificação automatizada da consistência dos modelos, integridade de dados e testes.
- Simplificação da manutenção de programas através de automatização de documentação, atualização de modelos através de recursos como engenharia reversa e atualização de código com base em alterações no modelo;
- Aumento da produtividade de equipes de desenvolvimento através de recursos como controle de versão e compartilhamento de informações como dados, modelos e documentações;
- Eliminação do trabalho monótono, com a conseqüente liberação dos desenvolvedores para trabalhos mais criativos do desenvolvimento;
- Facilita a prototipagem de sistemas;
- Facilita a desvinculação entre modelagem e decisões de implementação proporcionando aos seus usuários uma visão completamente conceitual do sistema, na qual decisões de implementação não influenciam;
- Facilita o desenvolvimento incremental de sistemas através da documentação e do re-projeto de sistemas;

A empresa Rational Software[8] possui uma série de ferramentas de apoio ao desenvolvimento de software, como por exemplo, Rose [Rose] para modelagem, RequisitePro [RequisitePro] para análise de requisitos e o RUP [Rup] uma ferramenta de apoio à utilização informatizada do Processo Unificado [Jacobson+99].

A ferramenta RequisitePro é uma ferramenta de gerência de requisitos que proporciona uma melhor comunicação entre a equipe de trabalho e reduz os riscos do projeto pois um bom projeto começa com uma boa definição dos requisitos do sistema.

Ferramentas como o RUP propiciam capacitação e agilidade do usuário no uso do Processo Unificado e auxílio na personalização do mesmo. Além disto, o RUP apresenta uma série de modelos para a elaboração dos artefatos de projeto, guias para realização das atividades e atribuição de tarefas. É uma ferramenta de gerência de projetos que faz referência às ferramentas de produção para definir atividades específicas do processo, como por exemplo, o RequisitePro. Maiores informações sobre este produto podem ser obtidas em [Rup] e [Kruchten00].

Existem poucos produtos disponíveis nesta área de apoio à gerência de projetos, mas ainda podemos citar o PSP Studio e o TSP Support Tool ferramentas de suporte aos processos PSP [Humphrey95] e TSP [Humphrey99], respectivamente.

A ferramenta Rose[Rose], principal produto da Rational, oferece suporte completo à modelagem orientada por objetos, atendendo desde o nível conceitual até a implementação física em algumas linguagens tais como Java, Visual Basic, C++ e Smalltalk, dentre outras. A ferramenta Rose oferece suporte a todos diagramas da UML, isto é, diagramas de casos de uso, classes, objetos, colaboração, atividades, componentes e distribuição. Uma outra ferramenta de modelagem UML e geração automática de código disponível no mercado é o Together [Together] da empresa TogetherSoft Corporation. Together foi implementado em Java e por isto é independente de plataforma. Já fora testado em Linux, Sun Solaris (SPARC), e Microsoft Windows (95/98/NT/2000).

2.3 Resumo

Este capítulo apresentou conceitos e terminologias das áreas de Tolerância a Falhas e Engenharia de Software. Alguns conceitos importantes serão usados no restante desta dissertação com uma semântica mais restrita. O termo *processo*, quando usado no contexto deste trabalho, designa um *processo de desenvolvimento* que especifica e implementa um sistema de software. Já quando nos referirmos a *componentes* estamos nos concentrados em sua representação em tempo de projeto, isto é, trata-se de um *componente lógico*. A definição de um componente lógico é recursiva, isto é, internamente um componente é formado por sub-componentes e assim recursivamente. Este capítulo mostrou ainda como um estilo arquitetural influencia os requisitos não-funcionais de um sistema e que o estilo arquitetural do componente ideal é favorável para a construção de sistemas tolerantes a falhas.

Até o momento apresentamos técnicas e termos que serão utilizados na definição de uma metodologia que auxilie a construção de sistemas tolerantes a falhas. Os capítulos seguintes se destinam a apresentar e exemplificar tal metodologia além de apresentar como os conceitos aqui definidos serão empregados para que sistemas tolerantes a falhas, flexíveis e reutilizáveis sejam construídos de forma compatível com o cronograma e orçamento previsto.

---

[8] http://www.rational.com

Capítulo 3

MDCE - Uma Metodologia para Definição do Comportamento Excepcional de Sistemas Confiáveis

Este capítulo apresenta a metodologia para definição do comportamento excepcional de sistemas tolerantes a falhas chamada de MDCE. MDCE é uma metodologia que concentra principalmente na definição, projeto e implementação do comportamento excepcional de um sistema visando prever as falhas e auxiliar o projeto de tratadores eficientes para os erros que podem ameaçar o correto funcionamento e disponibilidade de um sistema confiável.

3.1 Introdução

Os sistemas de software a serem desenvolvidos utilizando-se a metodologia MDCE proposta são aplicações tolerantes a falhas. Os principais atributos destas aplicações são:

- **Confiabilidade**[9]**:** nos dias de hoje não é admissível que um sistema venha produzir uma resposta incorreta ou mesmo não execute o serviço da forma como esperado pelo cliente. Um componente se comportando erroneamente pode influenciar o comportamento do resto do sistema levando-o ao caos. Qualquer aplicação deve se comportar de uma forma correta e previsível mesmo diante de situações de falha.
- **Segurança no funcionamento**[10]: o uso de computadores em aplicações críticas, que envolvem vidas humanas, requer que falhas que possam trazer riscos ao ambiente onde ele esteja inserido sejam evitadas. Por exemplo, um

[9] do inglês: *Reliability*

computador controlando o sistema de aquecimento de uma residência, caso venha a falhar, pode causar um super aquecimento e possivelmente resultar em um incêndio local. Sendo assim, estas aplicações devem garantir que problemas internos não tragam riscos para o meio aonde ela está inserida.

- **Disponibilidade**[11]: muitas vezes a indisponibilidade de um serviço é inaceitável dado a dependência do mundo moderno dos sistemas computacionais. Sistemas médicos, se faltarem por um instante apenas podem trazer resultados catastróficos. Em um âmbito menos crítico estão os sistemas Web que devem estar disponíveis 7 dias na semana, 24 horas por dia. A indisponibilidade de um serviço pode trazer desde conseqüências apenas desagradáveis como até levar a morte seres humanos ou ocasionar grandes perdas financeiras.
- **Tempo de resposta baixo**: Técnicas de tolerância a falhas geralmente se baseiam em replicação de elementos críticos do sistema e checagem das respostas fornecidas por cada um deles. Este é um agravante no desempenho da aplicação dado que uma mesma ação é executada mais de uma vez e ainda testada no fim. Deve-se estar atento para não penalizar o tempo de resposta e tornar o sistema inviável sob o ponto de vista prático.

Além dos atributos dos produtos é preciso levar em consideração os objetivos da própria metodologia MDCE e o que se pretende alcançar com sua implantação. O objetivo da metodologia MDCE é auxiliar o desenvolvimento de software que requer medidas de tolerância a falhas, mantendo os prazos, custos e qualidade sobre controle. Esse objetivo geral pode ser traduzido em metas mais específicas, tais como:

- **Minimizar o tempo de liberação do produto no mercado**: a complexidade adicional ao se considerar e tratar as situações excepcionais de um sistema pode penalizar o tempo de liberação deste sistema no mercado. Mais de dois terços do código de um sistema é dedicado à detecção e tratamento das exceções [Cristian89]. Sendo assim, as exceções, juntamente com os seus tratadores são partes significativas do projeto e implementação de sistemas e devem, portanto, serem consideradas desde a fase de levantamento de

[10] do inglês: *Safety*
[11] do inglês: *Availability*

requisitos para que a previsão dos prazos já as inclua. Além disto, é importante prover paralelismo das atividades destinadas à elaboração do comportamento normal do sistema e das atividades destinadas à elaboração do comportamento excepcional para que o tempo de confecção do sistema não seja inviável. Ferramentas que automatizam parte da sistemática de elaboração do comportamento excepcional ajudariam a manter a viabilidade dos prazos de construção destes sistemas.

- **Auxiliar no cumprimento dos custos**: toda técnica de tolerância a falhas se baseia em replicação de elementos críticos do sistema. Dado ao investimento dobrado ou triplicado da compra de equipamentos e as vezes até mesmo replicação de projeto de software, o custo destes sistemas pode ser um fator complicador. O projeto de sistemas tolerantes a falhas deve ser um processo iterativo, envolvendo identificação de possíveis falhas que podem afetar o sistema e avaliando métodos alternativos para implementar tolerância a falhas. O objetivo de tais iterações é minimizar a redundância usada enquanto se maximiza a confiabilidade proporcionada, baseado nas restrições de custo do sistema [LeeAnderson90].
- **Minimizar o número de manutenções corretivas**: sabe-se que corrigir problemas depois do sistema pronto e em produção é muito mais difícil e caro do que prevenir ou mesmo corrigi-los ainda durante o desenvolvimento. O fato de dividir o sistema em liberações que são projetadas, implementadas, testadas, revisadas e integradas com as anteriores já é um grande passo no sentido de identificar problemas o quanto antes e corrigi-los enquanto o sistema ainda não é tão grande e complexo. Várias outras atividades auxiliam no cumprimento deste objetivo, tais como a definição completa dos requisitos considerando tanto o comportamento normal e excepcional do sistema, o projeto robusto e revisado da arquitetura do sistema como um todo e de cada uma de suas partes.
- **Favorecer o reuso:** quando se tem uma metodologia bem definida para construção de sistemas de software é possível controlar a complexidade dos sistemas através de reutilização sistemática de artefatos envolvidos no desenvolvimento do software como idéias, projetos, arquitetura, conceitos e, até mesmo, requisitos. A reutilização possibilita não apenas redução do custo e do prazo do desenvolvimento de sistemas. Além disto, possibilita a construção de sistemas mais robustos baseados em elementos previamente testados.

Componentes de software projetados para serem reutilizados de forma intensiva devem ser robustos, isto é, devem incorporar atividades de tratamento de erros de forma a se comportarem adequadamente mesmo na presença de falhas [Garcia+99].

- **Gerar produtos de fácil manutenção e expansão**: Uma característica essencial das metodologias de desenvolvimento de software atuais é a possibilidade de se gerar sistemas flexíveis em um curto espaço de tempo. Entende-se por sistema flexível, um sistema extensível e reutilizável [Charles99]. Um sistema é dito extensível se novas características podem ser acrescentadas a ele sem grandes dificuldades. Já um sistema reutilizável é tal que pode ser usado, inteiramente ou parte dele, em outros sistemas ou aplicações amenizando o velho problema de atualizações. Uma simples substituição de componentes individuais produzirá a evolução ou atualização desejada de forma muito mais rápida e eficaz. Para facilitar a manutenção e expansão, a metodologia de desenvolvimento deve auxiliar manter em separado as atividades normais e as excepcionais de um componente de software durante toda a sua confecção, isto é, desde a fase de especificação dos requisitos até a implementação.

Na seção seguinte daremos uma idéia geral da metodologia MDCE proposta que visa atender os objetivos do produto e da metodologia propriamente dita apresentados nesta seção.

3.2 Visão Geral da Metodologia MDCE

Existem duas maneiras para construir um sistema tolerante a falhas: (1) uma abordagem *top-down* (de cima para baixo) e (2) uma *botton-up* (de baixo para cima) [Avizienes97]. A abordagem *botton-up* se baseia em projetar componentes tolerantes a falhas e posteriormente integrá-los tomando o cuidado para que a interação entre eles ainda mantenha os requisitos de tolerância a falhas. Já a abordagem *top-down* permite que o sistema seja construído através de componentes já existentes que inclua ou não medidas de tolerância a falhas. Funções de monitoramento devem ser implementadas para proporcionar tolerância a falhas nestes sistemas.

Como guia para a construção de sistemas tolerantes a falhas usaremos o

trabalho apresentado por Avizienes [Avizienes97], que adota a abordagem *botton-up* de desenvolvimento. Avizienes apresenta uma visão geral de uma abordagem sistemática para incorporar técnicas de tolerância a falhas nos componentes de software e como integrar estes componentes durante o projeto. O projeto desta abordagem sistemática de Avizienes baseia-se na sobreposição de etapas na tentativa de minimizar erros, descuidos e inconsistências que possam ocorrer durante a implementação destes sistemas.

A metodologia MDCE, proposta neste trabalho, visa detalhar a construção *botton-up* de sistemas tolerantes a falhas ao nível de atividades e artefatos. Os conceitos de construção de sistemas tolerantes a falhas definidos em [Avizienes97] são alcançados através do emprego de técnicas de tratamento de exceções. Apesar das técnicas de tratamento de exceções estarem bem difundidas, ainda existem muitos problemas em aplicá-las na prática. Os projetistas não estão suficientemente preparados para definir e tratar o comportamento excepcional de um sistema de software. Na verdade, pouca informação está disponível para auxiliá-los no uso apropriado de exceções e seus tratadores [Martin+00]. Geralmente esta é uma questão considerada apenas na fase de projeto, até mesmo porque nem a própria UML oferece suporte completo para a representação de exceções e tratadores em seus modelos. O resultado disto é um projeto mal estruturado que gera estruturas de exceções complicadas e tratadores pouco adequados [Martin+00].

O objetivo da metodologia MDCE é apresentar guias e sugestões de como identificar, representar, projetar e implementar exceções e seus tratadores durante as etapas de construção de um sistema, isto é, durante análise, projeto e implementação. A metodologia MDCE mantém a representação do comportamento excepcional do sistema no modelo de casos de uso, no projeto da arquitetura e em modelos dinâmicos da UML como diagramas de estados, seqüência e colaboração. Além disto, a metodologia MDCE deve ser genérica o suficiente para ser aplicada a processos de desenvolvimento já existente na literatura.

Na seção seguinte apresentamos uma abordagem padrão para tratamento de exceções apresentado em [Dellarocas+98]. Faremos uso desta abordagem para compor a metodologia MDCE que será apresentada em detalhes na seção 3.2.2.

3.2.1 Abordagem Padrão para Tratamento de Exceções

Uma abordagem padrão para tratamento de exceções foi apresentado em [Dellarocas+98]. Esta abordagem consiste nas seguintes atividades:

1. **Antecipar e Preparar-se para Exceções:** O primeiro passo é tentar antecipar as possíveis situações excepcionais que podem levar o sistema a um fracasso. Uma vez antecipadas as exceções passa ser possível projetar o sistema de forma que estas exceções possam ser detectadas e evitadas.
2. **Estabelecer Métodos para Detectar e Prevenir Exceções:** Para cada tipo de exceção identificado, o projetista deve decidir como projetar o sistema para **detectar** estas exceções. Um sistema pode falhar de diversas formas, cada falha possui diferentes manifestações, incluindo perda de prazos, violação de restrições, extrapolação de recursos disponíveis, entre outras. Sendo assim, para cada tipo de exceção deve existir um método de identificação associado. Por exemplo, para exceções do tipo "item atrasado" ou "serviço não disponível" o método de identificação pode ser um mecanismo de *time-out*. Além de se projetar detectores de exceções, processos para **prevenir** manifestações excepcionais também podem ser estabelecidos.
3. **Diagnosticar as Exceções:** Várias situações excepcionais podem se manifestar de uma mesma forma, por exemplo, "item atrasado" ou "serviço não disponível" se manifestarão através de *time-out*. Sendo assim, antes de tratar uma exceção, é preciso dar o seu diagnóstico, isto é, identificar qual foi o motivo que levou ao *time-out* pois dependendo do motivo, o tratamento será diferenciado. Por exemplo, se a causa foi "item atrasado", o cliente pode requisitar ao servidor o reenvio do item. Por outro lado, se a causa foi "serviço não disponível" a tentativa de recuperação será requisitar o serviço de um servidor alternativo.
4. **Tratar as Exceções:** Considerando que uma exceção já fora identificada e diagnosticada, é o momento de especificar como a exceção deve ser tratada a fim de retornar o sistema para um estado estável.

3.2.2 As Fases da Metodologia MDCE

A abordagem para tratamento de exceções apresentado na seção anterior, quando aplicado a uma metodologia de desenvolvimento, será diluída em todas as suas etapas. Isto quer dizer que, assim como a construção da funcionalidade de um sistema se dá através de refinamentos de sua especificação em atividades de projeto e implementação, o mesmo ocorre com a especificação de sua atividade excepcional. As situações excepcionais são antecipadas, detectadas, diagnosticadas e tratadas durante a especificação de requisitos, projetadas durante a fase de projeto e posteriormente implementadas e testadas como mostra a Figura 5.

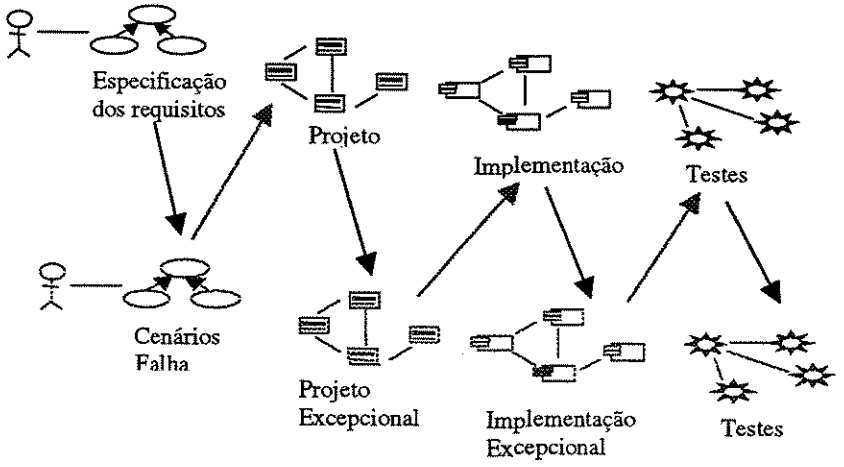

Figura 5 – Atividades de Processo de Desenvolvimento de Sistemas Confiáveis

Quando se usa um processo disciplinado, poucas falhas são introduzidas ou se mantém não identificadas até as etapas finais do ciclo de vida dos sistemas [Rushby93]. Além disto, podemos afirmar que, para falhas antecipadas desde a especificação dos requisitos do sistema, seu diagnóstico será bem mais preciso e o seu tratamento bem mais eficaz. Além disto, a antecipação destas situações excepcionais podem resultar em restrições para as etapas seguintes de desenvolvimento evitando retrocessos no processo de construção do sistema.

As seções seguintes apresentarão em detalhes cada um das fases de construção de sistemas computacionais confiáveis estendidas para auxiliar na identificação e representação do comportamento excepcional do sistema. As atividades referentes a especificação dos requisitos visam obter o enunciado completo, claro e preciso dos requisitos de um produto de software e estão apresentadas na seção 3.3. O conjunto de técnicas empregadas para levantar, detalhar, documentar e validar os requisitos de um produto forma a Engenharia de Requisitos. Projetos de produtos

mais complexos geralmente precisam de maior investimento em Engenharia de Requisitos que projetos de produtos mais simples. Uma boa Engenharia de Requisitos é um passo essencial para o desenvolvimento de um bom produto, em qualquer caso [Paula01]. Por estes motivos concentramos a maior parte de nossos esforços na elaboração de uma sistemática que conduza a uma boa Engenharia de Requisitos. Uma boa Engenharia de Requisitos é aquela que resulta em requisitos de alta qualidade, isto é, requisitos claros, completos, sem ambigüidade, implementáveis, consistentes e testáveis.

Já as seções 3.4 e 3.5 se dedicam às atividades de projeto que foram subdivididas em *projeto arquitetural* e *projeto interno* de cada componente. Na seção 3.6 tratamos assuntos relevantes à implementação dos sistemas confiáveis especificados e projetados até o momento.

3.3 Especificação dos Requisitos

Esta atividade visa identificar o contexto aonde o sistema será inserido, entender o problema, regras de negócio, estabelecer terminologias e definir requisitos como desempenho, confiabilidade, escalabilidade e objetivos de reuso. Além disto, é preciso levar em conta as restrições de arquitetura como equipamentos, sistema operacional, distribuição e middleware e restrições de planejamento e projeto como orçamento, pessoal e tarefas.

Este é o ponto de partida para a especificação de tolerância a falhas. Entretanto, é preciso levar em conta algumas questões que podem facilitar ou limitar a implantação da tolerância a falhas necessária [Avizienes97].

- **Requisitos de qualidade:** Determinar os requisitos de qualidade de cada módulo do sistema. Por exemplo, para alguns componentes do sistema o critério essencial é a disponibilidade, outros a confiabilidade, a segurança, o desempenho ou até mesmo a combinação de alguns destes requisitos de qualidade.
- **Suporte externo:** Determinar a disponibilidade de se reparar o sistema por agentes externos ou suporte remoto e ainda a forma como isto será feito (sob demanda ou periódico).
- **Alocação de recursos:** Deve-se estabelecer antecipadamente quais são os recursos disponíveis para implantação de tolerância a falha, pois questões financeiras e de hardware podem limitar a disponibilidade de recursos.

Durante a especificação dos requisitos, além de detalhar a funcionalidade do sistema, é importante considerar a impossibilidade de prover tal funcionalidade. O sistema torna-se incapaz de prover uma funcionalidade caso uma *situação excepcional* se manifeste por problemas internos ao sistema ou por falhas externas que afetam o sistema através de sua interface de entrada/saída, canais de comunicação, iteração homem-máquina ou comportamento inadequado de atores evolvidos com o sistema. É importante definir as *situações excepcionais* para que seja possível avaliar a qualidade do serviço fornecido.

Considere um sistema de arquivos distribuídos cuja funcionalidade básica é prover acesso a arquivos remotos. Não é suficiente descrever sua funcionalidade já que vários são os motivos que impedem o sistema de realizá-la, ou seja, de entregar o arquivo ao cliente. Entre os problemas que podem ocorrer, podemos citar: quebra do canal de comunicação, arquivo não encontrado, acesso não permitido, arquivo violado, entre outros. Para que o sistema de arquivos seja confiável é importante considerar as situações excepcionais e estabelecer formas de recuperação.

A forma como cada *situação excepcional* vai ser tratada depende de qual foi o motivo de sua manifestação. Sendo assim, torna-se extremamente importante ter uma sistemática para lidar com estas situações, pois se estas diferenças não forem consideradas, o tratamento nem sempre será adequado. Além disto, considerar estas diferenças desde o primeiro momento impede que diferentes situações excepcionais se manifestem de uma mesma forma dispensando a necessidade de **diagnosticar as exceções** como previsto em [Dellarocas+98] e apresentado na seção 3.2.1 deste trabalho.

O modelo de casos de uso proposto por [Jacobson92] é uma ferramenta amplamente empregada para especificação dos requisitos de um sistema de software. Na seção seguinte apresentaremos a importância de um modelo de casos de uso bem definido e completo que descreva tanto a funcionalidade que o sistema deve prover assim como as situações excepcionais que impediriam seu sucesso, além de descrever a forma como tais situações devam ser tratadas.

3.3.1 O modelo de casos de uso

O modelo de casos de uso especifica a funcionalidade oferecida pelo sistema

sob uma perspectiva do usuário, definindo o que faz parte do sistema e o que está fora dele. Este modelo usa **atores** para representar os diferentes papéis que um usuário desempenha junto ao sistema e **casos de uso** para representar como um usuário deve interagir com o sistema.

O comportamento de um caso de uso é descrito por uma seqüência de atividades chamada de *cenário*. Todas as atividades de cenário podem ser bem ou mal sucedidas. O *comportamento normal* de um caso de uso é sua execução livre de erros, isto é, o sucesso de todas as atividades do cenário. Já o *comportamento excepcional* de um caso de uso é representado por desvios do curso normal para tratamento de situações excepcionais ocorridas em atividades do cenário normal.

O modelo de casos de uso guia todo o projeto de um sistema, não servindo apenas para especificar suas funcionalidades [Jacobson92]. Baseado no modelo de casos de uso, os desenvolvedores projetam, implementam e testam todo o sistema. Desta forma, o modelo de casos de uso estabelece uma interdependência entre todos os modelos, mantendo a integridade do projeto e facilitando o entendimento e a manutenção do sistema. O *comportamento excepcional* do sistema, estando presente no modelo de casos de uso, permite verificar as conseqüências de falhas, em um caso de uso, no restante do sistema. Além disto, permite estabelecer uma análise do impacto das falhas em um nível de abstração mais alto. Uma vez que as falhas e seus impactos foram especificados, tem-se a base para uma especificação robusta, pois os mecanismos de proteção contra falhas e seus relacionamentos, podem ser incorporados desde as fases iniciais do desenvolvimento.

Sendo assim, além de descrever a funcionalidade do caso de uso, é necessário apurar todas as *situações excepcionais* que impediriam seu sucesso, além de descrever a forma como tais situações devam ser tratadas. As *situações excepcionais* e seus *tratadores* constituem o comportamento excepcional do caso de uso.

Alguns trabalhos começaram a despertar para a importância de se considerar situações excepcionais de caso de uso no momento em que está sendo especificado [Cockburn01] e [Schneider+00]. No entanto, quando o fazem, descrevem as situações excepcionais e a forma como tratá-las em conjunto com a funcionalidade do caso de uso, como extensão ao fluxo normal do caso de uso. Não separar o que é comportamento normal do que é excepcional tornam os casos de uso muito complexos e conseqüentemente difíceis de serem entendidos, estendidos e

mantidos.

O modelo de caso de uso que propomos inclui a descrição completa de um caso de uso, isto é, a definição de seu *comportamento normal* e *excepcional*. Além disto, mantém em separado os conceitos de normalidade e anormalidade, o que é de suma importância para manter a legibilidade do caso de uso e facilitar a sua descrição de forma completa como será mostrado a seguir.

O *comportamento normal* de um caso de uso possui um **cenário primário** e zero ou mais **cenários alternativos**. O *cenário primário* representa as atividades mais importantes do caso de uso, o curso mínimo para realização da funcionalidade. Variações do curso mínimo, como condições e iterações, são descritas como *cenários alternativos*.

Já o *comportamento excepcional* é descrito por cenários que constituem os desvios do comportamento normal do caso de uso quando *situações excepcionais* acontecem mantendo as medidas de recuperação do sistema. *Cenários excepcionais* podem ser de dois tipos: **cenários recuperáveis** e os **cenários de falha**. A diferença entre estes dois cenários está no tipo de falha que cada um trata. *Cenários recuperáveis* são desvios do curso normal para tratamento de atividades mal sucedidas, visando trazer o sistema de volta para um estado estável. Entretanto, algumas falhas representam situações inaceitáveis para o funcionamento do sistema por expor o sistema ou o ambiente aonde ele se encontra em perigo. Estas situações devem ser evitadas a todo custo, entretanto, quando ocorrem, é sinal que existem falhas de projeto, implementação ou interferência maléfica do meio externo sobre o sistema. Os *cenários de falhas* mantêm providências de segurança ativadas mediante uma situação inaceitável a fim de evitar uma catástrofe maior.

A Figura 6 mostra os cenários que constituem uma descrição completa do caso de uso. A execução do caso de uso começa no *cenário primário* e pode eventualmente ser desviada para *cenários alternativos*. Durante a execução de um cenário alternativo, novos cenários alternativos podem ser chamados, ou a execução pode voltar para o cenário primário. A execução do caso de uso pode terminar de forma normal tanto em um cenário primário quanto em um cenário alternativo.

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* conseguir tratar o erro com sucesso, os cenários normais voltarão a ser executados ou senão o caso de uso terminará de forma excepcional.

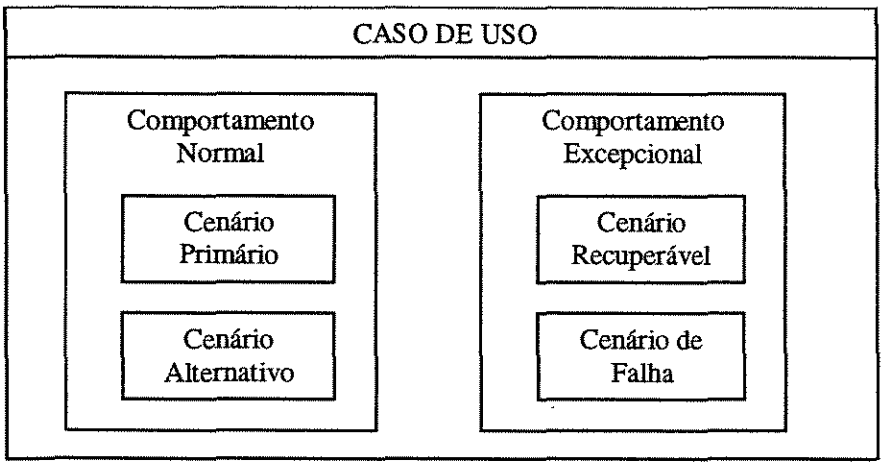

Figura 6 – Dinâmica de um caso de uso

No modelo de casos de uso deve existir uma clara separação entre seus comportamentos para facilitar a legibilidade, entendimento, manutenção e expansão do modelo. Veja na seção 5.2 uma extensão proposta para o modelo de casos de uso da UML onde o *comportamento excepcional* está representado por casos de uso excepcionais estereotipados por <<handler>> que estendem o caso de uso normal.

O maior esforço da atividade de especificação de requisitos deve ser construir um modelo de casos de uso que seja claro, completo e sem ambigüidades. De fato, esta é uma tarefa não trivial e exige uma sistemática para definição do *comportamento excepcional* de forma que a complexidade adicional se mantenha sob controle e o projetista não perca o controle da situação. Por este motivo, concentramos a maior parte dos nossos esforços deste trabalho na construção do modelo de casos de uso e propomos, na seção seguinte, uma metodologia sistemática para descrição e identificação destes casos de uso completos.

3.3.2 Metodologia para Construção dos Casos de Uso

A metodologia proposta para construção de casos de uso completos especifica

atividades separadas para a definição do *comportamento normal* e do *comportamento excepcional* de cada caso de uso. O fato de separar os conceitos de funcionalidade e dos requisitos de tolerância a falhas facilita a construção dos casos de usos permitindo uma definição mais completa e flexível do sistema. Além disto, possibilita um paralelismo destas atividades reduzindo o tempo gasto com a especificação dos requisitos do sistema. Veja Figura 7.

O primeiro passo para se especificar um sistema é a identificação de seus limites. Isto significa identificar o que está fora do sistema (atores) e o que está dentro (casos de uso), ou seja, o que será implementado pelo sistema. Tal informação será armazenada em um modelo de casos de uso que estará em constante atualização. Posteriormente os casos de uso são priorizados e detalhados. Detalhar um caso de uso significa descrever seus cenários primários, alternativos e excepcionais. O objetivo da metodologia que estamos propondo é auxiliar na identificação do *comportamento* de um caso de uso. Sendo assim, consideramos que os limites do sistema já foram previamente identificados e nos concentramos na atividade de detalhamento dos requisitos descrevendo os comportamentos normais e excepcionais de cada caso de uso. Para maiores esclarecimentos de como identificar estes limites consulte [Jacobson+99].

A metodologia aqui proposta consiste em três etapas para identificação completa e consistente dos comportamentos de um sistema além da modelagem e análise da interdependência entre atores e casos de uso e casos de uso entre si.

1. **Especificação do comportamento normal** do sistema através da descrição dos cenários primários e alternativos de cada caso de uso.
2. **Especificação do comportamento excepcional** do sistema, acrescido ao modelo de casos de uso previamente definido, através dos cenários excepcionais e de falha de cada caso de uso;
3. **Detalhamento do modelo de casos de uso** incluindo diagramas de colaboração, atividades e estados para o comportamento normal e excepcional.

A Figura 7 mostra as atividades da metodologia proposta. Cada atividade produz resultados com cenários e diagramas. Pode haver um paralelismo entre as atividades, o que ajuda a manter os prazos de especificação do sistema sob controle.

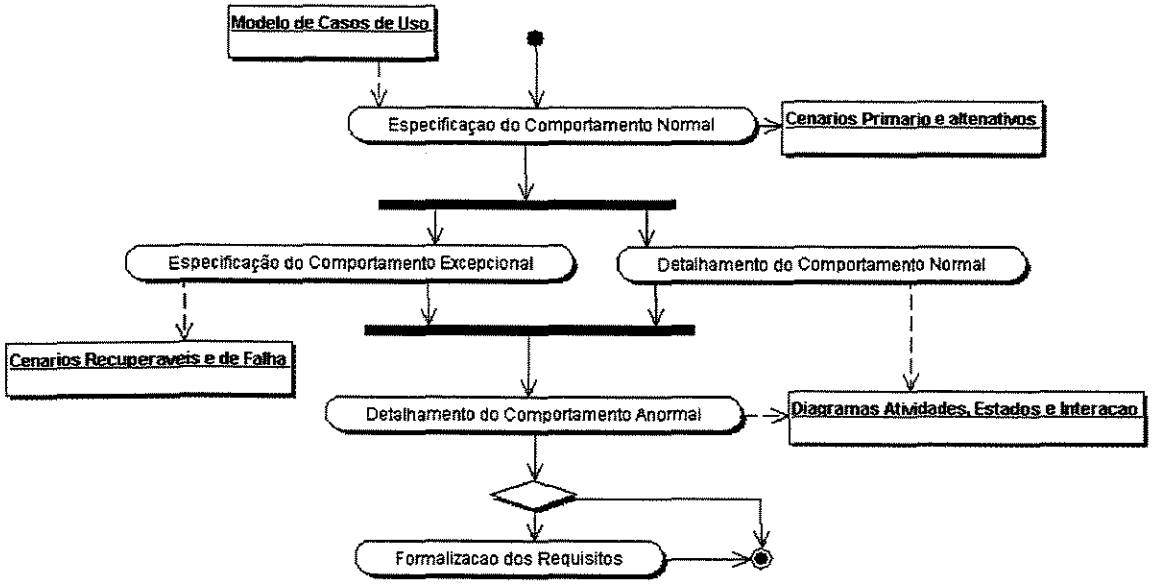

Figura 7 – Atividades de Detalhamento de Casos de Uso

As sub-seções 3.3.2.1, 3.3.2.2 e 3.3.2.3 apresentam em mais detalhes cada uma das etapas de especificação do comportamento normal, especificação do comportamento excepcional, detalhamento dos comportamentos respectivamente. A formalização dos requisitos está prevista como trabalho futuro.

