**Uma Abordagem Arquitetural para Tolerância a Falhas em Sistemas de Software Baseados em Componentes**

*Paulo Asterio de Castro Guerra*

**Tese de Doutorado**

**Instituto de Computação**
**Universidade Estadual de Campinas**

# Uma Abordagem Arquitetural para Tolerância a Falhas em Sistemas de Software Baseados em Componentes

Paulo Asterio de Castro Guerra

Junho de 2004

**Banca Examinadora:**

Profª. Drª. Cecília Mary Fischer Rubira (Orientadora)

Prof. Carlos José Pereira de Lucena, Ph.D.
Pontifícia Universidade Católica do Rio de Janeiro (PUC-Rio)

Profª. Claudia Maria Lima Werner, D.Sc.
Instituto Alberto Luiz Coimbra de Pós-graduação e Pesquisa de Engenharia (COPPE/UFRJ)

Prof. Ivan Luiz Marques Ricarte, Ph.D.
Faculdade de Engenharia Elétrica e de Computação (FEEC/UNICAMP)

Profª. Eliane Martins, Docteur
Instituto de Computação (IC/UNICAMP)

Prof. Edmundo Roberto Mauro Madeira, Doutor
Instituto de Computação (IC/UNICAMP)

Prof. Ariadne Maria Brito Rizzoni Carvalho, Ph.D. (Suplente)
Instituto de Computação (IC/UNICAMP)

**FICHA CATALOGRÁFICA ELABORADA PELA BIBLIOTECA DO IMECC DA UNICAMP**


Guerra, Paulo Asterio de Castro

G937u Uma abordagem arquitetural para tolerância a falhas em sistemas de software baseados em componentes / Paulo Asterio de Castro Guerra -- Campinas, [S.P. :s.n.], 2004.

Orientadora: Cecília Mary Fischer Rubira.

Tese (doutorado) - Universidade Estadual de Campinas, Instituto de Computação.

1. Tolerância a falha (Computação). 2. Software - Reutilização. 3. Software - Arquitetura. 4. Software - Confiabilidade. 5. Engenharia de software. I. Rubira, Cecília Mary Fischer. II. Universidade Estadual de Campinas. Instituto de Computação. III. Título.

# Uma Abordagem Arquitetural para o Desenvolvimento de Sistemas de Software Tolerantes a Falhas Baseados em Componentes

Este exemplar corresponde à redação final da Tese devidamente corrigida e defendida por Paulo Asterio de Castro Guerra e aprovada pela Banca Examinadora.

Campinas, 12 de Julho de 2004

Profª. Drª. Cecília Mary Fischer Rubira

(Orientadora)

Tese apresentada ao Instituto de Computação, UNICAMP, como requisito parcial para obtenção do título de Doutor em Ciência da Computação.

**Termo de Aprovação**

Tese defendida e aprovada em 12 de Julho de 2004, pela Banca Examinadora composta pelos Professores Doutores.

Prof. Carlos José Pereira de Lucena, PhD
Pontifícia Universidade Católica do Rio de Janeiro (PUC-Rio)

Profa. Claudia Maria Lima Werner, D.Sc.
Instituto Alberto Luiz Coimbra de Pós-graduação e Pesquisa de Engenharia (COPPE/UFRJ)

Prof. Ivan Luiz Marques Ricarte, Ph.D.
Faculdade de Engenharia Elétrica e de Computação (FEEC/UNICAMP)

Prof. Edmundo Roberto Mauro Madeira, Doutor
Instituto de Computação (IC/UNICAMP)

Prof. Eliane Martins, Docteur
Instituto de Computação (IC/UNICAMP)

Profª. Drª. Cecília Mary Fischer Rubira
Instituto de Computação (IC/UNICAMP)

© Paulo Asterio de Castro Guerra, 2004.
Todos os direitos reservados.

*À Elizabeth, com minha gratidão pelo incentivo*

*e apoio nos momentos mais difíceis.*

# Agradecimentos

Agradeço aos Professores: Pedro de Rezende pela minha recepção aqui no IC como candidato ao curso, Cecília Rubira, pela orientação recebida durante toda a pesquisa, Cecília Baranauskas, Cid Carvalho, Cláudia Medeiros, Edmundo Madeira, Eliane Martins, Heloisa Vieira, Ricardo Anido e Ricardo Dahab, pelas disciplinas ministradas durante o curso.

Agradeço ao Professores Rogério de Lemos, da Universidade de Kent, e Alexander Romanovsky, da Universidade de Newcastle, que contribuiram de forma decisiva para essa pesquisa, de várias formas: com idéias, produção conjunta de artigos, dedicação de tempo e recursos durante os períodos em que estive visitando as suas instituições e, principalmente, grandes doses de estimulo, em vários momentos decisivos.

Agradeço aos colegas do IC: Fernando, Gisele, Milton, Moacir, Ricardo, Patrick, Rodrigo e Vinicius, pelo companheirismo e pelas muitas contribuições que fizeram a essa pesquisa, em trocas de idéias e publicações conjuntas.

Finalmente, meus agradecimentos ao grande amigo Paulo Kato, diretor da Autbank - Projetos e Consultoria Ltda., e a todas da sua equipe, em especial à Alessandra, Denise, Michelle, Paulo Freire, Tomohiro e Vanusa, pelo apoio e colaboração durante o desenvolvimento dessa pesquisa.

# Resumo

Esse trabalho se concentra na inclusão do tratamento sistemático de requisitos de confiabilidade no desenvolvimento de sistemas críticos modernos como, por exemplo, automação bancária e comércio eletrônico. Tais sistemas evoluem rapidamente e estão sujeitos a fortes restrições de prazos e custos de desenvolvimento. Em conseqüência, o desenvolvimento desses sistemas críticos modernos geralmente é baseado em software de grande complexidade e que integra componentes de software já existentes. O desenvolvimento de sistemas confiáveis que dependem de software com tais características é um problema ainda em aberto. Esse trabalho contribui para a solução desse problema através de uma abordagem arquitetural para tolerância a falhas em sistemas de software baseados em componentes reutilizáveis. A abordagem proposta se desdobra em: (i) uma arquitetura de software baseada em componentes ideais tolerantes a falhas; (ii) uma solução arquitetural para transformar componentes de prateleira (*Off-the-Shelf Components*) em componentes ideais tolerantes a falhas; (iii) uma estratégia geral para tratamento de exceções em sistemas de software baseados em componentes; e (iv) a proposta de um ambiente integrado para desenvolvimento de software baseados em componentes, centrado na arquitetura do software e no processo de desenvolvimento. O presente trabalho inclui também a aplicação prática da abordagem proposta em diversos estudos de casos, incluindo sistemas de automação bancária reais desenvolvidos por uma empresa de software independente. Os resultados obtidos permitem concluir pela eficácia da abordagem proposta para elevar a confiabilidade de sistemas de software complexos e baseados em componentes reutilizáveis. Conclui-se também pela necessidade de um suporte de ferramentas especializadas que aumentem eficiência da abordagem proposta através da automação de métodos e sua melhor integração no processo de desenvolvimento de software.

# Abstract

This work concentrates on the systematic treatment of dependability requirements during the development of modern critical software systems, such as banking and electronic commerce. Such software systems constantly evolve and are constrained by short time-to-market and low development costs requirements. As a result, the development of these modern critical software systems is increasingly being based on the integration of preexisting components. The development of dependable systems built in this way is still an open problem. This work contributes to the solution of this problem by proposing an architectural approach for adding fault tolerance to software systems based on reusable components. The proposed approach includes: (i) a software architecture based on idealised fault tolerant components; (ii) an architectural solution for transforming off-the-shelf software components in idealised fault tolerant components; (iii) a general strategy for exception handling in component-based software systems; and (iv) a proposal for an integrated development environment for component-based software systems, which is processed- and architecture-centered. The present work also includes two case studies where the proposed approaches were applied, one of them being based on a banking software system developed by an independent software house. The results achieved during these case studies allow us to conclude that the approach proposed can increase the dependability properties of complex software systems built from reusable components. We also conclude about the need for more specialized tools to increase the effectiveness of the proposed approach and better integrate it within a software development process.

# Sumário

Capítulo 1 Introdução ........................................................................... 23

1.1. Contexto do Problema ........................................................... 23

1.2. Descrição do Problema ........................................................... 27

1.3. Solução Proposta ........................................................... 29

1.4. Contribuições Originais ........................................................... 32

1.5. Organização da Tese ........................................................... 34

Capítulo 2 Fundamentos Teóricos ........................................................... 35

2.1. Desenvolvimento de Software Baseado em Componentes ........................ 35

2.1.1. Conceitos Básicos de Componente de Software ........................ 35

2.1.2. Especificação, Implementação e Instanciação de Componente... 36

2.1.3. As Duas Fases do DBC ........................................................... 38

2.1.4. Classificação de Componentes ........................................... 39

2.1.5. Processos de Desenvolvimento Orientados a DBC .................... 41

2.2. Arquitetura de Software ........................................................... 43

2.2.1. Linguagens de Descrição de Arquitetura .................................. 43

2.2.2. Análise de Arquitetura ........................................................... 44

2.2.3. Estilos Arquiteturais e Padrões Arquiteturais ........................... 45

2.2.4. O Estilo Arquitetural C2 ........................................................... 46

2.3. Confiabilidade em Sistemas Computacionais ........................................... 49

2.3.1. Conceitos de Defeito ........................................................... 49

2.3.2. Atributos de Confiabilidade ........................................................... 49

2.3.3. Conceitos de Erro e Falha ........................................................... 51

2.3.4. Categorias de Falhas e Hipóteses de falhas ............................... 51

2.3.5. Prevenção de Falhas ........................................................... 53

2.3.6. Tolerância a Falhas ........ 54
2.3.7. Detecção de Erros ........ 55
2.3.8. Recuperação de Erros ........ 57
2.3.9. Mecanismos de Tratamento de Exceção ........ 57
2.3.10. Componente Ideal Tolerante a Falhas ........ 58
2.4. Confiabilidade de Software no Paradigma DBC ........ 59
2.4.1. Definições de Interface ........ 60
2.4.2. Composição de Componentes ........ 61
2.4.3. Implementação das Interfaces ........ 62
2.4.4. Testes de Componentes ........ 62
2.4.5. Evolução de Componentes ........ 63
2.4.6. Infra-estrutura de Componentes ........ 64
2.5. Considerações Finais ........ 64
Capítulo 3 Arquiteturas de Software Tolerantes a Falhas ........ 67
3.1. O Componente C2 Ideal Tolerante a Falhas (iC2C) ........ 67
3.1.1. Estruturação do Componente AtividadeNormal ........ 70
3.1.2. Estruturação do componente AtividadeAnormal ........ 72
3.1.3. Integração do iC2C em Configurações C2 ........ 72
3.1.4. Modelo Formal do iC2C ........ 75
3.2. Um Componente Ideal Baseado em COTS (iCOTS) ........ 79
3.2.1. Conceito de Invólucros Protetores ........ 80
3.2.2. Passos para Desenvolvimento de iCOTS ........ 81
3.2.3. Passos do Estágio de Provisionamento ........ 81
3.2.4. Passos do Estágio de Montagem ........ 83
3.2.5. Arquitetura Geral do iCOTS ........ 85
3.3. Considerações Finais ........ 87
Capítulo 4 Uma Abordagem Arquitetural para Tratamento de Exceções em DBC ........ 89

4.1. Descrição do Problema .......................................................................... 89

4.2. Um Modelo para Tratamento de Exceções ............................................ 91

4.2.1. As Abordagens de Flaviu Cristian e Bertrand Meyer ................ 91

4.2.2. Condições Excepcionais Antecipadas....................................... 94

4.2.3. Condições Excepcionais Não Antecipadas ............................... 95

4.2.4. Pontos de Saída de uma Operação ........................................... 96

4.2.5. Semântica do Defeito............................................................... 97

4.3. Solução Proposta .......................................................................... 98

4.3.1. Hierarquia de Tipos de Exceções Proposta ............................. 98

4.3.2. Implementação Java da Hierarquia de Tipos de Exceções Proposta100

4.3.3. A Estratégia Intra-componente ............................................... 101

4.3.4. Aplicação da Estratégia Construindo um Componente "do Zero"104

4.3.5. Aplicação da Estratégia a Componentes Reutilizáveis .............. 106

4.3.6. A Estratégia Inter-Componente .............................................. 107

4.4. Considerações Finais ..................................................................... 110

Capítulo 5 Proposta de um Ambiente de Desenvolvimento de Software Confiável Baseado em Componentes ........................................................................................ 113

5.1. Requisitos Básicos do Ambiente Integrado Proposto ............................. 114

5.2. Diretrizes Gerais do Projeto de Desenvolvimento do Ambiente ............... 115

5.3. Concepção Inicial do Ambiente Integrado ............................................ 116

5.4. Situação Atual do Desenvolvimento do Ambiente ................................. 118

5.5. Considerações Finais ..................................................................... 122

Capítulo 6 Estudos de Casos ......................................................................... 125

6.1. Estudo de Caso 1: Sistema de Controle de Caldeira a Vapor .................. 125

6.1.1. Estágio de Provisionamento.................................................... 126

6.1.2. Estágio de Montagem............................................................. 128

6.1.3. Discussão do Estudo de Caso 1 .............................................. 136

6.2. Estudo de Caso 2: Sistema de Automação Bancária ............................... 136
6.2.1. Objetivo........................................................................ 136
6.2.2. Descrição do Sistema Boleto de Ativos.................................... 137
6.2.3. Especificação das Hipóteses de Falhas .................................... 141
6.2.4. Projeto do Comportamento Excepcional do Sistema ................ 144
6.2.4.1. A nova interface IPoolMgr ....................................... 144
6.2.4.2. Sinalização das Condições Excepcionais.................... 146
6.2.4.3. Detecção das Condições Excepcionais........................ 146
6.2.4.4. Tratamento das Condições Excepcionais ................... 146
6.2.5. Alterações no Código do Sistema ............................................ 147
6.2.6. Testes do Comportamento Excepcional do Sistema ................. 149
Capítulo 7 Conclusões e Trabalhos Futuros ................................................ 153
7.1. Objetivos e Contribuições Principais....................................................... 153
7.2. Lista de Publicações................................................................................ 154
7.3. Dificuldades Encontradas ........................................................................ 156
7.4. Limitações da Pesquisa............................................................................. 157
7.5. Trabalhos Futuros .................................................................................... 158
Referências Bibliográficas............................................................................. 161
Anexo 1 Especificação Formal do iC2C........................................................ 173
Arquivo iC2C.ta ............................................................................................ 173
Arquivo iC2C.q ............................................................................................. 177

# Lista de Figuras

Figura 1 - Representação UML de um componente de software ............ 36
Figura 2 - Camadas de um sistema DBC ............ 37
Figura 3 - Estilo arquitetural em camadas ............ 45
Figura 4 - Elementos básicos do estilo arquitetural C2 ............ 47
Figura 5 - O componente ideal tolerante a falhas ............ 58
Figura 6 - Hierarquia de tipos de mensagens do iC2C ............ 68
Figura 7 - O Componente C2 Ideal Tolerante a Falhas ............ 69
Figura 8 - Decomposição do Componente AtividadeNormal ............ 71
Figura 9 - Decomposição do componente AtividadeAnormal ............ 72
Figura 10 - Envolvendo um iC2C para elevar tolerância a falhas ............ 75
Figura 11 - Adaptando a interface de um iC2C para elevar tolerância a falhas ............ 75
Figura 12 - Modelo UPPAAL de mais alto nível do iC2C ............ 76
Figura 13 - Modelo refinado do processo AtividadeNormal ............ 78
Figura 14 - Estrutura Geral do iCOTS ............ 86
Figura 15 - Decomposição de um detector de erro ............ 86
Figura 16 - Exemplo de programa com pontos de saída excepcionais ............ 93
Figura 17 - Pontos de saída na abordagem de Projeto por Contrato ............ 94
Figura 18 - Hierarquia Comum de Tipos Abstratos de Exceções ............ 99
Figura 19 - Hierarquia de Tipos de Exceções Java ............ 100
Figura 20 - Elementos básicos da implementação de um componente ............ 105
Figura 21 - Implementação de componente com tratadores de exceções ............ 105
Figura 22 - Adaptação das interfaces excepcionais de um componente já existente ............ 107
Figura 23 - Tratadores de Exceções CLE ............ 107
Figura 24 - Arquitetura de software do ambiente integrado proposto ............ 117

Figura 25 - Exemplo de diagrama de atividades do processo UML Components ...................... 120
Figura 26 - Metamodelo GME para o Editor de Configurações ............................................. 121
Figura 27 - Tela principal da ferramenta de Teste de Componentes......................................... 122
Figura 28 - Partes Componentes do Sistema de Controle da Caldeira...................................... 126
Figura 29 - Arquitetura Inicial do Sistema de Controle da Caldeira ......................................... 127
Figura 30 - Decomposição do Componente AirFlowController.................................................. 130
Figura 31 - Arquitetura de Software Final do Sistema de Controle da Caldeira ........................ 132
Figura 32 - Diagrama de seqüência UML para uma exceção PIDTimeout.................................. 133
Figura 33 - Implementação parcial Java do detector inferior do AirFlowController ...................... 134
Figura 34 - Implementação do tratador de erros do AirFlowController. ...................................... 135
Figura 35 - Arquitetura de software em camadas do sistema Boleto de Ativos ........................ 137
Figura 36 - Visão física do Sistema Boleto de Ativos............................................................. 138
Figura 37 - Caso de uso Cadastramento de Proposta de Leasing............................................. 139
Figura 38 - Diagrama parcial de componentes do Sistema Boleto de Ativos............................ 140
Figura 39 - Diagrama de seqüência da operação listaProduto().................................................. 142
Figura 40 - Nova versão da interface IPoolMgr.......................................................................... 145
Figura 41 - Versão inicial do método listaProdutos() do componente EBTabelas........................ 148
Figura 42 - Nova versão do método listaProdutos() do componente EBTabelas .......................... 148
Figura 43 - Nova versão do conector baebtabelasfwpool .......................................................... 149

# Lista de Tabelas

Tabela 1 - Regras de tradução de tipos de exceções por um tratador CLE ............................... 109

Tabela 2 - Lista de Operações do Sistema de Controle da Caldeira .......................................... 128

Tabela 3 - Especificações de detecção de erros para o AirFlowController .................................... 129

Tabela 4 - Resumo das notificações excepcionais do sistema de controle da caldeira................ 131

Tabela 5 - Tipos de condições excepcionais ao listar produtos ................................................. 143

Tabela 6 - Casos de Testes de Condições Excepcionais .......................................................... 151

# Capítulo 1

# Introdução

*A expansão significa complexidade*
*e a complexidade, decadência.*
*Cyril N. Parkinson*

Este capítulo define o problema tratado no presente trabalho, situando-o no seu contexto mais geral, descreve sucintamente a solução proposta e relaciona os principais resultados obtidos. Atualmente não há um consenso, nas comunidades acadêmica e industrial da Língua Portuguesa, quanto à terminologia utilizada na área de **confiabilidade**[1] (ou **confiança no funcionamento**)[101]. Por esse motivo, apresentamos em notas de rodapé os termos na Língua Inglêsa correspondentes aos termos utilizados no presente trabalho, sempre nas suas primeiras ocorrências.

## 1.1. Contexto do Problema

Sistemas de software são empregados em praticamente todas as atividades da nossa sociedade, inclusive as mais essenciais. Telefonia e finanças são exemplos de atividades essenciais fortemente dependentes de sistemas de software. Sistemas como esses, em que a sua interrupção ou mal funcionamento representam riscos para vidas humanas, bens patrimoniais ou o meio-ambiente, são denominados **sistemas de missão crítica** ou, simplesmente, **sistemas críticos**. Um dos principais requisitos dos sistemas de missão crítica é a confiabilidade. Um sistema confiável é aquele para o qual podemos, justificadamente, confiar nos serviços que ele oferece [52].

Quando um sistema computacional deixa de oferecer as funcionalidades ou o faz em desacordo com suas especificações dizemos que ocorre um **defeito**[2] no sistema. A ocorrência de um defeito

[1] do inglês: *dependability*.
[2] Nota: traduzimos *failure*, *error* e *fault* por **defeito**, **erro** e **falha**, respectivamente.

pode ser associada a um determinado estado interno do sistema que, sob determinadas entradas, provoca aquela ocorrência. Esse estado interno do sistema, que provoca o defeito, é chamado de **erro**. Para que haja um erro haverá necessariamente alguma causa, a qual chamamos de **falha** [54].

Os sistemas críticos mais freqüentemente citados na literatura são aqueles onde há risco para vidas humanas ou de danos materiais irreparáveis, tais como os empregados na pesquisa espacial ou na medicina. Tolerância a falhas é uma qualidade essencial para esses sistemas críticos, aqui chamados de sistemas críticos "clássicos". Os mecanismos de tolerância a falhas incorporados nesses sistemas impedem, por exemplo, que defeitos no próprio sistema coloquem em risco a vida dos pacientes submetidos a uma radioterapia computadorizada. Esses sistemas críticos "clássicos" são, em geral, construídos com software relativamente simples, o que também favorece a sua confiabilidade. Essa simplicidade decorre de algumas características normalmente presentes no projeto desses sistemas, destacando-se: (i) as funções vitais do sistema são implementadas por componentes de hardware projetados especificamente para essas funções; (ii) todos os componentes de software são projetados e desenvolvidos especificamente para uma única aplicação; e (iii) todo o sistema é fortemente isolado, evitando dependências de outros sistemas e exposição a agentes externos desconhecidos. Subsistemas e acessórios especiais são incluídos com essa finalidade como, por exemplo, geradores de energia próprios e barreiras físicas de proteção; e (iv) o sistema é operado e supervisionado por técnicos especialmente treinados no funcionamento daquele sistema.

À medida que os sistemas de software se tornam mais presentes, desempenham funções mais complexas e interagem entre si para compor sistemas maiores, suas missões tornam-se também mais críticas. Automação bancária e sistemas comerciais são exemplos atuais de áreas onde isso já ocorre e que estão se tornando criticamente dependentes de sistemas de software. Um sistema de automação bancária atual, por exemplo, deve operar de forma integrada e em tempo real com inúmeros outros sistemas como, por exemplo, sistemas comercias com pagamento eletrônico, sistemas de compensação eletrônica e sistema de controle dos bancos oficiais. Sistemas como

esses, aqui chamados de sistemas críticos "modernos" e que hoje se multiplicam, são de alto risco para as organizações que dependem desses sistemas e, muitas vezes, para o bem-estar da sociedade em geral.

Aspectos mercadológicos normalmente associados aos sistemas críticos "modernos" impõem fortes restrições aos prazos e custos de seu desenvolvimento. A utilização de hardware de prateleira (ou COTS[3]) como os micro-computadores produzidos em larga escala e com arquitetura padronizada é, assim, normalmente determinante para a viabilidade econômica desses sistemas. Da mesma forma, a construção de software através da integração planejada de componentes reutilizáveis, conhecido como **desenvolvimento baseado em componentes** (DBC) [11][96], é outra prática que vem se disseminando rapidamente no desenvolvimento desses sistemas críticos "modernos". No DBC, a ênfase é no projeto de partes, ou componentes, que se comunicam apenas através de interfaces bem definidas e que podem ser implementadas independentemente [11]. Infra-estruturas de componentes J2EE [95] e .NET [60] são exemplos de tecnologias de DBC amplamente utilizadas no desenvolvimento de sistemas de software atuais.

Um aspecto central do paradigma DBC é possibilitar a independência entre as atividades de desenvolvimento de componentes de software e de integração de sistemas. Essa divisão de trabalho permite que, para um mesmo sistema de software, essas duas atividades sejam desenvolvidas por equipes independentes e em momentos distintos. Durante a integração de sistemas são reutilizados os componentes de software desenvolvidos independentemente. Criam-se, assim, as condições para que surja um mercado de componentes de software de prateleira (ou COTS software) onde produtores de componentes de software oferecem seus produtos para os integradores de sistemas DBC. Freqüentemente esses componentes de software são fornecidos como "caixas-pretas", o que limita ou impede o acesso, pelo integrador do sistema, ao projeto interno e código fonte dos componentes.

[3] do inglês: *Commercial Off-the-Shelf components*.

As restrições impostas pelos aspectos mercadológicos citados, além da própria natureza das aplicações dos sistemas críticos "modernos", contribuem para que os projetos desses sistemas apresentem características que os tornam dependentes de sistemas de software de grande complexidade. Dentre essas características, substancialmente diversas daquelas dos sistemas críticos "clássicos", destacamos: (i) todas as funções do sistema, inclusive as mais críticas, são controladas por software; (ii) o software é construído reutilizando-se componentes já existentes, muitas vezes desenvolvidos por terceiros e integrados no sistema como "caixa-preta"; (iii) os sistemas interagem fortemente com outros sistemas e dependem de uma infra-estrutura de hardware e software complexa, sobre a qual os projetistas do sistema exercem pouco ou nenhum controle; e (iv) esses novos sistemas são operados por usuários leigos, sem conhecimento específico sobre o seu funcionamento.

Esse contraste entre características de projeto dos sistemas críticos "clássicos" e "modernos" impede a simples transposição das técnicas e métodos utilizadas com sucesso no desenvolvimento dos sistemas críticos "clássicos" para os sistemas críticos "modernos", hoje demandados. O desenvolvimento de sistemas confiáveis ainda permanece como uma atividade altamente especializada que exige técnicas e recursos exclusivos dos sistemas críticos "clássicos". As técnicas e práticas empregadas no desenvolvimento de software dos sistemas críticos "modernos" dão pouca ou nenhuma atenção aos aspectos relacionados com confiabilidade. Conseqüentemente, a confiabilidade desses sistemas críticos "modernos" é, em geral, altamente insatisfatória.

A arquitetura de um sistema de software mostra como o sistema é realizado através de uma coleção de componentes e as interações entre eles, abstraindo os detalhes de implementação desses componentes [80]. Sem um projeto arquitetural consistente e explicitamente representado, a análise de propriedades de um sistema requer um raciocínio envolvendo, simultaneamente, todas as partes desse sistema e suas interações. Para sistemas com centenas ou milhares de componentes, essa análise simultânea pode ser absolutamente inviável. A partir da arquitetura do software, podemos deduzir propriedades sistêmicas em função das propriedades de subsistemas

componentes que podem ser analisados separadamente, num processo recursivo de "divisão e conquista".