3.3.2.1 Especificação do comportamento normal

O comportamento normal de um caso de uso é o comportamento esperado dado que falhas não ocorrerão. A especificação do comportamento normal de um caso de uso é a descrição em detalhes das atividades implementadas por ele. As transições entre as atividades são realizadas por disparos de eventos e podem seguir os mais variados cursos como ilustrado na Figura 8.

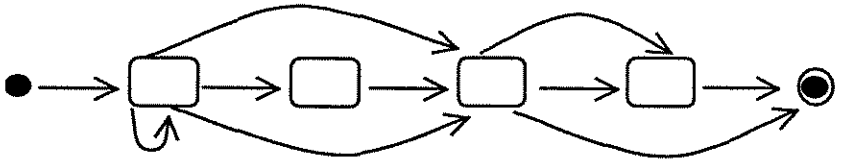

Figura 8 – Transições entre atividades de um caso de uso

Descrever o *comportamento normal* do caso de uso de forma completa consiste em descrever todas as seqüências possíveis de suas atividades. Uma boa prática para se fazer isto é fazer uma breve descrição da funcionalidade representada pelo caso de uso, escolher um fluxo completo como base e os outros caminhos como fluxos alternativos. Veja Figura 9. O fluxo base constitui o cenário

primário do caso de uso, isto é, contém o mínimo de ações necessárias para implementar a sua funcionalidade. Os fluxos alternativos constituem os cenários alternativos e são incluídos como extensões do cenário primário para agregar funcionalidade.

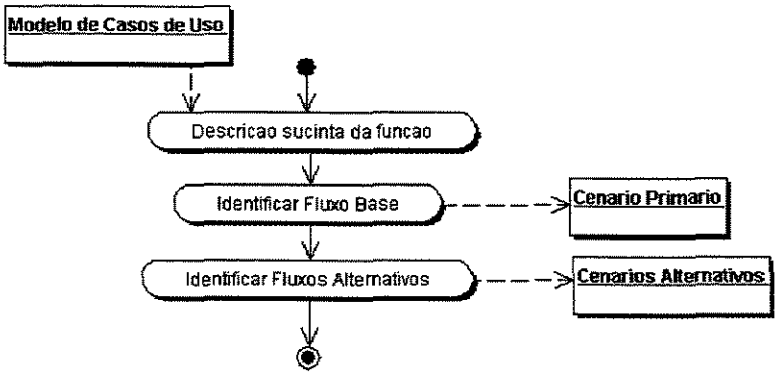

Figura 9 – Atividades para Especificação do Comportamento Normal do Sistema

Cenários alternativos ocorrem quando:

- Existe mais de uma opção para se realizar uma atividade, por exemplo, pagar uma mercadoria em dinheiro, cheque ou cartão.
- Mais de um ator está envolvido no caso de uso e a ação de um deles pode influenciar os caminhos seguidos pelos outros.
- Em qualquer momento um evento pode acontecer, como por exemplo, em qualquer momento o usuário pode cancelar uma transação.

Um caso de uso com o comportamento normal especificado deve ter pelo menos os campos apresentados no modelo expresso na Figura 10.

Nome do Caso de Uso
Descrição: Breve descrição do comportamento normal do caso de uso;
Participantes: Lista dos componentes participantes do caso de uso;
Pré-condições: Conjunto de condições que devem ser válidas quando o caso de uso iniciar;
Invariantes: Condições que devem ser mantidas durante todo o caso de uso;
Cenário primário: Conjunto mínimo de 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
2.1 Atividade executada após atividade 2

Pós-condições: Condições que devem ser válidas ao fim do caso de uso;

Figura 10 – Modelo de comportamento normal de um caso de uso

Os *cenários alternativos* devem estar associados aos passos do cenário primário onde o desvio acontece. A forma para definir esta associação fica a critério do projetista, no entanto, sugerimos fortemente que seja usada a numeração inteira para os passos do cenário primário e a numeração fracionada para passos dos cenários alternativos, como mostrado no modelo acima. Deixe clara qual foi a condição de desvio para o cenário alternativo e os passos que compõem o cenário alternativo.

Sistemas tolerantes a falhas necessitam de testes para identificar e remover falhas que possam comprometer os requisitos de confiabilidade e disponibilidade. Tolerância a falhas é requisito de qualidade dos sistemas, não faz parte da sua funcionalidade básica. Sendo assim, os testes para identificação das falhas que comprometem o sucesso do caso de uso devem ser modelados como cenários alternativos como extensão do cenário primário. Note que estamos falando em testes para identificação de falhas e não em tratamento da falha pois este fará parte dos cenários excepcionais que serão descritos a seguir.

Para que sistemas computacionais sejam capazes de realizar testes de identificação de falhas muitas vezes faz-se necessário replicar certos componentes do sistema ou até mesmo fazer uso de novos elementos de hardware ou software para que isto seja possível. Esta é uma questão importante no projeto destes sistemas, pois, os recursos financeiros e de hardware podem limitar a disponibilidade destes serviços [Avizienes97].

Uma vez tendo identificado e descrito o *comportamento normal* de um caso de uso, passemos para exploração de seu *comportamento excepcional.* Como identificar e especificar o comportamento excepcional de um caso de uso será apresentado na seção seguinte.

3.3.2.2 Especificação do comportamento excepcional

Para especificar o comportamento excepcional do sistema, os casos de usos são acrescidos de *cenários recuperáveis* e *cenários de falha.* Estes cenários, chamados de forma genérica de *cenários excepcionais*, incluem as medidas de recuperação do sistema dado a manifestação de falhas.

As atividades para especificação do comportamento excepcional de um caso de uso estão ilustradas na Figura 11. É importante explorar todas as condições de falha do caso de uso antes de pensar em como tratá-las. Definir como tratar uma situação excepcional, muitas vezes é uma tarefa exaustiva e pode consumir muito

tempo fazendo com que as atenções sejam desviadas do fluxo principal e impossibilitando de encontrar novas situações excepcionais. Por outro lado, se todas as condições de falhas estiverem listadas, isto servirá como um guia para os próximos passos e a especificação do *comportamento excepcional* do sistema certamente será feita de forma muito mais completa. A menos que o tratamento da falha seja simples o suficiente, ele poderá ser incluído no momento de sua identificação.

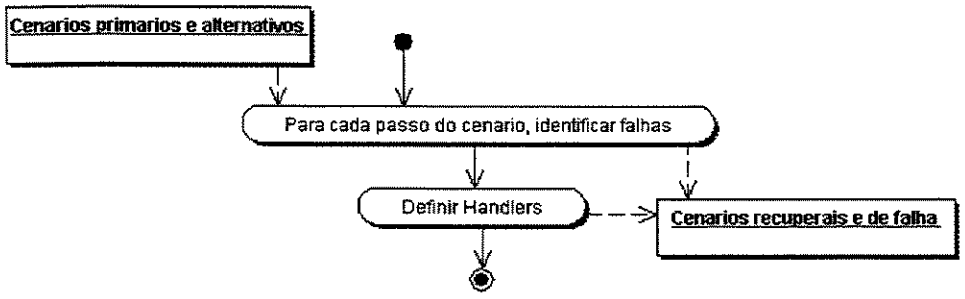

Figura 11– Atividades para Especificação do Comportamento Excepcional do Sistema

1 – Identificação de Situações Excepcionais

Situações excepcionais são derivadas de violação de contrato [Meyer97]. O contrato é representado pelas pré-condições, invariantes e pós-condições que representam os direitos e obrigações das partes envolvidas na negociação. Um contrato é sempre especificado entre um servidor e um cliente e, como estamos no contexto de casos de uso, clientes e servidores são *atores* ou *casos de uso*. Um caso de uso pode solicitar serviços de outros casos de uso ou mesmo de atores. Já atores solicitam serviços apenas a casos de uso.

A violação de pré-condição é a manifestação de problemas no cliente [Meyer97]

Existem várias maneiras de se violar uma pré-condição. A entrada do caso de uso pode estar incompleta ou inválida ou simplesmente inexistente para que sua pré-condição seja violada. De uma forma mais genérica, quebra de pré–condições são devido à:

- Comportamento inválido do cliente
- Inatividade do cliente
- Entrada inválida

A violação de pós-condição é a manifestação de problemas no servidor [Meyer97]

Já a violação de uma pós-condição é decorrente de problemas internos ao caso de uso que ocorreram impedindo-o de prover o serviço especificado. Quando uma pós-condição é violada, duas situações podem ocorrer:

- Falhas internas são detectadas e mascaradas pelo sistema;
- Falhas inesperadas são tratadas pelo sistema mas ocasionarão conseqüências externamente visíveis;

Vários motivos levam à violação de uma pós-condição:

- **Inatividade temporária**: A inatividade temporária de um serviço ou equipamento são falhas que podem ser detectas e mascaradas pelo sistema. Por exemplo, dentro do exemplo do caixa eletrônico, caso o mecanismo de liberação de dinheiro trave, o caso de uso de saque pode aciona o caso de uso que destrava o equipamento internamente sem que o cliente tenha consciência do acontecido.
- **Inatividade permanente**: Quando a inatividade é permanente (o equipamento quebrou, por exemplo) a falha pode ser mascarada apenas se existir um substituto para o elemento faltoso. Caso contrário a falha será repercutida externamente. Por exemplo, o canal de rede que liga o caixa eletrônico ao servidor central do banco caiu, caso exista um caminho alternativo, a falha pode ser mascarada, mas caso contrário, o sistema de caixa eletrônico se torna impossibilitado de acessar o servido, deixando o cliente ciente da situação.
- **Desempenho**: Quando o sistema apresenta problemas de desempenho impedindo que o serviço seja realizado a tempo, a pós-condição também não será cumprida. Dependendo da aplicação, problemas de desempenho são falhas que podem ou não ser mascaradas. Por exemplo, quando se trata de sistemas de tempo real, um pequeno atraso na realização do serviço pode ser desastroso e muitas vezes se torna impossível esconder do cliente a falha ocorrida.

Seguindo a terminologia do **componente ideal** (seção 2.1.2), violação de pré-condição leva à manifestação de **exceções de interface** e violação de pós condições leva à manifestação de **exceções internas** ou **exceções de defeito**, caso

a falha venha a ser mascarada ou não.

2 - Definição dos tratadores de exceções

Dado que as situações excepcionais foram identificadas na atividade anterior, agora é o momento de definir quais serão as medidas de recuperação do sistema. Neste ponto é importante considerar a disponibilidade de se reparar o sistema por agentes externos ou suporte remoto e ainda a forma como isto será feito (sob demanda ou periódico) [Avizienes97].

Uma medida de recuperação pode ser implementada por um simples passo na execução do caso de uso principal ou poderá constituir um novo caso de uso completo. Manter a medida de recuperação como parte do caso de uso facilita seu entendimento a menos que tal medida seja complexa o suficiente para extrapolar o limite de legibilidade de um caso de uso (no máximo duas páginas e três níveis de identação).

Um caso de uso que contém medidas de recuperação do sistema é modelado como extensão do caso de uso original representado pela associação <<extends>> (Veja na seção 5.2) e deve ser descrito de forma completa, considerado tanto o sucesso da recuperação quanto o fracasso [Cockburn01]. Apresentamos na Figura 12 um modelo para especificação completa dos cenários anormais dos casos de uso.

Cenários Anormais: **Recuperáveis ou de Falha**	Seqüência de atividades que implementam as medidas de recuperação em caso de falha do caso de uso
Sinal:	O estado de erro ou evento que identificou a situação excepcional;
Tratador:	Atividades de recuperação do sistema ou medidas de segurança;
Pós-condições:	Estado do sistema após execução do cenário anormal;

Figura 12 – Modelo para especificação do comportamento excepcional de casos de uso

Deve-se combinar situações de falhas independentes e avaliar qual seria o impacto no sistema e como este reagiria diante de manifestação quase simultânea de falhas com efeitos sobrepostos [Avizienes97]. Além disto, é necessário fazer análise do custo de implantação de tolerância a falhas se comparado com a disponibilidade e confiabilidade obtida depois de sua implantação [Avizienes97].

Dado que os comportamentos normais e excepcionais dos casos de uso foram especificados, este é o momento de refiná-los para tornar o modelo de casos de uso mais claro, além de remover possíveis ambigüidades e melhorar o entendimento dos requisitos do sistema.

3.3.2.3 Detalhamento do modelo de casos de uso

Nas etapas anteriores, os fluxos de requisitos foram descritos textualmente como cenários normais e excepcionais dos casos de uso. A terceira etapa consiste no detalhamento do modelo de casos de uso para melhorar o entendimento dos requisitos, a comunicação com o usuário final, as medidas de recuperação do sistema, além de possibilitar a identificação novas situações excepcionais. Este detalhamento é ortogonal à estrutura proporcionada pelas etapas anteriores e deve existir uma clara transição entre os dois modelos para que exista consistência na descrição e especificação dos requisitos.

Ao se refinar os requisitos do sistema, faz-se necessário representar os cenários graficamente através de diagramas. O uso de diagramas facilita o entendimento da dinâmica do caso de uso e é particularmente importante quando se considera o comportamento excepcional. Quando uma situação excepcional acontece, o fluxo normal de execução de um caso de uso é desviado para um fluxo excepcional aonde o problema será tratado. É importante que o projetista tenha consciência dos desvios causados pelas situações excepcionais e para isto, pelo menos dois caminhos devem ser considerados: o fluxo normal e o excepcional.

Os fluxos de um caso de uso podem ser descritos por qualquer diagrama da UML que represente os aspectos dinâmicos do sistema, isto é, diagrama de seqüência, colaboração, estados ou atividades. Cada diagrama possui uma particularidade que descreve melhor certas características da dinâmica do caso de uso. No Capítulo 5 os Diagramas de Atividades, Diagramas de Interação e Diagramas de Estados da UML foram usados para representar os desvios e estados inconsistentes causados pela manifestação de situações excepcionais.

Tendo em mãos uma especificação completa dos requisitos do sistema, agora ele será projetado e posteriormente implementado. Durante o projeto de um sistema confiável é necessário representar e identificar novas situações excepcionais a nível arquitetural, chamadas de exceções de configurações como veremos a seguir na seção 3.4. Também durante o projeto, o sistema é modelado em termos de classes com seus atributos, operações e associações. As exceções são representadas como classes estereotipadas por <<exception>> e organizadas hierarquicamente como veremos na seção 3.5.

3.4 Projeto arquitetural

O projeto arquitetural descreve como partes do sistema são conectadas para interagir e proporcionar a funcionalidade desejada. Consiste na definição da arquitetura do sistema em termos de componentes e conectores. Como foi dito na seção 2.2.3 - Arquitetura de Software, as propriedades arquiteturais são derivadas dos requisitos não-funcionais do sistema e influenciam, direcionam e restringem o ciclo de vida do software. A presença de uma determinada propriedade arquitetural pode ser obtida através da utilização de estilos arquiteturais que garantam a preservação dessa propriedade durante o desenvolvimento do sistema.

O projeto arquitetural é muito importante quando se deseja um projeto robusto e flexível de um sistema confiável. A escolha do estilo arquitetural baseado no modelo do componente ideal apresentado na seção 2.2.3 favorece a construção destes sistemas.

Além disto, considerar tratamento de exceções a nível arquitetural é de fundamental importância por três motivos:

1. **Mapeamento do fluxo excepcional:** O estilo arquitetural define a forma de interconexão entre componentes o que estabelece tanto o fluxo de execução normal quanto o excepcional. Sendo assim, através do projeto arquitetural, é possível avaliar, sob um alto nível de abstração, o caminho percorrido por uma exceção e os componentes arquiteturais por ela afetados.
2. **Localização dos tratadores:** Propagação automática de exceções não deve ser adotada em projeto de sistemas confiáveis pois torna a identificação do caminho percorrido pela exceção uma tarefa impraticável. Conseqüentemente os tratadores não são suficientemente hábeis para tratar a exceção ou ainda pode acontecer da exceção ser capturada por um tratador genérico que não possui informações suficientes para recuperar o sistema de forma efetiva. Exceções devem ser tratadas com muita cautela pois em sua presença o sistema se encontra em um estado inconsistente e a continuação normal do serviço nestas condições pode levar a novas ocorrências excepcionais ou até mesmo ao colapso do sistema [Cristian89]. O projeto arquitetural auxilia na localização dos tratadores uma vez que define o fluxo de exceções e torna possível identificar quais exceções atingem

determinados componentes. Para toda exceção que alcance um componente, direta ou indiretamente, definimos que é necessária a existência de um tratador para esta exceção. O fato de existir tratadores para todas as exceções, mesmo que eles sejam usados apenas para propagá-las, é útil para a identificação do caminho que uma exceção percorreu e conseqüentemente identificação do contexto de manifestação da falha e projeto de um tratamento efetivo.

3. **Tratamento de exceções de configuração:** Muitas vezes, uma exceção não é conseqüência de falhas em um serviço específico, e sim um problema de âmbito mais geral que diz respeito ao sistema como um todo. Por exemplo, considere um sistema de arquivos distribuídos onde um arquivo não pôde ser acessado em um servidor remoto por problemas de quebra no canal de comunicação. Toda exceção lançada por um conector ou componente deve ser propagada para o cliente da requisição. Entretanto, o cliente, nesta situação, não é capaz de lidar com a exceção levantada, pois se trata de um problema de reconfiguração do sistema para ativação de uma réplica de servidor que esteja disponível. A este tipo de exceção foi dado o nome de *exceções de configuração* [Valerie00] que devem ser tratadas em um nível de abstração mais alto, durante o projeto arquitetural.

O estilo arquitetural baseado no modelo do componente ideal foi sugerido como estilo base para os sistemas alvos deste trabalho. Sugerimos que as interações excepcionais sejam explicitamente definidas no projeto arquitetural possibilitando a visualização da propagação de exceções e colocação dos tratadores como exemplificado nas

Figura 13 e 14.

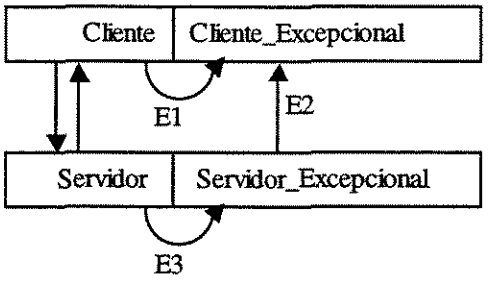

Figura 13 – Representação do fluxo excepcional no modelo arquitetural

O componente `Cliente` solicita serviços do componente `Servidor`. O `Servidor`, ao executar os serviços, poderá encontrar situações excepcionais e lançar a exceção interna `E3`. Sendo assim, o `Servidor` deverá manter um tratador para a exceção `E3` e caso este tratador não consiga recuperar o sistema, uma nova exceção `E2` é lançada externamente para o componente `Cliente`. O `Cliente`, por sua vez, que deve manter um tratador para a exceção `E2`, propagada pelo Servidor, e ainda para a sua exceção interna `E1`. Neste momento o modelo arquitetural pode ser refinado e conter não apenas o fluxo das exceções internas e externas como também os tratadores que cada componente deve manter para as exceções que lhe atingem. Veja a Figura 14.

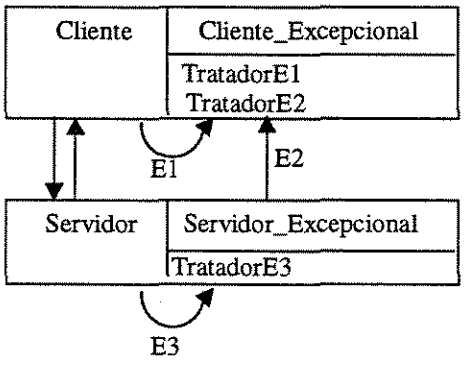

Figura 14 – Representação do fluxo excepcional no modelo arquitetural

Até o momento estamos tratando exceções derivadas de problemas na realização de um serviço específico, isto é, exceções internas a componentes ou derivadas de interações entre componentes, ambas tratadas internamente pelos componentes. O trabalho [Valerie00] apresenta a introdução de tratamento de exceções a nível arquitetural com o enfoque em *exceções de configuração* de forma complementar ao tratamento de exceções realizado internamente a componentes. Como este tipo de exceção é tratado a nível arquitetural, o seu tratador também deve ser especificado durante o projeto arquitetural.

A linguagem ADL foi estendida em [Valerie00] de forma a acrescentar a especificação das *exceções de configuração* e seus tratadores além da especificação de componentes, conectores, seus relacionamentos normais e excepcionais. Usando o exemplo apresentado na Figura 14 e considerando que a interação entre `Cliente` e `Servidor` se dá através de um conector com protocolo de comunicação RPC apresentamos a seguir uma idéia geral da

descrição da arquitetura de software considerando tratamento de exceções.

```
COMPONENT Cliente:
   /* operações fornecidas pelo Cliente */
     PROVIDES m1() RAISES E1, failure;
   /* operações requeridas pelo Cliente */
   REQUIRES m2(...) RAISES E2, E3, failure;
...
COMPONENT Servidor:
   /* operações fornecidas pelo Servidor */
   PROVIDES m2(...) RAISES E2, E3, failure;
...
CONNECTOR Rpc:
   PORT Clt;    PORT Srv;
CONFIGURATION Sistema:
    COMPONENTS:
          /* o sistema é composto por várias */
          /* instâncias de clientes e servidores */
          instClt[]: Cliente
          instSrv[]: Servidor
   CONNECTOR:
          /* O sistema dispõe de um número em */
          /* aberto de conectores RPC de */
          /* acordo com o número de instâncias de */
          /* componentes disponíveis */
          instRpc[]: UNSHARED Rpc;
   BINDINGS:
          /* Associação entre serviços requeridos */
          /* e fornecidos através de uma instância */
          /* do conector Rpc */
          instClt().m2 AS Rpc.Clt TO Rpc.Srv USING instRpc();
   EXCEPTION HANDLING:
          CONFIGURATION EXCEPTION: /* definição */
          HANDLERS: /* definição */
```

Fora a parte de definição das exceções, a declaração acima já existe em linguagens de descrição de arquiteturas (ADLs). As exceções esperadas por componentes e conectores estão declaradas dentro da especificação deles próprios. *Failure* é uma exceção lançada e tratada na presença de uma exceção não esperada. Uma arquitetura está associada com um certo número de *exceções de configuração* e cada exceção está sintaticamente associada a um tratador que possui um conjunto de alterações de configuração a serem feitas diante da

ocorrência da exceção.

Descrever uma exceção de configuração significa definir:

- O nome da exceção e seus parâmetros;
- A SUBCONFIGURAÇÃO: uma cláusula que define as instâncias de componentes e conectores cuja interação resultou na manifestação da exceção;
- A INTERAÇÃO: cláusula que fornece o conjunto de interações entre instâncias de componentes e conectores da subconfiguração cujo comportamento ocasionou a manifestação da exceção.
- A OCORRÊNCIA: cláusula que fornece a condição associada com a ocorrência da exceção que diz respeito ao conjunto de interações definido acima.

Dado a especificação da *exceção de configuração*, o tratador associado irá realizar alterações de configuração em tempo de execução. A reconfiguração se baseia em refinamentos dos elementos da arquitetura que fazem parte da subconfiguração. De forma mais específica, a declaração de um tratador se baseia na definição de: novas instâncias de componentes e conectores, refinamento de elementos da subconfiguração detalhando a reconfiguração a ser feita.

3.5 Projeto Interno dos Componentes

Os componentes arquiteturais apresentados durante o projeto arquitetural serão agora projetados e construídos até o nível de interfaces e classes ou componentes preexistentes. O objetivo desta atividade é definir uma estrutura interna e interações que satisfaçam os requisitos funcionais e não-funcionais de um componente. Para componentes complexos, é necessário um particionamento em subcomponentes, que devem ser especificados e implementados recursivamente.

Nesta fase é o momento de definir o modelo das classes que compõem o componente. Cada componente é projetado internamente por pelo menos duas hierarquias de classes: a hierarquia normal e a hierarquia excepcional. A hierarquia normal implementa a funcionalidade do componente e mantém explícito no modelo as exceções lançadas pelas classes desta hierarquia. A hierarquia excepcional correspondente deve manter tratadores para todas as exceções que a hierarquia normal lançar e ainda deve deixar explícitas as exceções por ela lançadas. O fato de organizar as classes excepcionais hierarquicamente possibilita

reuso da parte excepcional do componente pois os tratadores especificados na superclasse não precisam ser redefinidos na subclasse. Veja um exemplo na Figura 15.

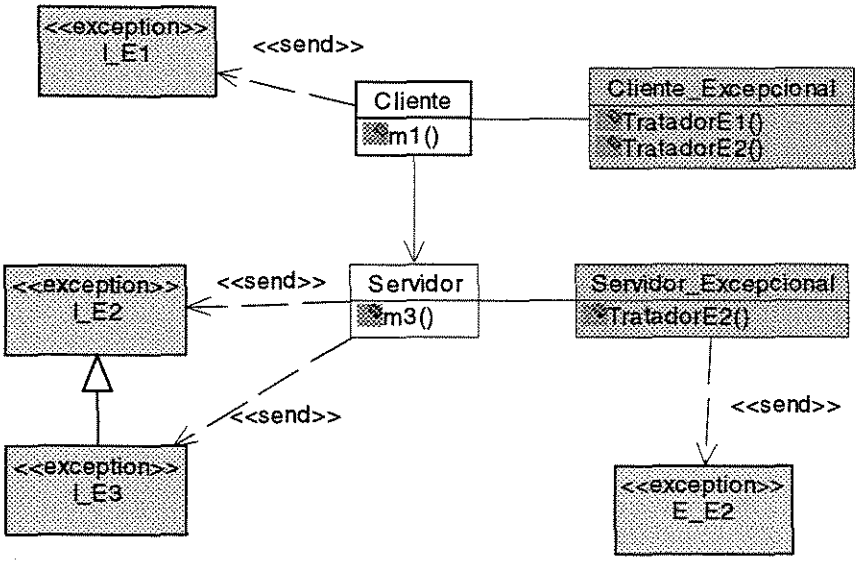

Figura 15 – Projeto interno de um componente

Vale observar que exceções lançadas pela hierarquia normal são tratadas internamente ao componente, isto é, são tratadas pela hierarquia excepcional. Isto é, exceções lançadas pela hierarquia normal são *exceções internas*. Já as exceções da hierarquia excepcional são *exceções externas* lançadas quando a parte excepcional do componente não foi capaz de tratar o problema. Para facilitar a especificação de exceções, definimos uma notação para nomenclatura de exceções especificando seu tipo (interna ou externa) e seu nome. O tipo pode ser {`I`, `E`}, onde "`I`" representa exceção interna e "`E`" representa exceção externa. O modelo é o seguinte:

```
Tipo_NomeExceção
```

A confecção do projeto interno como proposto acima e exemplificado na Figura 15 é demasiadamente trabalhosa. Enquanto tradicionalmente o projetista escreveria apenas a hierarquia normal, formada por apenas duas classes, de acordo com o modelo proposto pela MDCE, o projetista precisaria escrever ainda a hierarquia excepcional (duas classes) e a hierarquia das exceções (cinco classes), totalizando sete classes adicionais. Para manter a viabilidade deste modelo,

criamos um script que automatiza a construção destas classes adicionais através da ferramenta de modelagem Rose. O projeto e implementação deste script está apresentado no Apêndice A.

Localização dos Tratadores

Muitas vezes, exceções devem ser tratadas de forma diferente dependo do contexto de sua manifestação. No modelo proposto, exceções devem representadas como classes hierarquicamente organizadas o que permite que tratadores possam lidar tanto com exceções genéricas quanto com exceções mais especializadas, mais uma vez proporcionando reuso da parte excepcional. Além disto, exceções podem estar associadas a um componente, a uma classe, a um método, a um objeto ou a uma exceção de acordo com a necessidade [Garcia+99].

Na Figura 15 temos exceções associadas a classe `Servidor` organizadas hierarquicamente onde `I_E2` é supertipo de `I_E3`. A classe excepcional associada a `Servidor`, o `Servidor_Excepcional` mantém um tratador para a exceção `I_E2` que conseqüentemente pode tratar a exceção `I_E3`. Quando as exceções `I_E2` ou `I_E3` não forem tratadas localmente, uma exceção externa `E_E2` será lançada pela classe `Servidor_Excepcional` para o nível superior da cadeia de chamadas.

Quando dizemos que uma exceção está associada a objetos, os tratadores para esta exceção devem ser diferentes pois a recuperação do sistema depende do objeto que lançou a exceção. Considere o exemplo da Figura 16. Os objetos `A`, `B` e `C`, instâncias da classe `Servidor`, lidam de forma diferente com as exceções lançadas por esta classe. Para isto, tratadores distintos foram definidos em classes excepcionais separadas, cada qual associada a um objeto: `A_Excepcional`, `B_Excepcional` e `C_Excepcional`.

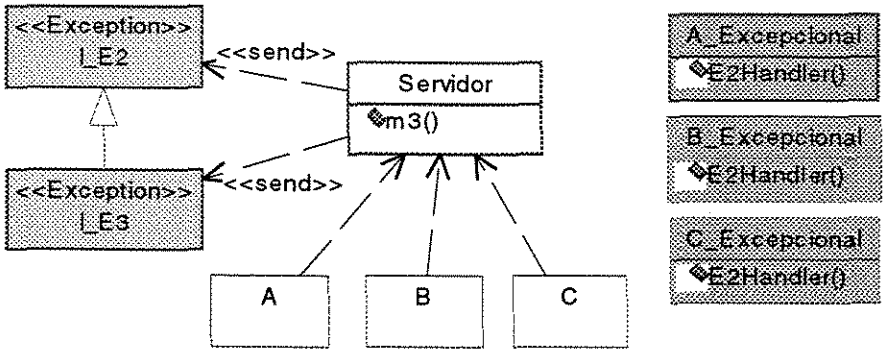

Figura 16 – Exceções associadas a objetos

Depois que o sistema foi projetado ao nível de classes, atributos, associações, exceções e tratadores, o próximo passo é transportar este modelo para uma linguagem de programação. Algumas questões relativas à implementação de sistemas confiáveis estão discutidas na seção seguinte.

3.6 Implementação

Nesta etapa o projeto é descrito em forma de código fonte em uma linguagem de programação. A condição mínima necessária para a escolha de uma linguagem de programação é a presença de um mecanismo de tratamento de exceções para sinalização de exceções e localização do tratador adequado. A separação entre o comportamento normal e excepcional, que vem sendo mantida desde o inicio do projeto, preferencialmente também deve permanecer durante a implementação.

Para implementar o projeto exatamente como definido na Figura 15 propomos duas soluções:

1. Em [Garcia+99] os autores apresentaram a especificação e implementação de um mecanismo de tratamento de exceções que oferece suporte a separação explicita entre a atividade normal e seus tratadores. O mecanismo foi implementado em Java e utiliza recursos de um protocolo de meta-objetos (MOP) chamado Guaraná [Oliva+98]. Neste ambiente, os componentes da aplicação são implementados no nível base enquanto os meta-objetos implementam as responsabilidades do mecanismo de exceções. Quando uma classe normal de um componente sinaliza uma exceção, ela é interceptada pelo MOP e os meta-objetos encontrarão o tratador adequado para esta exceção na classe excepcional deste componente. As classes excepcionais são

hierarquicamente organizadas de forma ortogonal à hierarquia de classes normais permitindo que subclasses herdem tratadores de suas superclasses e conseqüentemente permitindo reuso de código excepcional.

2. Se o mecanismo proposto por [Garcia+99] não estiver disponível, ainda assim é possível fazer uso de tradicionais mecanismos de exceções presentes em linguagens de programação orientadas a objetos. A solução proposta para manter a separação entre as partes normais e anormais está mostrada na Figura 17. Considere que a linguagem de programação escolhida foi Java, o bloco try-catch não mantém o tratador em si, mas sim uma chamada para ele que está implementado como um método da classe excepcional associada.

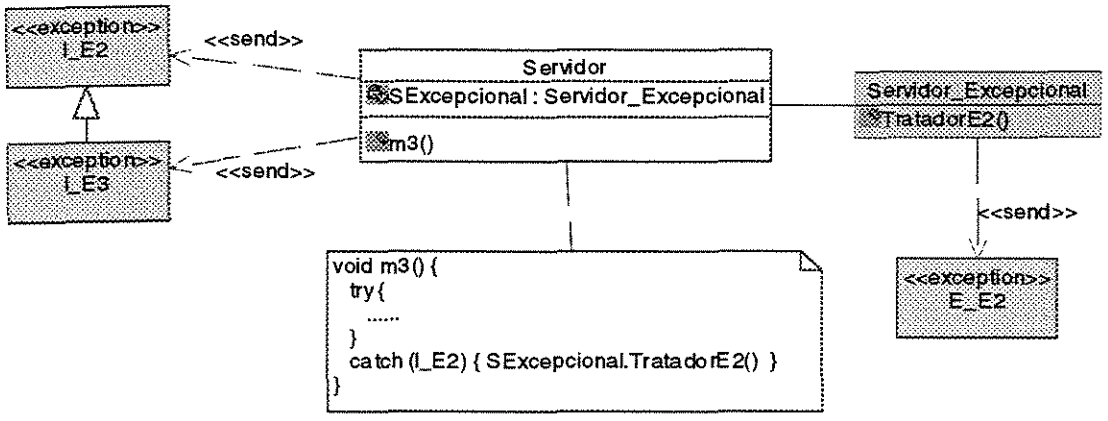

Figura 17 – Implementação da atividade normal e excepcional de forma disjunta

No entanto, apenas manter a separação entre as partes normais e excepcionais de um componente não é suficiente para se obter uma implementação robusta. É imprescindível que as exceções lançadas por um método estejam explicitamente representadas em sua interface pública, além de não considerar propagação automática de exceções. Na seção seguinte discutiremos a flexibilização proporcionada pela linguagem de programação Java quanto à definição de exceções na interface dos métodos e a existência de tratadores para todas exceções.