A abordagem de arquitetura de software é complementar ao paradigma DBC. Componentes de software reutilizáveis são, em geral, desenvolvidos sem o conhecimento do contexto específico de todos os sistemas onde são empregados. Sendo assim, cabe à arquitetura do software a responsabilidade pela integração desses componentes de forma a se obter as qualidades desejadas para o sistema em desenvolvimento. Essa complementariedade se evidencia nos principais processos de DBC [4] [17] [25], que são também centrados na arquitetura do software.

A arquitetura do software é, portanto, de fundamental importância no desenvolvimento dos sistemas críticos "modernos", que são sistemas complexos e desenvolvidos com base em componentes de software reutilizáveis.

## 1.2. Descrição do Problema

Um dos principais meios para se alcançar confiabilidade num sistema computacional é torná-lo tolerante a falhas [2]. **Tolerância a falhas** é definida como a capacidade de um sistema continuar a prover serviços, em conformidade com suas especificações, mesmo na presença de falhas [52]. Uma condição necessária para se obter tolerância a falhas num sistema é o emprego de alguma forma de **redundância** [34]. Um sistema é redundante quando possui algo que, em condições ideais de funcionamento, não é necessário. Alguns exemplos de formas de redundância utilizadas para tolerar falhas são: (i) o uso de réplicas, que são cópias idênticas de um componente do sistema e capazes de assumir as funções daquele componente, em caso de falha; (ii) a adição de rotinas para tratamento de condições excepcionais, que são ativadas em caso de falha para eliminar os erros introduzidos no sistema e permitir a continuidade do processamento, ou para interromper as atividades do sistema de maneira ordenada e segura.

O principal obstáculo à tolerância a falhas [2] é a imprevisibilidade de ocorrência de uma falha e dos seus efeitos sobre o sistema. A ação dos mecanismos de tolerância a falhas só é iniciada com a detecção de erros já presentes no sistema. Uma vez detectado um erro, este deve ser removido

e, quando necessário, o sistema deve ser reconfigurado para isolar o agente provável causador da falha. Essas atividades são chamadas, genericamente, de recuperação de erro. Após a recuperação do erro espera-se que o sistema possa prosseguir com seu funcionamento normal, sem apresentar defeito.

Nos sistemas críticos "clássicos" uma parcela expressiva do seu código e complexidade está relacionada com a provisão de detecção e recuperação de erros [72]. Paradoxalmente, isso implica na elevação da probabilidade de ocorrência de falhas de projeto e, conseqüentemente, pode afetar negativamente a confiabilidade do sistema como um todo. A introdução de tolerância a falhas no software de um sistema exige, portanto, técnicas de estruturação de software que minimizem a interferência negativa do código relacionado com detecção e recuperação de erros nas funções normais do software. Nesse contexto, os chamados mecanismos de tratamento de exceções possuem uma importância fundamental ao permitir a separação do código responsável pela recuperação do erro do código normal. Quando utilizados esses mecanismos, a detecção de um erro durante a execução do código normal é sinalizada através do **lançamento de uma exceção** e a recuperação do erro é feita por rotinas especiais, chamadas de **tratadores de exceção**, separadas do código normal. Os mecanismos de tratamento de exceção são responsáveis pelo controle do fluxo de execução desde o momento em que é lançada uma exceção até o retorno do processamento ao seu fluxo normal, após a execução de um tratador de exceções apropriado [76].

O desenvolvimento de arquiteturas de software tolerantes a falhas, baseadas em componentes, é um problema ainda em aberto e de difícil solução [92]. Freqüentemente, a especificação de um componente de software reutilizável não abrange o seu comportamento em caso de falha. Como resultado, o comportamento excepcional de um componente de software é, muitas vezes, definido de forma "ad hoc" e inteiramente dependente da sua implementação. Quando especificado um comportamento excepcional, as hipóteses de falhas assumidas podem não ser compatíveis com aquelas definidas para um novo sistema. Conseqüentemente, na integração de um sistema baseado

em componentes reutilizáveis, podem surgir conflitos entre o comportamento excepcional da implementação de um componente e o comportamento excepcional previsto para o sistema.

Nesse contexto, são duas as questões principais abordadas neste trabalho:

(i) como incluir o tratamento de condições excepcionais em arquiteturas de software baseadas em componentes, de forma a permitir a estruturação de sistemas tolerantes a falhas que preservem a independência de implementação dos seus vários componentes [22] [89]; e

(ii) como estruturar o tratamento de condições excepcionais num projeto de integração de um sistema baseado em componentes reutilizáveis, de forma a resolver os conflitos que possam haver entre o comportamento excepcional das implementações de componentes utilizadas e o comportamento excepcional especificado na arquitetura do software [24] [28] [74].

Os processos de desenvolvimento de software hoje mais difundidos, tais como o *Unified Software Process,* também conhecido como *RUP* [47] e *UML Components* [17], são focados no atendimento aos requisitos funcionais do sistema e não incluem um tratamento sistemático dos requisitos não-funcionais (ou de qualidade) do sistema. Portanto, o comportamento do sistema em condições excepcionais não é tratado de maneira sistemática, de forma a incluir no sistema a redundância necessária para torná-lo tolerante a falhas. Sendo assim, duas outras questões são também abordadas no presente trabalho: (i) como inserir as técnicas e métodos ora propostos num processo de desenvolvimento de software que seja adequado a sistemas reais; e (ii) quais ferramentas de suporte para desenvolvimento de software são necessárias para apoiar a inclusão de tratamento de exceções nesses sistemas.

## 1.3. Solução Proposta

O foco central desse trabalho é no desenvolvimento de técnicas, métodos e ferramentas para o desenvolvimento centrado em arquitetura de software de sistemas tolerantes a falhas, baseados em componentes [96].

Um princípio fundamental da solução proposta é prover, na arquitetura do software, a máxima separação das responsabilidades pelo comportamento normal do sistema daquelas relativas a

tolerância a falhas. No caso de sistemas baseados em componentes, que integram componentes do tipo "caixa-preta" e de baixa confiabilidade, essa separação é, mais do que desejável, absolutamente necessária. No presente trabalho, essa separação de interesses é obtida através de dois modelos para configurações tolerantes a falhas, denominados iC2C (*idealised C2 Component*) [37] e iCOTS (*idealised COTS component*) [36] [38] [39] [40]. O iC2C é uma abordagem para estruturação de sistemas baseados em componentes arquiteturais ideais tolerantes a falhas, utilizando o estilo arquitetural C2 [98]. O iCOTS é uma solução arquitetural para transformar componentes de software de prateleira em componentes ideais tolerantes a falhas, adicionando aos mesmos invólucros protetores[4] [71]. Essas duas configurações integram componentes responsáveis pelo comportamento normal do sistema com outros elementos arquiteturais distintos, aos quais são atribuídas as responsabilidades por detecção e recuperação de erros. O interesse, neste trabalho, pelo estilo arquitetural C2 foi motivado pelas suas regras de comunicação entre componentes: somente através de mensagens enviadas por meio de conectores explícitos, que atuam como mediadores entre os vários componentes de um sistema. Essas restrições na comunicação permitem a integração de sistemas DBC com componentes heterogêneos e independentes, propiciando um maior grau de reutilização de software.

O nível de abstração da arquitetura do software impõe que, durante a implementação do sistema, sejam adicionados ao sistema diversos novos elementos, num processo de refinamentos sucessivos até a obtenção do código executável. Um dos principais problemas no desenvolvimento de software centrado na arquitetura é garantir a conformidade entre a implementação final assim obtida e a arquitetura do software, tal como projetada. Essa garantia é necessária para que possamos afirmar que o sistema obtido apresenta, de fato, as propriedades da arquitetura do software.

Na solução aqui proposta, a conformidade da implementação com a arquitetura do software é facilitada através da adoção de um modelo de componentes que oferece um mapeamento direto

[4] do inglês: *protective wrappers*.

das abstrações do nível arquitetural para elementos básicos das linguagem de programação utilizadas correntemente, como Java [94] e C# [61]. Esse modelo é uma extensão do modelo COSMOS [88] com os elementos básicos necessários para tratamento de exceções em sistemas DBC. Essa extensão inclui uma hierarquia de tipos de exceções e um conjunto de tipos de tratadores de exceções. O modelo COSMOS assim estendido permite a implementação de maneira sistemática de arquiteturas de software tolerantes a falhas baseada em componentes, tais como as baseadas nos modelos do iC2C e do iCOTS.

O desenvolvimento de sistemas com a complexidade dos sistemas críticos "modernos" normalmente requer uma equipe de desenvolvedores atuando de acordo com um processo de desenvolvimento de software definido. Esse desenvolvimento geralmente é apoiado por um ambiente integrado de desenvolvimento de software (IDE[5]), também denominado ambiente de engenharia de software. Esses ambientes integram um repositório de artefatos de software, tais como diagramas de classes e módulos de código fonte, e um conjunto de ferramentas que auxiliam os desenvolvedores nas tarefas de produção e análise desses artefatos. Uma classe importante de IDE são os chamados ambientes de engenharia de software centrados no processo (PCSEE[6]) [1]. Um PCSEE integra também a especificação do processo de desenvolvimento de software empregado. Essa visão do processo de desenvolvimento permite associar as funções oferecidas pelas várias ferramentas disponibilizadas pelo ambiente ao contexto específico das tarefas executadas pelos desenvolvedores. Essa contextualização serve de foco para a especificação das funções a serem oferecidas aos usuários do ambiente, o que evita a proliferação de funcionalidades de baixa utilidade e, conseqüentemente, torna o ambiente mais simples e fácil de usar.

A solução aqui proposta inclui ainda: (i) a extensão do processo UML Components para incluir o tratamento sistemático dos requisitos de tolerância a falhas, nas várias etapas do desenvolvimento; e (ii) a proposta de um ambiente integrado voltado para o desenvolvimento de sistemas confiáveis

[5] do inglês: *Integrated Development Environment*.
[6] do inglês: *Process-Centered Software Engineering Environment*.

baseados em componentes, centrado nesse processo de desenvolvimento estendido. O processo *UML Components* [17] foi escolhido como base para essa etapa do presente trabalho por ser um processo simples, de fácil entendimento, e que estende o processo *Unified Software Process* [47]. O RUP é um processo de desenvolvimento, baseado no paradigma de orientação a objetos, bem consolidado e amplamente utilizado no desenvolvimento de sistemas de informação atuais. A extensão do processo UML Components é tema de dois outros trabalhos de pesquisa que se desenvolvem em paralelo com o presente trabalho. No primeiro trabalho [68] é desenvolvido um estudo de caso onde esse processo é aplicado em conjunto com a metodologia MDCE [29]. A metodologia MDCE foi originalmente desenvolvida para o contexto do método Catalysis [25] e visa incorporar o comportamento excepcional no desenvolvimento de sistemas de software baseados em componentes. Como resultado desse estudo de caso, obteve-se uma primeira versão estendida do processo UML Components que incorpora a MDCE. No segundo trabalho [10] essa primeira versão estendida do processo é refinada e descrita formalmente.

Para desenvolvimento do ambiente integrado de desenvolvimento proposto neste trabalho, foi escolhida a plataforma Eclipse [26]. Eclipse é uma plataforma genérica para construção de IDE desenvolvida e mantida por um projeto de software de código aberto (*Open Source*). Embora relativamente recente, o projeto Eclipse já congrega uma grande comunidade de usuários e desenvolvedores de ferramentas. O desenvolvimento do ambiente proposto é também tema de outro trabalho de pesquisa que se desenvolve em paralelo com o presente trabalho [99].

## 1.4. Contribuições Originais

As principais contribuições do presente trabalho são:

1. Uma discussão dos conceitos do paradigma DBC relacionados com os aspectos de confiabilidade dos sistemas de software e do seu impacto nos mecanismos de tolerância a falhas [76].
2. O iC2C [37] [38], que é uma abordagem para estruturação de sistemas tolerantes a falhas baseados em componentes de software e no estilo arquitetural C2.

3. O iCOTS [36] [39] [40] [41], que é especialização do iC2C para transformação de componentes de software de prateleira em componentes ideais tolerantes a falhas, através da adição de invólucros protetores.
4. Uma extensão do modelo de componentes COSMOS para inclusão de uma estratégia geral e um conjunto de diretrizes para o desenvolvimento de componentes de software reutilizáveis e sua integração em sistemas confiáveis baseados em componentes [42].
5. A proposta de um ambiente integrado de desenvolvimento de software para construção de sistemas confiáveis baseados em componentes de software.

Dada a amplitude dos objetivos propostos, parte da pesquisa desenvolvida foi executada em colaboração com outros alunos de pós-graduação do Instituto de Computação. Os resultados dessas pesquisas estão apresentados em dissertações de mestrado e publicações em co-autoria, como segue:

6. Uma abordagem arquitetural para a composição confiável de sistemas componentes baseada em contratos de composição e num esquema de tratamento de exceções que considera a execução concorrente de componentes arquiteturais [84] [86].
7. O modelo de componentes COSMOS [87], que é um modelo independente de plataforma tecnológica para projeto e implementação de arquiteturas de software baseadas em componentes.
8. Uma extensão do processo UML Components [17] para incluir os modelos e atividades prescritas pela abordagem MDA (*Model Driven Architecture*) [63] [90].
9. O arcabouço de software orientado a objetos[7] FaTC2 [15] para construção de sistemas tolerantes a falhas baseados em componentes, com base no iC2C e no arcabouço de software do estilo arquitetural C2 chamado `c2.fw` [100].
10. O sistema de tratamento de exceções ALEx [14], que promove a adição, no nível arquitetural, de tolerância a falhas em sistemas baseados em componentes.

[7] do inglês: *object-oriented framework*.

Em continuidade ao trabalho dessa pesquisa estão em desenvolvimento duas outras dissertações de mestrado e uma nova tese de doutorado: a primeira dissertação [10] visa a formalização de uma extensão do processo UML Componentes para o desenvolvimento de sistemas tolerantes a falhas; a segunda dissertação [99] visa implementar o ambiente de integrado de desenvolvimento aqui proposto; a tese de doutoramento [16] dá continuidade ao desenvolvimento de arquiteturas de software tolerantes a falhas para sistemas baseadas em componentes de prateleira.

## 1.5. Organização da Tese

O restante deste documento está organizado da seguinte forma. O Capítulo 2 resume os fundamentos teóricos básicos desse trabalho, nas áreas de desenvolvimento baseado em componentes, arquiteturas de software e confiabilidade em sistemas computacionais.

O Capítulo 3 descreve os modelos do iC2C e do iCOTS, que são os principais resultados desse trabalho na área de arquiteturas de software tolerantes a falhas. O Capítulo 4 descreve a abordagem proposta para tratamento de exceções em DBC, que estende o modelo de componentes COSMOS. No Capítulo 5 é proposto um ambiente integrado para desenvolvimento de sistemas confiáveis baseados em componentes, que apoia a aplicação prática dos resultados descritos nos Capítulos precedentes. O Capítulo 6 apresenta dois estudos de caso, onde foram aplicados os resultados descritos nos capítulos anteriores. Finalmente, o Capítulo 7 apresenta as conclusões desse trabalho e propõe alguns temas para trabalhos futuros.

# Capítulo 2

# Fundamentos Teóricos

*Há sempre um modo correto e um modo errado.*
*O modo errado parece sempre mais razoável.*
*George Moore*

## 2.1. Desenvolvimento de Software Baseado em Componentes

Desenvolvimento de software baseado em componentes, ou DBC, é definido como o desenvolvimento de sistemas de software através da integração planejada de componentes de software reutilizáveis [11]. Embora o paradigma DBC só tenha se tornado realidade recentemente, a idéia de componentes de software surgiu ainda no final da década de 1960. Em outubro de 1968 [57] McIlroy propõe a criação de uma indústria de componentes de software, à semelhança da indústria de componentes de hardware, que servisse de base para a produção de sistemas de software em grande escala. Nessa seção apresentamos os principais conceitos de DBC que, após 35 anos, realizam essa visão precursora da indústria de software.

### 2.1.1. Conceitos Básicos de Componente de Software

Um **componente de software** é uma unidade de composição com interfaces especificadas contratualmente e somente dependências explícitas de contexto [97]. Uma **interface** identifica um ponto de interação entre um componente e o seu ambiente [33]. Esse ambiente pode ser constituído por outros componentes de software e partes de software não baseadas em componentes. As interfaces de um componente podem ser de dois tipos: (i) uma **interface provida** identifica um ponto de acesso a serviços que o componente é capaz de prover para o ambiente; e (ii) uma **interface requerida** identifica um ponto de acesso a serviços que o componente requer do ambiente. Na Figura 1 está representado, usando a notação UML [64], um componente de software com três interfaces: uma provida e duas requeridas.

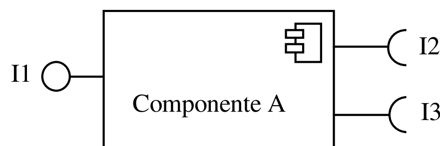


**Fig. 1.** Representação UML de um componente de software.

As dependências de contexto de um componente são especificadas através de um **modelo de componentes**. Um modelo de componentes especifica os padrões e convenções impostas aos desenvolvedores de componentes [5] e os pressupostos básicos que podem ser assumidos a respeito do ambiente. A forma de comunicação entre componentes de software e os protocolos correspondentes são exemplos de padrões normalmente especificados no modelo de componentes. Os modelos de componentes mais difundidos atualmente são J2EE [95] e .NET [60].

Um modelo de componentes pode especificar também uma **infra-estrutura de componentes**, a ser integrada ao ambiente dos sistemas. Essa infra-estrutura oferece serviços básicos para todos os componentes de software, tais como serviços de persistência e segurança de acesso. O modelo de componentes J2EE especifica um padrão aberto de infra-estrutura que possui diferentes implementações, tais como Websphere [45] e JBOSS [48]. O modelo .NET baseia-se numa infra-estrutura proprietária, que é integrada aos sistemas operacionais da família Microsoft Windows. A Figura 2 mostra a estrutura básica, em camadas, de um sistema DBC típico.

### 2.1.2. Especificação, Implementação e Instanciação de Componente

Um aspecto fundamental do paradigma DBC é a separação entre especificação de um componente de software e sua implementação. Uma **especificação de componente** define o comportamento observável externamente do componente, com a abrangência e precisão necessárias para sua integração em diferentes sistemas, porém abstraindo detalhes de qualquer implementação específica. Uma **implementação de componente** fornece um modelo concreto para instanciação, em diferentes ambientes, de componentes que realizam uma dada especificação.

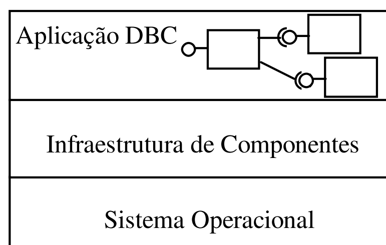


**Fig. 2.** Camadas de um sistema DBC.

Podemos dividir a especificação de um componente de software em quatro níveis contratuais [8], conforme a seguir. No nível **sintático** são especificadas as operações que podem ser executadas em cada interface, os parâmetros de entrada e de saída requeridos por essas operações e os tipos de condições excepcionais que podem ocorrer durante a execução dessas operações. No segundo nível, que é o **comportamental**, são especificadas as obrigações mútuas, entre o cliente dos serviços da interface e o componente que provê esses serviços. Essas obrigações são especificadas através de pré-condições e pós-condições, aplicáveis a cada operação num contexto puramente seqüencial. No nível de **sincronização** é especificado o comportamento global do componente, considerando suas várias interfaces e as dependências que possam existir entre os diversos serviços oferecidos pelo componente. No quarto e último nível, que é o de **qualidade de serviço**, são especificados os parâmetros quantitativos esperados para o comportamento das interfaces, tais como tempo de resposta, precisão dos resultados, dentre outros, ou, alternativamente, é especificada uma forma de negociação dinâmica desses parâmetros entre o cliente dos serviços e o componente que os provê.

Uma especificação de componente é, portanto, um componente de software abstrato descrito por artefatos de software que compõem o **memorial de especificação do componente**[8]. Esse memorial é utilizado como elemento básico no projeto de um sistema DBC. Em [6] é apresentada uma lista de características do componente a serem especificadas nesse memorial, classificadas em características obrigatórias, recomendadas e opcionais. Idealmente, essas características devem

[8] do inglês: *component blueprint*.

ser descritas formalmente como, por exemplo, interfaces descritas numa linguagem de definição de interfaces (IDL[9]). Entretanto, as limitações dos formalismos existentes obrigam, na prática, a utilização de modelos semiformais, como uma seqüência de operações descrita através de um diagrama UML, ou mesmo descrições informais.

A implementação de um componente de software requer um projeto interno que pode, recursivamente, ser também baseado em componentes. Um implementação de componente pode, ao mesmo tempo, ser composta por subcomponentes menores e ser parte de um componente maior. A implementação de um componente, no seu nível mais concreto, requer a utilização de uma linguagem de programação. A escolha de uma linguagem específica está sujeita apenas às restrições porventura impostas pela infra-estrutura de componentes adotada. Quando utilizada uma linguagem orientada a objetos, as operações especificadas nas interfaces providas pelo componente são implementadas através de uma ou mais **classes de implementação**.

Uma implementação de componente é, portanto, um componente de software concreto composto por vários artefatos de software, tais como: (i) o projeto interno do componente; (ii) o código fonte da implementação, numa linguagem de programação específica; e (iii) o módulo executável correspondente, no código binário especificado pela infra-estrutura de componentes adotada. Esses artefatos compõem o **pacote de implementação** do componente.

A **instanciação do componente** ocorre apenas no momento da execução de um sistema onde o componente está integrado. Quando a implementação do componente utiliza uma linguagem orientada a objetos, essa instanciação resulta na criação de um ou mais objetos das suas classes de implementação.

### 2.1.3. As Duas Fases do DBC

A separação entre especificação e implementação de componente visa, fundamentalmente, permitir a divisão da atividade de desenvolvimento de software em duas atividades independentes: a **produção de componentes de software** e a **integração de sistemas**. Os elos de ligação entre

[9] do inglês: *Interface Definition Language*.

essas duas atividades são os **repositórios de componentes**, onde os produtores de componentes depositam os artefatos produzidos (especificações e implementações de componentes). A integração de um sistema, idealmente, se resumiria a: (i) busca e seleção, naqueles repositórios, de componentes já existentes que ofereçam as funções requeridas pelo sistema; e (ii) ligação das interfaces providas e requeridas dos componentes selecionados, num ambiente gráfico para composição de componentes que dispensa o uso de uma linguagem de programação. No entanto, quando se deseja integrar componentes desenvolvidos independentemente, esse grau de simplicidade na integração do sistema dificilmente é possível. No caso mais geral, a integração de um sistema normalmente requer também um projeto de integração que pode incluir o desenvolvimento de **adaptadores de interfaces** e a especificação e implementação de novos componentes de software. Adaptadores de interfaces são partes de software responsáveis por resolver conflitos que possam existir entre as interfaces providas e requeridas de dois componentes que devam interagir. Um exemplo de adaptador de interface é um conversor de protocolo JDBC, utilizado por uma aplicação Java ao acessar um banco de dados, para o protocolo ODBC, requerido por um banco de dados relacional específico. Os novos componentes de software produzidos para um sistema específico podem também ser depositados num repositório de componentes, para reutilização posterior.

### 2.1.4. Classificação de Componentes

Quanto à **granularidade** da sua especificação, podemos distinguir três tipos componentes: de baixa, média e alta granularidade. Um componentes de baixa granularidade encapsula uma abstração simples e provê uma única interface com um conjunto reduzido de serviços. Os chamados **componentes visuais**, que encapsulam os elementos básicos de uma interface gráfica com o usuário, tais como botões e caixas de texto, são exemplos de componentes de baixa granularidade. No extremo oposto, um componente de alta granularidade encapsula um conjunto de abstrações de alto nível e, em geral, oferece uma variedade de serviços, através de diferentes

interfaces. Os chamados **componentes de negócio**[10], que são componentes de software autônomos abrangendo as funcionalidades de uma área de negócios como, por exemplo, o processamento de cartões de crédito [102] são exemplos de componentes de alta granularidade. Na escala intermediária, um componente de média granularidade encapsula uma regra de negócio específica ou um aspecto da infra-estrutura tecnológica como, por exemplo, a manutenção de uma conta-corrente ou o controle de acesso ao sistema. Componentes de média granularidade são utilizados como elementos básicos que colaboram entre si para compor sistemas ou outros componentes de maior granularidade.

Quanto à **visibilidade interna** da sua implementação, podemos distinguir três tipos de componentes: caixa-preta, caixa-cinza e caixa-branca. Um componente caixa-preta é uma implementação de componente que oculta, do integrador do sistema, o projeto interno do componente e o seu código fonte. Os chamados componentes comerciais de prateleira (COTS) como, por exemplo, bancos de dados comerciais proprietários, são exemplos desse tipo de componente. No extremo oposto da escala, um componente caixa-branca é uma implementação de componente que oferece, ao integrador do sistema, uma visão irrestrita de todos os detalhes do seu projeto interno e código fonte. Os componentes de software oferecidos sob licenças ditas "de código aberto"[11] são exemplos desse tipo de componente. Na escala intermediária, um componente caixa-cinza é uma implementação de componente que oferece, ao integrador do sistema, alguns detalhes selecionados do seu projeto interno e código fonte.

Quanto à **adaptabilidade** do componente a diferentes contextos, podemos distinguir duas formas de adaptação que podem ser oferecidas pelo próprio componente: através de interfaces de configuração ou de uma implementação extensível. Com relação às interfaces de configuração, podemos distinguir três tipos de especificação de componentes: **não adaptáveis**, **adaptáveis estaticamente** e **adaptáveis dinamicamente**. Com relação à extensibilidade da implementação, podemos distinguir dois tipos de implementação de componentes: **não extensíveis** e **extensíveis**.

[10] do inglês: *business components*.
[11] do inglês: *open source*.

Uma **interface de configuração** provê operações que permitem ao integrador do sistema adaptar o comportamento do componente a diferentes contextos previstos na especificação do componente. Uma interface de configuração pode ser estática, se habilitada apenas durante a fase de instanciação do componente, ou dinâmica, caso permita também a reconfiguração do componente durante a fase de execução do sistema. Componente visuais são, em geral, exemplos de componentes adaptáveis através de interfaces de configuração, que permitem ao integrador modificar atributos como dimensões e cores utilizadas.

Um **componente extensível** expõe ao integrador do sistema partes selecionadas da sua implementação (**pontos de extensão**) que permitem o refinamento da sua estrutura interna ou do seu comportamento. Dessa forma, o componente pode ser adaptado a novos contextos, não previstos na sua especificação. Componente extensíveis são comumente encontrados nos arcabouços de software baseados em componentes, como o SanFrancisco [62].