3.6.1.1 Exceções em Java

Uma questão de fundamental importância para garantir a construção de

sistemas robustos, a ser considerada durante a implementação em Java, é o tipo da exceção a ser lançada. A linguagem de programação Java possui dois tipos de exceções: `Exception` e `RuntimeException`, aonde a primeira é super tipo da segunda.

Em Java, quando uma exceção do tipo `Exception` é lançada ela pode ser tratada localmente ou propagada para o cliente da requisição. Se a exceção for propagada, obrigatoriamente ela deve estar declarada na interface pública do método que a lança, como mostrado abaixo. Clientes deste método podem tanto manter tratadores para esta exceção como também declará-la em sua interface deixando claro que poderão propagá-la.

```
public void writeList() throws IOException {...}
```

Já a classe `RuntimeException` é um tipo especializado de `Exception` que representa exceções ocorridas internamente na máquina virtual Java em tempo de execução. Um exemplo de uma `RuntimeException` é a exceção `NullPointerException`, que ocorre quando um método tenta acessar um objeto através de uma referência nula. A biblioteca de Java define várias classes derivadas de `RuntimeException` que podem ser tratadas exatamente como qualquer outra exceção. A diferença é que o compilador não exige a presença de um tratador para exceções do tipo `RuntimeException` e nem mesmo exige que estas exceções estejam explicitas na interface do método.

Em Java um tratador é representado pelo bloco Try-Catch como já mostrado na Figura 17. Sendo assim, em outras palavras, quando se empregam exceções do tipo `RuntimeException` o programador não é obrigado a definir blocos Try-Catch e nem a declarar a exceção em sua interface. Este procedimento, apesar de ser menos trabalhoso, proporciona um código pouco robusto pois o contrato não fica explícito no código.

Um componente deve deixar explícitas todas as exceções por ele lançadas, direta ou indiretamente, para que o cliente decida o que fazer com estas exceções. Se a exceção não está explícita na interface do método e o compilador também não acusa sua existência, o cliente nunca saberá que deve manter um tratador para esta exceção e conseqüentemente ela será propagada a procura de um tratador que nunca será encontrado.

Por outro lado, muitas vezes encarece testar toda chamada a um método muito

acessado durante a execução do programa, como por exemplo, testar se uma exceção do tipo `NullPointerException` acontece todas as vezes que acessar uma posição do vetor. É por este motivo que existem exceções do tipo `RuntimeException`. Entretanto, quando se usar este tipo de exceção, é necessário considerar algumas regras:

- não use exceções do tipo `RuntimeException` apenas para não se aborrecer com declarações da exceção e definição de blocos try-catch.
- use `RuntimeException` apenas quando tiver certeza de que seu cliente possui informações suficientes sobre as restrições do contrato de forma que elas não precisem ser verificadas pelo compilador. Por exemplo, é viável que uma exceção `ArrayIndexOutOfBoundsException` seja `RuntimeException`. Métodos de acesso a elementos de vetor são muito freqüentes e testar os limites do vetor todas as vezes que estes acessos fossem feitos seria muito caro. Além disto, programadores possuem familiaridade suficiente com vetores e certamente conhecem como eles funcionam e quais são suas restrições de operação.
- dentro do componente que você está projetando, considera-se que você tenha total controle das exceções. Neste caso, consideramos o uso de `RuntimeException` possível em métodos que não façam parte da interface pública do componente, ou seja, em métodos que vão ser usados por clientes deste componente. Em outras palavras, use `RuntimeException` para exceções internas, e `Exception` para exceções externas.
- `RuntimeException` podem ainda ser usada quando se desejar sinalizar erros de implementação que certamente serão identificados em fase de testes. Como é muito difícil ter esta certeza, este uso de `RuntimeException` deve ser evitado.

Mas, para usar `RuntimeException` é necessário ter disciplina. Mesmo que o compilador não exija sua presença na interface do método, é importante explicitá-la para deixar o cliente ciente do que pode ser retornado como resposta da sua requisição. Para uma discussão mais profunda sobre este assunto, direcionamos o leitor para [Ferreira+01a].

3.7 Resumo

Este capítulo apresentou os objetivos e atividades da metodologia proposta para descrição do comportamento excepcional de sistemas chamada de MDCE. O objetivo desta metodologia é auxiliar a construção de sistemas confiáveis mantendo os custos e prazos sob controle apesar da complexidade adicional ao especificar, projetar e implementar não apenas o comportamento normal do sistema, mas também seu comportamento mediante as situações excepcionais.

MDCE é metodologia genérica que contém apenas atividades estritamente necessárias para especificar e construir um sistema, isto é, as atividades de especificação de requisitos, projeto arquitetural, projeto detalhado e implementação.

Durante a etapa de **especificação dos requisitos**, os requisitos são modelados sob a forma de casos de uso que possuem a descrição de seu comportamento normal associada, mas disjunta, da descrição de seu comportamento excepcional. Estes casos de uso são refinados e representados sob a forma de diagramas dinâmicos da UML visando esclarecer a colaboração entre atores e casos de uso e aumentar o entendimento sobre os requisitos e funcionalidade do sistema a ser construído. O comportamento excepcional descrito ao nível de casos de uso propicia uma análise, sob um alto nível de abstração, do impacto das falhas sobre o projeto do sistema e também sobre o sistema propriamente dito. O modelo de casos de uso guiará todo o ciclo de vida do sistema servindo de base para as atividades seguintes de projeto e implementação e testes.

Durante as atividades de **projeto**, destacamos a importância em se representar exceções e tratadores no projeto arquitetural do sistema. Além disto, apresentamos como representar exceções e seus tratadores em modelo de classes robusto onde existe uma hierarquia normal de classes, que implementam a funcionalidade normal do componente, associada a uma hierarquia excepcional de classes que matem os tratadores para as exceções lançadas pela hierarquia normal. Além disto, exceções internas e externas estão representadas explicitamente no modelo como classes estereotipadas por <<exception>> e associadas à classe que a lança.

Para a atividade de **implementação**, apresentamos duas sugestões de como transpor o modelo de classes para uma linguagem de programação e ainda manter a separação entre as atividades normais e excepcionais do componente. Destacamos ainda a importância de deixar explícito no código o contrato

estabelecido entre cliente e servidor, incluindo as respostas excepcionais que podem ser retornadas mediante falhas na realização do serviço solicitado.

A metodologia MDCE é genérica o suficiente para ser aplicada aos processos de desenvolvimento presentes na literatura pois mantém apenas as atividades básicas para construção de sistemas computacionais. No capítulo seguinte realizaremos sua aplicação ao Processo Catalysis [Desmond98].

Capítulo 4

Metodologia MDCE Aplicada ao Catalysis

Neste capítulo, faremos a aplicação da metodologia MDCE apresentada no Capítulo 3 ao processo Catalysis [Desmond98]. O que diferencia Catalysis dos outros processos, e o motivo que nos atraiu para ele, são suas *colaborações*. Catalysis considera que as decisões sobre a interação entre os objetos são a chave para um projeto bem feito considerando as colaborações entre componentes como unidades básicas de projeto. No contexto de desenvolvimento de sistemas tolerantes a falhas, as colaborações são particularmente úteis, pois consideramos que um componente sozinho não é capaz de prover os recursos necessários para tratamento de uma falha de forma efetiva. Portanto, é necessário levar em conta o as interações entre os elementos do sistema.

Na seção 4.1 daremos uma visão geral do Processo Catalysis, cobrindo seus conceitos e atividades. Nas seções seguintes detalharemos como cada uma das atividades do Catalysis foi estendida para incorporar a representação do comportamento excepcional do sistema a ser construído.

4.1 Visão Geral do Processo Catalysis

Catalysis é um processo de desenvolvimento de sistemas orientados a objetos e baseados em componentes que se baseia em três conceitos de modelagem: *Colaborações*, *Tipos* e *Refinamentos*

- *Colaborações* são utilizadas para modelar interações entre grupo de objetos;
- *Tipos* definem o comportamento externo e visível de um objeto, abstraindo os detalhes de sua representação interna;

- *Refinamentos* são relações entre duas descrições de uma mesma coisa em dois níveis de abstração diferentes.

De forma similar a outras metodologias orientadas a objetos, Catalysis define cinco fases de desenvolvimento: (1) o domínio do problema ou especificação de requisitos, (2) a especificação do comportamento externamente visível do sistema e (3) o projeto interno dos componentes, (4) implementação, (5) testes. A

Figura 18 mostra as fases principais de construção de um sistema, apenas implementação e testes não estão presentes.

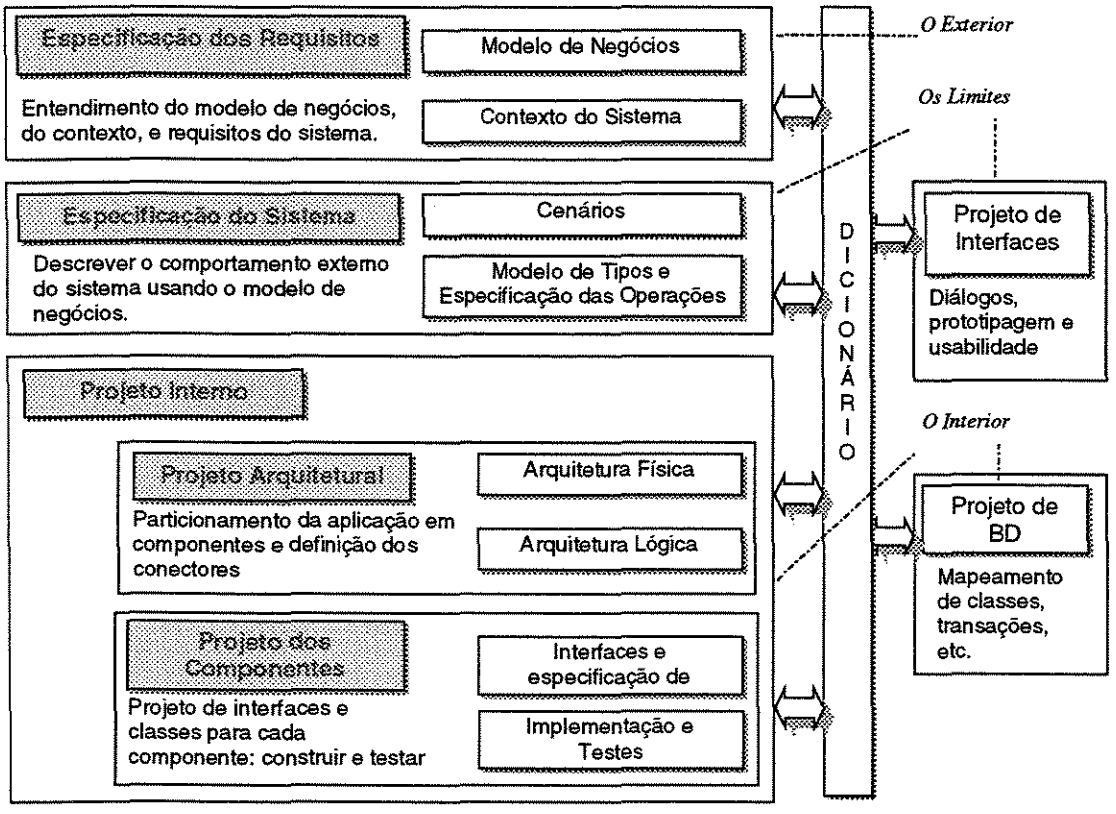

Figura 18 – Principais Atividades do Processo Catalysis

Em linhas gerais, durante a especificação dos requisitos do sistema definimos *seu exterior* onde o objetivo é estabelecer uma terminologia, entender o modelo de negócio, regras, colaborações e também o ambiente aonde o sistema será inserido. Já durante a especificação do sistema, *seus limites* serão definidos, isto é, seu comportamento externamente visível, suas responsabilidades e interfaces. Por fim, durante o projeto interno, abre-se a caixa preta e *o interior* do sistema é projetado

descrevendo como as pequenas partes que compõem o sistema são estruturadas e interagem para prover a funcionalidade especificada. A definição de um sistema é recursiva e, conseqüentemente, o processo de modelagem e projeto também é recursivo até que se depare com um sistema ou componentes implementáveis ou já disponíveis.

As seções seguintes detalham cada uma das etapas do processo Catalysis e a forma como cada uma delas foi estendida para incorporar a representação do comportamento excepcional do sistema a ser construído. A seção 4.2 define *o exterior*, a seção 4.3 *os limites* e, por fim, a seção 4.4 define *o interior* do sistema.

4.2 Especificação dos Requisitos: O Exterior

O maior objetivo da atividade de especificação dos requisitos é a definição do *modelo de negócios* e requisitos do sistema a ser construído. *Modelo de negócios* é o modelo de objetos do mundo real e suas interações. O termo negócio cobre quaisquer conceitos que sejam de relevância para os clientes e não necessariamente negócio no sentido de empreendimento comercial. É possível que haja várias visões de um negócio, pois a preocupação do diretor de marketing e do gerente de pessoal conterá partes em comum embora devam apresentar muitas divergências.

Catalysis especifica algumas técnicas e construções para definição do modelo de negócios e requisitos. Entre estas técnicas encontra-se o modelo de casos de uso chamado de *colaboração do negócio*[12] por este processo. Desta forma, a especificação dos requisitos de um sistema, segundo o processo Catalysis, se dá de forma semelhante ao que foi especificado pela metodologia MDCE e apresentado na seção 3.3 - Especificação dos Requisitos. Sendo assim, esta atividade pode ser realizada exatamente da forma como fora especificado anteriormente para a metodologia MDCE, não necessitando ser redefinida para este processo.

4.3 Especificação do Sistema: Os Limites

Esta etapa reúne as atividades que visam definir o comportamento externo do componente ou sistema que fora especificado na etapa de requisitos. Esta é uma atividade importante quando se considera um projeto baseado em componentes, pois quando se lida com componentes não se tem acesso a sua estrutura interna e,

portanto seu comportamento externo deve estar muito bem definido e separado de sua implementação interna.

Catalysis utiliza a noção de *tipo* para descrever o comportamento visível nos limites entre o sistema e seu ambiente. Um *tipo* é uma representação genérica de um conjunto de **objetos** que possuem um comportamento semelhante diante de determinadas ações. Especificar um *tipo* envolve identificar seus atributos e a lista de **ações** das quais ele participa, definir como cada ação o modifica internamente e a forma como ele responde a estas ações.

Os termos *objeto* e *ação* são bem amplos. *Objeto* pode ser tanto um objeto de linguagem de programação quanto um componente ou programa ou uma rede de computadores, hardware, pessoa, organização, ou seja, qualquer coisa que apresente um comportamento encapsulado e bem definido com o mundo a sua volta. *Ação* não é apenas uma mensagem ou chamada a procedimentos, mas um diálogo completo entre objetos. Pode-se lidar apenas com os efeitos de uma ação sem necessariamente especificar exatamente quem a iniciou ou como sua execução é realizada em detalhes.

Cada *ação* é representada como uma *operação* do *tipo* e cada operação está sujeita a condições que limitam sua aplicação a uma situação particular. Estas limitações devem ser expressas sob a forma de *pré* e *pós-condições* que podem ser descritas tanto em uma linguagem informal quanto com expressões formais. Uma pré-condição define as condições para que a ação possa ser executada enquanto as pós-condições são usadas para documentar as mudanças de estado e saídas garantidas pela correta realização da operação.

A figura abaixo exemplifica uma definição de *tipo* onde as operações foram especificadas sob a forma de pré e pós condições descritas formalmente em uma linguagem baseada em OCL (Object Constraint Language).

---

[12] do inglês *Business Collaboration*

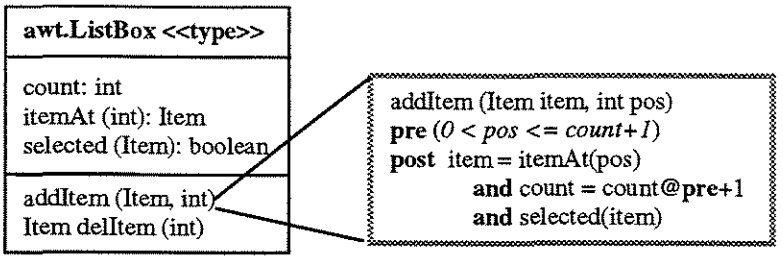

Figura 19 – Um Modelo de Tipo

Quando se define o comportamento de um sistema através de tipos e operações, como definido pelo processo Catalysis, trata-se apenas da especificação do comportamento normal ou esperado do sistema. O objetivo desta atividade deveria ser especificar o comportamento externo do sistema de forma completa. Isto é, especificar também as respostas excepcionais devolvidas quando a interação não for bem sucedida, pois ambiente externo deve estar preparado para desvios de comportamento do sistema causados por situações excepcionais.

Desta forma, propomos que as operações de um *Tipo* mantenham as exceções lançadas explicitamente em sua interface. Além disto, baseado no modelo do componente ideal apresentado na seção 2.1.2, um *Tipo* deve conter, além de uma estrutura interna que define seu comportamento normal, uma outra estrutura que defina seu comportamento excepcional. Isto é, associado ao *Tipo* deve existir uma outra estrutura chamada de *Tipo_Excepcional* que mantenha as medidas de recuperação do *Tipo* dado que situações excepcionais se manifestem internamente ou através de interações mal sucedidas com o ambiente externo. Manter em separado as atividades normais e excepcionais do *Tipo* estabelece o mapeamento direto entre a estrutura dos casos de uso. Esta separação facilita a manutenção da consistência entre os modelos do projeto.

O tipo que anteriormente especificava apenas o comportamento normal do sistema agora mantém também as respostas excepcionais dada a interações mal sucedidas como exemplificado na Figura 20.

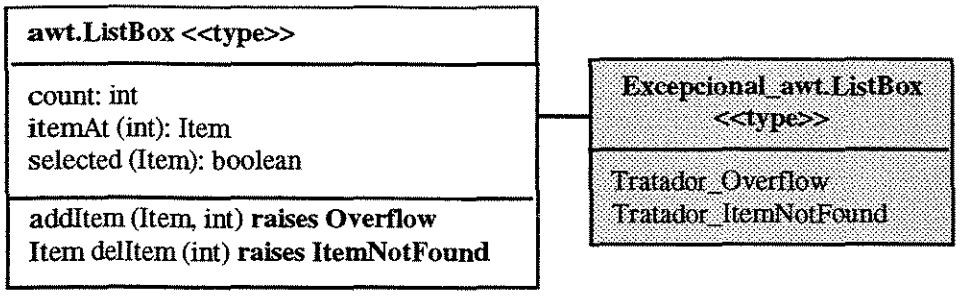

Figura 20 – Modelo básico de especificação de um tipo

As exceções que estão representadas neste modelo tipos só serão identificadas durante o projeto das colaborações e então o modelo de tipos deverá ser refinado a fim de incorporar sua representação e incluir os tratadores que são representados como ações dos tipos excepcionais. Ou seja, num primeiro momento o modelo de tipos é definido da forma como sugerido pelo processo Catalysis e após o projeto das colaborações ele deverá ser refinado a fim de conter a representação do comportamento excepcional identificado a sim como proposto neste trabalho.

O comportamento de um tipo, isto é, seu aspecto dinâmico, pode ser descrito através de diagramas de estados. Quando uma situação excepcional se manifesta, o sistema entra em um estado excepcional e após intervenção bem sucedida de recuperação do sistema ele volta para seus estados normais. Diagramas de estados devem ser adaptados para que incluam representação dos *estados excepcionais* do sistema. Caso a medida de recuperação adotada não seja efetiva o sistema termina de forma excepcional representado por *estados finais anormais*. Veja na seção 5.4.3 a extensão de um diagrama de estados da UML para representação de estados e transições excepcionais.

Depois de especificar os limites do sistema o próximo passo é estruturá-lo internamente através do projeto arquitetural, do projeto das colaborações e do projeto interno ao nível de classes, com suas operações e atributos.

4.4 Projeto do Sistema: O Interior

Projetar o interior de um sistema significa definir sua arquitetura onde é modelada a forma de interação entre os componentes arquiteturais, projetar as colaborações onde é definido como os componentes interagem e, finalmente como cada componente é projetado internamente. Estas atividades estão descrita nas seções 4.4.1, 4.4.2 e 4.4.3 respectivamente e incluem os procedimentos para modelagem, identificação e representação do comportamento excepcional do

sistemas.

4.4.1 Projeto Arquitetural

Projeto arquitetural é a modelagem da disposição e interação entre os componentes do sistema em um nível alto de abstração. Já apresentamos a importância em se considerar as exceções a nível arquitetural na seção 3.4 e também o estilo arquitetural baseado no modelo do componente ideal que favorece a construção de sistemas confiáveis.

Para o processo Catalysis, a sugestões propostas para definição do projeto arquitetural de sistemas confiáveis são as mesmas que as apresentadas pela metodologia MDCE no capítulo anterior, e portanto não serão repetidas aqui. Passemos então para uma atividade específica do processo Catalysis, o projeto das colaborações entre os componentes do sistema.

4.4.2 Projeto das Colaborações

Durante o projeto das colaborações os tipos definidos na seção 4.3 são implementados através de uma coleção de objetos e ações que cooperam para prover a funcionalidade desejada. A figura abaixo exemplifica como o tipo **ListBox** pode ser implementado através da colaboração entre dois elementos: **List** e **Item.**

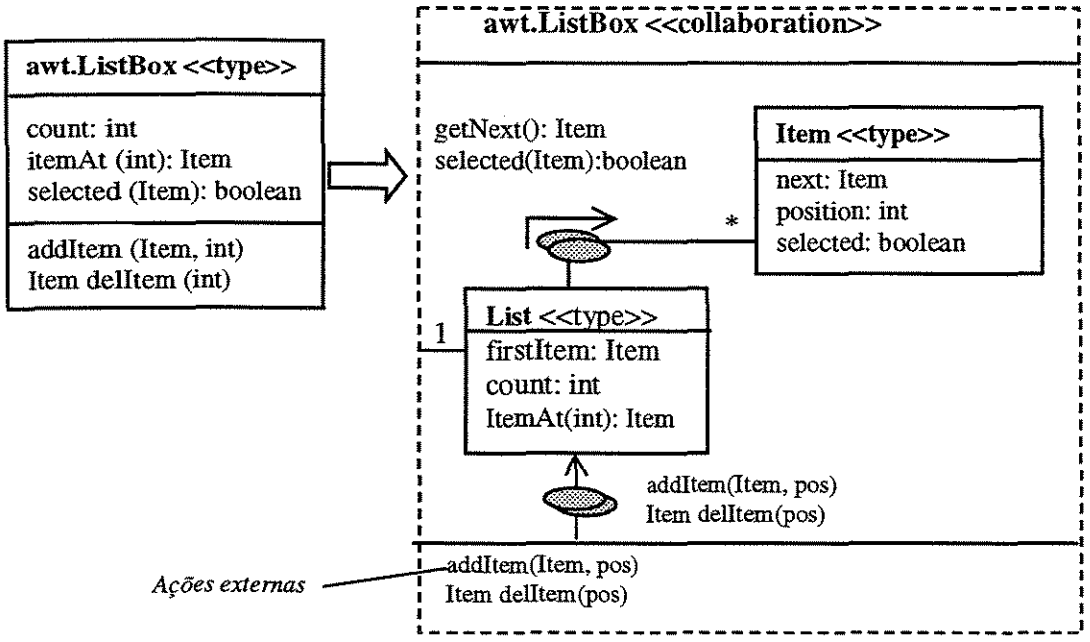

Figura 21 – Implementação de um *Tipo* sob a forma de uma *Colaboração*

A colaboração da Figura 21 mostra as duas ações externas que o tipo **ListBox** está submetido: inserção (addItem) e remoção (delItem) de um item na lista. As ações internas visam prover estas funcionalidades. Como foi dito, o tipo **ListBox** é implementado através da colaboração entre os tipos **List** e **Item**. Cada **Item** possui uma *referência* para seu sucessor na lista, além de um inteiro que indica sua *posição* e um atributo que indica se ele é o item *selecionado* ou não. Quando uma operação de inserção (addItem) é solicitada ao tipo **ListBox**, o tipo **List** é responsável por localizar a posição de inserção do novo Item, acrescentá-lo na lista, atualizando seu sucessor e as posições dos itens subseqüentes da lista.

O comportamento normal de uma colaboração é especificado através de ações definidas através de suas pré e pós-condições. Para cada ação, seja ela interna ou externa, *situações excepcionais* podem ocorrer. Estas situações excepcionais correspondem à violação de pré-condições, invariantes ou pós-condições de ações e são representadas por *exceções* que definem o *comportamento excepcional* da colaboração. Além de especificar a *exceção*, o *comportamento excepcional* inclui as medidas de recuperação do sistema e as condições garantidas após tratamento bem sucedido da exceção.

Sugerimos um modelo para a descrição do comportamento normal e excepcional de uma colaboração apresentado na Tabela 1

Comportamento Normal		Comportamento Anormal	
Ação:	A atividade especificada na colaboração;	**Sinal:**	O estado errôneo ou evento que identificou a falha;
Pre-condição:	Conjunto de condições que devem ser válidas ao iniciar a ação;	**Tratador:**	As ações que devem tentar tartar a falha;
Invariante:	Condições que devem ser mantidas durante a ação;	**Pós-Condição:**	Condições garantidas após finalizar o tratamento da exceção;
Pós-condição:	Conjunto de condições que devem ser válidas quando a ação terminar;		

Tabela 1 – Modelo para descrição de ações de uma colaboração robusta

Para facilitar a especificação de exceções, definimos uma notação para nomenclatura de exceções especificando seu tipo (interna ou externa) e a colaboração a qual está associada. O tipo pode ser {I, E}, onde "I" representa exceção interna e "E" representa exceção externa. O modelo é o seguinte:

```
Tipo_NomeDaExceção_NomeDaColaboração
```

A descrição das ações de uma colaboração, com suas pré, invariantes e pós-condições, podem ser feitas tanto informalmente quanto de maneira formal. Catalysis defende o emprego de formalismos pois considera que os modelos gerados terão menos ambigüidades e inconsistências. Para fazer uso de formalismos ao descrever as pré- condições, as invariantes e a pós-condições a Tabela 2 apresenta alguma notação básica.

num@**pré**	indica o estado ou valor de *num* antes da execução da operação.
exceção!	sinalização lançamento da *exceção*
exceção?	sinalização captura da *exceção*

Tabela 2 – Nomenclatura para descrição formal de ações

Para exemplificar, considere a colaboração apresentada na Figura 21. A descrição de suas ações, incluindo tanto o comportamento normal quanto o excepcional está apresentada na Tabela 3.

Comportamento Normal	Comportamento Anormal

Ação: addItem (Item item, int pos) **Pre-condição:** a posição de inserção do novo elemento deve estar entre 0 e a ultima posição + 1 **pre** (0 < pos <= count+1) **Pós-condição:** o item foi inserido na posição indicada e selecionado; o número de elementos da lista aumentou uma unidade; todos os elementos que se encontravam a partir da posição pos foram movidos para cima. **post** item = itemAt(pos) **and** count = count@**pre**+1 **and** selected(item) **and** movedUp (pos,count@**pre**)	**Sinal:** a posição de inserção não se encontra dentro dos limites esperados pos = 0 **or** pos > count+1 **Tratador:** sinalizar posição inválida através do lançamento de uma exceção E_Overflow_ListBox! **Pós-Condição:** o item não foi inserido na posição indicada e o número de elementos da lista se mantive inalterado **post** itemAt(pos) = itemAt(pos)@**pre** **and** count = count@**pre**
Ação: Item item delItem (int pos) **Pre-condição:** a posição de remoção de um elemento deve estar entre 1 e a ultima posição **pre** (1 < pos <= count) **Pós-condição:** o item foi retirado da posição indicada; o número de elementos da lista diminuiu uma unidade; todos os elementos que se encontravam a partir da posição pos + 1 foram movidos para baixo. **post** item = itemAt(pos) **and** count = count@**pre**-1 **and** movedDown (pos+1,count@pre)	**Sinal:** a posição de remoção não se encontra dentro dos limites esperados pos < 1 **or** pos > count **Tratador:** sinalizar inexistência de elemento através do lançamento de uma exceção E_ItemNotFound_ListBox! **Pós-Condição:** nenhum elemento foi retirado; o número de elementos da lista se manteve inalterado. **post** itemAt(pos) = itemAt(pos)@**pre** **and** count = count@**pre**

Tabela 3 – Comportamento Excepcional devido a falhas em ações

O comportamento excepcional não está restrito a falhas internas a um tipo. Além disto, outras situações excepcionais são advindas de interações mal sucedidas entre dois ou mais elementos. Por exemplo, o tipo `List` solicita a um objeto do tipo `Item` uma referência para seu sucessor através da ação `getNext()`. Suponhamos que esta referência não tenha sido encontrada e a ação `getNext()` retorne a exceção `E_NextNotFound_ListBox`. Esta situação deve ser tratada, fazendo parte do comportamento excepcional do tipo `List` como mostrado na Tabela 4.

Comportamento Normal	Comportamento Anormal

Ação:	Item item getNext ()	**Sinal:**	sucessor não foi encontrado e o tipo Item sinalizou o problema com a seguinte exceção capturada pelo tipo List
Pós-condição: sucessor.	o item retorna uma referencia para seu		E_NextNotFound_ListBox?
	post item = Item.next	**Tratador:**	a lista está descontinuada. Não há nada que o tipo List possa fazer, a não ser sinalizar o problema e em qual item a lista termina.
			Alarme(Lista descontinuada)
		Pós-Condição:	o item não foi retornado.
			post item == NULL

Tabela 4 – Comportamento Excepcional advindo de interações

Um cenário ilustra a dinâmica de uma colaboração, seja ele descrito de forma narrativa ou através de um diagrama de interação (seqüência ou colaboração). Cenários e diagramas de interação são ferramentas importantes para ilustrar a manifestação de situações anormais dentro da realização de uma colaboração. Para cada ação de um cenário, duas situações são possíveis: uma resposta normal e uma excepcional. Para cada ação, identifique os efeitos da execução bem ou mal sucedida da operação sob as instâncias (objetos) dos tipos envolvidos na colaboração. Veja em 5.4.2 - Diagramas de Interação como representar situações excepcionais em diagramas de interação da UML.

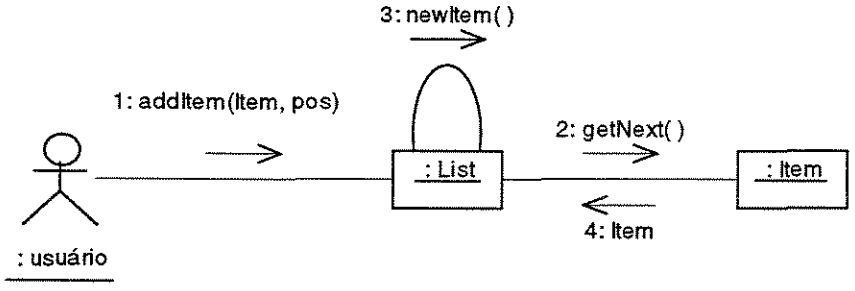

Figura 22 – Colaboração normal para adição de um item à lista

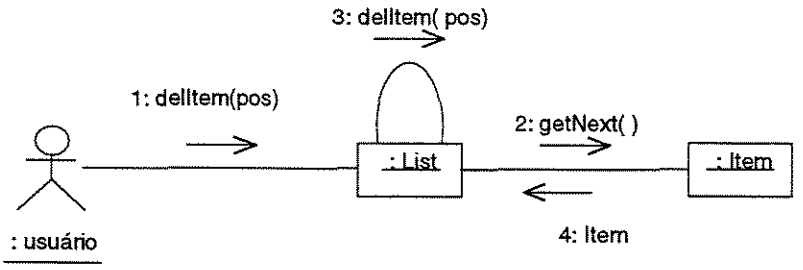

Figura 23 – Colaboração normal para remoção de um item da lista

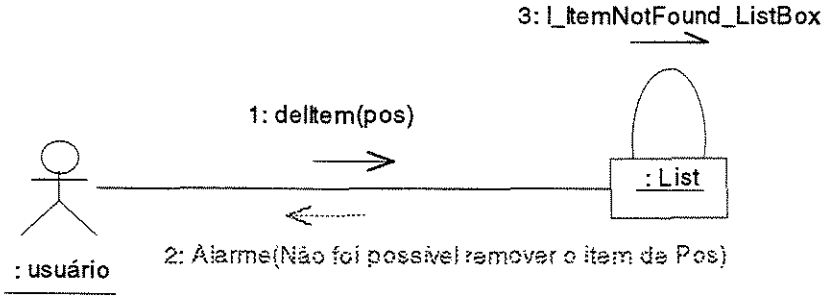

Figura 24 – Colaboração excepcional ao realizar a operação de remoção de item

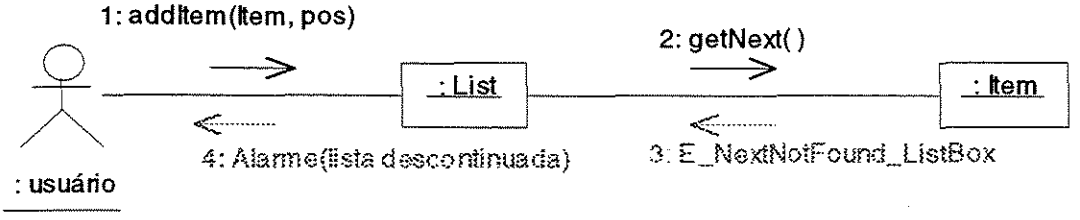

Figura 25 – Colaboração excepcional ao percorrer a lista

4.4.3 Projeto Interno dos Componentes

Os componentes individuais da aplicação são projetados e construídos até o nível de interfaces e classes ou componentes preexistentes. O objetivo desta atividade é definir uma estrutura interna e interações que satisfaçam os requisitos comportamentais, não funcionais e tecnológicos de um componente. Para componentes complexos, é necessário um particionamento em subcomponentes, que devem ser especificados e implementados recursivamente.

Esta fase é o momento de definir as classes que compõem o componente. O

modelo inicial do Processo Catalysis é feito pela materialização de cada *Tipo* em uma *Classe*. Na extensão que propomos ao Catalysis, um *Tipo* não é mais representado por uma única classe e sim por duas classes: classe normal e classe excepcional. A classe normal implementa a funcionalidade do *Tipo* e a classe excepcional implementa o *Tipo_Excepcional* já previamente identificado. A classe normal mantém explícitas no modelo as exceções internas por ela lançadas. A classe anormal mantém os tratadores para todas as exceções que a classe normal lançar e ainda deixa explícitas as exceções externas por ela lançadas. A partir daí, o projeto interno de um componente segue a estrutura já definida para metodologia MDCE e apresentada na seção 3.5.

No exemplo usado, o Tipo **ListBox** foi projetado internamente como mostra a Figura 26. A partir daí, o modelo será implementado seguindo as sugestões já apresentadas para a metodologia MDCE na seção 3.6.

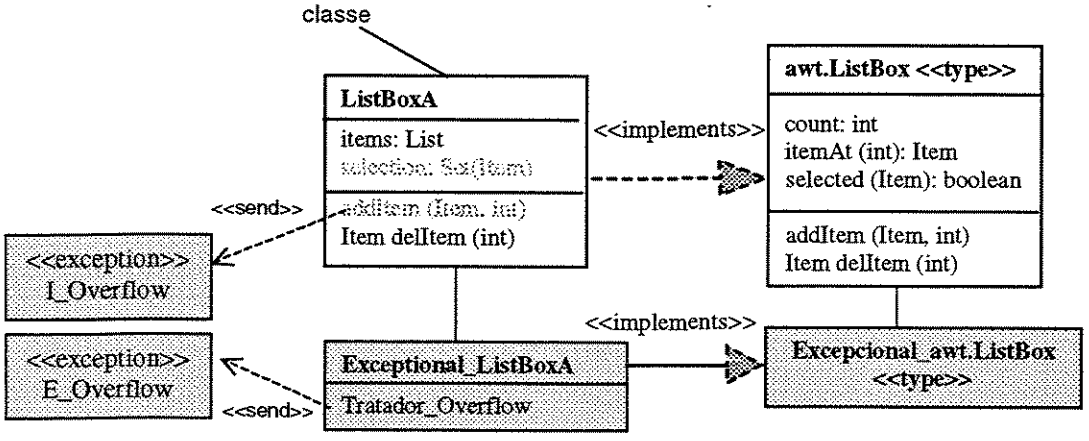

Figura 26 – Implementação de um tipo

4.5 Resumo

Este capítulo apresentou a metodologia MDCE aplicada ao processo Catalysis. Catalysis foi o processo escolhido para aplicação de MDCE pois tem uma construção particularmente interessante para projeto de sistemas confiáveis: *as colaborações*. Situações excepcionais ocorrem pela interação entre os elementos do sistema, isto é, são situações advindas de colaborações. Por este motivo, o projeto das colaborações é essencial para identificar as situações excepcionais além de possibilitar um projeto mais eficiente de recuperação que envolve o trabalho de mais de um componente do sistema.

A metodologia MDCE apresentou no capítulo 3 as atividades básicas para

construção de sistemas confiáveis que são: (1) especificação de requisitos, (2) projeto arquitetural, (3) projeto interno e (4) implementação. Catalysis mantém duas outras atividades para o desenvolvimento de sistemas computacionais. As atividades do Catalysis são: (1) especificação de requisitos, (2) especificação dos limites, (3) projeto arquitetural, (4) projeto das colaborações, (5) projeto interno e (6) implementação. Sendo assim, a aplicação da metodologia MDCE ao processo Catalysis se resume em definir como o comportamento excepcional de sistemas confiáveis é identificado e representado durante as atividades 2 e 4, isto é, durante a especificação dos limites através do modelo de tipos e durante o projeto das colaborações.

No capítulo seguinte mostramos como a UML foi estendida para que o comportamento excepcional projetado segundo MDCE ou Catalysis possa ser representado em seus modelos de caso de uso, classes, estados, colaborações ou seqüência.

Capítulo 5

Propostas de Extensões da UML para Representação do Comportamento Excepcional

Este capítulo apresenta como exceções são modeladas em UML [Booch+99] e ainda extensões da linguagem, propostas por este trabalho, que especificam novos estereótipos com semântica e restrições adicionais para a representação do comportamento excepcional. Além disto, apresentamos instruções de como fazer uso dos próprios diagramas da UML para representar o comportamento excepcional dos sistemas que fazem uso de tratamento de exceções para manterem sua confiabilidade e disponibilidade.

5.1 Estendendo UML com Estereótipos

UML possui uma série de mecanismos de extensões, mas os mais usados são os estereótipos. Um estereótipo vem enclausurado por <<>>. Exemplo de estereótipo são o <<exception>> para representar classes de exceções, <<interface>> que representa um conjunto de operações usadas para especificar os serviços de uma classe ou componente, <<type>> usado para especificar um conjunto de objetos, seus atributos e operações aplicáveis a objetos daquele tipo.

Um estereótipo é a abertura para se criar novos elementos de modelagem em UML. Neste trabalho, sugerimos o uso de dois novos estereótipos no projeto de sistemas confiáveis segundo a metodologia MDCE.

- **O estereótipo <<abnormal>>**: Na metodologia proposta neste trabalho, apresentamos o conceito de classe excepcional ou anormal. Uma classe anormal(ou excepcional) é aquela que implementa a parte excepcional de um componente, isto é, mantém os tratadores para exceções que o componente

pode lançar. Classes anormais devem ser modeladas explicitamente através do estereótipo <<abnormal>>. Veja um exemplo na Figura 27 onde a classe *listBox_Excepcional* apresenta os tratadores para as exceções que a classe *listBox* pode lançar. Através do uso do estereótipo fica claro no modelo que a classe *listBox_Excepcional* representa uma classe excepcional mesmo se o projetista não deseje utilizar um nome que indique isto.

- **O estereótipo <<handler>>** : Um caso de uso definido de forma completa possui um fluxo normal, descrito por um cenário normal, e vários fluxos excepcionais. Um fluxo excepcional é o desvio do fluxo normal para tratamento de falhas e podem ser de dois tipos: *Recuperável* e *de Falha*. Muitas vezes, o tratamento da falha é complexo o suficiente para que faça parte de um caso de uso completo e não seja representado apenas por um cenário de um caso de uso. Um caso de uso que contém medidas de recuperação do sistema é identificado pelo estereótipo <<handler>>. Veja Figura 28.

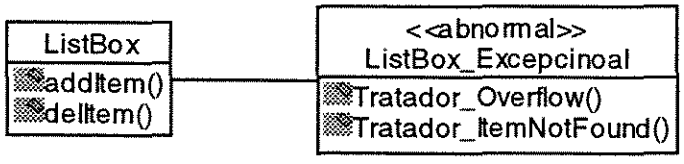

Figura 27 – Estereótipo para classes anormais

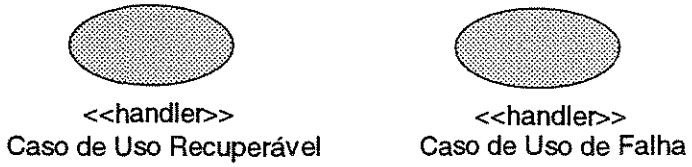

Figura 28 – Estereótipos para casos de uso anormais

5.2 Modelando Exceções em UML

Em UML, exceções são sinais modelados como classes estereotipadas por <<exception>>. Um sinal representa um objeto que pode ser disparado (lançado) assincronamente por um objeto e recebido por outro. Para cada operação de classe ou interface, as exceções lançadas são representadas explicitamente através da dependência <<send>> entre uma operação e uma exceção. As exceções também devem estar hierarquicamente organizadas o que permite que tratadores genéricos

possam ser empregados para tratar uma família de exceções.

A Figura 29 mantém a hierarquia de exceções que podem ser lançadas pela classe *listBox*. Esta hierarquia é iniciada pela classe abstrata *Exception* e inclui três outras exceções especializadas: *Duplicate, Overflow e Underflow*. Como mostrado, a operação *add* da classe *listBox* lança as exceções *Duplicate* e *Overflow* e a operação *remove* lança *Underflow*. Como uma alternativa, estas dependências podem ser colocadas apenas na especificação da operação.

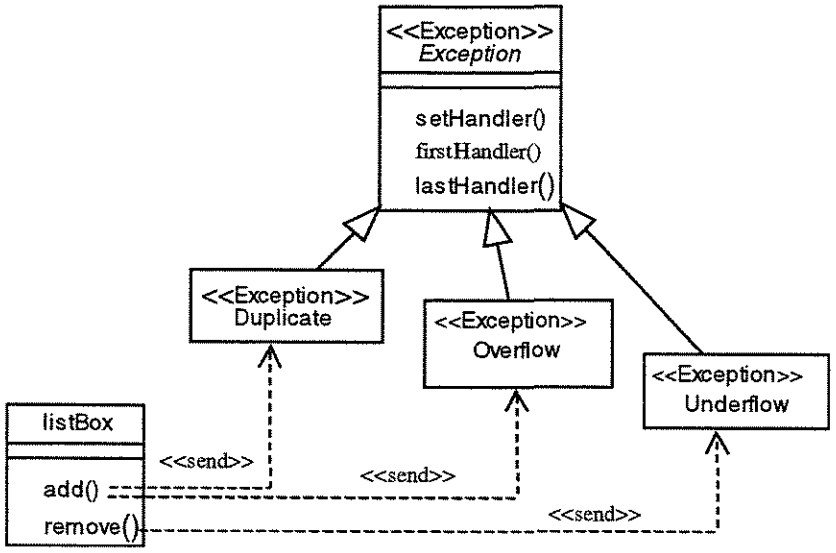

Figura 29 – Modelando Exceções

5.3 Organização dos casos de uso

Notações especiais são utilizadas para facilitar descrições mais complexas de casos de uso. Entre estas notações, destacam-se os casos de usos secundários, que simplificam o comportamento dos casos de uso primários através dos mecanismos de **inclusão** e **extensão**. O caso de uso B **estende** o caso de uso A quando B representa uma situação opcional ou de exceção, que normalmente não ocorre durante a execução de A. A associação de extensão pode ser vista como uma interrupção do caso de uso original que ocorre quando um novo caso de uso será inserido. O caso de uso original não sabe se a interrupção ocorrerá ou não. A associação de extensão é estereotipada por <<extend>>.

No contexto deste trabalho, um caso de uso é interrompido quando uma situação excepcional é identificada. Neste momento, medidas de recuperação são acionadas. Quando estas medidas de recuperação constituem um caso de uso

completo, aqueles estereotipados por <<handler>> (seção 5.1), a notação <<extend>> deve ser usada. Veja Figura 30.

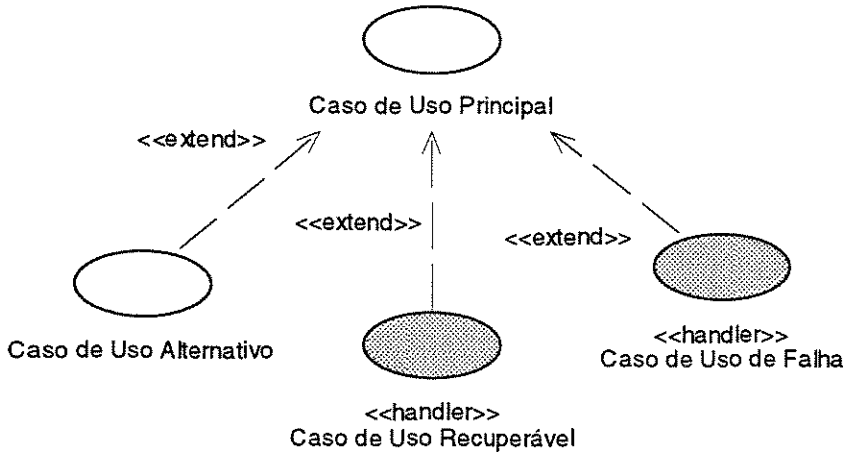

Figura 30 – Associação de extensão entre casos de usos

5.4 Diagramas de modelagem dinâmica

Quando projetamos sistemas tolerantes a falhas devemos nos concentrar no seu aspecto dinâmico, no seu comportamento, pois é durante a execução do sistema que situações excepcionais acontecem. Os diagramas da UML que representam os aspectos dinâmicos do sistema são: seqüência, colaboração, estados e atividades. Nas seções seguintes apresentamos como cada um destes diagramas de modelagem dinâmica da UML podem ser usados para incluírem a representação de exceções e desvios causados pelas tentativas de recuperação do sistema.

5.4.1 Diagramas de Atividades

Diagramas de atividades são usados para modelagem do fluxo de controle entre atividades de um sistema. Através deste diagrama torna-se possível modelar o fluxo normal e os desvios causados pela manifestação de situações excepcionais em casos de uso ou funções do sistema. Em um diagrama de atividades, cada atividade é representada por retângulos com bordas arredondadas e as transições entre atividades por setas que indicam a direção do fluxo de execução. Quando se trata da representação das atividades de um caso de uso, esta diagramação facilita o entendimento da dinâmica e aprimora a comunicação com o cliente do sistema.

As atividades podem ser separadas por grupos que são responsáveis por suas

realizações. Em UML, cada grupo é chamado de raia. Cada raia é responsável por parte das atividades do diagrama e é geralmente implementada por uma ou mais classes. Para manter a separação entre o fluxo normal e excepcional de um caso de uso ou função do sistema, as atividades de cada um destes fluxos devem ser representadas em raias separadas, a raia do fluxo ou cenário normal e as raias dos fluxos ou cenários excepcionais (Recuperáveis e de Falha) como mostrado na Figura 31. Sobre cenários excepcionais e normais de um caso de uso consulte 3.3.1 - O modelo de casos de uso.

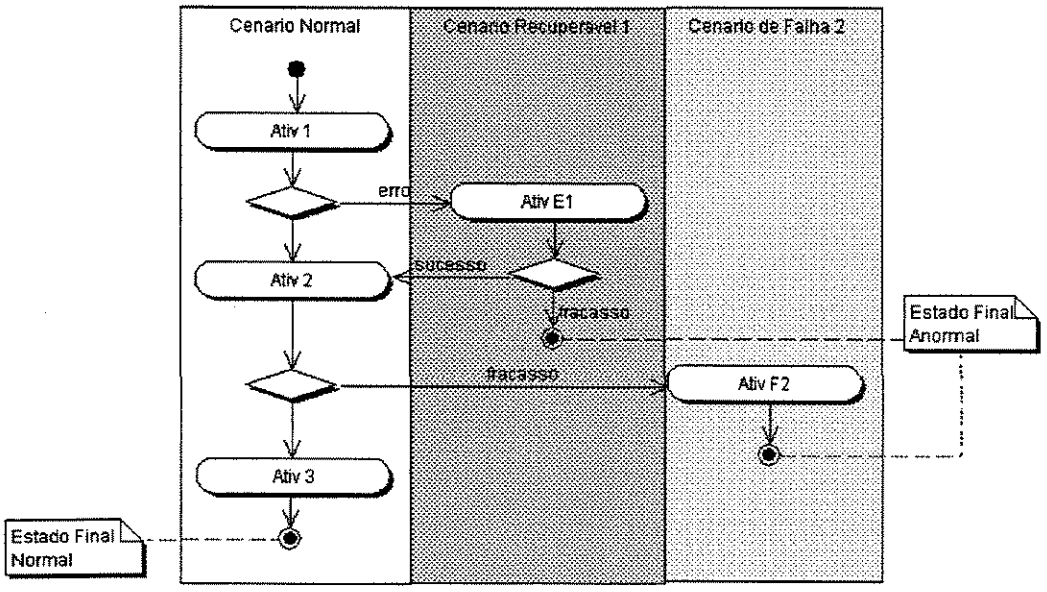

Figura 31 – Diagrama de Atividades de um Caso de Uso

Cada atividade do cenário normal pode ser bem ou mal sucedida. Dado que uma atividade do cenário normal fracassou, um cenário excepcional será executado para tentar retornar o sistema a um estado estável. No exemplo apresentado na Figura 31, a ocorrência de um erro na *Ativ1* levou a execução do *cenário recuperável 1*. A realização de um cenário excepcional pode ou não obter sucesso na recuperação do sistema. Caso o problema seja recuperado com sucesso, o cenário normal volta a ser executado e caso contrário, o sistema terminará de forma excepcional. No caso de uma situação de falha, como ocorrido com a *ativ2* do exemplo, o *cenário de falha 2* será executado a fim de tomar algumas medidas de precaução mas não existe a chance de retomar o fluxo de execução normal e o sistema sempre terminará de forma excepcional.

5.4.2 Diagramas de Interação

Tanto os diagramas de seqüência, quanto os de colaboração são tipos de diagrama de interação. Um diagrama de interação pode mostrar tanto um conjunto de atores, casos de usos e suas relações quanto um conjunto de objetos e suas interações.

No contexto deste trabalho, diagramas de interação não são importantes apenas para modelagem do comportamento excepcional de um sistema, mas também porque, muitas vezes, um elemento isolado não é capaz de recuperar o sistema de forma eficiente por não possuir recursos suficientes para tratar a falha tornando necessária a cooperação de outros elementos do sistema. Além disto, dois objetos se considerados de forma isolada podem estar em um estado válido, entretanto, se combinados, representam uma composição inválida o que traz riscos ao funcionamento do sistema.

Um diagrama de seqüência enfatiza a ordem das mensagens no tempo enquanto o diagrama de colaboração enfatiza a organização estrutural dos objetos que interagem. Ambos os diagramas representam um conjunto de objetos e as mensagens enviadas e recebidas por eles. A figura abaixo exemplifica o uso tradicional dos diagramas de interação.

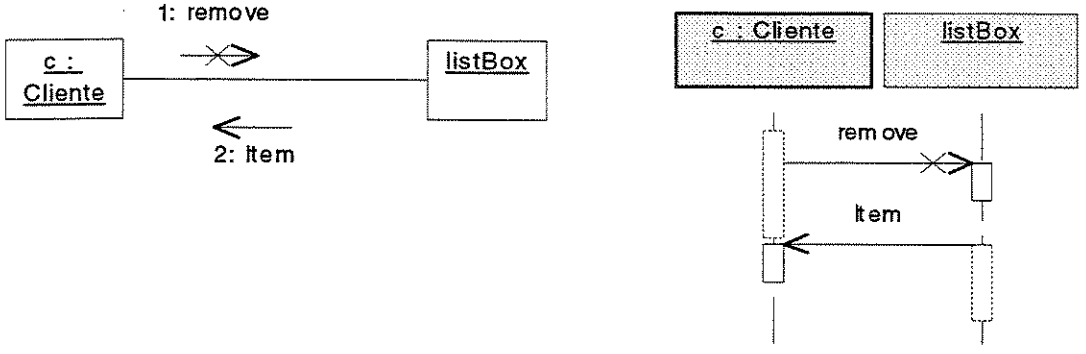

Figura 32 – Diagramas de Interação Normal

Abaixo encontra-se o diagrama de colaboração e seqüência excepcional da iteração mostrada na Figura 32. A exceção *underflow* foi lançada a retirar mais um item da lista quando esta se encontrava vazia. Como exceção é um evento, a mensagem é assíncrona.

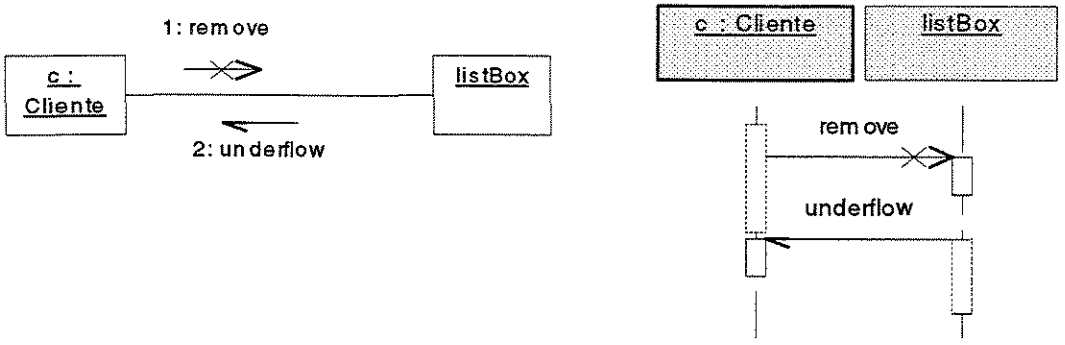

Figura 33 – Diagramas de Interação Excepcional

5.4.3 Diagramas de Estados

Para entender um caso de uso, ele pode ser visto como um gráfico de transição de estados. A cada estimulo enviado pelos atores, alguma atividade é realizada e dependendo do estado em que o sistema se encontra, ele é levado a executar uma transição de estado. O diagrama de estados também é importante para modelar a dinâmica de um modelo de tipos como foi falado na seção 4.3.

No contexto de casos de uso, cada transição de estados representa a realização de uma atividade do caso de uso. Quando uma atividade falhar, o sistema será levado a um estado excepcional. Seguindo a terminologia definida para casos de usos, um estado excepcional pode ser um *estado recuperável* ou um *estado de falha*. No exemplo da Figura 34, o sistema entra em um *estado recuperável* quando ocorreu um erro durante sua execução, mas este erro pode ser recuperado. Caso a atividade excepcional do sistema recupere o erro com sucesso, o sistema volta para um estado livre de erros. Caso contrário o sistema terminará de forma excepcional, em um estado final anormal (ou excepcional). Por outro lado, um sistema entra em um *estado de falha* quando uma situação não prevista ou uma situação inaceitável ocorre. Durante o estado de falha algumas atividades serão realizadas apenas para manter um estado estável e o sistema será sempre finalizado de forma excepcional, em um estado final excepcional.

Além disto, quando se considera tratamento de exceções, não se pode assumir

que uma transição de estados é sempre atômica [Miller+97]. Uma exceção pode ocorrer antes que a transição de estados se complete levando o sistema a estados excepcionais. Em [Miller+97] estes estados excepcionais são denominados *estados parciais*. Normalmente, os projetistas consideram a transição entre estados uma ação atômica, nunca observando os estados parciais. Entretanto, esta é uma observação importante para tomada de decisões sobre como retornar eficientemente o sistema para um estado livre de erros. Estas decisões incluem decidir se os objetos de um sistema em um estado excepcional devem ser mantidos ou destruídos.

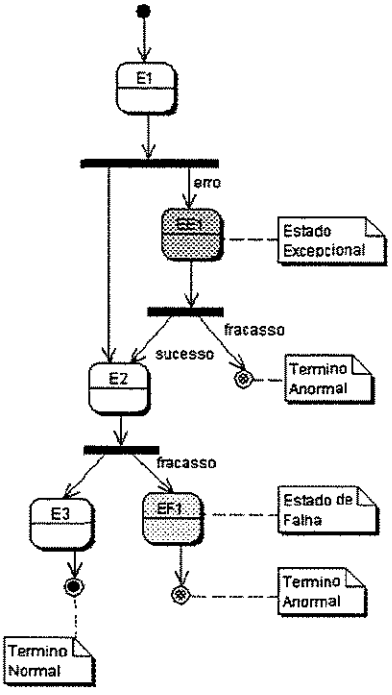

Figura 34 – Representação de estados normais e anormais de um sistema

Matrizes de transição e definição de estados

O diagrama de estados provê uma visão do comportamento que é diferente da proporcionada pelas especificações das ações. O diagrama de estados focaliza em como as ações afetam uma especificação do sistema, com ênfase na seqüência de transições. Duas tabelas propostas em [Desmond98] podem ajudar a verificar a completude e consistência do diagrama de estados.

A primeira delas é a matriz de transição de estados mostrada na Tabela 5. Para uma especificação robusta, é importante que prever as transições excepcionais, isto

é, transições de estados que só ocorrerão mediante existência de falhas de projeto ou falhas que afetam o sistema externamente e levariam o sistema a problemas de funcionamento.

A construção de uma matriz de transição de estados auxilia na identificação de possíveis transições excepcionais, pois o objetivo é preencher todas as lacunas da matriz, isto é, considerar todas as possibilidades de transições. Mesmo que no projeto uma transição nunca aconteça é importante considerá-la. Existem situações que, para um projeto bem feito, nunca iriam ocorrer, mas ter certeza de que se trata de um projeto bem feito é uma consideração difícil de ser afirmada. Sendo assim, devemos prever a possibilidade desta transição e sinalizá-la através de uma exceção de falha de projeto representada na matriz por *"DF"* (Design Fault).

Algumas situações apresentam riscos para o funcionamento do sistema e devem ser tratadas com mais cuidado. Estes eventos devem ser representados por "*X*" na tabela de transição de estados e o diagrama de estados deve ser revisto para inclusão de estados excepcionais que serão ativados mediante sua ocorrência. Existem ainda situações que a manifestação de um evento não traz conseqüência alguma para o sistema. Estes eventos são representados por "|" na matriz de transição de estados.

	E1	E2	E3
e1	[E2]	X	|
e2	[E3]	[E3]	|
e3	|	DF	[E1]

Tabela 5 – Matriz de Transição de Estados

A outra matriz que nos referimos anteriormente, é a matriz de definição de estados. Cada estado é definido em termos de seus atributos e associações como mostrado na Tabela 6.

	E1	E2	E3
atrib1	>3	null	<> null
atrib2	on	off	null
atrib3	>0	<>null	-

Tabela 6 – Matriz de Definição de Estados

5.5 Resumo

Este capítulo apresentou propostas de extensões da linguagem de modelagem UML para representação do comportamento excepcional de um sistema confiável

modelado segundo a metodologia MDCE, apresentada no Capítulo 3.

Sugerimos novos estereótipos para diferenciar casos de uso e classes responsáveis pela implementação do comportamento excepcional do sistema. Além disto, apresentamos como os diagramas de modelagem dinâmica da UML, diagramas de atividades, estados, colaboração e seqüência, podem ser utilizados para representar as atividades e estados excepcionais de um sistema. As maiores preocupações foram como manter em separado e diferenciado as atividades, estados e mensagens excepcionais. Apresentamos ainda, uma matriz de transição de estados que auxilia a identificação de transições excepcionais.

Capítulo 6

Estudo de Caso: O Sistema de Mineração

Este capítulo apresenta um estudo de caso para exemplificar as extensões feitas ao processo Catalysis que visam identificar e representar o comportamento excepcional de sistemas confiáveis. O estudo de caso escolhido é baseado na aplicação de um sistema de controle de mineração apresentado em [Sloman+87]. Este sistema é responsável pelo controle da extração de mineral que é um processo que além de produzir água, também libera gás metano no ambiente. O sistema de controle da mineração deve ainda manter o nível de água acumulada em um reservatório e nível de metano da mina sob controle. Consideramos este exemplo um bom estudo de caso pois está submetido a uma série de falhas incluindo mal funcionamento dos dispositivos de extração e riscos de explosão caso a bomba (mecanismo de extração de água) opere quando o nível de metano estiver elevado.

Na seção 6.1 apresentamos em detalhes em que consiste o sistema de mineração. Na seção 6.2 seus requisitos são apresentados sob a forma de casos de uso que descrevem o comportamento completo do sistema, incluindo as medidas para identificação das situações excepcionais e recuperação do sistema. Na seção 6.3 o sistema é especificado sob a forma de modelos de tipos e na seção 6.4 projetado recursivamente até o nível de componentes implementáveis ou já disponíveis no mercado.

6.1 O Sistema de Mineração

O sistema de mineração consiste em três estações de controle: uma que monitora o nível de água no reservatório, outra que monitora o nível de metano no ambiente e outra que monitora a extração de mineral propriamente dita. A

representação esquemática do sistema está apresentada na Figura 35. A extração de mineral é realizada através de um braço mecânico que durante sua operação, se o nível de água atingir a capacidade máxima do reservatório, é desligado e uma bomba é ligada para drenar o reservatório. Quando o nível de água alcançar o limite mínimo, a bomba será desligada e o braço deverá retomar o seu funcionamento normal. Da mesma forma, se o nível de metano estiver alto durante a extração de mineral, o braço mecânico é desligado e um exaustor é ligado para remover o excesso do gás metano no ambiente. Quando o nível de metano estiver sob controle o exaustor será desligado e o braço mecânico retornará a operar normalmente. Durante a operação dos componentes do sistema, sensores de níveis sinalizam as situações do nível de água e de metano.

A bomba e o exaustor são dispositivos não confiáveis que podem falhar durante a operação. Além disto, por razões de segurança, a bomba não pode funcionar quando o nível de metano estiver alto, pois haveria riscos de explosão. Para manter a confiabilidade e a disponibilidade do sistema, além da segurança do ambiente onde ele está inserido, medidas de tolerância a falhas devem ser adotadas. Sensores confiáveis que indiquem a presença de fluxo de água e ar são incorporados ao sistema para indicar o correto funcionamento dos dispositivos bomba e exaustor respectivamente. Além disto, todo o sistema é monitorado pelo operador através de uma interface aonde o sistema sinaliza as emergências para que o operador possa intervir.

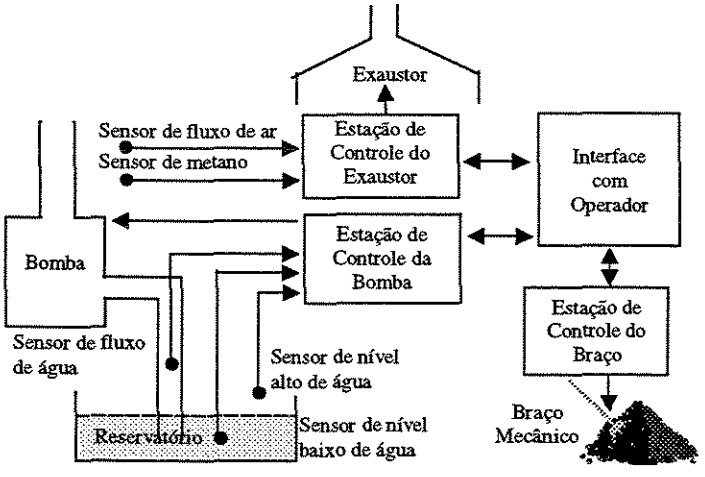

Figura 35– Representação esquemática da extração de mineral

6.2 Especificação dos Requisitos do Sistema de Mineração

Esta seção contém a especificação dos requisitos do sistema adotado como estudo de caso. A primeira etapa para especificação dos requisitos é identificação dos atores que interagem com o sistema, apresentados na seção 6.2.1, e a definição dos limites do sistema, apresentados na seção 6.2.2. A seguir, o comportamento normal e excepcional do sistema serão descritos como apresentado nas seções 6.2.3 e 6.2.4 respectivamente. Para finalizar, o comportamento do sistema será refinado como especificado em 6.2.5.

6.2.1 Identificação dos Atores

- **Operador:** Operador é quem inicia e interrompe a operação de extração de mineral através de sua interface.
- **O braço mecânico:** dispositivo responsável pela extração de mineral;
- **A bomba:** dispositivo responsável por drenar o reservatório;
- **O exaustor:** dispositivo responsável por retirar ar que contém metano do ambiente;
- Sensor **MuitoMetano**: sensor de metano que indica nível elevado de metano no ambiente;
- Sensores **MuitaAgua** e **PoucaAgua:** sensores que detectam nível alto e baixo de água no reservatório respectivamente;
- Sensor **FluxoAgua:** sensor que indica que a bomba está funcionando corretamente, isto é, está sendo retirada água do reservatório que passa pelos canos aonde se encontra o sensor;
- Sensor **FluxoAr:** sensor que indica que o exaustor está retirando metano do ambiente, isto é, está funcionando corretamente.

6.2.2 Definição dos Limites do Sistema

Os principais casos de uso do sistema de mineração são facilmente identificados. O caso de uso principal está relacionado à extração de mineral (caso de uso *ExtrairMineral*), como esta operação produz água e metano, é necessário parar a extração de mineral para retirar a água do reservatório (caso de uso *DrenarReservatório*) e metano da mina (caso de uso *ExtrairMetano*). Como o caso

de uso *ExtrariMineral* será interrompido para realização do caso de uso *DrenarReservatório* ou *ExtrairMetano*, estes devem ser modelados como extensões do caso de uso original como mostra a Figura 36.