### 2.1.5. Processos de Desenvolvimento Orientados a DBC

Os processos de desenvolvimento de software convencionais não incluem nos seus objetivos a reutilização de componentes de software. Através desses processos, essa reutilização pode ocorrer apenas ocasionalmente e de maneira não sistemática. Por outro lado, um processo de desenvolvimento orientado a DBC deve, como um dos seus principais objetivos, promover essa reutilização de componentes em larga escala.

Tipicamente, podemos dividir um processo de desenvolvimento baseado em componentes em seis estágios [17]: requisitos, especificação, provisionamento, montagem (ou integração), teste e instalação. O estágio de requisitos visa identificar os requisitos do sistema. Durante a fase de especificação o sistema é decomposto num conjunto de componentes abstratos com responsabilidades específicas, que interagem de forma a satisfazer os requisitos do sistema. Esses componentes são instanciados no estágio de provisionamento. Durante esse estágio o integrador do sistema decide se um componente abstrato pode ser concretizado por um componente de prateleira, aqui chamado **instância COTS**, ou se irá requerer um esforço de implementação, caso em que é chamado de **instância doméstica**. Durante a fase de montagem o integrador do sistema

constrói o sistema final através da integração das instâncias COTS e domésticas. Esse esforço de integração inclui o desenvolvimento do código necessário para interligação das interfaces dos vários componentes, o que inclui a especificação e implementação de invólucros[12] para adaptação das interfaces das instâncias COTS utilizadas. Durante o estágio de teste o sistema já integrado é testado e são feitas as correções necessárias para assegurar que o sistema está em conformidade com suas especificações e os seus requisitos estão sendo satisfeitos. Na fase de instalação o sistema final é instalado no ambiente onde será utilizado.

A característica essencial desses processos é a separação entre os estágios de provisionamento e de integração de sistemas. Um dos primeiros processos descritos na literatura que possui essa característica é o IIDA (*Infrastructure Incremental Development Approach*) [30]. O IIDA combina e estende métodos em cascata e em espiral para adaptá-los ao desenvolvimento de infra-estruturas de software baseadas em componentes COTS. Outros processos orientados a DBC, mais recentes e que obtiveram maior divulgação e aceitação, são o Catalysis [25], o UML Components [17] e o KobrA [4]. O Catalysis é, dentre esses três métodos, o mais abrangente e complexo. Essa complexidade tem se mostrado um sério obstáculo à sua adoção de maneira generalizada. O UML Components é, ao contrário, o mais restrito e simples dos três. O UML Components é uma extensão do processo RUP [47] que visa, especificamente, o desenvolvimento de sistemas baseados em componentes e estruturados de acordo com uma arquitetura de software geral pré-estabelecida, em quatro camadas. Essa arquitetura geral pré-estabelecida é adequada a sistemas de informações baseados na Web, que constitui uma parcela expressiva dos sistemas atuais. Essa restrição arquitetural contribui para tornar o processo mais simples e de fácil entendimento. O processo KobrA situa-se numa escala intermediária de abrangência e complexidade. O objetivo central do KobrA é o estabelecimento de um processo, baseado em transformações de modelos, para desenvolvimento de uma **família de produtos**[13] de derivados de uma arquitetura de software geral definida durante esse processo.

---

[12] do inglês: *wrappers*.

## 2.2. Arquitetura de Software

A arquitetura de um sistema de software mostra como o sistema é realizado por um conjunto de **componentes arquiteturais** e as interações entre eles [80]. Seu foco principal é em como as responsabilidades do sistema são divididas entre os vários componentes arquiteturais e na forma como esses componentes interagem, abstraindo os detalhes de implementação internos aos componentes. Os componentes arquiteturais podem ser definidos em diferentes níves de abstração, podendo representar tanto rotinas de uma biblioteca procedural, como componentes de software como definidos no paradigma DBC ou grandes subsistemas. Em geral, o termo **componente** é utilizado, conforme o contexto, para designar tanto um componente arquitetural como um componente de software do paradigma DBC. Para que haja interação entre componentes é necessária alguma forma de comunicação entre eles. A responsabilidade pelo estabelecimento dessa comunicação é atribuída a elementos arquiteturais denominados **conectores**. Um conjunto de componentes interligados por conectores define uma **configuração** arquitetônica. A importância da abordagem de arquitetura do software é sentida à medida em que aumenta o porte e a complexidade dos sistemas, sendo um fator condicionante de diversos aspectos qualitativos do sistema, tais como desempenho, escalabilidade e segurança [18].

Embora a disciplina de arquitetura de software não se limite a sistemas desenvolvidos de acordo com o paradigma DBC, há uma forte sinergia entre essas duas disciplinas.

### 2.2.1. Linguagens de Descrição de Arquitetura

A arquitetura de uma sistema é normalmente expressa através de um conjunto de diagramas representando diferentes **visões arquiteturais** do sistema, como: a estrutura funcional, a estrutura do código, a estrutura de concorrência, a estrutura física, e a estrutura de desenvolvimento [51]. Embora tais diagramas sejam extremamente eficazes para a comunicação entre arquitetos, desenvolvedores e usuários do sistema, não possuem o formalismo necessário para que possam ser empregados em outras etapas do desenvolvimento, como análise da arquitetura e geração de

---

[13] do inglês: *product line*.

código, por exemplo. Para isso, são utilizadas linguagens formais de descrição de arquitetura (ADL[14]) tais como UniCom [79], Acme [33] e xADL [21]. As abstrações principais de uma ADL são componentes, conectores e configurações. Os componentes são descritos através das suas interfaces. No tocante a essas interfaces, as mesmas limitações de expressividade das IDLs (Seção 2.4.1) são encontradas também nessas ADLs.

### 2.2.2. Análise de Arquitetura

A análise da arquitetura de software de um sistema visa determinar, ainda num estágio inicial do desenvolvimento, problemas que possam vir a se manifestar após a implementação do sistema. A partir de estimativas de latência, velocidade de transmissão, capacidade de processamento e carga do sistema, por exemplo, devemos poder estimar o tempo de resposta esperado para uma determinada transação considerando-se diferentes arquiteturas de software.

Em [7] é descrito o método denominado ATAM (*Architecture Tradeoff Analysis Method*), baseado em modelos matemáticos para os diferentes atributos em que se está interessado, que são desenvolvidos para determinados cenários de uso considerados representativos para o sistema em questão.

O uso efetivo de DBC para sistemas corporativos de grande porte depende não só de tecnologia para implementação de componentes como também para integração desses componentes de maneira sistemática e previsível. Isso requer técnicas para especificar o comportamento desse componentes de forma a permitir o raciocínio sobre os mesmos, a análise e comparação de componentes abstraindo-se dos seus detalhes de implementação [11].

Em [35] é proposta uma abordagem para composição de sistemas que diverge da visão de componentes e conectores. Nessa abordagem, denominada ACTI (*Abstract, Concrete, Templates, Instances*), a interação entre componentes (ou *instances*) é feita sempre através de gabaritos de software (*templates*) que definem novos componentes, cujas propriedades podem ser deduzidas diretamente das propriedades de suas instâncias. Essa propriedade de análise modular (*modular*

[14] do inglês: *Architecture Description Language*.

*reasoning*) é considerada essencial para que se possa controlar a complexidade do trabalho de análise de arquitetura.

### 2.2.3. Estilos Arquiteturais e Padrões Arquiteturais

Um estilo arquitetural define um conjunto de tipos de componentes e conectores que podem ser utilizados para compor um sistema e um conjunto de restrições impostas às configurações que podem ser criadas dessa forma [82]. O estilo arquitetural chamado "em camadas" (Figura 3) é um exemplo de estilo freqüentemente encontrado. Nesse estilo, um sistema é dividido em duas ou mais camadas que se sobrepõem. Cada camada corresponde a um nível diferente de abstração, que é sempre crescente de uma camada inferior para outra superior. A camada mais inferior abstrai a plataforma de hardware, ocultando a complexidade e os detalhes de uma plataforma de hardware específica e oferecendo, para as camadas superiores, uma interface de acesso de mais alto nível. Cada camada superior abstrai outros aspectos do sistema como, por exemplo, o mecanismo de persistência de dados utilizado, até se atingir, na camada mais superior do sistema, o nível de abstração dos serviços oferecidos pelo sistema para os seus usuários.

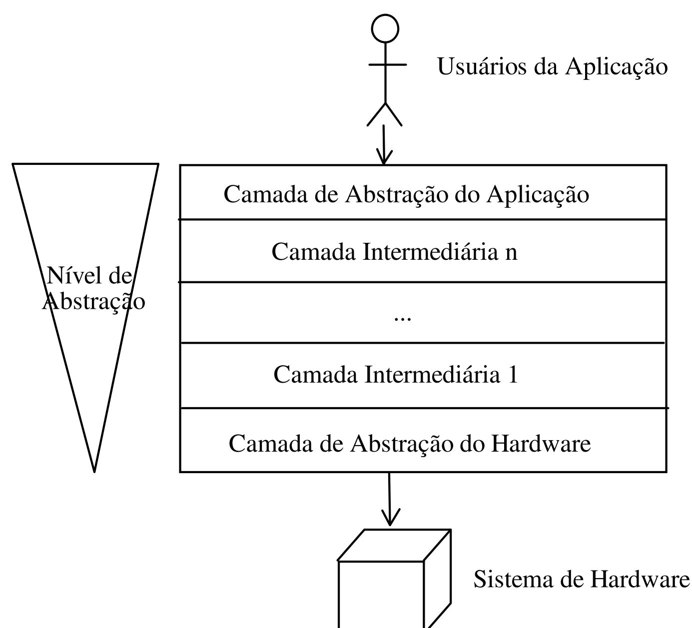


**Fig. 3.** Estilo arquitetural em camadas.

Os estilos arquiteturais mais conhecidos foram descritos inicialmente de maneira informal e não sistematizada, através de definições discursivas e diagramas ilustrativos. A comunidade de padrões se dedica a documentar soluções já comprovadas, incluindo estilos arquiteturais, no contexto de tipos de problemas específicos em que cada solução é aplicável [82]. Surgiram, assim, os chamados **padrões arquiteturais** [12], que são estilos arquiteturais organizados e documentados de forma sistemática.

Um estilo arquitetural define uma família de arquiteturas de software, num nível de abstração mais elevado que o de uma configuração específica de componentes e conectores. A análise de um estilo arquitetural, nesse nível de abstração, permite que se deduzam propriedades que se irão se verificar em qualquer configuração que possa ser criada em conformidade com aquele estilo. São essas propriedades gerais dos vários estilos arquiteturais que, no desenvolvimento de sistema em particular, motivam a escolha de um ou mais estilos arquiteturais para guiar o projeto de sua arquitetura de software.

### 2.2.4. O Estilo Arquitetural C2

C2 [98] é um estilo arquitetural voltado para desenvolvimento baseado em componentes, que visa alcançar um elevado grau de reutilização de componentes e permitir a composição desses componentes de forma flexível, enfatizando o fraco acoplamento entre os componentes. O estilo C2 surgiu no desenvolvimento de interfaces de usuário gráficas. No estilo C2, os componentes interagem por meio de mensagens, em geral assíncronas, mediadas por conectores explícitos. Os conectores são responsáveis pelo roteamento, difusão e filtragem das mensagens. Os componentes podem ser encapsulados em invólucros que resolvem **conflitos arquiteturais**[15] [32] que possam existir entre os vários componentes, adaptando as interfaces e serviços de infra-estrutura requeridos pelo componente às interfaces e serviços de infra-estrutura providos pela configuração onde está sendo inserido. Dessa forma, um componente pode interagir num sistema formado por componentes que lhe são desconhecidos e desenvolvidos com base em suposições

[15] do inglês: *architectural mismatches*.

arquiteturais conflitantes. Essa característica é especialmente adequada para integração de componentes de software desenvolvidos independentemente, como componentes de prateleira, que podem ser baseados em estilos arquiteturais, linguagens de programação e infra-estruturas de componentes heterogêneos.

No estilo C2, tanto componentes como conectores (Figura 4) possuem duas interfaces, que se convencionou chamar de **interface de cima** e **interface de baixo**. As configurações são formadas usando um estilo em camadas: uma interface de cima (baixo) só pode ser conectada a uma interface de baixo (cima) de outro elemento arquitetural. Um componente só pode ser conectado a, no máximo, dois conectores: um acima e outro abaixo. Um conector pode ser conectado a um número qualquer de componentes ou outros conectores.

São dois os tipos de mensagens em C2: **requisições** e **notificações**. Por convenção, as requisições fluem para cima, através das várias camadas do sistema, e as notificações fluem para baixo. Um componente emite requisições através da sua interface de cima. Essas requisições seguem, através do conector ligado à interface de cima do componente, para um ou mais elementos (componentes ou conectores) situados na camada imediatamente superior, que estejam ligados à interface de cima daquele conector. Dessa forma, e dependendo das regras de difusão e filtragem definidas pelos conectores, uma mesma requisição pode atingir um número qualquer de componentes, que as recebem através de suas interfaces de baixo. Ao receber uma requisição um componente pode executar uma de suas operações e, eventualmente, emitir uma notificação através de sua interface

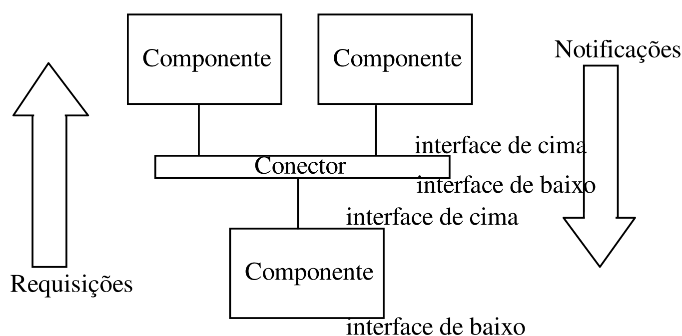


**Fig. 4.** Elementos básicos do estilo arquitetural C2.

de baixo. Essas notificações seguem o caminho inverso das requisições, podendo também alcançar um número qualquer de componentes situados em camadas inferiores da arquitetura, incluindo o componente que emitiu aquela requisição. Ao receber uma notificação um componente pode reagir de forma similar à uma requisição, executando uma de suas operações e, eventualmente, emitindo uma nova notificação.