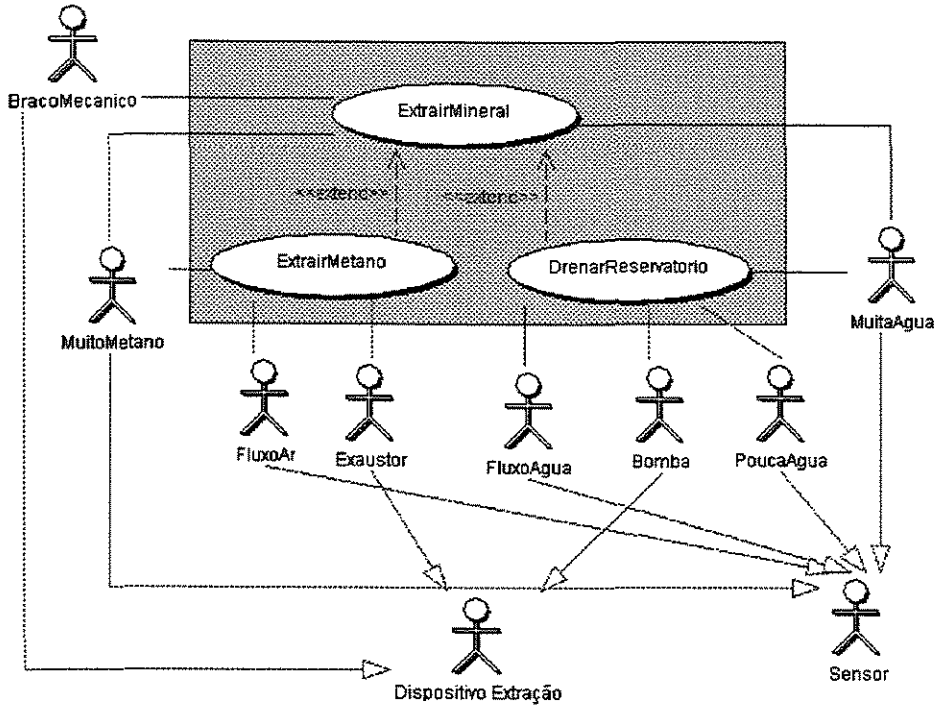

Figura 36 – Diagrama de Casos de Uso do Sistema de Mineração

Nas seções seguintes, apresento um único caso de uso do sistema da Mineração, o caso de uso *DrenarReservatório*, para exemplificar o processo de especificação de requisitos. O sistema está sendo projetado para tolerar as falhas dos dispositivos bomba e exaustor, considerando que os sensores são todos elementos confiáveis. Além disto, estamos levando em conta que a bomba não deve funcionar enquanto o nível de metano estiver alto devido ao risco de explosão. Os sensores de fluxo de ar e água são usados para identificar as situações excepcionais do exaustor e bomba respectivamente.

6.2.3 Especificação do Comportamento Normal

A descrição textual do comportamento normal do caso de uso *DrenarReservatório* está apresentada na Figura 37. A descrição do comportamento normal de um caso de uso inclui, além de seu fluxo de execução normal, os participantes envolvidos na realização do caso de uso, suas pré-condições,

invariantes e pós-condições.

Caso de Uso DrenarReservatório

Descrição:	Quando o nível de água subir, a bomba deve ser ligada para abaixa-lo. A bomba deve ser desligada quando a água atingir seu nível mínimo;
Participantes:	Bomba, sensor MuitaAgua, sensor PoucaAgua e sensor FluxoAgua, sensor MuitoMetano;
Pré-condições:	Bomba desligada, sensor FluxoAgua desligado, sensor MuitaAgua ligado, sensores MuitoMetano , PoucaAgua e FluxoAr desligados
Invariantes:	sensor MuitoMetano desligado, sensor FluxoAgua ligado, bomba ligada e sensor FluxoAr desligado;
Cenário Primário:	1. Ligar a bomba 2. Desligar a bomba quando o sensor de nível baixo de água ligar
Pós-condições:	Bomba desligada, sensor FluxoAgua desligado, sensor MuitaAgua desligado, sensores MuitoMetano , FluxoAr desligados, sensor PoucaAgua ligado

Figura 37 – Descrição do comportamento normal do caso de uso DrenarReservatório

Dado que o comportamento normal do caso de uso já fora especificado, este é o momento de descrever o comportamento do sistema dado a manifestação de situações excepcionais ou inaceitáveis para a segurança do sistema.

6.2.4 Especificação do Comportamento Excepcional

Situações excepcionais são decorrentes de violação das pré-condições, invariantes ou pós-condições do comportamento normal do caso de uso. Violação de pós-condição ou invariantes representam que o serviço não pode ser realizado devido a falhas internas ao caso de uso como problemas na execução de qualquer uma de suas atividades. Violação de pré-condições representam falhas no cliente ao solicitar a realização do serviço fornecido pelo caso de uso.

Na Figura 38 apresentamos três cenários excepcionais: dois cenários recuperáveis e um cenário de falha relacionados a problemas de manutenção das invariantes e pós-condições do caso de uso *DrenarReservatório*.

Caso de Uso DrenarReservatório	
Cenário Recuperável 1 **Violação de invariante**	Durante a extração de água o sensor FluxoAgua acusou ausência de fluxo, o que significa mal funcionamento da bomba;
- Sinal:	sensor FluxoAgua desligado e Bomba ligada
- Tratador:	iniciar o caso de uso BombaFalhou
- Pós-condições:	Bomba ligada, sensor FluxoAgua ligado
Cenário Recuperável 2 **Violação de pós-condição**	Após ter desligado a bomba, o sensor de fluxo FluxoAgua permanecia ligado, o que indica que a bomba não foi corretamente desligada;
- Sinal:	sensor FluxoAgua ligado e Bomba desligada
- Tratador:	Tentar desligar a bomba mais uma vez; Enviar alarme para operador.
- Pós-condições:	Bomba desligada, sensores FluxoAgua desligado e PoucaAgua ligado.
Cenário de Falha 1 **Violação de invariante**	Durante a extração de água o sensor de metano acusou alto nível do gás metano no ambiente, o que propicia risco de explosão da mina;
- Sinal:	MuitoMetano ligado e Bomba ligada
- Tratador:	Desligar bomba; Enviar alarme para operador.
- Pós-condições:	Bomba desligada e MuitoMetano ligado

Figura 38 – Comportamento excepcional do caso de uso DrenarReservatório

Como foi falado durante a descrição do problema na seção 6.1, manter a bomba funcionando em presença de alto nível de metano trás riscos de explosão do sistema. Este é o motivo pelo qual este cenário (bomba ligada e nível alto de metano) foi tratado com cenário de falha e não um cenário recuperável. Note que o comportamento do sistema em um cenário de falha não é de tentar recuperar o sistema e sim de tentar manter as condições seguras sob situações de risco.

Problemas de fluxo de água são decorrentes de mau funcionamento da bomba. Já era previsível que falhas acontecessem durante seu funcionamento pois se trata de um dispositivo não confiável. A medida de recuperação do cenário recuperável 1 consiste em um caso de uso completo chamado *BombaFalhou* que será descrito inteiramente a seguir.

Por fim, o cenário recuperável 2 indica que a bomba não foi corretamente desligada, pois ainda existe fluxo de água. Dado esta situação o sistema pode fazer novas tentativas de desligar a bomba e caso continue fracassando sua única possibilidade é avisar o operador para intervir e evitar que a bomba se queime por funcionar de forma incorreta.

6.2.4.1 Caso de uso Excepcional *BombaFalhou*

Este caso de uso trata de recuperação de problemas decorrentes da execução normal de caso de uso *DrenarReservatorio*. Este é um caso de uso ativado apenas diante de uma situação excepcional, sendo assim, trata-se de um caso de uso excepcional (ou anormal) estereotipado por <<handler>> como apresentado na seção 5.1 e na Figura 39.

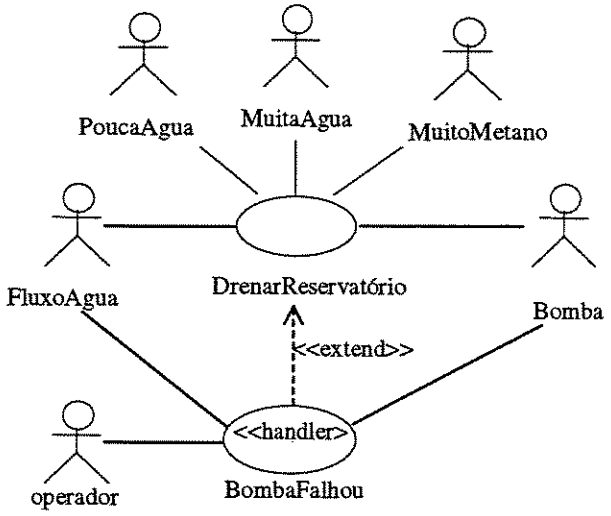

Figura 39 – Caso de Uso Excepcional BombaFalhou

Para decidir como a falha será tratada deverá ser avaliada a possibilidade de replicar o dispositivo não confiável *Bomba*. Consideramos que este dispositivo não será replicado por dois motivos: (1) não há recursos financeiros suficientes; (2) deixar a *Bomba* sem operar por alguns instantes não é uma situação extremamente critica pois a água do reservatório pode ser retirada de outra forma, por exemplo, manualmente. Note que a mesma decisão não poderia ser tomada com relação ao exaustor, pois manter nível alto de metano no interior da mina é uma situação muito mais arriscada do que manter nível alto de água. Além do mais, não é possível retirar metano de outra forma sem recursos de um exaustor. Desta forma, a estratégia de recuperação do sistema se resume em uma nova tentativa de reativar a bomba como mostrado nos cenários da Figura 40.

Caso de Uso Excepcional BombaFalhou	
Descrição:	Bomba foi ligada, mas sensor de fluxo de água não acusou funcionamento correto. Este caso de uso implementa uma nova tentativa de religar a bomba.
Participantes:	Bomba, sensor FluxoAgua, operador;
Pré-condições:	Bomba ligada, sensor FluxoAgua desligado;
Cenário Primário:	1. ligar a bomba
Pós-condições:	Bomba ligada, sensor FluxoAgua ligado;
Cenário Excepcional 1 **Violação de pós-condição**	A tentativa de ligar a bomba não foi bem sucedida pois o sensor de fluxo de água se manteve desligado;
- Sinal:	sensor FluxoAgua desligado e bomba ligada
- Tratador:	Desligar a bomba; Enviar alarme para operador.
- Pós-condições:	Bomba desligada, sensor FluxoAgua desligado

Figura 40 – Descrição dos comportamentos do caso de uso excepcional BombaFalhou

Os comportamentos normal e excepcional dos outros casos de uso do Sistema de Mineração, isto é, caso de uso *ExtrairMineral* e caso de uso *ExtrairMetano*, devem ser descritos de forma semelhante ao apresentado para o caso de uso *DrenarReservatorio*. Por questões didáticas não foram apresentados neste estudo de caso. Daremos continuidade à especificação dos requisitos de sistema de mineração apresentado um detalhamento dos casos de uso já identificados e especificados.

6.2.5 Detalhamento dos Casos de Uso

Os casos de uso descritos textualmente podem ser detalhados sob a forma de diagramas de modelagem dinâmica da UML. Estes diagramas podem ser usados para manter a representação explícita do comportamento excepcional do caso de uso como descrito na seção 5.4 - Diagramas de modelagem dinâmica.

Neste estudo de caso, mostraremos as atividades dos casos de uso *DrenarReservatório* e *BombaFalhou* sob a forma de:

- **Diagrama de atividades:** Na Figura 41 mostramos a seqüência de atividades dos casos de uso onde as atividades excepcionais estão separadas das atividades normais através de raias.
- **Diagrama de colaboração e seqüência normais:** Já a Figura 42 apresenta a diagramação do comportamento normal do caso de uso *DrenarReservatorio* através de diagramas de seqüência da UML que mostram a interação entre atores e casos de uso sem a manifestação de

falhas, onde tudo ocorre como esperado.

- **Diagrama de colaboração e seqüência excepcionais**: A Figura 43 apresenta a diagramação do cenário recuperável 1 do caso de uso *DrenarReservatorio* que destaca a manifestação de falha ao ligar a bomba e a medida de recuperação tomada que consiste numa nova tentativa de religá-la. A Figura 44 apresenta a diagramação do cenário recuperável 2 onde a bomba não é desligada corretamente e dentro de um prazo máximo o sensor de fluxo de água não sinaliza que o fluxo cessou.

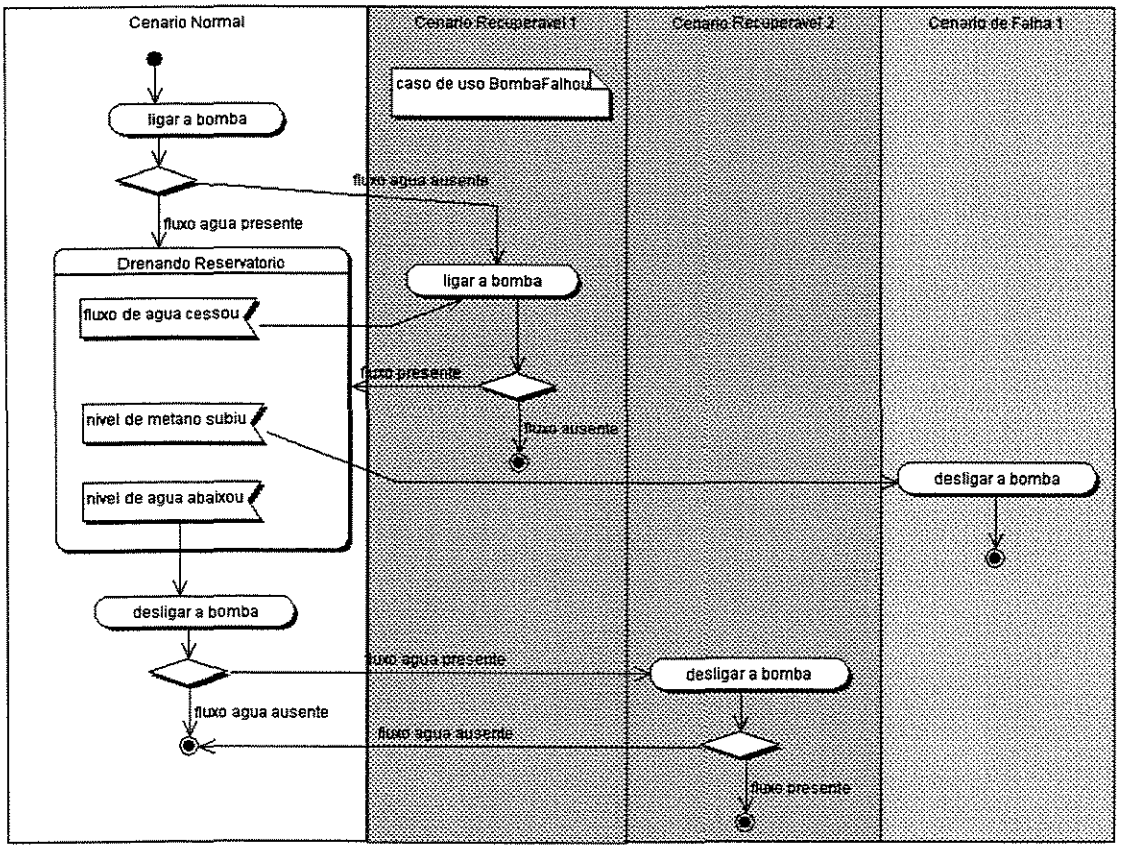

Figura 41 – Atividades normais e excepcionais do caso de uso DrenarReservatório

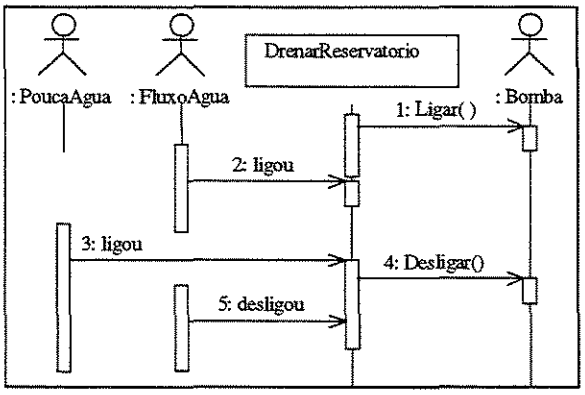

Figura 42 – Diagrama de Seqüência Normal do caso de uso DrenarReservatório

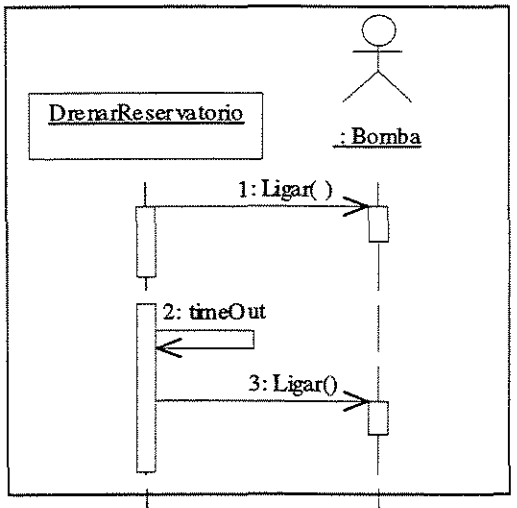

Figura 43 – Cenário recuperável 1 do caso de uso DrenarReservatório

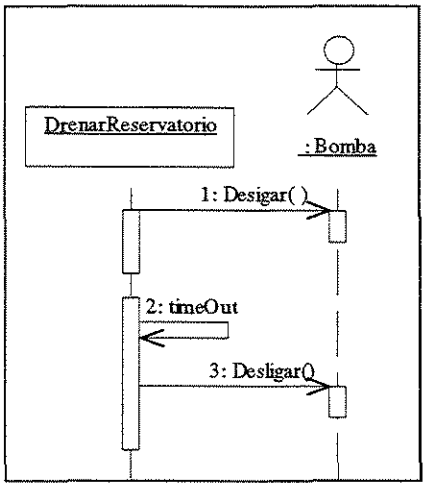

Figura 44 – Cenário recuperável 2 do caso de uso DrenarReservatorio

Além da diagramação dos comportamentos de casos de uso, a fase de detalhamento visa apresentar o modelo de estados do sistema. Observe que esta atividade vai auxiliar na identificação de transições excepcionais que provavelmente passariam despercebidas. A Figura 45 apresenta os estados mais genéricos do sistema dado que nenhuma situação excepcional aconteça.

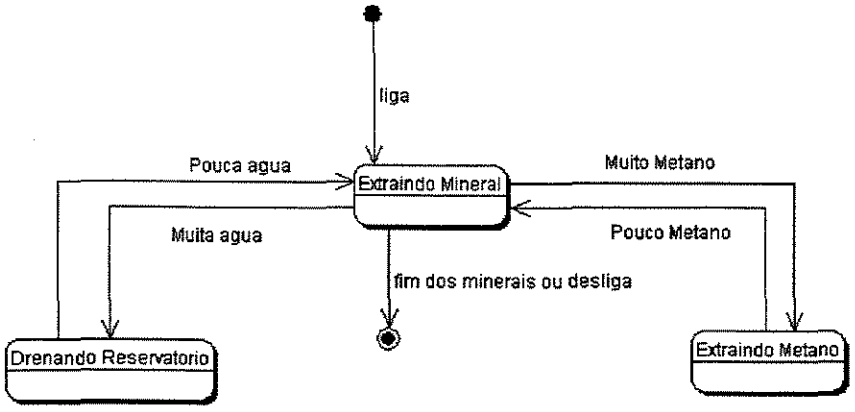

Figura 45 – Estados normais do Sistema de Mineração

A matriz de transição de estados correspondente ao diagrama de estados apresentado na Figura 45 encontra-se na Tabela 7.

Estados		Extraindo Minerais[1]	Extraindo Metano[2]	Drenando Reservatório[3]
MuitaAgua	ligar			
	desligar	[3]		
PoucaAgua	ligar			
	desligar			[1]
Muito Metano	ligar			
	desligar	[2]	[1]	

Tabela 7 – Matriz de Transição de Estados do Sistema de Mineração

6.2.5.1 Análise de Transições Inválidas de Estados

A matriz de transição de estados (Tabela 7) não está completa, consequentemente nem todos os eventos associados aos estados foram previstos. Alguns destes eventos implicariam em transições e estados excepcionais que, para um sistema confiável, deveriam ser previstos e especificados. Um exemplo de uma transição excepcional não prevista no diagrama de estados da Figura 45 é o nível de metano subir (sensor MuitoMetano ligar) durante o estado DrenandoReservatório. Diante desta situação, existem duas possibilidades: (1) interromper a drenagem do reservatório e retornar uma exceção avisando que uma situação inesperada ocorreu ou (2) uma solução mais robusta seria interromper a drenagem de água por alguns instantes e iniciar a extração de metano, que é prioritária nesta situação, e depois retomar a drenagem de água.

Uma análise semelhante deve ser feita ao considerar que o nível de água pode subir durante a extração de metano. O sistema deverá continuar extraindo metano

até que o nível de metano abaixe e em seguida deverá iniciar a extração de água ao invés de retornar a extração de minerais. Note que uma nova transição entre os estados DrenandoReservatorio e ExtraindoMetano deve ser criada e uma restrição adicionada ao evento de saída do estado ExtraindoMetano veja Figura 46.

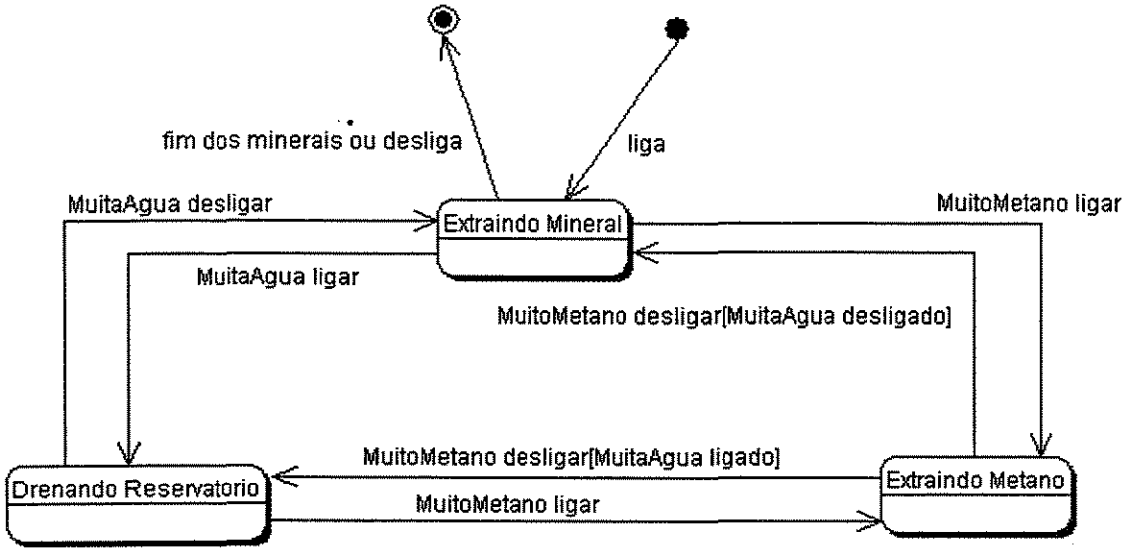

Figura 46 – Diagrama de Estados Robusto do Sistema de Mineração

A matriz de transição de estados correspondente ao diagrama de estão apresentada na Tabela 7 será revista para incluir o novo modelo de transição de estados da Figura 46 e os eventos de alteração dos estados dos sensores de fluxos. A nova matriz de transição de estados está completa e apresentada na Tabela 8.

Conforme foi explicado na seção 5.4.3, o símbolo | significa que determinado evento não traz mudança de estado e nem conseqüências maiores; símbolo X indica que determinado evento não deve ocorrer em determinado estado e DF significa falha de projeto caso o evento aconteça.

		Extraindo Minerais[1]	Extraindo Metano[2]	Drenando Reservatório[3]
MuitaAgua	ligar	[3]	|	DF
	desligar	DF	|	|
PoucaAgua	ligar	|	|	[1]
	desligar	|	|	DF
Muito Metano	ligar	[2]	DF	[2]
	desligar	DF	se muita água [3] senão [1]	DF
FluxoAgua	ligar	X	X	|
	desligar	X	|	|

FluxoAr	ligar	X	|	X
	desligar	X	|	|

Tabela 8 – Matriz de Transição de Estados Completa do Sistema de Mineração

1. Análise do estado **Extraindo Mineral**: Ao iniciar a extração de mineral os níveis de água e metano estão baixos, isto é, os sensores MuitoMetano e MuitaAgua estão desligados e sendo assim o evento *desligar* para estes dois sensores não ocorrerão em hipótese alguma, a não ser sob falha de projeto ou implementação. Quanto ao sensor de PoucaAgua, este poderá tanto ligar quanto desligar o que não afetará o estado corrente. Caso o sensor MuitaAgua ligue, o sistema deverá mudar para o estado Drenando Reservatório [3] e caso o sensor MuitoMetano ligue, o sistema deverá mudar para o estado Extraindo Mineral [2]. Durante a extração de mineral a bomba e exaustor estão desligados, portanto, não se considera que haverá mudança dos estados dos sensores de fluxo.
2. Análise do estado **Extraindo Metano**: Ao iniciar a extração de metano, o sensor MuitoMetano está ligado e, portanto o evento *ligar* não faz sentido para este sensor. Durante extração de metano, se o sensor MuitaAgua ligar o sistema continua extraindo metano mas sua próxima transição será para o estado DrenandoReservatorio [3]. Quando o sensor MuitoMetano desligar, a extração de metano será interrompida e, se o nível de água estiver baixo, o sistema voltará ao estado Extraindo Mineral [1]. Durante a extração de metano o sensor de fluxo de água não deve ter alteração no estado, caso ocorra, representa riscos de explosão da mina. O sensor de fluxo de ar vai ligar e desligar de acordo com o estado do exaustor de ar.
3. Análise do estado **Drenando Reservatório**: Ao iniciar a extração de água o sensor MuitaAgua está ligado, o PoucaAgua e o MuitoMetano desligados, portanto considera-se que o evento *ligar* para o MuitaAgua não ocorrerá e o evento *desligar* para MuitoMetano e PoucaAgua também não ocorrerão. Se o nível de metano alterar (sensor MuitoMetano ligar) o sistema interrompe a drenagem de água, começa a extração de metano, isto é, entra no estado Extraindo Metano [2]. Quando o nível de água do reservatório começar a baixar, o sensor MuitaAgua irá desligar mas isto ainda não é suficiente para interromper a extração. Só quando o sensor PoucaAgua ligar o sistema

deixa de extrair água para passar a extrair mineral novamente. Durante a extração de água o sensor de fluxo de ar não deve ter alteração no estado.

6.2.5.2 Situações Excepcionais decorrentes de transição de estados

A Figura 47 mostra que a transição entre dois estados passa por estados intermediários acarretando situações excepcionais durante a transição. O sistema começa em um estado ExtraindoMineral onde a bomba está desligada, e o nível de água está baixo (sensor MuitaAgua desligado). Assim que o nível de água subir (sensor MuitaAgua ligar), o sistema passa para o estado intermediário E1 onde a bomba continua desligada mas agora o nível de água está alto. Em seguida, a bomba será ligada e o sistema passa a drenar água (sub estado E2.1). Quando o nível de água atingir o mínimo (sensor PoucaAgua ligar), o sistema ainda está drenando o reservatório (sub estado E2.2) e em seguida a bomba será desligada, interrompendo a drenagem, e retornando para o estado inicial ExtraindoMineral.

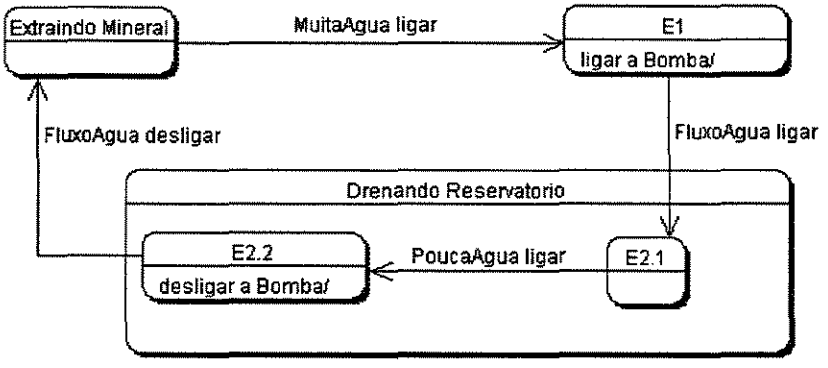

Figura 47– Fluxo normal de execução do caso de uso DrenarReservatorio

A Tabela 9 apresenta a situação dos componentes do sistema em cada um dos estados descritos.

Estado	Descrição do Estado				
	Bomba	FluxoAgua	MuitaAgua	PoucaAgua	MuitoMetano
Extraindo Mineral	Off	Off	Off	Off	Off
E1	Off	Off	On	Off	Off
E2.1	On	On	On / Off	Off	Off
E2.2	On	On	Off	On	Off

Tabela 9– Características dos Estados Válidos do Sistema de Mineração

Já a Figura 48 apresenta um diagrama dos estados normais e excepcionais pelos quais o sistema de mineração pode passar durante a mudança de estados. Este diagrama é uma descrição mais completa do apresentado na Figura 47 pois considera além das possibilidades de sucesso, os fracasso que podem levar o sistema a estados inconsistentes e por isto deve ser preferencialmente adotado.

Quando a bomba é ligada (estado E1), o sensor de fluxo de água pode ligar ou pode manter-se desligado. Caso o sensor de fluxo de água não ligue, o sistema entra em um estado excepcional (EE1) onde a bomba está ligada mas não existe fluxo de água. Neste estado excepcional o caso de uso BombaFalhou será executado visando recuperar o correto funcionamento da bomba. Consideremos agora que o sensor de fluxo de água tenha acusado o correto funcionamento da bomba. O sistema passa para o estado onde o reservatório está sendo drenado. Durante este estado, duas situações anormais podem acontecer: (1) sensor de fluxo de água desligar e o sistema entrar no estado excepcional EE1; (2) sensor de metano acusar um alto nível do gás no ambiente e o sistema entra no estado de falha EF1. No estado de falha EF1 a bomba será desligada e o caso de uso terminará de forma excepcional sem chances de recuperação.

Suponhamos que tudo tenha corrido como esperado durante a drenagem do reservatório e a bomba será desligada. Caso o sensor de fluxo de água indique que o fluxo cessou, o sistema volta para o estado inicial ExtraindoMineral. Caso contrário, o sistema entra no estado excepcional EE2 onde haverá uma nova tentativa de desligar a bomba. Caso a tentativa de recuperação não obtenha sucesso, o caso de uso terminará de forma excepcional.

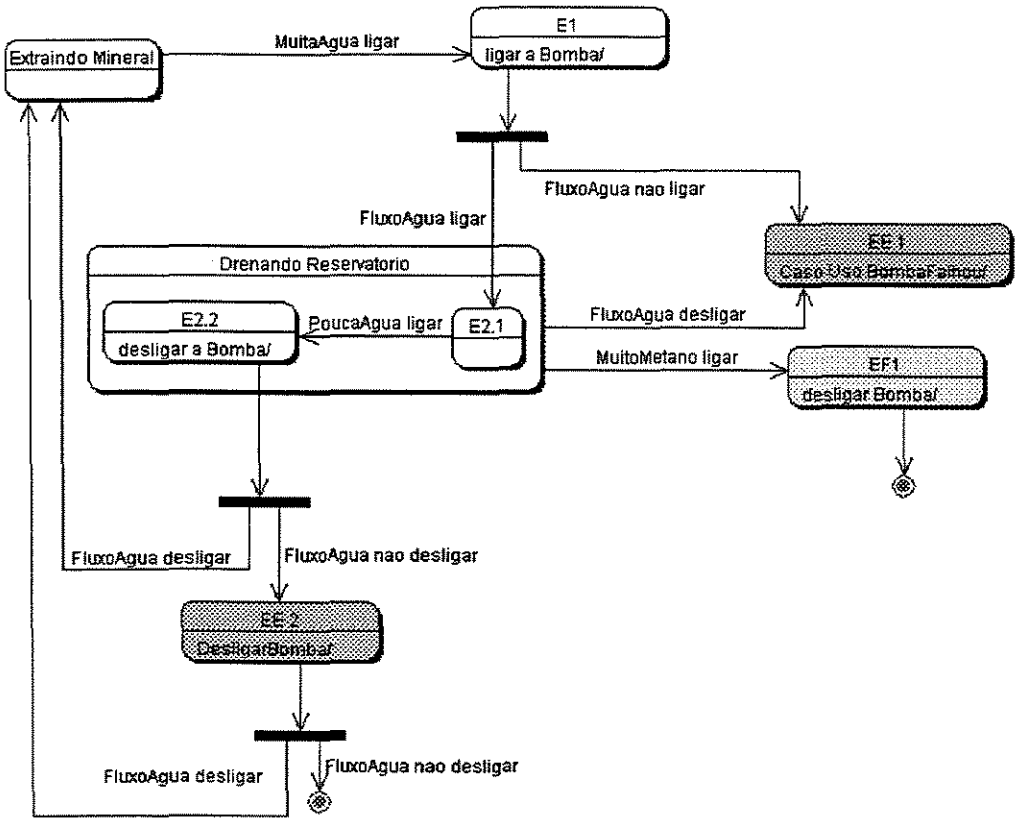

Figura 48 – Fluxo completo de execução do caso de uso DrenarReservatorio

A Tabela 10 apresenta as características dos estados anormais do sistema.

Estado	Descrição do Estado				
	Bomba	FluxoAgua	MuitaAgua	PoucaAgua	MuitoMetano
EE1	On	Off	On	Off	Off
EE2	Off	On	Off	On	Off
EF1	On	On	On / Off	Off	On

Tabela 10– Características dos Estados Excepcionais do Sistema de Mineração

Dado que os requisitos foram bem especificados, é o momento de definir os limites do sistema de mineração através do modelo de tipos e projeta-lo internamente. Este é um processo recursivo onde um tipo pode ser formado pela combinação de outros tipos que também necessitam ser especificados e projetados. Esta recursividade acontecerá no sistema de mineração como poderá ser visto a seguir.

6.3 Especificação dos Limites do Sistema de Mineração

Os limites são definidos através de modelos de tipos. Especificar um tipo envolve descrever seus componentes internos e seu comportamento externo através de sua interface pública. Sob um alto nível de abstração, o sistema de mineração pode ser decomposto em quatro grandes componentes como mostra a Figura 49. Os elementos são:

- **Interface com Operador**: representa o console onde as emergências são sinalizadas e onde o operador interage com o sistema ligando-o e desligando-o.
- **Sensores:** representam os sensores de níveis tais como MuitoMetano, MuitaAgua e PoucaAgua, além dos sensores de fluxos, FluxoAgua e FluxoAr.
- **Dispositivos de extração**: representam os equipamentos usados para extração de água, metano e minério, que são bomba, exaustor e braço mecânico respectivamente.
- **Estações de controle**: controladoras dos dispositivos de extração que implementam as três extrações identificadas durante a etapa de especificação de requisitos: extração de minerais, metano e água.

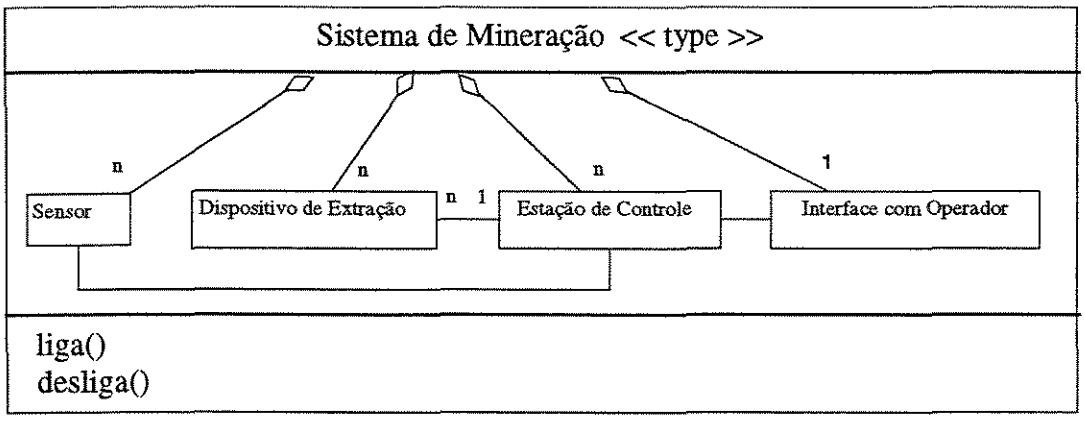

Figura 49 – O Tipo "Sistema de Mineração"

6.4 Projeto do Sistema de Mineração

Nesta seção, o sistema de mineração será projetado internamente segundo as três atividades determinadas pelo processo Catalysis: projeto arquitetural, projeto

das colaborações e projeto interno. Estas três atividades que compõem o projeto de sistemas são aplicadas recursivamente no Sistema de Mineração e seus componentes. Como foi especificado anteriormente, o sistema de mineração é composto por: uma *interface*, *sensores*, *dispositivos de extração* e *estação de controle*.

6.4.1 Projeto Arquitetural do Sistema de Mineração

Um sistema confiável deve ser decomposto em um conjunto de componentes tolerantes a falhas ideais organizados hierarquicamente como especificado na metodologia MDCE. Neste estudo de caso, os componentes do sistema de mineração foram organizados como mostra a Figura 50.

Nesta estrutura, as estações de controle requisitam serviços dos dispositivos de extração e recebe informações sobre alteração dos estados dos sensores.

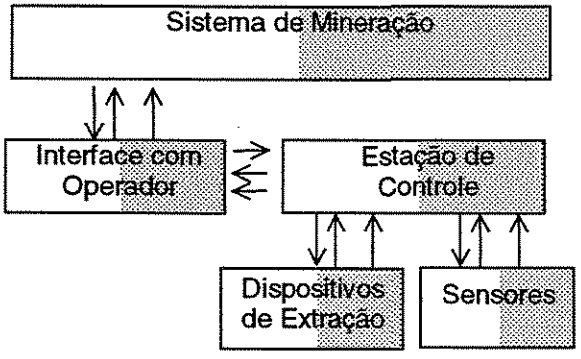

Figura 50 – Arquitetura do "Sistema de Mineração"

6.4.2 Projeto das Colaborações

O tipo especificado na seção 6.3 deve ser implementado por colaborações que descrevem como os componentes internos se interagem para prover o comportamento externo especificado. Na Figura 51 podemos observar que o operador inicia o sistema através do componente *Interface com Usuário* que por sua vez solicita à *Estação de Controle* que a extração se inicie. A estação de controle liga e desliga os *Dispositivos de Extração* e recebe informações sobre alterações nos estados dos *Sensores*.

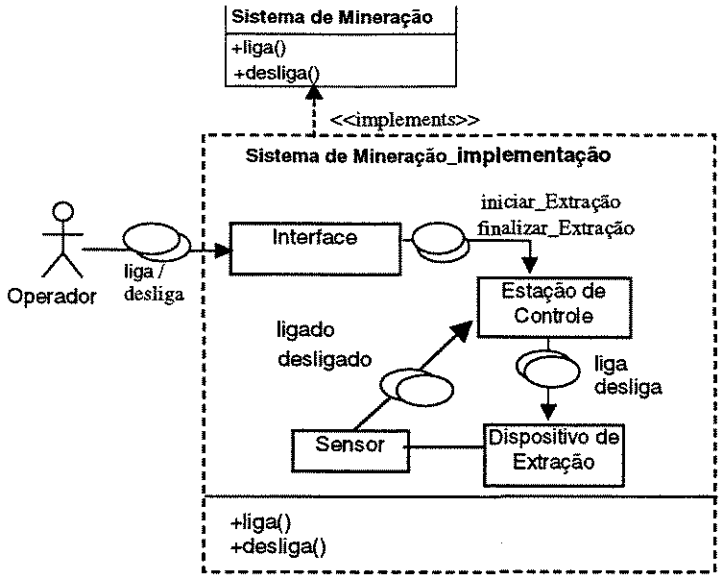

Figura 51 – Colaboração do SistemaDeMineração

Uma situação excepcional que pode ser identificada neste momento é o problema de perda de mensagens enviadas pelos sensores para a estação de controle. Se a estação de controle perder a mensagem que o sensor lhe enviou para comunicar alteração em seu estado, esta informação não poderá ser recuperada e o funcionamento do sistema fica comprometido. Por exemplo, considere que o sensor de nível alto de água (MuitaAgua) e o sensor de nível alto de metano (MuitoMetano) ligaram ao mesmo tempo e sinalizaram suas alterações para a estação de controle também ao mesmo tempo. Se a estação de controle recebeu apenas a mensagem do sensor MuitaAgua, a extração de água será iniciada em presença de um alto nível de metano no ambiente, o que representa risco de explosão da mina. Caso a estação de controle receba apenas a mensagem do sensor MuitoMetano, a situação é menos grave, mas apenas o excesso de metano será retirado podendo trasbordar o reservatório de água. A solução para este problema está na sincronização das mensagens dos sensores para a estação de controle.

6.4.3 Projeto do Sistema de Mineração

Este é o momento de definir internamente o tipo *Sistema de Mineração* através da especificação e projeto de seus componentes. A *Estação de Controle* é um componente do Sistema de Mineração que merece ser especificado e projetado pois trata-se de componente complexo formado internamente por outros três

componentes como será visto a seguir. A *Estação de Controle* será especificada na seção 6.4.4 e projetada na seção 6.4.5.

6.4.4 Especificação da Estação de Controle

A *Estação de Controle* é o componente do sistema de mineração responsável por garantir as condições adequadas para a extração de mineral, isto é, drenando a água do reservatório e expelindo o metano do ambiente quando os níveis de água e metano estiverem elevados comprometendo a extração de mineral.

Entretanto, a fim de obter componentes com responsabilidades mais específicas, a *Estação de Controle* é instanciada em três componentes menores como mostra a Figura 52: (1) *Controle da Bomba*, componente responsável pela extração de água; (2) *Controle do Exaustor*, componente responsável pela extração de metano do ambiente, (3) *Controle do Braço*, componente responsável por operar o braço mecânico que realiza a extração de minerais.

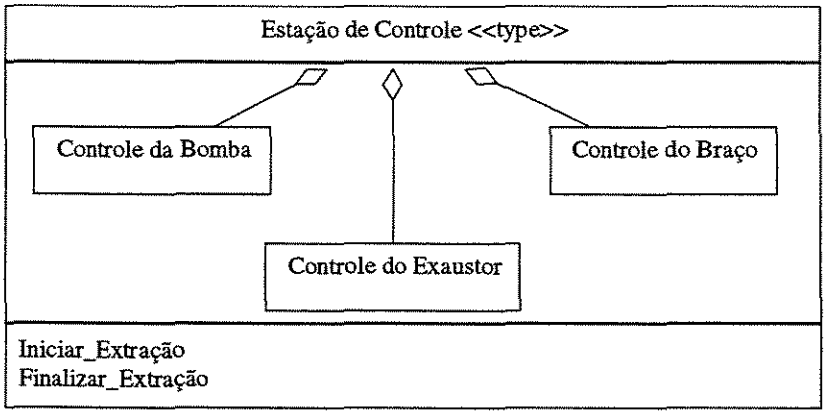

Figura 52 – Tipo "Estação de Controle"

A seguir o componente *Estação de Controle* será projetado para definir a organização e forma de interação entre seus componentes internos

6.4.5 Projeto da Estação de Controle

Como foi feito para o Sistema de Mineração na seção 6.4, o subsistema *Estação de Controle* será projetado da forma como definido pelo processo Catalysis, isto é, apresentando seu projeto arquitetural (seção 6.4.5.1), o projeto de suas colaborações (seção 6.4.5.2) e seu projeto interno (seção 6.4.5.3).

6.4.5.1 Projeto arquitetural

A estação de controle é um componente de mais alto nível que controla,

ativando e desativando, cada uma das estações de controle que o compõe como mostrado na Figura 53. Cada um dos elementos internos da *Estação de Controle* é um componente ideal tolerante a falha que mantém uma atividade excepcional, ou anormal, responsável pelo tratamento das situações excepcionais que eventualmente ocorram.

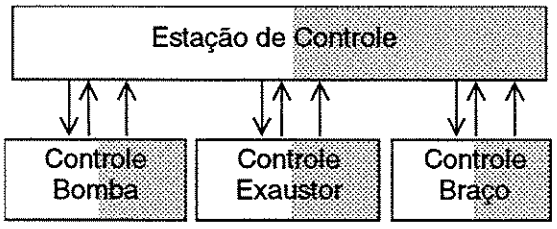

Figura 53 – Arquitetura do componente "Estação de Controle"

6.4.5.2 Projeto das colaborações

A colaboração que implementa o Tipo *Estação de Controle* está apresentada na Figura 54. O componente *Estação de Controle* inicia e finaliza suas atividades de acordo com as determinações do operador através do componente *Interface com Operador*. Quando a *Estação de Controle* é iniciada, ela repassará a mensagem de inicialização para uma de suas sub estações, dependendo da situação dos sensores MuitaAgua e MuitoMetano. Para ser mais precisa, quando o sensor MuitaAgua estiver ligado, a mensagem de inicia_Extração será enviada para o componente *Controle da Bomba*. Quando há alteração dos estados dos sensores, o componente *Estação de Controle* é avisado e repassa a mensagem para a sub estação que estiver ativa no momento. No caso, a *Estação de Controle* repassará a mensagem para o componente *Controle da Bomba* que interpretará a mensagem segundo sua especificação. Por exemplo, caso a mensagem seja que o sensor PoucaAgua ligou, ele cessará a drenagem do reservatório desligando a bomba. Caso a mensagem seja de que o sensor MuitoMetano ligou, esta é uma situação de risco que será sinalizada sob a forma de uma exceção. Cada controladora possui sua própria interpretação para as mesmas mensagens decorrente do estado do sistema no momento.

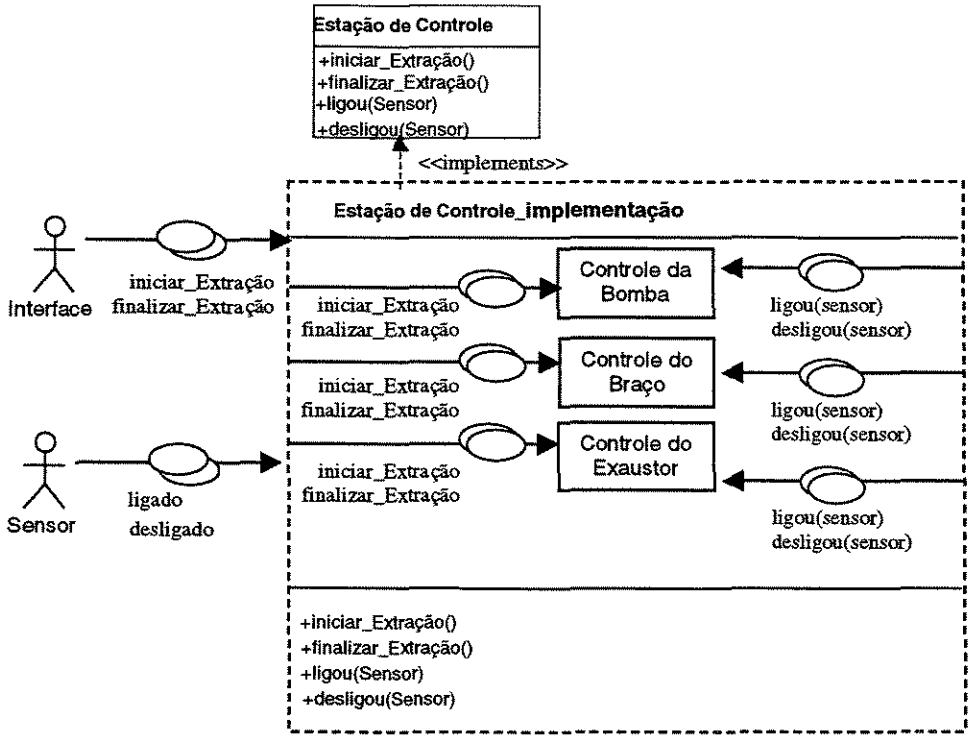

Figura 54 – Colaboração de EstaçãoDeControle

Esta colaboração está descrita sob a forma de ações com suas invariantes, pré e pós condições na Tabela 11. As exceções associadas a esta colaboração são propagandas pelos subcomponents: *Controle da Bomba* e *Controle do Exaustor*. Para simplificar nosso estudo de caso, consideremos que o componente *Controle do Braço* é confiável.

Comportamento Normal	Comportamento Excepcional

Comportamento Normal	Comportamento Excepcional
Ação(ControleBomba):iniciarExtração **Pre:** a extração de água vai ser iniciada quando o nível de água estiver elevado, o nível de metano baixo, a bomba desligada e o sensor FluxoAgua desligado Bomba==desligada e FluxoAgua==desligada e MuitaAgua==ligado e MuitoMetano==desligado **Pós:** a bomba é ligada e passa a existir fluxo de água Bomba==ligada e FluxoAgua==ligado e MuitaAgua==ligado e MuitoMetano==desligado	**Sinal:** uma exceção é propagada pelo componente ControleBomba; E_excBombaFalhou_DR? **Tratador:** o processo de extração deve ser interrompido e um alarme é enviado para o operador; E_excBombaFalhou_DR? implica (EstacaoControle(finalizarExtração) e E_excLançarAlarme_DR!) **Pós:** o processo de extração foi interrompido, o nível de água não está baixo e o nível de metano continua normal; Extração==desligada e Bomba==desligada e FluxoAgua==desligado e MuitoMetano==desligado e PoucaAgua=desligado
Ação(ControleBomba):finalizarExtração **Pre:** a extração de água é interrompida quando o nível de água está baixo, a bomba está ligada Bomba==ligada e PoucaAgua==ligado **Pós:** a bomba é desligada e o fluxo de água cessa Bomba==desligada e FluxoAgua==desligado	**Sinal:** uma exceção é propagada pelo componente ControleBomba; E_excBombaFalhou_DR? **Tratador:** o processo de extração deve ser interrompido e um alarme é enviado para o operador; E_excBombaFalhou_DR? implica (EstacaoControle(finalizarExtração) e E_excLançarAlarme_DR!) **Pós:** o processo de extração foi interrompido, o nível de água não está baixo e o nível de metano continua normal; Extração==desligada e Bomba==desligada e FluxoAgua==desligado e MuitoMetano==desligado e PoucaAgua=desligado

Comportamento Normal	Comportamento Excepcional
Ação(ControleExaustor):iniciarExtração **Pre:** a extração de metano vai ser iniciada quando o nível de metano está elevado, o exaustor está desligado, o sensor FluxoAr está desligado Exaustor==desligado e FluxoAr==desligado e MuitoMetano==ligado **Pós:** o exaustor é ligado e passa a existir fluxo de ar Exaustor==ligado e FluxoAr==ligado	**Sinal:** uma exceção é propagada pelo componente ControleExaustor E_excExaustorFalhou_EMe? **Tratador:** o processo de extração deve ser interrompido e um alarme é enviado para o operador; E_excExaustorFalhou_EMe? implica (EstacaoControle(finalizarExtração) e E_excLançarAlarme_DR!) **Pós:** o processo de extração foi interrompido, o nível de metano não está baixo e apesar do exaustor estar desligado, não se tem certeza se o fluxo de ar cessou. Extração==desligada e Exaustor==desligado e FluxoAr==? e MuitoMetano==ligado
Ação(ControleExaustor):finalizarExtração **Pre:** a extração de metano vai ser finalizada quando o nível de metano se normalizar e o exaustor estava ligado Exaustor==ligado e MuitoMetano==ligado **Pós:** o exaustor é desligado e o fluxo de ar cessa Exaustor==desligado e FluxoAr==desligado	**Sinal:** uma exceção é propagada pelo componente ControleExaustor E_excExaustorFalhou_EMe? **Tratador:** o processo de extração deve ser interrompido e um alarme é enviado para o operador; E_excExaustorFalhou_EMe? implica (EstacaoControle(finalizarExtração) e E_excLançarAlarme_DR!) **Pós:** o processo de extração foi interrompido, o nível de metano não está baixo e apesar do exaustor estar desligado, não se tem certeza se o fluxo de ar cessou. Extração==desligada e Exaustor==desligado e FluxoAr==? e MuitoMetano==ligado
Ação(ControleBraco):iniciarExtração **Pre:** a extração de mineral vai ser iniciada quando o nível de metano e agua estiverem sob controle e o braço estiver desligado Braco==desligado e MuitaAgua==desligado e MuitoMetano==desligado **Pós:** o braço é ligado Braco==ligado	Componente confiável

Comportamento Normal	Comportamento Excepcional
Ação(ControleBraco):finalizarExtração **Pre:** a extração de mineral vai ser finalizada quando o nível de metano ou água estiverem elevado e o braço estiver ligado Braco==ligado e (MuitaAgua==ligado e/ou MuitoMetano==ligado) **Pós**: o braço é desligado Braco==desligado	Componente confiável

Tabela 11 – Comportamento da colaboração EstaçãoDeControle

A Figura 55 ilustra um cenário normal da colaboração da Figura 54 e a Figura 56 um cenário excepcional onde a bomba não funciona corretamente durante a extração de água.

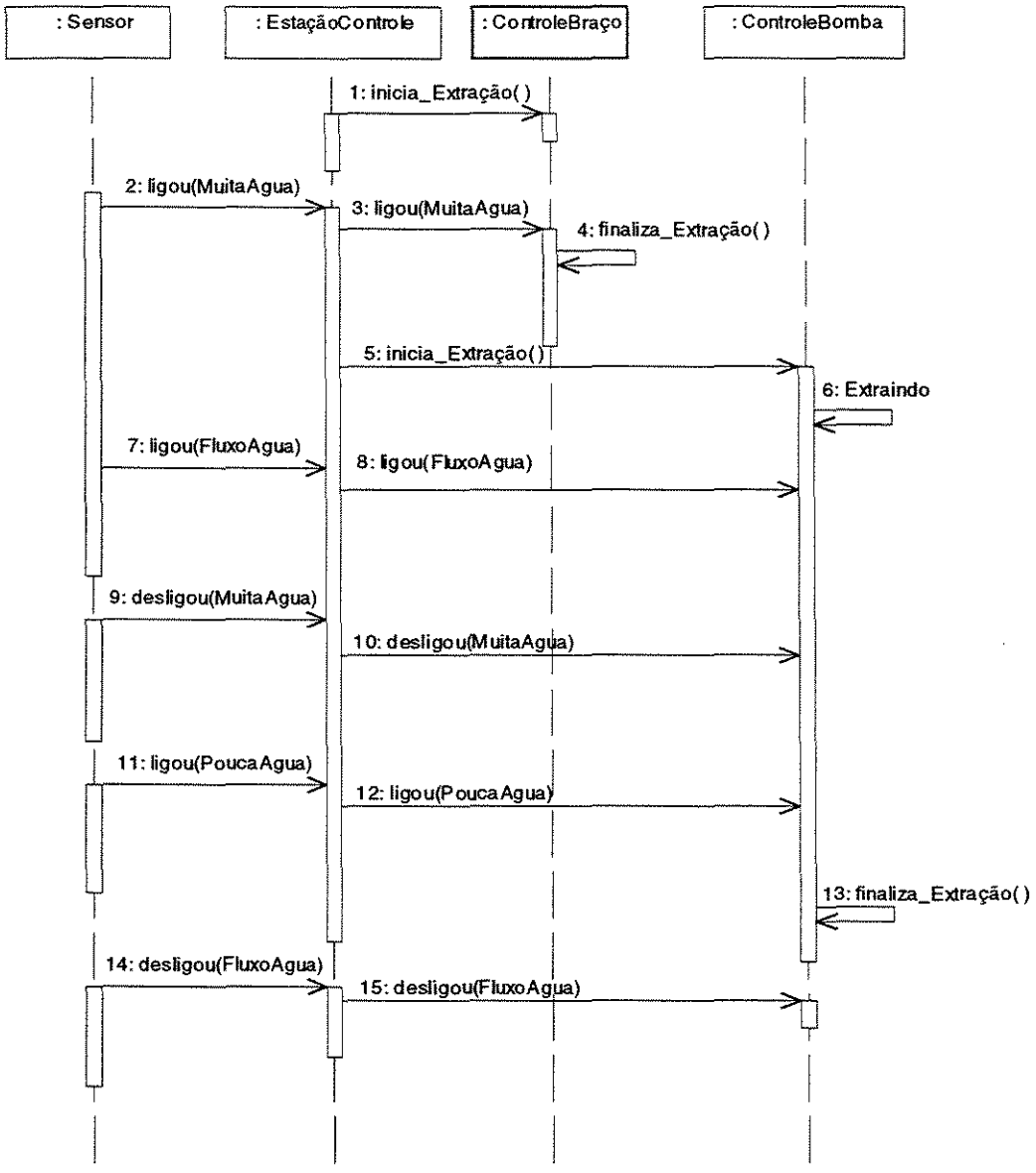

Figura 55 – Cenário normal da colaboração EstaçãoDeControle

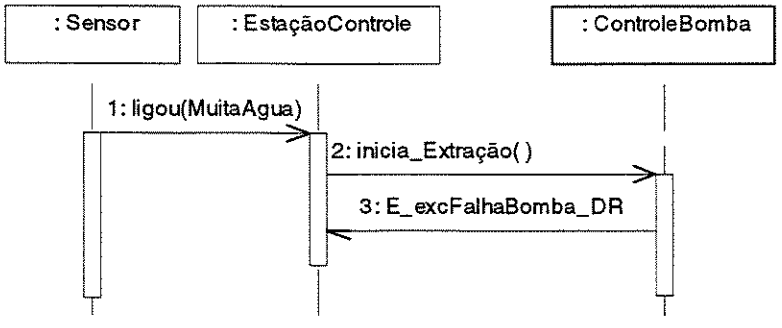

Figura 56 – Cenário excepcional da colaboração EstaçãoDeControle

6.4.5.3 Projeto Interno da Estação de Controle

Como dissemos anteriormente, cada sub componente da estação de controle interpreta uma mensagem de acordo com o estado do sistema no momento. Por exemplo, quando a estação de *Controle da Bomba* estiver ativa, é sinal de que a bomba está ligada e receber uma mensagem do tipo sensor MuitoMetano ligou é interpretada como se uma situação de risco para o ambiente. Por outro lado, se esta mesma mensagem for enviada a estação de *Controle do Braço* ela significa apenas um sinal de que a extração de mineral deve ser interrompida para que a extração de metano seja ativada.

Esta situação é adequada para a aplicação do padrão de projeto *State* [Gamma+95]. Sendo assim, cada subestação de controle foi implementada por uma classe derivada de EstadoEstacaoControle, que representa os estados da Estação de Controle, isto é, controlando a bomba ou controlando o exaustor ou controlando o braço.

A Figura 57 ilustra o modelo do projeto interno do componente *Estação de Controle*. Neste momento, as exceções lançadas pelas classes de controle aparecem explícitas no modelo sob a forma de classes estereotipadas por <<exception>> como sugerido pela metodologia MDCE. O componente *Estação de Controle* é formado por todas as classes representadas abaixo, inclusive pela classe Excepcional_EstaçãoControle que mantém os tratadores para as exceções propagadas pelas classes ControleBomba e ControleExaustor.

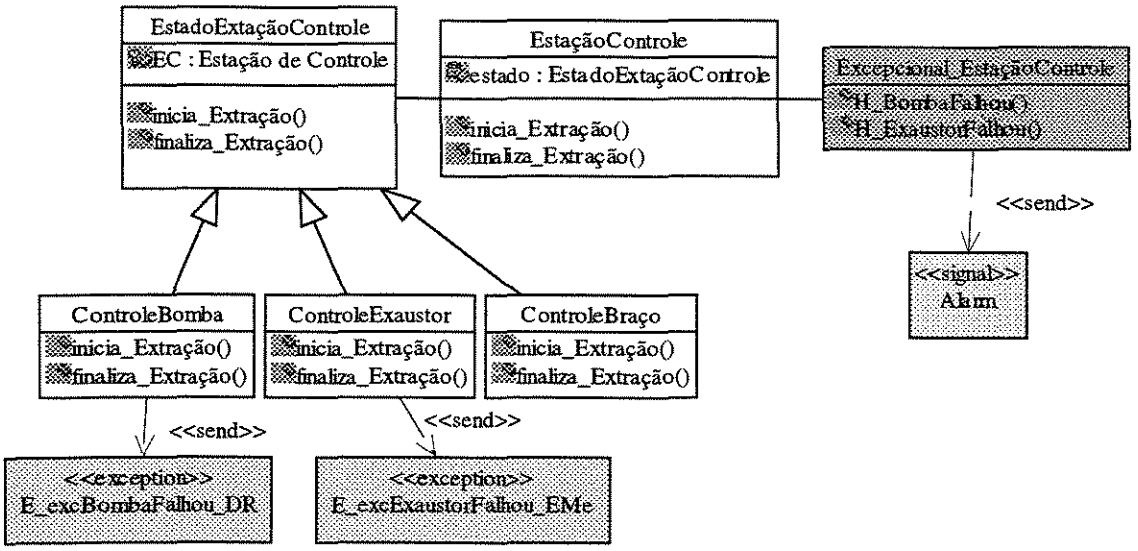

Figura 57 – Projeto Interno do Componente EstaçãoControle

Controle da Bomba, *Controle do Exaustor* e *Controle do Braço* são componentes complexos que também devem ser especificados e projetados internamente. A seção 6.4.5.4 se destina a especificação e projeto do componente *Controle da Bomba.* Os componentes *Controle do Braço* e *Controle do Exaustor,* assim como o componente *Controle da Bomba,* deveriam ser especificados e projetados internamente, mas por questões de espaço e legibilidade do estudo de caso não o fizemos.

6.4.5.4 Especificação e Projeto do *Controle da Bomba*

Controle da Bomba é um tipo simples que possui duas operações: uma de inicialização e outra de finalização da extração. Dado a simplicidade de especificação deste tipo e arquitetura do sub sistema, passaremos direto ao projeto das colaborações que implementam o controle da bomba para identificação das situações excepcionais as quais está sujeito.

A Figura 58 ilustra a colaboração DrenarReservatorio que implementa o tipo ControleBomba. A colaboração é formada por ações de ligar e desligar o dispositivo de extração de água, a bomba. Estas ações internas são desencadeadas pelas mensagens externas de iniciar ou finalizar a extração ou ainda pelas alterações dos estados dos sensores, tais como sensor PoucaAgua ligou que indica que a bomba pode ser desligada e a colaboração DrenarReservatorio finalizada.

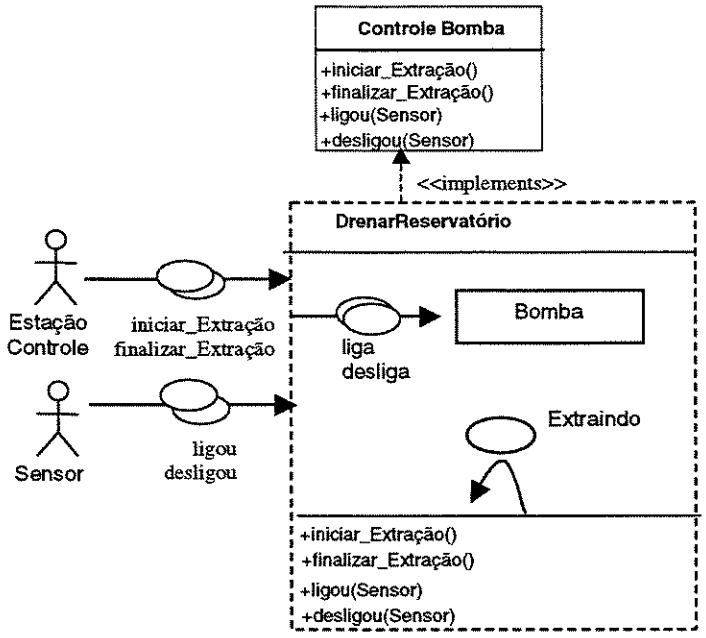

Figura 58 – Colaboração DrenarReservatório

Os comportamentos normais e excepcionais das ações internas à colaboração DrenarReservatóiro estão descritas na Tabela 12.

Comportamento Normal	Comportamento Excepcional
Ação(Bomba)::ligar **Pre:** a bomba está desligada, não há fluxo de água Bomba==desligada e FluxoAgua==desligado **Pós:** a bomba é ligada e há fluxo de água Bomba==ligada e FluxoAgua==ligado	**Violação da pós-condição** **Sinal:** A bomba foi ligada mas não existe fluxo de água (Bomba==ligada e FluxoAgua==desligado) implica I_excLigarBomba_DR! **Tratador:** tentar novamente se não conseguir, desligar a bomba e lançar exceção; Bomba(ligar); se FluxoAgua==desligado implica Bomba(desligar) e E_excBombaFalhou_DR **Pós:** bomba e fluxo de água ligados Bomba==ligada e FluxoAgua==ligado

Comportamento Normal	Comportamento Excepcional
Ação(Bomba)::desligar **Pre:** a bomba está ligada e há fluxo de água Bomba==ligada e FluxoAgua==ligado **Pós:** a bomba é desligada e o fluxo de água cessa Bomba==desligada e FluxoAgua==desligado	**Violação da pós-condição** **Sinal**: A bomba foi desligada mas o fluxo de água ainda continua (Bomba==desligada e FluxoAgua==ligado) implica I_excDesligarBomba_DR! **Tratador:** lançar exceção E_excBombaFalhou_DR! **Pós:** bomba desligada e fluxo de água ligado Bomba==desligada e FluxoAgua==ligado
Ação()::Extraindo **Pre:** a bomba está ligada e há fluxo de água, o nível de água está alto Bomba==ligada e FluxoAgua==ligado e MuitaAgua==ligado **Inv.:** nível de metano está baixo e existe fluxo de água e não existe fluxo de ar MuitoMetano == desligado e FluxoAgua== ligado e FluxoAr==desligado	**Violação da invariante** **Sinal**: o nível de metano subiu (Bomba==ligada e FluxoAgua==ligado e MuitoMetano== ligado) implica I_excMuitoMetano_DR! **Tratador:** interromper a drenagem de água e iniciar extração de metano; Bomba(desligar) **Pós**: bomba desligada e sensor de muito metano ligado Bomba==desligada e FluxoAgua == desligado e MuitoMetano==ligado
Pós: a bomba está ligada, há fluxo de água mas o nível de água está baixo Bomba==ligada e FluxoAgua==ligado e PoucaAgua==ligada	**Violação da invariante** **Sinal**: sensor de fluxo de água desligou durante a extração (Bomba==ligada e FluxoAgua==desligado) implica I_excBombaFalhou_DR! **Tratador:** tentar religar a bomba se não conseguir, desliga-la e lançar exceção; Bomba(ligar); se FluxoAgua==desligado implica Bomba(desligar) e E_excBombaFalhou_DR! **Pós**: bomba e fluxo de água ligados Bomba==ligada e FluxoAgua==ligado

Comportamento Normal	Comportamento Excepcional
	Violação da invariante **Sinal**: sensor de fluxo de ar ligou durante a extração de água (Bomba==ligada e FluxoAgua==ligado e FluxoAr==ligado) implica I_DF_DR! **Tratador:** desligar a bomba e lançar exceção; Bomba(desligar) e E_excDF_DR! **Pós**: bomba e fluxo de água desligados e fluxo de ar ligado Bomba==desligada e FluxoAgua==desligado e FluxoAr==ligado

Tabela 12 – Ações da colaboração DrenarReservatório

Internamente o componente *ControleBomba* é formado por três tipos de classes: (1) uma classe que implementa sua funcionalidade normal do componente: a classe ControleBomba; (2) uma classe responsável pelo tratamento das situações excepcionais: a classe Excepcional_ControleBomba; (3) exceções lançadas internamente (I_excLigarBomba_DR, I_excBombaFalhou_DR, I_excDesligarBomba_DR e I_excDF_DR) e externamente (E_excBombaFalhou_DR, E_excDF_DR) por este componente.