## 2.3. Confiabilidade em Sistemas Computacionais

### 2.3.1. Conceitos de Defeito

Quando um sistema computacional deixa de oferecer as funcionalidades previstas ou o faz em desacordo com suas especificações dizemos que ocorre um defeito no sistema [52]. A emissão de um extrato bancário com o saldo incorreto é um exemplo de defeito para um sistema bancário.

De forma mais abrangente, definimos defeito como um desvio entre o comportamento esperado de um sistema e o seu comportamento real. Isso porque as especificações do sistema podem ser omissas ou mesmo inconsistentes quanto ao "comportamento esperado" em determinada situação. Um defeito é dito catastrófico quando resulta no colapso total do sistema. Em outros casos, pode haver apenas uma degradação nos serviços oferecidos pelo sistema, que mantém parte de sua funcionalidade intacta. Se o próprio sistema é capaz de administrar essa degradação, minimizando as conseqüências negativas do defeito, dizemos que há uma degradação suave.

Três métricas importantes, relacionados com os defeitos de um sistema, são:

1. Tempo Médio até um Defeito (MTTF[16]) - é uma medida do tempo esperado de operação normal e contínua de um componente, até que apresente algum defeito;
2. Tempo Médio de Reparo (MTTR[17]) - é uma medida do tempo esperado necessário para que, após um defeito, o componente seja reparado ou substituído e colocado novamente em operação normal;
3. Tempo Médio entre Defeitos (MTBF[18]) - é a soma das duas medidas acima (MTTF + MTBF).

### 2.3.2. Atributos de Confiabilidade

Todos os atributos de confiabilidade estão relacionados com o conceito de defeito apresentado na seção anterior. Intuitivamente, quanto menor a probabilidade de ocorrência de algum defeito e

[16] do inglês: *Medium Time To Fail.*
[17] do inglês: *Medium Time To Repair.*
[18] do inglês: *Medium Time Between Failures.*

menos severas forem as suas conseqüências, maior será a confiança que podemos depositar num sistema, ou seja, maior será a sua confiabilidade.

A **disponibilidade** de um sistema expressa a garantia de que o sistema estará apto a fornecer seus serviços no momento em que desejarmos. Ela é definida como a probabilidade de que o sistema esteja em funcionamento em algum instante $t$ futuro. Sistemas de telefonia são exemplos clássicos de aplicações que exigem elevada disponibilidade. Paralisações freqüentes (baixo MTTF) ou prolongadas (alto MTTR) afetam negativamente esse atributo.

A **confiabilidade**[19] de um sistema expressa a garantia de que o sistema é capaz de manter-se em funcionamento ininterrupto pelo tempo exigido por uma dada missão. É definida como a probabilidade de que o sistema funcione ininterruptamente por um período de tempo $\Delta t$. Sistemas embarcados em satélites, nos quais as possibilidades de intervenção para reparo são mínimas, são exemplos clássicos de aplicações que exigem alta confiabilidade. Um único defeito catastrófico implica na perda do sistema inteiro e, eventualmente, no fracasso da missão.

Enquanto os dois atributos acima estão relacionados primordialmente com a probabilidade de ocorrência de um defeito, os demais atributos referem-se à gravidade das conseqüências de algum defeito no sistema.

A **segurança no uso**[20] de um sistema expressa a garantia de que o sistema é capaz de preservar a integridade e confidencialidade das informações que mantém, mesmo na presença de defeito. Aplicações militares e bancárias são exemplos de aplicações que exigem sistemas seguros.

A **proteção**, ou **segurança no funcionamento**[21], expressa a garantia de que o sistema é capaz de evitar conseqüências catastróficas, tais como, perdas de vidas humanas ou danos materiais, em decorrência da sua utilização, mesmo na presença de defeito. Aplicações médicas e industriais, como no controle de usinas nucleares, são exemplos de aplicações com requisitos de segurança no funcionamento máxima.

---

[19] do inglês: *reliability*.
[20] do inglês: *security*.
[21] do inglês: *safety*.

### 2.3.3. Conceitos de Erro e Falha

O comportamento de um sistema é função de seu estado interno e das entradas que recebe. A ocorrência de um defeito, portanto, pode ser associada a um determinado estado interno do sistema que, sob determinadas entradas, provoca a ocorrência do defeito. Esse estado interno do sistema é chamado de erro. O fornecimento de um saldo incorreto no extrato bancário, por exemplo, pode ser provocado por um erro no valor de um lançamento registrado no sistema. Um erro é, portanto, de natureza estática e interna ao sistema, podendo ou não provocar um defeito.

Para que haja um erro e, eventualmente, venha a ocorrer um defeito haverá necessariamente alguma causa, a qual chamamos de falha. Voltando ao exemplo acima, o erro no valor do lançamento pode ter sido causado por uma descarga eletromagnética que tenha interferido na gravação dos dados do lançamento no banco de dados. Uma falha é, portanto, um evento que ocorre aleatoriamente e que acarreta uma alteração indevida no seu estado interno, provocando um erro.

Há sempre uma latência entre o momento em que ocorre uma falha, o surgimento de um erro e a ocorrência de um defeito. Nem sempre uma falha resulta em erro, ou este em defeito. Usando o mesmo exemplo acima, o defeito só ocorreu no momento da emissão do extrato, possivelmente horas ou dias após o surgimento do erro. Aquele valor incorreto poderia ter sido corrigido pelo próprio sistema antes da emissão do extrato, por exemplo, através de uma operação de atualização ou sincronização do banco de dados. Um defeito num componente pode significar uma condição de falha para outro componente que dele dependa. Uma sucessão de eventos em que uma falha num componente produz outra falha em outro componente é denominada **propagação de falha**.

### 2.3.4. Categorias de Falhas e Hipóteses de falhas

Sistemas computacionais funcionam, em geral, interagindo com um ambiente externo diversificado e em constante mutação, formado por usuários, operadores, desenvolvedores, dispositivos de entrada e saída, redes de comunicação, sistemas de fornecimento de energia, outros sistemas computacionais, o meio-ambiente natural, dentre outros possíveis agentes.

Essa complexidade nos obriga, durante o desenvolvimento de um sistema, a criarmos um modelo simplificado do ambiente externo: ignora-se determinados agentes e definem-se padrões de interação idealizados para os demais. Quanto maior a distância entre esse modelo de projeto e as condições reais do ambiente em que é utilizado, maior será a probabilidade de ocorrerem eventos capazes de perturbar o funcionamento normal do sistema, e que serão percebidos pelo mesmo como falhas.

Uma falha é dita **permanente** quando exige uma ação externa de manutenção do sistema para remover seus efeitos e reparar o agente causador, como no caso do rompimento de um cabo de comunicação. **Falhas intermitentes** são aquelas que, embora também exijam uma ação externa para reparar o agente causador, não se manifestam permanentemente. O desgaste de um componente mecânico, por exemplo, pode, sob certas circunstâncias, interferir no funcionamento de algum dispositivo, de modo intermitente. Finalmente, **falhas transientes** só se manifestam por um curto período de tempo desaparecendo espontaneamente sem que seja necessária qualquer ação externa, como no caso de uma sobretensão momentânea na rede de alimentação elétrica.

Uma classe importante de falhas engloba as **falhas de projeto**, que ocorrem nas fases de projeto e implementação de um sistema computacional. Uma falha de projeto introduz no próprio sistema, de forma permanente, "mecanismos" capazes de levá-lo a um estado interno inconsistente. Quanto mais complexo for um componente, maior a probabilidade de conter falhas desse tipo. **Falhas de *software***, em componentes de natureza imaterial, são sempre falhas de projeto. Componentes materiais, por outro lado, estão sujeitos também a falhas ambientais ou por desgaste físico, também chamadas **falhas de *hardware***.

Uma questão fundamental no desenvolvimento de sistemas confiáveis é o estabelecimento de hipóteses a respeito dos tipos de falhas a que o sistema estará sujeito. Durante o desenvolvimento de um sistema são assumidas diversas hipóteses de falhas, que limitam o escopo das várias estratégias de confiabilidade utilizadas no desenvolvimento do sistema. As hipóteses de falhas assumidas na fase de prevenção de falhas, por exemplo, podem não ser as mesmas assumidas na

fase de tolerância a falhas. É essencial, portanto, que em cada fase desse processo sejam definidas claramente as hipóteses de falhas assumidas [49].

### 2.3.5. Prevenção de Falhas

Uma vez que a ocorrência de alguma falha é a causa fundamental de qualquer defeito, a abordagem mais intuitiva para elevarmos a confiabilidade de um sistema é adotar medidas preventivas que evitem a ocorrência de falhas. Para isso precisamos identificar os tipos de falha a que o sistema está exposto e os agentes causadores das mesmas para então adotarmos as medidas preventivas adequadas, como a seguir.

**Falhas de projeto** estão entre as principais causas de defeitos em sistemas computacionais e merecem uma atenção especial para sua prevenção. Para isso, podemos atuar sobre três aspectos do desenvolvimento do sistema:

a) Qualificação das pessoas encarregadas do desenvolvimento e manutenção do sistema, o que inclui o treinamento e a certificação desses profissionais;

b) Qualidade do processo de desenvolvimento, o que inclui o emprego de técnicas de engenharia de *software*, tais como métodos de especificação rigorosa, e a certificação do processo de desenvolvimento;

c) Qualidade do produto final obtido, o que inclui o emprego de técnicas sistemáticas de teste e certificação do sistema.

**Falhas de componentes** incluem as falhas em componentes de prateleira, produzidos por terceiros e que estejam integrados ao sistema em questão. Para prevenção desse tipo de falha podemos atuar sobre:

d) A quantidade de componentes utilizados, evitando o uso de componentes comerciais de prateleira de procedência ou qualidade duvidosas;

e) A qualidade dos componentes utilizados, selecionando componentes cujas especificações estejam em conformidade com as exigidas pelo sistema tanto nos aspectos funcionais como de qualidade. Inclui a certificação de qualidade tanto do produto como do seu fornecedor, quando aplicável;

f) A complexidade dos componentes utilizados, privilegiando o uso de componentes mais "leves" e evitando componentes que introduzam mais dependências ou funcionalidade desnecessária;

g) A integração do componente no sistema, incluindo testes sistemáticos do componente nas condições em que será empregado.

Outros tipos de falhas, não consideradas no presente trabalho, são as falhas humanas e ambientais. **Falhas humanas** são aquelas provocadas acidentalmente ou intencionalmente por um operador, usuário ou qualquer pessoa que tenha acesso ao sistema. **Falhas ambientais** são aquelas causadas pelos agentes da natureza e do meio-ambiente físico em que o sistema é utilizado.

### 2.3.6. Tolerância a Falhas

Embora a prevenção de falhas seja essencial para conferir confiabilidade a um sistema, a completa eliminação da possibilidade de ocorrência de falhas é uma meta inalcançável. A complexidade, o dinamismo e a autonomia dos agentes causadores dessas falhas limitam fortemente o controle que podemos exercer sobre os mesmos. Há também limitações práticas quanto ao custo e ao tempo que podem ser despendidos nessa atividade.

Tolerância a falhas é uma abordagem, complementar à prevenção de falhas, em que se considera, no projeto do sistema, a possibilidade de ocorrências de determinados tipos de falhas. Dessa forma, são incluidos no próprio sistema mecanismos para reagir a essas ocorrências de forma previsível, sem apresentar defeito. Sistemas com essas características são ditos **tolerantes a falhas** ou **robustos**.

O principal obstáculo à implementação de tolerância a falhas é a imprevisibilidade de ocorrência de uma falha e dos seus efeitos sobre o sistema. A ação dos mecanismos de tolerância a falhas só é iniciada com a **detecção de erros** já presentes no sistema, ou seja, erros resultantes de alguma falha ocorrida anteriormente.

Uma vez detectado um erro, este deve ser removido e, quando necessário, o sistema deve ser reconfigurado para isolar o agente provável causador da falha. Essas atividades são chamadas,

genericamente, de **recuperação de erro**. Após a recuperação do erro espera-se que o sistema possa prosseguir em seu funcionamento normal, sem apresentar defeito.

Dizemos que um erro é **mascarado** quando a detecção e recuperação do erro ocorrem simultaneamente, isto é, sem afetar o comportamento externo do componente onde foi detectado o erro. No caso mais geral, a detecção de um erro resulta apenas na sinalização de sua existência para que o sistema possa executar as ações de recuperação mais apropriadas ao contexto atual e seu estado global. Essa sinalização é feita normalmente por meio do lançamento de **exceções** e a recuperação do erro em rotinas especiais de tratamento dessas exceções, chamadas de **tratadores de exceção**.