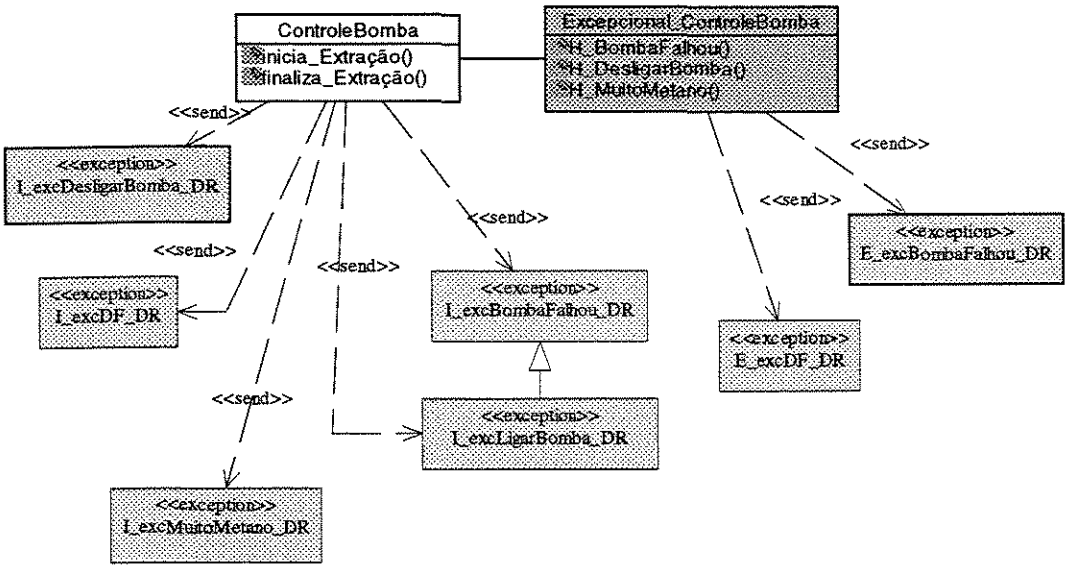

Figura 59 – Projeto Interno do Componente ControleBomba

Note que a classe de exceção I_excLigarBomba_DR foi modelada como subclasse de I_excBombaFalhou_DR pois o tratamento destas exceções é realizado da mesma forma. Sendo assim, basta existir um tratador para a exceção de nível

superior que a suas descendentes serão capturadas e tratadas de forma semelhante.

6.5 Detalhamento do Modelo de Tipos

A metodologia MDCE prevê que o modelo de tipos deve ser detalhando para manter a especificação completa de sua interface publica, incluindo as respostas excepcionais que podem ser retornadas. Sendo assim, apresentaremos como o tipo *Controle da Bomba* ficaria após o projeto das colaborações e identificação das exceções pelo tipo lançadas.

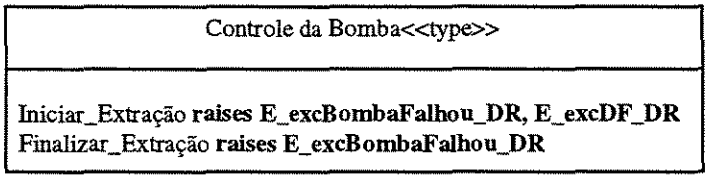

Figura 60 – Detalhamento do tipo *Controle da Bomba*

6.6 Detalhamento do Projeto Arquitetural

A arquitetura do Sistema de Mineração projetada na seção 6.4.1 deve ser detalhada para manter as exceções e tratadores explícitos em seu projeto e obter uma visão de alto nível do caminho percorrido pelas exceções e os componentes afetados por elas.

A Figura 61 apresenta o detalhamento da arquitetura, incluindo as exceções lançadas por cada componente interna e externamente além dos tratadores que cada componente deve manter para as exceções que lhe atingem.

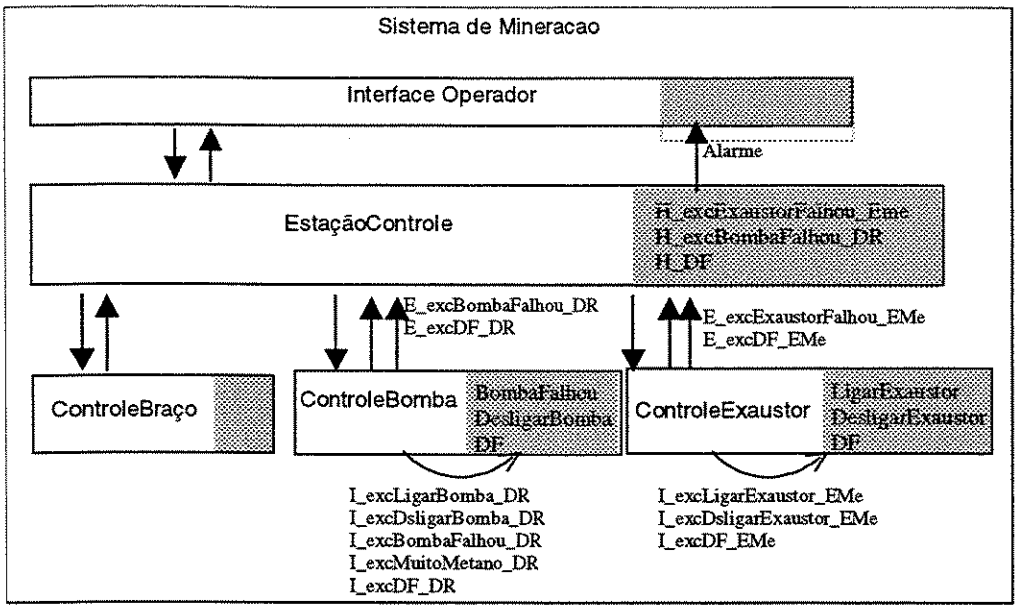

Figura 61 – Detalhamento da arquitetura do Sistema de Mineração.

6.7 Implementação

O sistema de mineração foi implementado na linguagem Java conforme especificado e projetado neste capítulo. A associação entre as classes normais e excepcionais de cada componente foi realizada da seguinte forma

```
//associação com o objeto excepcional
protected Excepcional_ControleBomba exceptional;

private void ligar(Dispositivo dispositivo) {
        try {
                dispositivo.ligar();
        }
        catch (I_excBombaFalhou_DR e) {
                exceptional.H_BombaFalhou();
        }
 }
```

Para verificar o projeto, implementação e tolerância a falhas do Sistema de Mineração, projetamos e implementamos um simulador de ambiente e um pequeno injetor de falhas que estão descritos em detalhes no Apêndice B.

Foi possível comprovar a afirmativa de que mais de dois terços do código de

um sistema confiável é dedicado à detecção e tratamento das exceções [Cristian89]. A implementação de um Sistema de Mineração tolerante a falhas mantém 27 classes e 669 linhas de código enquanto a implementação de um Sistema de Mineração não tolerante a falhas possui apenas 4 classes e 200 linhas de código.

6.8 Resumo

Neste capítulo exemplificamos a aplicação da metodologia de construção de sistemas confiáveis através do estudo de caso do Sistema de Mineração. O Sistema de Mineração é composto de uma interface com o operador, sensores, dispositivos de extração e uma estação de controle. A *Estação de Controle* é um componente complexo composto de três componentes menores e com funções mais específicas: o *Controle da bomba*, o *Controle do exaustor* e o *Controle do braço mecânico*. O processo Catalysis foi então aplicado recursivamente sobre o componente *Estação de Controle* e sobre seus sub componentes.

Este estudo de caso foi suficiente para mostrar que situações excepcionais são identificáveis e devem ser modeladas desde a especificação de requisitos e se estender por todo o restante do projeto. O projeto das colaborações é de fundamental importância para que as exceções de interação possam ser identificadas e um eficiente modo de recuperação projetado. O projeto arquitetural, quando mantém a representação de exceções e seus tratadores, é uma poderosa ferramenta para análise de alto nível do impacto das exceções sobre o sistema como um todo. O modelo de tipos deve manter na descrição de sua interface as exceções que eventualmente podem ser por ele lançadas.

Capítulo 7

Conclusões e Trabalhos Futuros

Apesar das técnicas de tratamento de exceções estarem bem difundidas, ainda existem muitos problemas em aplicá-las na prática. Os projetistas não estão suficientemente preparados para definir e tratar o comportamento excepcional de um sistema de software. Desta forma, o projeto das exceções e tratadores acaba sendo adiado para as etapas finais de desenvolvimento, realizados de acordo com a intuição do projetista, resultando em um complexo modelo de exceções e tratadores pouco eficientes.

A utilização de uma metodologia que auxilie a identificação e representação das exceções e projetos eficientes de tratadores é de fundamental importância para o sucesso do projeto e construção de sistemas que realmente sejam confiáveis. Algum cuidado deve ser tomado para que tal metodologia não se torne um procedimento burocrático e, assim, um *overhead* inaceitável na entrega do produto final. Além disto, técnicas de tolerância a falha se baseiam em replicação de elementos críticos do sistema podem afetar os custos e tornar sua aplicação impraticável. O importante é prever os custos e prazos adicionais de um sistema confiável desde o primeiro momento para evitar surpresas desagradáveis.

Este trabalho apresentou uma sistemática para identificação, representação, projeto e implementação do comportamento excepcional de um sistema sob a forma de uma metodologia que chamamos de MDCE – Metodologia para Definição do Comportamento Excepcional. A metodologia MDCE cobre as etapas básicas de construção de um sistema de software, isto é, a especificação de requisitos, projeto e implementação. Sendo assim trata-se de uma metodologia genérica que pode ser aplicada a vários processos de desenvolvimento existentes na literatura. Neste trabalho realizamos sua aplicação ao processo Catalysis e exemplificamos através do estudo de caso do sistema de mineração.

MDCE possui duas regras básicas: (1) manter uma clara separação entre a especificação, modelagem e implementação das atividades normais e excepcionais do sistema durante todo o processo de construção. Esta separação facilita o entendimento e manutenção do sistema além de manter sua complexidade sobre controle. (2) Manter a preocupação com o *comportamento excepcional* do sistema desde a etapa de especificação de requisitos. Esta atitude traz diversos benefícios dentre os quais podemos citar: (i) possibilita análise da complexidade adicionada pela inclusão do comportamento excepcional e assim possibilita prever novos prazos e custos para a confecção do sistema; (ii) direciona o projeto do *comportamento excepcional* pelas etapas subseqüentes do processo possibilitando a identificação de erros sem a perda do contexto e conseqüentemente tratando-os de forma mais eficiente; (iii) informações obtidas em uma fase indicarão restrições para as etapas seguintes; (iv) os esforços de garantia da confiabilidade estarão distribuídos em todo o processo, garantindo um sistema confiável de melhor qualidade.

7.1 Contribuições

Como contribuições deste trabalho ressaltamos:

1. **A metodologia MDCE:** A principal contribuição deste trabalho foi a proposta de uma metodologia de definição do comportamento excepcional de sistemas confiáveis que mantém uma sistemática para identificação, representação, projeto e implementação das exceções e seus tratadores. Tal metodologia foi chamada de MDCE – Metodologia para Definição do Comportamento Excepcional - e seus objetivos, princípios e atividades estão descritos no Capítulo 3.
2. **Aplicação de MDCE ao processo Catalysis**: A aplicação de MDCE ao processo Catalysis foi realizada através da adaptação de fases que são específicas do processo Catalysis para que estas também mantenham a representação do comportamento excepcional em seus modelos e atividades. O novo processo Catalysis, que mantém a preocupação com o comportamento excepcional em todas as suas atividades de construção de sistemas confiáveis pode ser encontrado no Capítulo 4.
3. **Extensões dos modelos da UML:** Propusemos extensões da linguagem UML para representação do comportamento excepcional através da

criação de novos estereótipos com restrições e semântica adicionais. Além disto, mostramos como o comportamento excepcional deve estar representado de forma explicita nos diagramas de modelagem dinâmica da UML, tais como diagramas de casos de uso, atividades, estados, colaboração e seqüência. As extensões da linguagem UML podem ser encontradas no Capítulo 5.

4. **Exemplo da utilização do processo Catalysis:** Exemplificamos como aplicar a teoria apresentada para construção do comportamento excepcional, usando o processo Catalysis, através de um estudo de caso do Sistema de Mineração. Este estudo de caso possibilitou verificar a factibilidade do novo processo Catalysis e está descrito no Capítulo 6.
5. **Automatização de Tarefas através da Ferramenta Rose**: A metodologia MDCE adiciona complexidade na construção de sistemas computacionais uma vez que mantém o tratamento das atividades excepcionais juntamente com as atividades normais. Desta forma, a necessidade de ferramentas que automatizem partes da construção do sistema, ou auxilie no controle da complexidade e verificação da consistência dos modelos, se tornam ainda mais importantes. A ferramenta Rose [Rose] é amplamente usada para modelagem de sistemas orientados a objetos e possui uma interface que permite estende-la e gerar scripts que automatizam atividades, dentre outras coisas. Construímos *scritps* no Rose para automatizar a construção de algumas classes e exceções. O projeto e implementação destes *scripts* estão descritos no Apêndice A e seu código fonte pode ser encontrado em http://www.ic.unicamp.br/~ra995478/
6. **Simulador do Sistema de Mineração**: No Apêndice B apresentamos o projeto de um simulador para o sistema de mineração onde estão presente os sensores, os dispositivos de extração, o reservatório de água, tudo como especificado no exemplo. O objetivo era identificar novas situações excepcionais durante sua implementação na linguagem Java. O projeto e implementação deste simulador podem ser encontrados em http://www.ic.unicamp.br/~ra995478/

Publicações

Este trabalho resultou em três publicações em conferências nacionais e

internacionais. São elas:

- Explicit Representation of Exception Handling in the Development of Dependable Component-Based Systems

  Autores: Gisele R. M. Ferreira, Cecília M. F. Rubira e Rogério de Lemos

  Conferência: High Assurance Software Engineering -HASE'2001- Boca Raton – Flórida – USA, Outubro 2001
- Designing Reliable, Robust and Reusable Components with Java Exceptions

  Autores: Gisele R. M. Ferreira e Lucas C. Ferreira

  Conferência: III Simpósio de Segurança em Informática – SSI'2001 - São José dos Campos – SP – Brasil, Outubro 2001
- Tratamento de Exceções no Desenvolvimento de Software Confiável baseado em Componentes

  Autores: Gisele R. M. Ferreira e Cecília M. F. Rubira

  Conferência: Workshop de Testes e Tolerância a Falhas - WTF'2000- Curitiba – Paraná - Brasil, Julho 2000

As cópias destes trabalhos encontram-se no Apêndice C ou disponíveis no endereço http://www.ic.unicamp.br/~ra995478/publicacoes.html

7.2 Trabalhos Futuros

Podemos propor algumas linhas de pesquisa como extensões e melhorias do trabalho apresentado nesta dissertação.

Em primeiro lugar está a necessidade de formalização do modelo de caso de uso para remover possíveis ambigüidades e inconsistências. Casos de uso podem ser descritos de várias formas dependendo do contexto do projeto. Estas formas variam desde descrições informais até uma descrição sistemática, detalhada, completa e formal do caso de uso [Cockburn01]. Em alguns projetos, tanto cuidado ao descrever um caso de uso pode representar perda de tempo e dinheiro. Entretanto, quanto se lida com sistemas grandes ou sistemas confiáveis, estes cuidados extras podem significar economia de custos e maior garantia da confiabilidade destes sistemas. Nestes casos, os casos de uso devem ser o mais completo possível, além de formalizados para evitar ambigüidades e desentendimentos entre os projetistas.

Uma outra necessidade seria aplicar a metodologia MDCE ao Processo Unificado devido à popularidade deste processo, tanto no mercado comercial quanto na comunidade acadêmica. O Processo Unificado consiste de várias interações que são divididas em cinco atividades: especificação dos requisitos, análise, projeto, implementação e testes. A metodologia MDCE mantém as atividades de especificação dos requisitos, projeto e implementação. Sendo assim, aplicar MDCE ao Processo Unificado se resume em definir como o comportamento excepcional deve ser modelado durante a *análise* e ainda como ele pode servir de base para os *testes* de tolerância a falhas. Existe ainda a necessidade de estabelecer quais seriam os recursos de pessoal necessários para realização das novas atividades propostas para definição do comportamento excepcional.

Segundo a UML, exceções são representadas como classes organizadas hierarquicamente. Esta estrutura possibilita construção de exceções com semânticas adicionais além de possibilitar construção de tratadores mais genéricos que podem ser empregados tanto no tratamento de uma exceção quanto de suas especializações. No entanto, algumas linguagens de programação, como por exemplo Eiffel [Meyer92], não permitem este tipo de construção de exceções. Desta forma, fica a necessidade de estudar como este projeto de exceções e tratadores poderia ser transportando para uma linguagem tipo Eiffel. Ou então, uma outra alternativa seria encontrar um projeto de exceções e tratadores mais genérico e aplicável a um conjunto maior de linguagens de programação.

Destacamos ainda a continuação deste trabalho: (1) geração de novos *scripts* na ferramenta Rose para automatização de outras atividades e também para verificação da consistência do modelo no que diz respeito ao comportamento excepcional. Por exemplo, um script que verifique se para toda exceção existe um tratador no módulo cliente do método que a lança seria de grande valia. (2) Melhoria do simulador do sistema de mineração e análise quantitativa em número de linhas de código dedicadas à identificação de falhas, propagação de exceções e tratamento das exceções.

Capítulo 8

Referências Bibliográficas

[Booch98] Grady Booch. *The Visual Modeling of Software Architecture for the Enterprise* - Rational Software Coorporation, 1998.

[Avizienes85] Algirdas Avizienes *The N-Version Approach to Fault-Tolerant System* - IEEE Trans. Software Engineering, vol 11, no 12, 1985

[Avizienes97] Algirdas Avizienes *Toward Systematic Design of Fault-Tolerant Systems* – IEEE Computer – 1997

[Booch+99] Grady Booch, Ivar Jacobson e James Rumbaugh. *The Unified Modeling Language User Guide* - Addison-Wesley, Reading -MA, 1999.

[Charles99] Charles Richter *Designing Flexible Object-Oriented Systems with UML* - MacMillan Technical Publishing 1999

[Cheesman+01] John Cheesman e John Daniels *UML Components: A Simple Process for Specifying Component-Based Software* - Addison-Wesley 2001

[Cockburn01] Alistair Cockburn *Writting Effective Use Cases* Addison-Wesley 2001

[Corba] *OMG CORBA - The Common Object Request Broker Architecture.* http://www.corba.org

[Cristian89] Flaviu Cristian *Exception Handling* - Dependability of Resilient Computers (ed. T. Anderson), pp.68-97, Blackwell Scientific Publications, 1989.

[Dellarocas+98] Chrysanthos Dellarocas e Mark Klein *A Knowledge-Base Approach for Handling Exceptions in Business Processes* - Workshop on Information Technologies and Systems (WITS'98), Helsinki, Finlândia, Dezembro 1998

[Desmond98] Desmond d'Souza e Alan C. Will. *Objects, Components and Frameworks with UML. The Catalysis Approach* Addison-Wesley 1998

[Ferreira+01a] Gisele R. M. Ferreira e Lucas C. Ferreira *Designing Reliable, Robust and Reusable Components with Java Exceptions* – III Simpósio de Segurança em Informática, São José dos Campos-SP, Outubro 2001

[Gamma+95] Erik Gamma, R. Helm, R. Johnson e J. Vlissides *Design Patterns – Elements of Reusable Object-Oriented Software* Addison-Wesley Publishing Company, 1995

[Garcia+99] Alessandro Garcia, Delano Beder e Cecilia Rubira *An Exception Handling Mechanism for Developing Dependable Object-Oriented Software based on Meta-level Approach* - Software Reliability Engineering, USA 1999

[Hatton97] L. Hatton *N-Version Design versus One Good Version* - IEEE Software, vol 14 no 6 Nov./Dec. 1997

[Humphrey95] Watts S Humphrey *A discipline for Software Engineering* Addison Wesley 1995

[Humphrey99] Watts S Humphrey *Introduction to Team Software Process* Addison Wesley 1999

[Jacobson+99] Ivar Jacobson, James Rumbaugh e Grady Booch *Unified Software Development Process* - Addison-Wesley, Reading -MA, 1999.

[Jacobson92] Ivair Jacobson *Object-Oriented Software Engineering - A Use Case Driven Approach* - Addison-Wesley 1992

[Kanoun01] Karama Kanoun *Real-World Design Diversity: A Case Study on Cost* - IEEE Software, vol 18, No. 4, Jul./Aug. 2001

[Kruchten00] Philippe Kruchten *The Rational Unified Process: An Introduction* 2ª edição Addison-Wesley 2000

[LeeAnderson90] Lee and Anderson *Fault-Tolerance: Principles and Practice* - Springer-Verlag 2ª Edição, Jan-1990

[Martin+00] P. Robillard Martin e Gail Murphy *Designing Robust Java Programs with Exception* Software Engineering Notes Nov 2000

[Meyer92] Bertrand Meyer *"Eiffel: the Language"* Prentice Hall, 2ª edição, 1992

[Meyer97] Bertrand Meyer *Object-Oriented Software Construction* - Second Edition Prentice-Hall 1997

[Miller+97] Robert Miller and Anand Tripathi *Issues with Exception Handling in Object-Oriented Systems* - ECOOP'97 — Object-Oriented Programming, pp.85-103, LNCS-1241, Finland, 1997.

[Monroe+97] Monroe, Robert T. e Kompanek, Andrew e Melton, Ralph e Garlan, David *Architectural styles, design patterns, and objects* IEEE Software, páginas 43 a52, Janeiro 1997.

[Oliva+98] Alexandre Oliva e Luiz Eduardo Buzato *Reflective Programming in C++ and Java* OOPSLA'98, Vancouver, Canadá, October 1998

[Paula01] Wilson de Pádua Paula Filho. *Engenharia de Software: Fundamentos, Métodos e Padrões*. LTC Editora. Rio de Janeiro – RJ, 2001.

[Pressman01] Roger S. Pressman *"Software Engineering: A practitioner's Approach"* 5ª edição – McGrawHill - 2001

[RequisitePro] *Rational RequistitePro* http://www.rational.com/products/reqpro/index.jsp

[Richter99] Charles Richter *Designing Flexible Object-Oriented Systems with UML* - Macmillan Technical Publishing 1999

[Rose] *Rational Rose* http://www.rational.com/products/rose/index.jsp

[Rumbaugh+99] James Rumbaugh, Ivar Jacobson e Grady Booch. *Unified Modeling Language Reference Manual* - Addison-Wesley, Reading -MA, 1999.

[Rup] *Rational RUP - Rational Unified Process* http://www.rational.com/products/rup/index.jsp

[Rushby93] John Rushby *Critical System Properties: Survey and Taxonomy* Computer Science Laboratory - SRI International . Technical Report CSL-93-01, Maio 1993

[Schneider+00] Geri Schneider and Jason P. Winters *Applying Use Cases - A practical Guide* - Addison-Wesley 2000

[Shaw+96] Shaw, Mary e Garlan, David *Software Architecture: Perspectives on an Emerging Discipline* - Prentice Hall, 1996.

[Sloman+87] M. Sloman, and J. Kramer *Distributed Systems and Computer Networks* - Prentice Hall. 1987.

[Sommerville01] Ian Sommerville *Software Engineering* - 6a edição. Addison Wesley - 2001

[Szyperski98] Clemmens Szyperski *Component Software - Beyond Object-Oriented Programming* - Addison-Wesley 1998

[Taisy01] Taisy Silva Weber *Tolerância a falhas: conceitos e exemplos* Programa de Pós-Graduação em Computação - Instituto de Informática – UFRGS http://www.inf.ufrgs.br/~taisy/projetos/TFSDconceitos.htm

[Together] http://www.togethersoft.com/together

[Valerie00] Valérie Issarny, Jean-Pierre Banâtre *Architecture-Based Exception Handling* In *Proceedings of the 34th Annual Hawaii International Conference on System Sciences (HICSS'34)*, January 2001, Hawaii, HA, USA.

[Weiss01] Gerson Mizuta Weiss *"Adaptação de componentes de software para o desenvolvimento de sistemas confiáveis"* Dissertação de Mestrado, IC/Unicamp, Junho de 2001

Apêndice A

Automatização de Tarefas Usando a Ferramenta Rose

Dado a complexidade dos sistemas tolerantes a falhas, a confecção de um projeto bem estruturado, consistente e correto é uma tarefa quase impossível de ser realizada sem a ajuda de ferramentas de auxílio automatizado, ou seja, ferramentas CASE[13]. Desta forma, além da elaboração de uma metodologia para definição do comportamento excepcional de um sistema, a metodologia MDCE, também faz parte deste trabalho disponibilizar ferramentas de auxílio ao emprego das soluções apresentadas pela metodologia MDCE.

Já dispondo de ferramentas poderosas como a Rational Rose, apresentada na seção 2.2.4.1 desta dissertação, nossa proposta inclui estender suas funcionalidades para incorporar particularidades dos modelos de sistemas confiáveis, além de automatizar algumas das atividades de construção destes sistemas e ainda verificar a consistência dos modelos gerados de acordo com a definição de um sistema confiável e robusto segundo a metodologia MDCE.

A ferramenta Rose permite automatizar algumas de suas atividades ou ainda criar novas funções e disponibilizá-las através de sua interface. Para isto, a ferramenta Rose disponibiliza seu meta-modelo através do Rose Extensibility Interface – REI. O Modelo REI é essencialmente a exposição dos pacotes, classes, propriedades e métodos que definem e controlam a aplicação Rose e suas funcionalidades. Pode-se interagir com o REI através de *Rose Scripts* ou *Rose Automation*. *Rose Script* é uma extensão da linguagem BasicScript que permite automatizar funções especificas do Rose, além possibilitar extração, criação e manipulação de informações de modelos e diagramas. *Rose Automation* permite

13 Computer Aided Software Engineering

integração de outras aplicações com o Rose.

Fizemos uso da tecnologia REI para auxiliar a construção dos modelos propostos pela metodologia MDCE pois estes modelos são demasiadamente trabalhosos de serem construídos manualmente. Os modelos propostos pela metodologia MDCE seguem os princípios do componente ideal tolerante a falhas apresentado no Capitulo 2 desta dissertação. O que essencialmente caracteriza o componente ideal é a separação entre atividade normal, que implementa os serviços normais do componente, e a atividade anormal, que implementa as medidas de tolerância a falhas do componente. As atividades normais são implementadas por uma hierarquia de classes normal ortogonal a uma hierarquia de classes excepcional que implementa as atividades excepcionais. Além disto consideramos que para um projeto robusto e confiável as seguintes regras básica devem ser seguidas:

- um componente deve tentar tratar internamente suas exceções antes de propaga-las para seu cliente. Para isto, para toda exceção lançada pela parte normal do componente deve existir um tratador capaz de tratá-la localmente, encapsulando sua ocorrência. No modelo de classes, as exceções devem estar representadas explicitamente e os tratadores são modelados como métodos das classes excepcionais do componente.
- Não considere propagação automática de exceções pois isto adiciona mais complexidade ao projeto do comportamento excepcional do que o já existente. Desta forma, um componente deve manter um tratador para toda exceção que o alcance, direta ou indiretamente, de forma a garantir que esta exceção possa ser eficientemente tratada.
- UML exceções são representadas como classes e organizadas hierarquicamente o que possibilita utilização do um tratador de uma classe para lidar com suas subclasses. Além disto, é importante deixar explicito no modelo de classes a associação de <<send>> entre uma classe e as exceções que esta pode lançar.

Como dissemos, a confecção do modelo sugerido acima é bastante trabalhosa. O número de classes que implementam o componente é no mínimo dobrado pois para cada classe normal deve existir uma classe excepcional associada. Além disto ainda devem estar representadas no modelo todas as exceções internas, tratadores e

exceções externas do componente em questão.

Desta forma, para automatizar parte da confecção deste modelo construímos um *script* que é executado internamente a ferramenta de modelagem Rational Rose e cria algumas das classes e associações propostas pelo novo modelo. A seguir, na seções 2 e 3, apresentamos os casos de uso que definem a funcionalidade do script criado e na seção 4 apresentamos um exemplo de sua execução.

A.1 Breve descrição dos requisitos

As atividades de modelagem que podem ser automatizadas através da ferramenta Rose estão representadas como casos de uso na Figura 1.

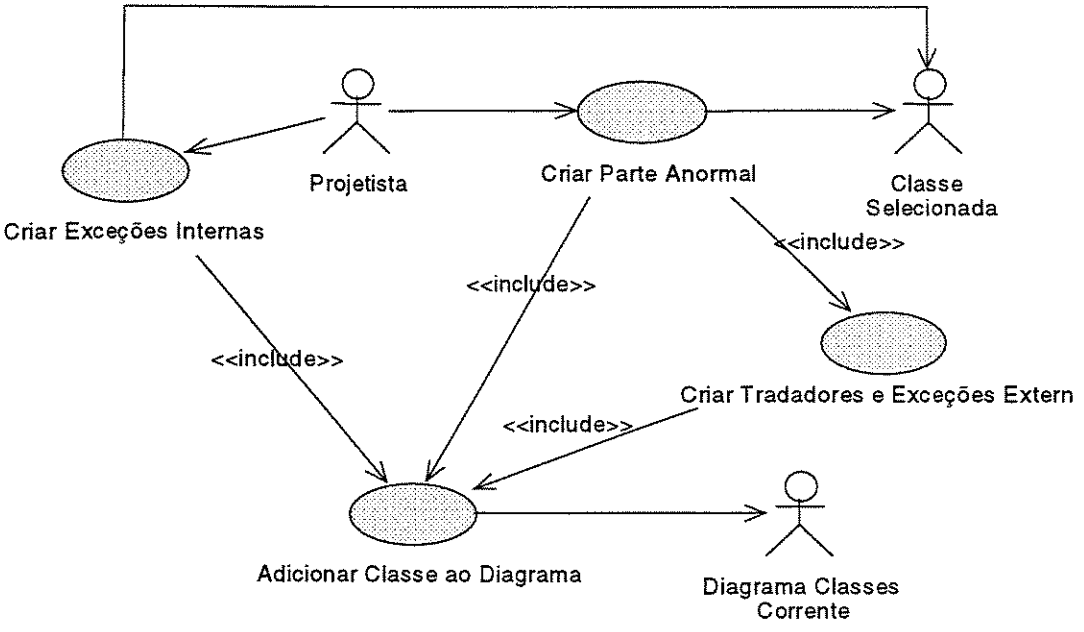

Figura 62 – Casos de Uso Primários

Veja abaixo uma breve descrição destes casos de uso.

Criar Exceções Internas

Dado uma classe normal selecionada no diagrama corrente, este caso de uso realiza as seguintes atividades: (1) identifica as exceções lançadas pelas operações desta classe; (2) cria as classes de exceção (se estas ainda não existirem); (3) associa a exceção à classe que a lança.

Criar Parte Anormal

Dado uma classe normal selecionada no diagrama corrente, este caso de uso cria uma classe anormal a ela associada (se esta ainda não existir). Este caso de uso inclui o caso de uso *Criar Tratadores e Exceções Externas* descrito a seguir.

Criar Tratadores e Exceções Externas

Para cada exceção interna, lançada pela classe normal, este caso de uso cria automaticamente um tratador que é representado por um método da classe anormal já criada pelo caso de uso *Criar Parte Anormal.* Cada tratador pode lançar uma exceção externa caso não consiga tratar localmente a falha, desta forma as classes que representam exceções externas são criadas automaticamente (caso não existam) e são associadas à classe que as lança, ou seja, à classe anormal.

Adicionar Classe ao Diagrama

Este caso de uso adiciona uma classe que lhe é passada como parâmetro ao diagrama de classes corrente.

A.1.1 Detalhamento dos casos de uso

Nesta seção apresentamos em um detalhe as atividades dos casos de uso mais complexos.

Caso de Uso *Criar Exceções Internas*

pré-condição: pelo menos uma classe normal está selecionada no diagrama de classes corrente

comportamento normal

1 para cada classe selecionada e para cada operação da classe

1.1 encontre as exceções lançadas por seus métodos

1.2 para cada exceção lançada

1.1.1 crie a classe de exceção se esta já não existir

1.1.2 crie a associação entre a classe normal e a classe de exceção

1.1.3 caso de uso Adicionar Classe ao Diagrama(exceção)

pós-condição: caso as operações da classe selecionada lancem exceções, novas classes de exceções são criadas e associadas a ela.

comportamento anormal

violação de pré-condição

sinal: não há classe selecionada ou a classe selecionada não é uma classe normal ou o diagrama corrente não é um diagrama de classes

tratador: notificar usuário através de uma mensagem

pós-condição: não há classes selecionadas, e nenhuma classe foi criada automaticamente.