### 2.3.7. Detecção de Erros

A detecção de erros num sistema pode ser feita através de mecanismos integrados ao *hardware* ou através de código adicionado ao sistema de *software*. A detecção de erros por *hardware* surgiu com os primeiros computadores em decorrência da baixa confiabilidade dos componentes eletrônicos e eletromecânicos então utilizados, tais como válvulas e relés. Isso obrigava a inclusão de circuitos especiais dedicados a verificar o funcionamento de cada módulo. A incorporação desses mecanismos no *hardware*, porém, aumenta substancialmente sua complexidade e custo. Atualmente, a elevada confiabilidade dos circuitos integrados e "*microchips*" tem motivado a eliminação desses mecanismos em projetos de *hardware* de uso geral, como processadores e memórias, em favor do aumento de sua capacidade, funcionalidade e redução de preço.

As técnicas básicas utilizadas por esses mecanismos são: (i) códigos redundantes, como *bits* de paridade e códigos CRC, que são verificados a intervalos regulares de tempo ou sempre que a informação codificada é processada; (ii) estabelecimento de limites de tempo para que determinadas funções sejam completadas; e (iii) módulos redundantes, onde uma mesma operação é executada simultaneamente por dois componentes idênticos (réplicas) e os resultados obtidos são comparados.

Dentre inúmeras variações dessas técnicas, algumas permitem o mascaramento das falhas, como por exemplo:

a) Códigos ECC, que são códigos com elevada redundância e que permitem a correção de certos tipos de erros;

b) Redundância tripla ou múltipla (TMR ou NMR[22]), que consiste no uso de três ou mais réplicas de um mesmo componente para permitir a comparação dos resultados e a seleção de um resultado considerado correto;

c) Diversidade de projeto, que é uma variação das técnicas TMR e NMR na qual, ao invés de réplicas idênticas de um mesmo componente, são utilizados diferentes implementações de componentes funcionalmente equivalentes. Essas implementações se diferenciam em aspectos como: projeto interno, linguagem de programação utilizada, procedência, dentre outros. O objetivo é permitir a detecção e o mascaramento também de falhas de projeto, que não são detectadas com o uso de réplicas idênticas.

A principal vantagem da detecção de erros por *software* é o baixo custo desses mecanismos, quando comparados com os mecanismos de *hardware*. Isso justifica, em muitos casos, a migração das funções de verificação, antes incluídas no *hardware*, para as camadas de *software*.

Desta forma, algumas das técnicas de detecção de erros por *hardware* também são utilizadas no *software* com as devidas adaptações, tais como, códigos redundantes, limites de tempo e módulos redundantes, tanto com o uso de réplicas idênticas como com diversidade de projeto.

Algumas técnicas mais específicas para detecção de erros por *software*, são:

d) Verificações estruturais - são procedimentos executados periodicamente para verificar a integridade das estruturas de dados do sistema;

e) Assertivas executáveis - são testes incluídos em pontos estratégicos da computação visando a verificar se determinadas condições previstas no projeto do sistema estão presentes. Testes de pré-condições, pós-condições e invariantes são exemplos desse tipo de verificação;

[22] do inglês: *Triple Module Redundancy* e *N-Module Redundancy*.

f) Detecção algorítmica - são algoritmos especializados para verificar a validade dos resultados fornecidos por uma função do programa em função das entradas que recebe, como quando fazemos uma "prova dos nove", para verificar se o resultado de uma soma é compatível com suas parcelas.

### 2.3.8. Recuperação de Erros

A recuperação de um erro consiste em restabelecer a consistência do estado interno do sistema. De uma forma geral, há dois tipos de estratégias para recuperação de erros. **Recuperação de erros por avanço** tenta levar o sistema a um estado livre de erros aplicando correções ao estado danificado do sistema. Esta técnica requer um certo conhecimento dos erros que possam acontecer ao nível da aplicação. Exceções e tratamento de exceções constituem um mecanismo comumente aplicado para a provisão de recuperação de erros por avanço. **Recuperação de erros por retrocesso** tenta retornar o sistema para um estado prévio livre de erros, não requerendo nenhum conhecimento específico dos erros que possam ocorrer. Como falhas de *software* e os erros causados por elas são de natureza imprevisível, recuperação de erros por retrocesso é geralmente aplicada para tolerar esse tipo de falha. Bancos de dados transacionais e mecanismos de *checkpoint* / *restart* são exemplos de sistemas que empregam recuperação de erros por retrocesso. Quando as características das falhas são bem conhecidas, recuperação de erros por avanço proporcionam uma solução mais eficiente [13].

### 2.3.9. Mecanismos de Tratamento de Exceção

Uma parcela expressiva do código e da complexidade de um sistema de alta confiabilidade está relacionada com as rotinas de detecção e recuperação de erros [72]. Paradoxalmente, isso implica na elevação da probabilidade de ocorrência de falhas de projeto e, conseqüentemente, pode afetar negativamente a confiabilidade do sistema como um todo.

A introdução de tolerância a falhas no *software* de um sistema exige, portanto, técnicas de desenvolvimento que minimizem a interferência negativa do código das rotinas relacionadas com a detecção e recuperação de erros nas funções normais do *software*. Nesse contexto, os chamados mecanismos de tratamento de exceções possuem uma importância fundamental ao permitir a

separação física do código das rotinas de tratamento de exceções do código normal. Esses mecanismos são também responsáveis pelo controle do fluxo de execução desde o momento em que é lançada uma exceção até o retorno do processamento ao seu fluxo normal, após a execução do tratador de exceção apropriado.

### 2.3.10. Componente Ideal Tolerante a Falhas

A complexidade e eficácia dos mecanismos de tolerância a falhas são fortemente dependentes da arquitetura do *software* [8]. A abordagem denominada **componente ideal tolerante a falhas** [72] visa a reduzir a complexidade desses mecanismos utilizando um estratégia de divisão e conquista, conforme descrito a seguir (Figura 5).

O sistema é subdividido em componentes, atribuindo-se a cada um uma parcela da responsabilidade pelas funções de tolerância a falhas. Esses componentes, por sua vez, podem ser recursivamente subdivididos em componentes menores, utilizando-se a mesma estratégia. Os componentes se comunicam apenas por meio de mensagens de requisições de serviços e respostas a essas requisições. Quando um componente não é capaz de atender a uma requisição é lançada uma exceção, que pode ser de uma das seguintes categorias:

1. Exceção de interface, que sinaliza o recebimento de uma requisição de serviço inválida

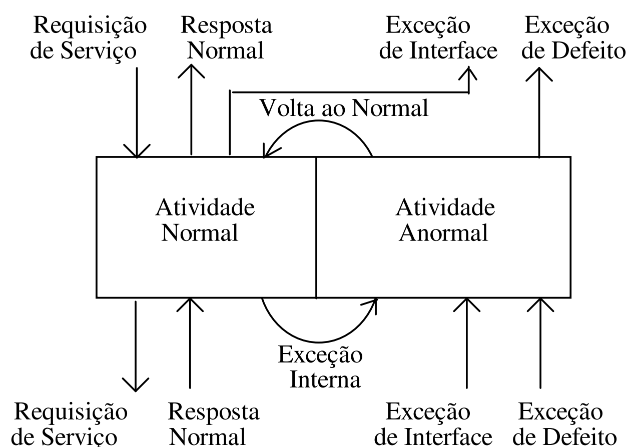


**Fig. 5.** O componente ideal tolerante a falhas.

para o componente e que deve ser tratada pelo componente que a enviou;

2. Exceção interna, que sinaliza um erro a ser tratado pelo componente que recebeu a requisição o qual, após as ações de recuperação, poderá retornar uma resposta normal;
3. Exceção de defeito, que sinaliza um erro impossível de ser tratado pelo componente que recebeu a requisição e que deverá ser tratado pelo componente que enviou a requisição.

As interfaces de um componente ideal tolerante a falhas declaram explicitamente todos os tipos de exceções, de interface ou defeito, que possam ser lançadas em resposta aos serviços especificados por essas interfaces. A implementação de um componente ideal tolerante a falhas, por sua vez, contém tratadores apropriados para todos os tipos de exceções que possam ser lançadas em resposta aos serviços requeridos pelo componente, que são as exceções externas, além dos tratadores das exceções internas, levantadas pelo próprio componente.

## 2.4. Confiabilidade de Software no Paradigma DBC

Historicamente, o desenvolvimento de sistemas confiáveis sempre privilegiou soluções sob medida, nos quais todos os componentes, tanto de *hardware* como de *software*, são projetados e construídos especificamente para aquela aplicação. A reutilização de componentes já existentes sempre foi vista como um fator de risco a ser evitado. Do ponto de vista técnico, essa estratégia se justifica por várias razões, dentre as quais destacamos:

(i) Componentes reutilizáveis são mais complexos que componentes desenvolvidos para uma tarefa específica, consumindo recursos do sistema e diminuindo a sua confiabilidade;

(ii) A reutilização de componentes muitas vezes adiciona novos requisitos ao projeto do sistema e exige adaptações que o tornam ainda mais complexo;

(iii) Componentes de prateleira são, em geral, fornecidos como "caixa-preta", o que dificulta prever o seu comportamento em situações de falha [103];

(iv) Muitos componentes de prateleira não são desenvolvidos com o rigor necessário para se alcançar a qualidade exigida por uma aplicação crítica.

A restrição à reutilização de componentes em aplicações críticas se fundamenta também em experimentos práticos e casos reais de defeitos catastróficos, como o fracasso da missão do foguete Ariane 5 [55], atribuído a uma falha provocada pela reutilização de um componente de *software* desenvolvido para o Ariane 4, e os resultados do projeto Ballista [50], que atestam a vulnerabilidade de sistemas operacionais comerciais amplamente utilizados.

Há, por outro lado, fatores econômicos e mercadológicos que impõem uma mudança de estratégia e que vão além dos benefícios diretos, em termos de prazo e custo de desenvolvimento, associados à reutilização de componentes. A esse respeito, o comitê reunido em 1998 pelo governo Norte Americano para aconselhar no estabelecimento de diretrizes para a pesquisa e desenvolvimento em tecnologias de informação conclui, em seu relatório final:

> *"A demanda por software tem crescido muito mais rapidamente que os recursos que temos para produzi-lo. Como resultado necessitamos desesperadamente de software que não está sendo desenvolvido. Ainda mais, a nação precisa de software que seja muito mais fácil de usar, confiável e poderoso do que o que é produzido atualmente. Nós nos tornamos perigosamente dependentes de grandes sistemas de software cujo comportamento não é bem compreendido e que, freqüentemente, fracassam de forma imprevisível. Sendo assim, devemos dar a máxima prioridade a mais pesquisas em software. Deve ser dada ênfase especial ao desenvolvimento de software baseado em componentes e técnicas de produção, desenvolvimento e testes de sistemas confiáveis tolerantes a falhas."* [70]

O desenvolvimento de sistemas confiáveis baseados em componentes é, portanto, uma necessidade real e urgente. Nas seções seguintes são destacadas algumas das questões inerentes ao paradigma DBC que dificultam o desenvolvimento desse tipo de sistema.

### 2.4.1. Definições de Interface

As linguagens de definição de interfaces existentes permitem expressarmos apenas a sintaxe e, em alguns casos, a semântica das interfaces, sob a forma de assinaturas de métodos, pré-condições, pós-condições e invariantes. A especificação dos aspectos de sincronismo e qualidade de serviço, por sua vez, ainda é objeto de pesquisa [5].

Essa lacuna na definição das interfaces é de vital importância no desenvolvimento de sistemas confiáveis, pois os atributos de confiabilidade de um sistema serão sempre função da qualidade

dos componentes utilizados. Sem informações sobre o comportamento de um componente em situações de falha e sobre a qualidade do seu projeto, é impossível oferecer qualquer garantia quanto ao comportamento de um sistema que dependa daquele componente.

Uma forma de aumentar a expressividade das linguagens de definição de interfaces é utilizar credenciais para especificar seus atributos de qualidade [81]. Uma credencial é uma tupla composta pelo nome de um atributo, um valor associado ao atributo, e o grau de confidência no valor do atributo. Uma interface com a credencial `<linhas_de_código, 1000, verificada>`, por exemplo, especificaria que qualquer implementação dessa interface deve conter no máximo 1000 linhas de código, e que essa condição seja verificada por alguma ferramenta.

### 2.4.2. Composição de Componentes

Em geral, os riscos associados às aplicações de sistemas confiáveis dificultam enormemente a realização de experimentos com esses sistemas em condições reais de operação. A experimentação com os mecanismos de tolerância a falhas de sistemas críticos, em particular, pode simplesmente não ser possível. Por esse motivo, o projeto de sistemas confiáveis deve objetivar sua máxima previsibilidade, ou seja, facilitar a análise do sistema, de modo a possibilitar a compreensão do seu comportamento e a dedução das suas propriedades de maneira clara e inequívoca.

De um modo geral, a análise de um projeto será tanto mais fácil quanto menos elementos contenha e mais simples forem as suas conexões. Sistemas confiáveis baseados em componentes de *software*, no entanto, costumam abranger centenas de componentes que interagem de várias formas, o que exige soluções arquiteturais que confiram previsibilidade ao projeto do sistema.