Caso de Uso *Criar Parte Anormal*

pré-condição: pelo menos uma classe normal está selecionada no diagrama de classes corrente. As exceções lançadas pela classe normal já foram criadas e associadas a classe normal.

comportamento normal

1 Para cada classe normal selecionada

1.1 crie uma classe anormal (se ela ainda não existir)

1.2 associe classe normal e anormal

1.3 trate relações de herança para a classe anormal

1.4 para cada exceção lançada pela classe normal

1.4.1 caso de uso: *Criar Tratadores e Exceções Externas*

1.5 caso de uso: *Adicionar Classe ao Diagrama(classe anormal)*

pós-condição: para uma classe normal, uma classe anormal é associada e esta contém tratadores para as exceções internas e lança exceções externas.

comportamento excepcional

violação de pré-condição

sinal: não há classe selecionada ou a classe selecionada não é uma classe normal ou o diagrama corrente não é um diagrama de classes

tratador: notificar usuário através de uma mensagem

pós-condição: não há classes selecionadas, e nenhuma classe foi criada automaticamente.

O caso de uso *Criar Parte Anormal* inclui a realização de um novo caso de uso*Criar Tratadores e Exceções Externas* descrito a seguir.

Caso de Uso *Criar Tratadores e Exceções Externas*

pré-condição: há uma classe normal selecionada que já possui sua correspondente classe anormal e as exceções internas por ela lançadas.

comportamento normal

1 para cada exceção interna

1.1 se ainda não existir, crie um tratador representado por um método na classe anormal

1.2 crie uma exceção externa lançada pelo tratador

1.3 associe a exceção externa à classe anormal

1.4 caso de uso: *Adicionar Classe ao Diagrama(exceção externa)*

pós-condição: para cada classe selecionada, foram criados tratadores pra suas exceções internas e classes de exceções que representam as exceções lançadas externamente caso a falha não consiga ser tratada localmente.

A.2 Exemplo de Aplicação

Para fazer uso do nosso script, o projetista cria o modelo apenas com a hierarquia de classes normais que implementam a funcionalidade do componente com mostrado na figura abaixo. No exemplo, a classe A possui dois métodos op1() e op2() que lançam as exceções especificadas nas operações da classe. O Rose não mostra as exceções na interface do método, mas elas são as seguintes:

Classe SuperSuperA::op() throws I_E1

Classe SuperA::op11() throws I_E2

Classe A::op1() throws I_E1, I_E2

Classe A::op2() throws I_E1

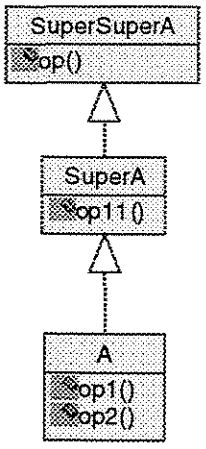

Figura 63 – Modelo atual

Após rodar o script, automaticamente será criado:

1. a hierarquia de classes anormais;
2. a associação entre as classes normais e anormais;
3. as classes que representam as exceções internas ao componente;
4. a relação de dependência entre a classe normal e as exceções por ela lançadas;
5. os tratadores para as exceções internas representados como métodos da classe anormal;
6. as classes de exceções externas caso o tratador não consiga recuperar a falha com sucesso;
7. a relação de dependência entre a classe anormal e as exceções externas por ela lançadas;

O modelo passará a ser o seguinte:

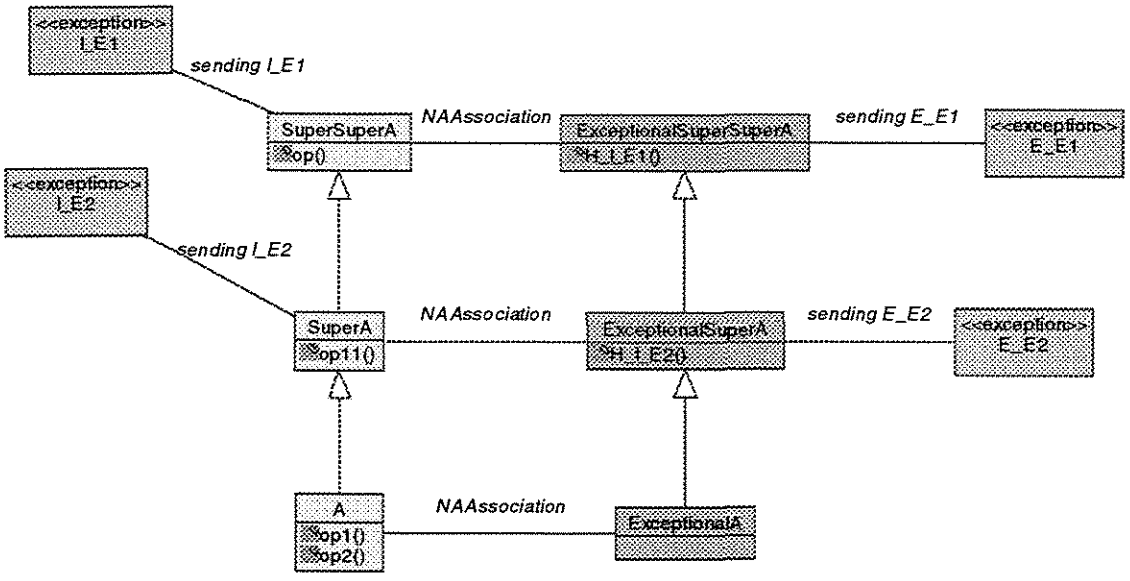

Figura 64 – Modelo gerado pelo script

Os estereótipos, layout e padrão de nomenclatura também são definidos automaticamente.

A.3 Como criar e executar um script

Como já foi dito, a linguagem para criação de scripts através do Rational Rose é uma extensão da linguagem BasicScript cujo editor pode ser acessado internamente através do menu *Tools* > *New Script* ou *Tools* > *Open Script*. A partir daí é só escrever seu script e para isto sugerimos que leia a ajuda da ferramenta Rose e ainda o código de scripts já prontos, além de executá-los e tentar entende-los. Alguns scripts podem ser encontrados dentro do diretório de instalação do Rose e subdiretório */scripts*. O código do script que geramos neste trabalho também pode ser útil e por isto estará disponível no endereço http://www.ic.unicamp.br/~ra995478/scritpRose.ebs.

Um script pode ser executado dentro do editor de scripts ou diretamente através da interface do Rose. Para executa-lo de dentro do editor é só fazer uso de sua barra de ferramentas aonde se encontra botões do tipo *start, break, end*. Por outro lado, executar um script através da interface do Rose exige um pouco mais de trabalho pois seu menu deve ser alterado para manter um atalho para realização do script. O conteúdo do menu do Rose é definido em um arquivo **Rose.mnu** e para realizar qualquer alteração em seu menu este arquivo deve ser editado. É possível criar novos menus ou adicionar comandos aos já existentes, para isto siga

as instruções da própria ferramenta no menu *Help > Contents and Index > Rational Rose Extensibility Interface > How to > Customizing Rational Rose Menus.*

A.4 Conclusões

Este apêndice mostrou que é possível e como fazer extensões na ferramenta de modelagem Rational Rose para automatizar algumas atividades de construção de um modelo de classes segundo especificação da metodologia MDCE proposta nesta dissertação.

Sugerimos, como trabalhos futuros, a construção de novos scripts que possam verificar a consistência entre os modelos e a metodologia MDCE. Isto é, verificar se para toda exceção existe um tratador, se todo tratador lança uma exceção externa caso não obtenha sucesso na recuperação da falha, se o cliente mantém um tratador para todas as exceções que lhe atingem, entre outros.

Apêndice B

Simulador do Sistema de Mineração

O sistema de mineração é um sistema computacional responsável pelo controle da extração de mineral. A extração de mineral é um processo que além de produzir água, também libera gás metano no ambiente. O sistema de mineração foi descrito em detalhes no Capitulo 6 desta dissertação, mas em linhas gerais, ele é responsável por controlar a extração de minerais e ainda manter o nível de água acumulada em um reservatório e nível de metano presente no ar atmosférico da mina sob controle. Para realizar estas extrações, o sistema de mineração faz uso dos dispositivos de extração como braço mecânico para extrair minerais, bomba para extrair água e exaustor para extrair o ar que contém metano de dentro da mina. Para ativar e desativar cada um destes dispositivos de extração, o sistema de mineração conta com mensagens dos sensores de níveis (MuitaAgua, PoucaAgua e MuitoMetano), que indicam que os níveis de água e metano estão altos ou baixos, e sensores de fluxo (FluxoAr e FluxoAgua) que indicam que está havendo fluxo de ar quando o exaustor estiver ligado e fluxo de água quando a bomba estiver operando.

Neste momento nos concentramos em descrever o simulador do sistema de mineração construído neste trabalho. O simulador do sistema de mineração é um sistema computacional que simula o ambiente onde o sistema de mineração está operando para que seja possível verificar seu correto funcionamento. O simulador mantém representação para o reservatório de água, o ar atmosférico e o nível de metano, os sensores e dispositivos de extração. Simula ainda a produção e consumo de água e metano durante o processo de extração. Este simulador não interfere no funcionamento básico do Sistema de Mineração pois funciona como um observador dos estados dos dispositivos de extração e altera a situação dos

sensores dada esta informação. Quando um sensor muda de estado, o controlador do sistema de mineração é avisado para que devidas atitudes sejam tomadas.

A seguir apresentaremos em linhas gerais o projeto do simulador do sistema de mineração.

B.1 Especificação dos Requisitos

No momento, estamos concentrados em descrever os requisitos do Simulador do Sistema de Mineração. Consideramos que o *Sistema de Mineração* é um elemento externo que interage como *Simulador* através dos dispositivos de extração e sensores que também são simulados pelo sistema *Simulador*.

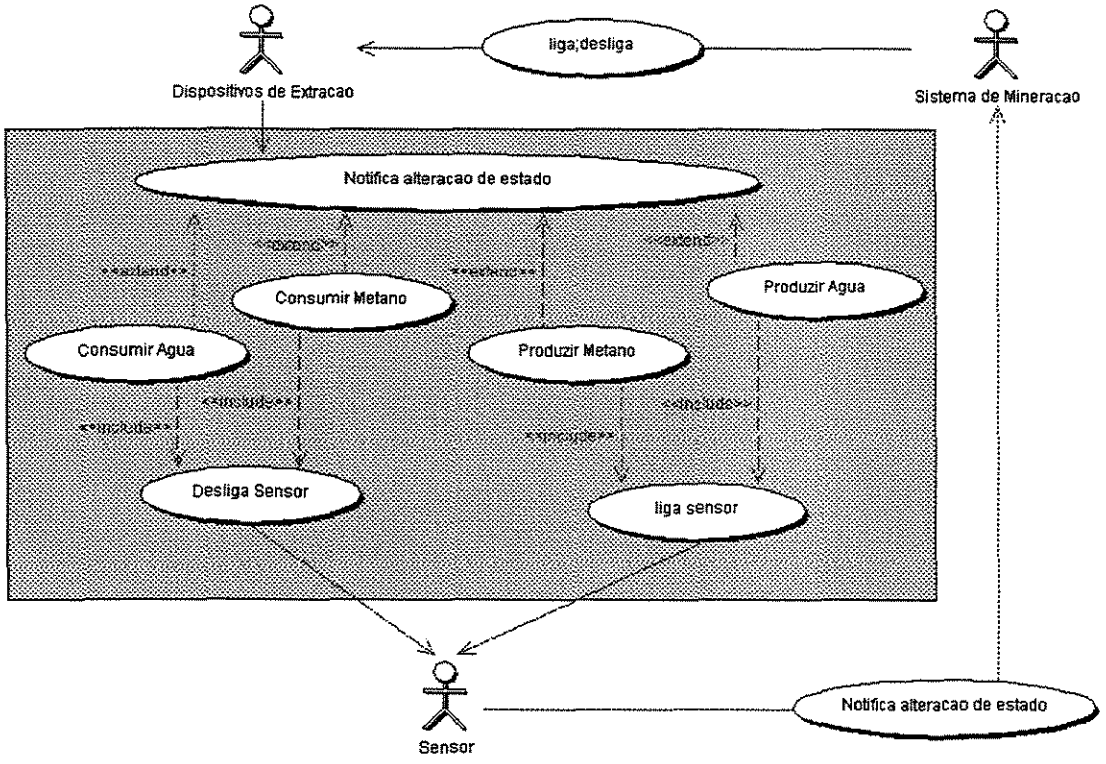

Figura 65 – Modelo de Casos de Uso do Simulador do Sistema de Mineração

O *Sistema de Mineração* liga e desliga os dispositivos de extração de acordo com a extração a ser realizada no momento. Abaixo daremos uma breve descrição dos casos de uso do Simulador.

Dispositivo de Extração Notifica Alteração de Estado

Quando um dispositivo de extração alterar seu estado, isto é, passar de

desligado para ligado ou o contrário, ele irá notificar o simulador sua alteração de estado através deste caso de uso. O Simulador, por sua vez, dado o estado atual dos dispositivos de extração saberá qual processo de extração está ativo no momento e se este consome ou produz água e/ou metano. Sendo assim, irá ativar um ou mais dos seguintes casos de uso: Consumir Metano, Consumir Água, Produzir Metano, Produzir Água.

Consumir Metano

Este caso de uso é ativado quando o dispositivo de extração Exaustor for ligado. Sua funcionalidade básica é abaixar o nível de metano do ambiente atmosférico da mina.

Consumir Água

Este caso de uso é ativado quando o dispositivo de extração Bomba for ligado. Sua funcionalidade básica é abaixar o nível de água nos reservatórios da mina.

Produzir Metano e Produzir Água

Estes dois casos de uso serão ativados quando o dispositivo de extração Braço Mecânico for ligado. Sua funcionalidade básica é aumentar o nível de metano do ambiente atmosférico da mina e o nível de água nos reservatórios da mina.

Desligar Sensor e Ligar Sensor

Quando metano e água são produzidos ou consumidos, através dos casos de uso descritos acima, seus níveis no ambiente atmosférico e no reservatório se alteram, elevando ou reduzindo respectivamente. As alterações entre máximos e mínimos permitidos para os níveis de água e metano fazem com que os sensores de níveis sejam ligados e desligados.

Sensor Notifica Alteração de Estado

Quando um sensor for ligado ou desligado pelo Simulador, este notificará sua alteração de estado ao Sistema de Mineração para que a extração atual possa ser finalizada e outra extração iniciada.

A dinâmica das interações entre Simulador, Sensores, Dispositivos de Extração e Sistema de Mineração pode ser vista na Figura 66.

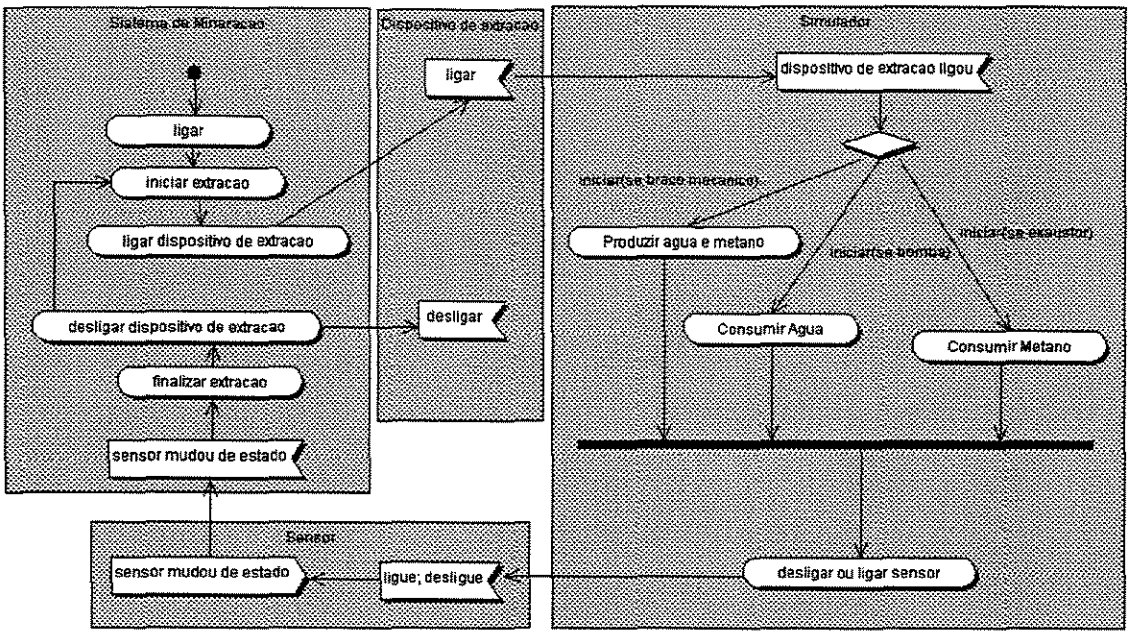

Figura 66 – Diagrama de Atividades do Simulador do Sistema de Mineração

Para facilitar o entendimento do diagrama acima vou descrevê-lo textualmente: O Sistema de mineração é ligado e inicia o processo de extração ligando um dispositivo de extração. Quando um dispositivo de extração é ligado, ele notifica seu novo estado ao Simulador que realizará uma das seguintes atividades: Produzir água e metano se o dispositivo ligado foi o Braço mecânico; consumir água se o dispositivo ligado foi a bomba; consumir metano se o dispositivo ligado foi o exaustor. Qualquer uma destas atividades realiza alteração dos níveis de água do reservatório e/ou de metano do ambiente atmosférico. Quando tais níveis atingirem seus limites máximos ou mínimos o sensor correspondente será ligado. Quando um sensor mudar de estado ele notifica sua alteração ao Sistema de Mineração que por sua vez irá interpretar a mensagem e, se for o caso, irá finalizar a extração corrente desligando o dispositivo de extração. O ciclo continua pois neste momento o dispositivo de extração irá avisar ao simulador sua alteração de estado e a produção ou consumo de água e metano cessará até que outro dispositivo seja ligado.

A seguir continuaremos com a definição dos limites do Simulador.

B.2 Definição dos Limites do Simulador

O simulador do sistema de mineração, assim como o sistema de mineração

propriamente dito, apresenta duas operações de interação com o operador: ligar e desligar. Internamente ele é formado pelos componentes do próprio sistema de mineração (*Estação de Controle*, *Sensor*, *Dispositivos de extração e Interface*) além dos componentes *Simula*. Veja Figura 67. O Componente *Simula* é responsável por simular a produção e consumo de água e metano no reservatório e no ar atmosférico e alterar o estado dos sensores de níveis quando o nível de metano ou água atingirem os limites mínimos ou máximos. Abaixo detalharemos alguns componentes complexos do simulador do sistema de mineração.

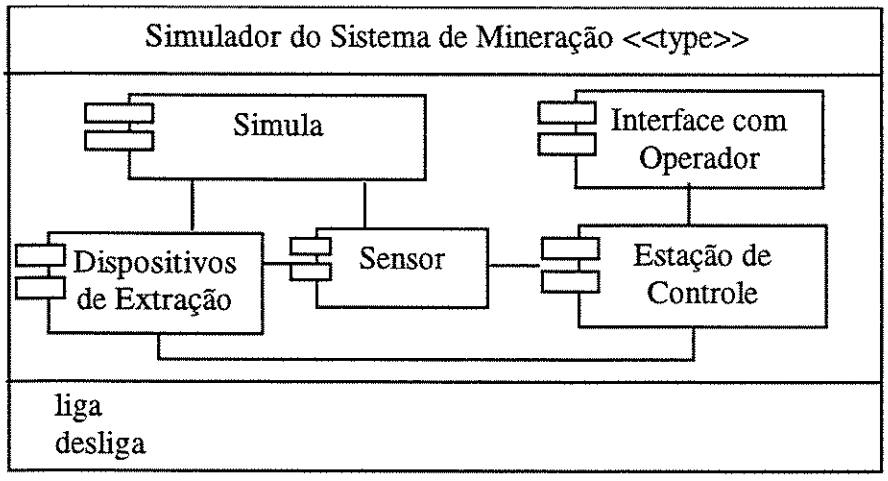

Figura 67 – Modelo de tipo do Simulador do Sistema de Mineração

B.2.1 Componente Sensor

O componente *Sensor* possui um componente que representa o sensor propriamente dito, com seu status de ligado e desligado, e um monitor de sensor. Monitor de Sensor é um observador do estado do sensor que notifica qualquer alteração ao controlador do sistema de mineração, ou seja, o componente Estação de Controle.

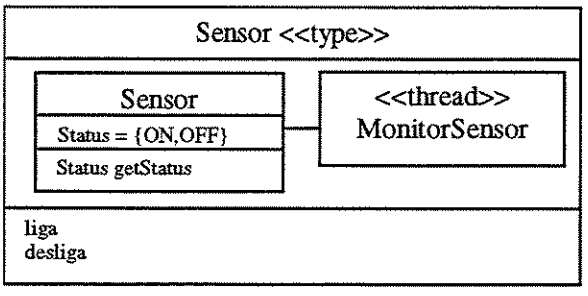

Figura 68 – Modelo de Tipo do Componente Sensor

B.2.2 Componente Simula

O *Simula* é um componente que internamente é formado por uma controladora, pelo reservatório de água e ambiente atmosférico como mostra a Figura 69. Tanto o ambiente atmosférico quanto o reservatório de água possuem indicativos de sua capacidade máxima para suportar metano ou água e ainda mantêm o nível atual de metano ou água respectivamente. Além disto, possuem ações externas que manipulam o nível de água ou metano conforme ordem recebida da controladora do simulador. A controladora recebe notificação de que um determinado dispositivo de extração ligou ou desligou, cabe a ela interpretar a mensagem e disparar a produção ou consumo de água ou metano. Como exemplo suponha que a controladora foi informada de que o braço mecânico ligou. Sendo assim, a controladora dispara a produção de metano e produção de água ao mesmo tempo pois quando mineral está sendo extraído pelo braço mecânico, água e metano são produzidos. Quando receber a notificação de que o braço mecânico desligou, a controladora interrompe a produção de metano e água.

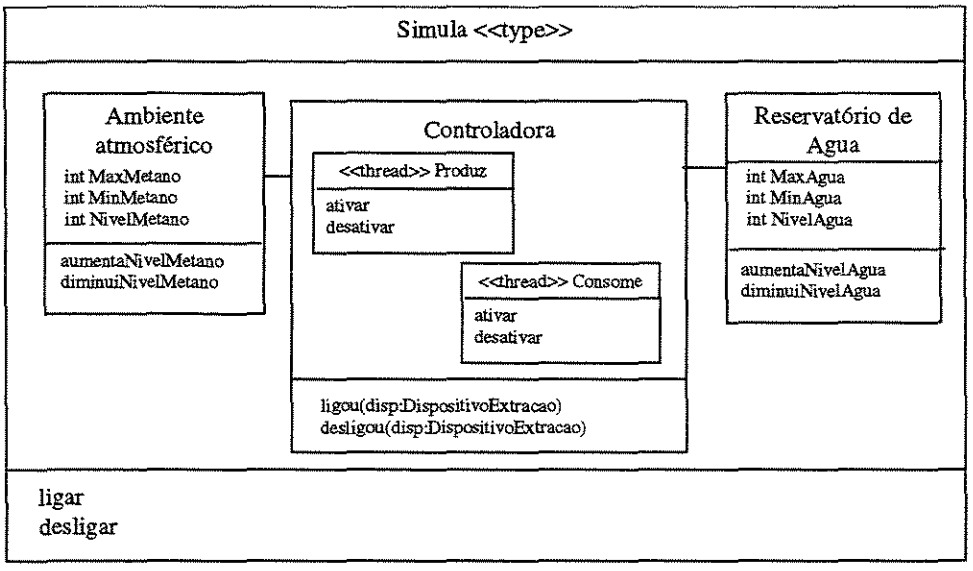

Figura 69 – Modelo de tipo do componente Simula

B.2.3 Componente Dispositivo de Extração

Assim, como o componente *Sensor,* o componente *Dispositivo de Extração* possui um componente que representa o dispositivo de extração propriamente dito e um monitor de dispositivos. Monitor de Dispositivo é um observador do estado do dispositivo de extração que notifica qualquer alteração à controladora do

simulador. Note que ao invés de notificar o sistema de mineração, o monitor de dispositivo notifica alterações dos dispositivos ao simulador pois dado esta informação o simulador saberá a hora de iniciar e interromper a produção ou consume de água e metano.

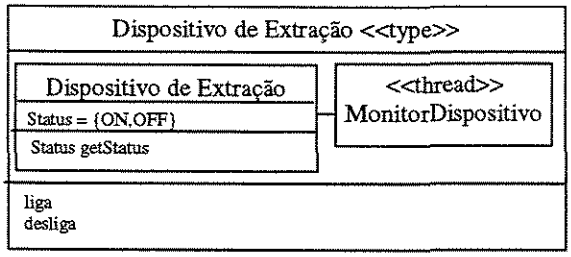

Figura 70 – Modelo de Tipo do Componente Dispositivos de Extração

A seguir continuaremos a projetar o Simulador definindo o projeto das colaborações entre seus elementos internos.

B.3 Projeto das Colaborações

Pro tratar-se de um sistema muito simples e formado por componentes que possuem grande interação, optei por apresentar um único projeto de colaboração que envolve todos os componentes do simulador. Este modelo único facilita o entendimento do sistema e está representado na Figura 71.

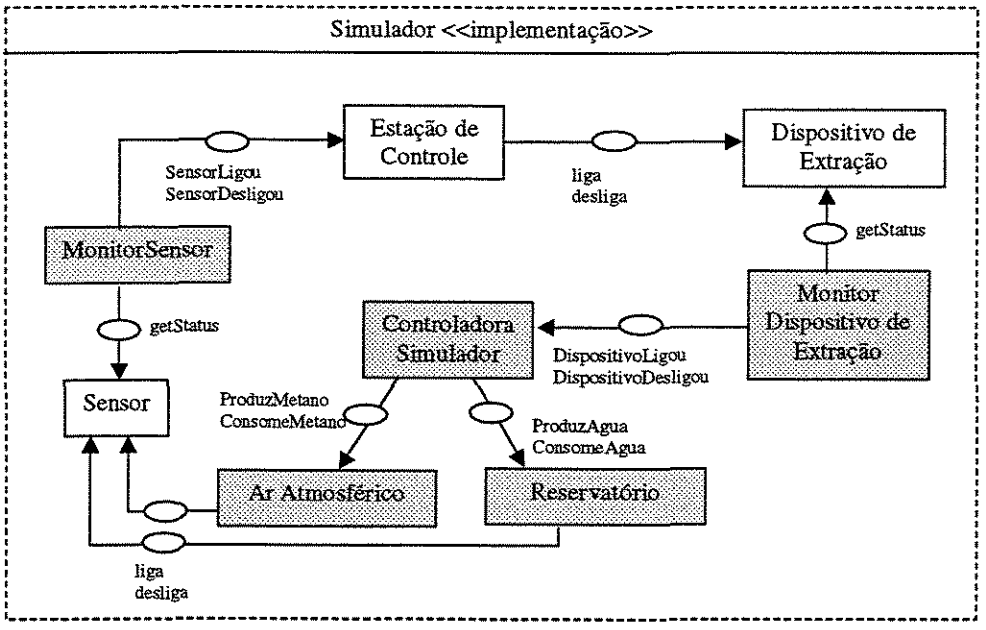

Figura 71 – Projeto de colaboração do Simulador

A estação de controle do Sistema de mineração liga e desliga dispositivos de extração de acordo com status dos sensores. Suponha que o dispositivo de Extração de Mineral foi ligado. A colaboração do simulador segue de acordo com o diagrama representado na Figura 72

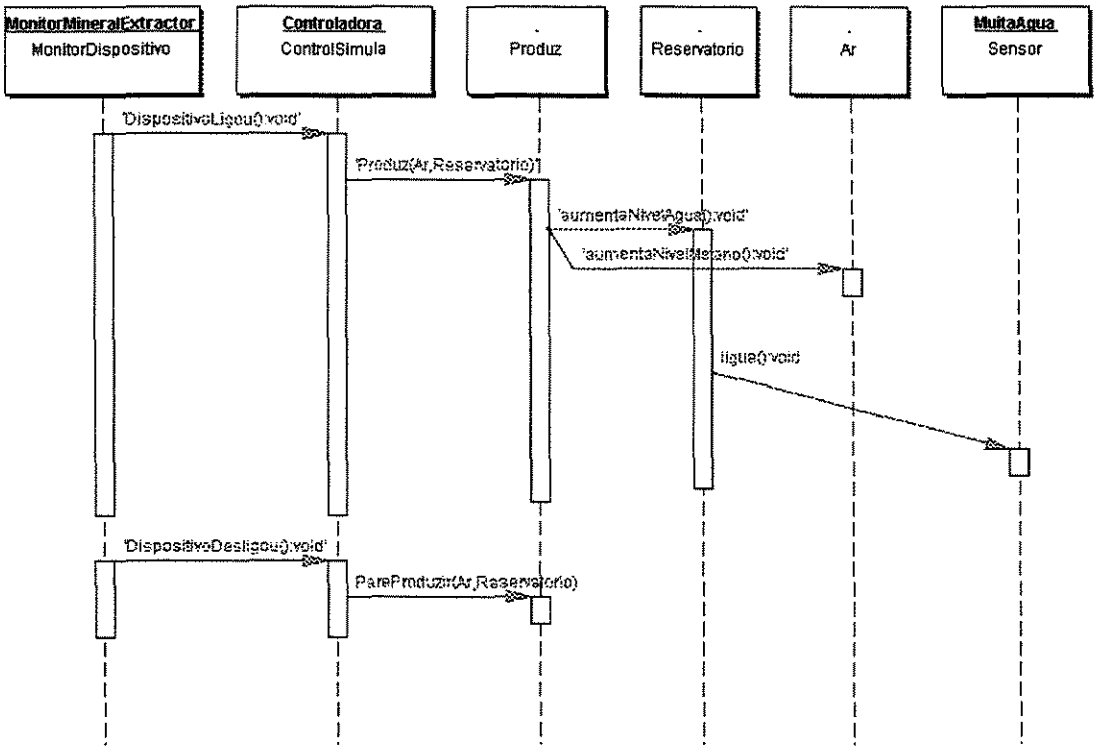

Figura 72 – Diagrama de seqüência de parte da colaboração do simulador

Consideramos que o Simulador é um componente confiável e não faremos a análise do seu comportamento excepcional para facilitar o entendimento do sistema.

B.4 Projeto Interno

Internamente o Simulador é então formado pelas seguintes classes e associações

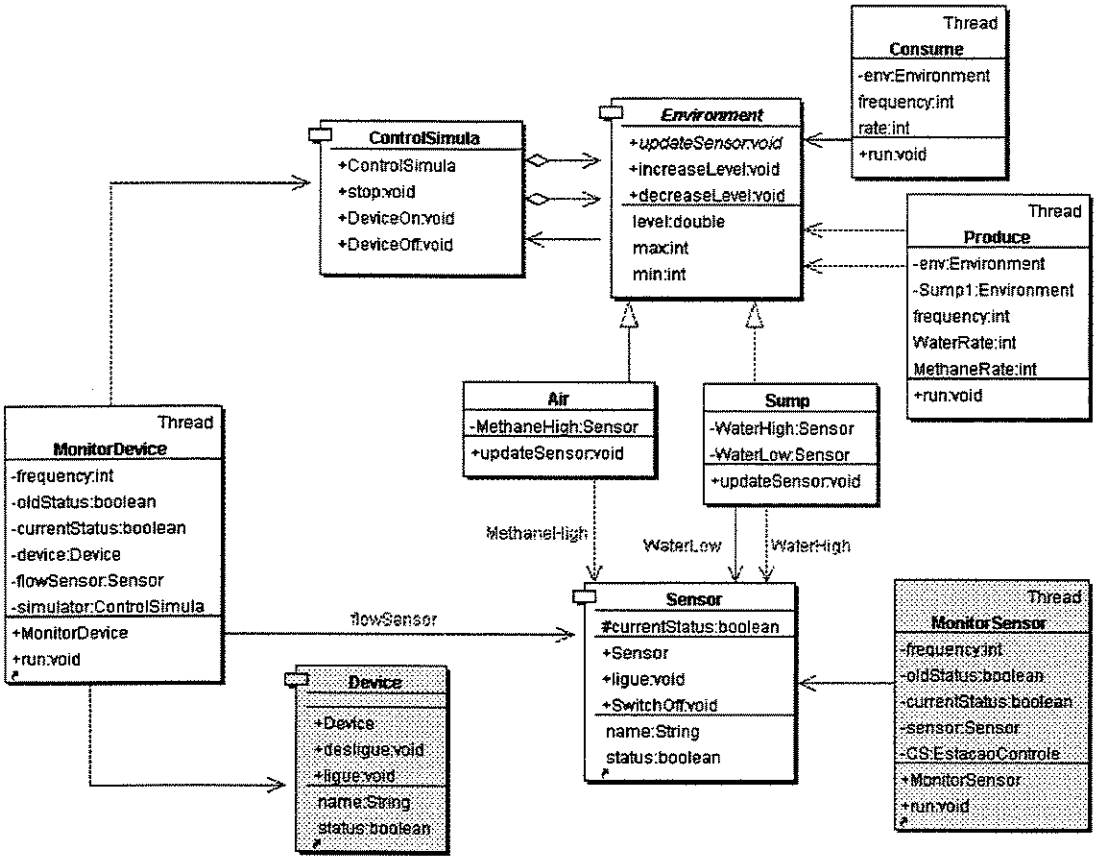

Figura 73 – Projeto Interno do Simulador

Além do simulador, projetamos e implementamos ainda um pequeno injetor de falhas que faz a bomba e/ou o exaustor falharem durante sua operação para testar a confiabilidade do sistema de mineração projetado e implementado segundo a metodologia MDCE proposta neste trabalho. A única funcionalidade do injetor de falhas é ligar ou desligar um dispositivo de extração e verificar como o sistema de mineração se comporta nestas situações.