O modelo de composição de componentes denominado ACTI [35] visa alcançar essa previsibilidade através da propriedade de **raciocínio modular**. Um projeto de sistema com essa propriedade permite prever o comportamento do sistema projetado a partir do conhecimento do comportamento de seus componentes isoladamente e de apenas suas interconexões locais com outros componentes. No modelo ACTI, o sistema é formado por um conjunto de componentes

interconectados através de um gabarito. Esses componentes podem ser decompostos, recursivamente, da mesma forma até o nível dos componentes mais elementares. Um gabarito define um padrão de interconexão entre componentes que resulta num novo componente, cujas propriedades são determinadas pelas propriedades dos seus componentes constituintes.

O uso de gabaritos de *software* como no modelo ACTI, é preconizado também por outros autores [5], sob diferentes denominações, tais como, configurações arquiteturais ou topologias [58] e contratos de interação [25].

Gabaritos de *software* podem ser aplicados em conjunto com a abordagem do componente ideal tolerante a falhas (Seção 2.3.10) para incorporar mecanismos de tolerância a falhas em sistemas confiáveis, através de modelos previamente definidos, como por exemplo, para inclusão de redundância modular.

### 2.4.3. Implementação das Interfaces

A implementação de uma interface possui propriedades que nem sempre fazem parte da sua definição. Memória utilizada e tempo de resposta são exemplos de propriedades inerentes a qualquer implementação e que muitas vezes não estão definidas nas interfaces. Uma vez que uma implementação é integrada a um sistema, essas propriedades podem criar dependências não previstas pelas interfaces dos componentes, num fenômeno chamado vazamento de propriedade. A simples substituição de uma implementação de uma interface por outra em perfeita conformidade com a mesma pode, nessas condições, resultar numa condição de falha para o sistema [5].

### 2.4.4. Testes de Componentes

Uma parcela significativa do esforço de implementação de qualquer artefato de *software* é dedicada às atividades de teste e de certificação que garantam a conformidade da implementação com suas especificações.

A confiabilidade de um componente de software é sempre relativa à sua utilização, ou seja: depende do tipo e freqüência dos serviços que lhe são requisitados, que é o seu perfil

operacional[23] [44]. No DBC, o perfil operacional de um componente de software muda de acordo com o sistema onde é integrado e só é conhecido no momento dessa integração. Essa variabilidade impede que se atribua a um componente de software reutilizável um grau de confiabilidade "absoluto". Apenas no momento da integração de um sistema, com base num perfil operacional específico, é possível determinar o grau de confiabilidade dos seus componentes de software.

O uso de componentes já existentes não elimina, portanto, a necessidade de testes desses componentes, tanto isoladamente como integrados ao sistema. Na prática, porém, limitações aos testes de componentes "caixa-preta", aliadas a restrições de custos e questões como a rápida evolução do software, inviabilizam essa avaliação precisa da qualidade dos componentes de software reutilizados. Verificação formal[24] e testes "caixa-branca" são exemplos de técnicas de prevenção de falhas que se tornam ineficazes quando a complexidade do software é elevada ou não se dispõe do seu código fonte.

### 2.4.5. Evolução de Componentes

Componentes de prateleira que evoluem independentemente podem impor um esforço continuado de atualização e reconfiguração dos sistemas nos quais são utilizados. Essa instabilidade na configuração do sistema representa um risco adicional no ciclo de vida de sistemas confiáveis baseados em componentes, que deve ser considerado explicitamente no projeto do sistema.

A arquitetura SIMPLEX, por exemplo, permite a evolução de sistemas de tempo real dinamicamente e de forma segura [78]. A substituição de um componente é supervisionada por um módulo de controle especializado que verifica o comportamento da nova implementação, comparando suas saídas com as produzidas pela implementação anterior. Essa verificação utiliza o conceito de redundância analítica que admite diferenças de comportamento entre as duas versões, desde que não violem determinadas condições de segurança no funcionamento do sistema pré-

[23] do inglês, *operational profile*
[24] do inglês, *model checking*

estabelecidas. Caso essas condições sejam violadas pela nova versão da implementação, a versão anterior é reativada automaticamente.

O arcabouço de software HERCULES também visa a evolução segura dos componentes de um sistema usando uma estratégia semelhante à da arquitetura SIMPLEX, mantendo no sistema diferentes versões de um componente [19].

### 2.4.6. Infra-estrutura de Componentes

As infra-estruturas de componentes geralmente incluem serviços de tolerância a falhas para replicação de componentes de software e objetos da aplicação em diferentes nós de uma rede de computadores. O estado dessas réplicas é mantido sincronizado por meio de um protocolo de comunicação em grupo confiável, que garante que todas as requisições de serviços sejam recebidas por todas as réplicas ativas no momento [43].

## 2.5. Considerações Finais

No desenvolvimento de software baseado em componentes (DBC), os sistemas são construídos a partir da integração de componentes já existentes. Esses componentes reutilizados são, muitas vezes, desenvolvidos de forma independente e fornecidos como "caixa-preta". O projeto desses sistemas privilegia a facilidade de construção do sistema, visando encurtar prazos e custos de desenvolvimento e, muitas vezes, em detrimento da simplicidade do software. A qualidade dos componentes reutilizados nem sempre pode ser avaliada precisamente. Dessa forma, a arquitetura do software tem, no contexto de DBC, uma importância ainda maior como determinante dos atributos de qualidade do sistema.

A confiabilidade de um sistema de software é uma resultante de diversos atributos, dentre os quais destacamos a corretude e robustez do software. Esses atributos são influenciados pelas características tanto do projeto do sistema de software como do processo de desenvolvimento empregado. Tradicionalmente, sistemas de software confiáveis são projetados "sob medida" e construídos a partir do zero, evitando qualquer complexidade desnecessária. Os processos de desenvolvimento empregados nesses sistemas visam, em primeiro lugar, garantir a corretude do

software, através da utilização intensa de técnicas de prevenção de falhas como, por exemplo, especificação e verificação formais e testes rigorosos. No projeto desses sistemas são empregadas técnicas de estruturação do software e tratamento de exceções que visam, essencialmente, prover tolerância a falhas através de componentes também robustos.

Por tudo isso, o desenvolvimento de sistemas confiáveis no paradigma DBC esbarra em restrições que dificultam a aplicação das técnicas de confiabilidade tradicionais. No que concerne a tolerância a falhas, a falta de garantias a respeito dos componentes reutilizáveis integrados a um novo sistema exige novas técnicas para estruturação do software e para tratamento de exceções. Os Capítulos 3 e 4 seguintes tratam, respectivamente, dessas novas técnicas de tolerância a falhas, aplicadas à arquitetura do software e abstraindo detalhes de implementação dos componentes individuais.

# Capítulo 3
# Arquiteturas de Software Tolerantes a Falhas

Neste capítulo são descritos os modelos para construção de configurações tolerantes a falhas denominados Componente C2 Ideal Tolerante a Falhas (ou iC2C[25]) [37] e Componente COTS Ideal Tolerante a Falhas (ou iCOTS[26]) [36] [38] [39] [40]. O objetivo desses modelos é prover, na arquitetura do software, a máxima separação entre os elementos responsáveis pelo comportamento normal do sistema daqueles responsáveis por prover tolerância a falhas ao sistema. O iC2C é uma abordagem para projeto de arquiteturas de software baseadas em componentes ideais tolerantes a falhas (Seção 2.3.10), utilizando o estilo arquitetural C2 (Seção 2.2.4). O iCOTS é uma especialização do iC2C para transformar componentes de software de prateleira em componentes ideais tolerantes a falhas, adicionando aos mesmos invólucros protetores [71].

## 3.1. O Componente C2 Ideal Tolerante a Falhas (iC2C)

Nessa seção é descrito o modelo do iC2C, utilizado para adaptar o comportamento excepcional de componentes de software a requisitos de tolerância a falhas específicos de um sistema. Essa adaptação é feita através da adição de novos elementos arquiteturais que envolvem aquele componente de software. A configuração obtida é um componente composto que possui uma estrutura e comportamento equivalentes aos de um Componente Ideal Tolerante a Falhas (ou iFTC[27]). A seguir é descrito o processo de estruturação do iC2C e a solução obtida.

---

[25] do inglês: ***i**dealised* ***C2 C****omponent.*

[26] do inglês: ***i**dealised* ***COTS*** *component.*

[27] do inglês: ***i**dealised* ***F****ault* ***T****olerant* ***C****omponent.*

Num primeiro passo, os tipos de mensagens do estilo C2 (Figura 4) foram utilizados como base de uma hieraquira com os diferentes tipos de mensagens definidas pelo iFTC (Figura 5). Requisições de serviços e respostas normais de um iFTC foram mapeadas como requisições C2 e notificações C2, respectivamente. Como exceções de interface e de defeito fluem na mesma direção que uma resposta normal, foram também consideradas como subtipos de notificações C2. A hierarquia final de mensagens é apresentada na Figura 6.

As duas partes do iFTC associadas, respectivamente, às atividades normal e anormal foram modeladas como dois subcomponentes distintos do iC2C. A estrutura geral resultante para o iC2C possui dois componentes e três conectores, como apresentado na Figura 7.

O componente AtividadeNormal implementa o comportamento normal do iC2C, processando *requisições de serviços* efetuadas por outros componentes e enviando de volta *respostas normais*. Esse mesmo componente é responsável pela detecção de erros durante o processamento normal, que são sinalizados por meio de mensagens de *exceções de interface* ou *exceções internas*. O componente AtividadeAnormal é responsável pela recuperação do erro e a sinalização de *exceções de defeito*. Para manter a consistência do estilo C2, exceções internas são mapeadas como subtipos de notificações C2 e a mensagem VoltarAoNormal, que flui na direção oposta, é mapeada

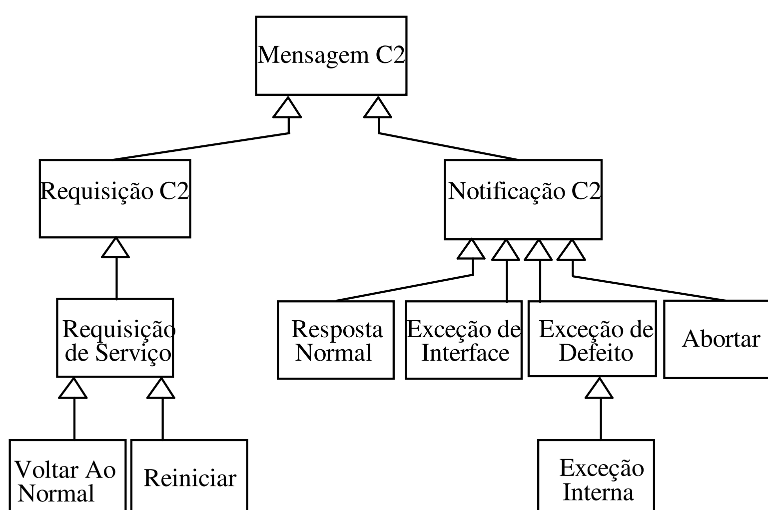


**Fig. 6.** Hierarquia de tipos de mensagens do iC2C.

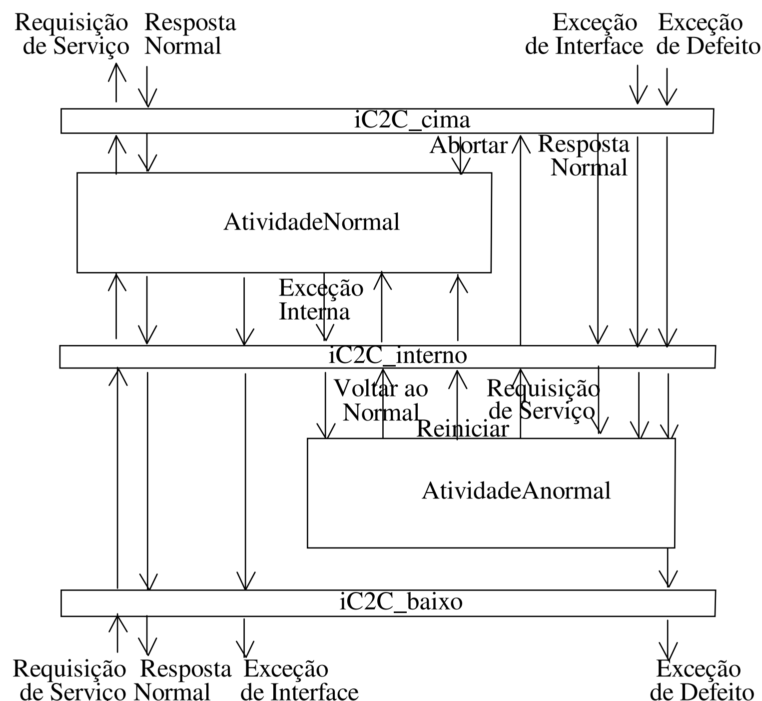


**Fig. 7.** O Componente C2 Ideal Tolerante a Falhas.

como subtipo de requisição C2 (Figura 6). Durante a recuperação de erros, o componente AtividadeAnormal pode enviar novas requisições de serviços e receber as notificações correspondentes, podendo interagir tanto com o componente AtividadeNormal como com outros componentes ligados à interface de cima do iC2C.

Dois outros tipos de mensagens que podem ser recebidas pelo componente AtividadeNormal completam o conjunto de tipos de mensagens definidos para o iC2C: notificações de Abortar e requisições de Reiniciar. Uma notificação de Abortar força o término do processamento de uma requisição de serviço pelo componente AtividadeNormal, após uma exceção externa ser recebida pelo iC2C e antes que o controle seja passado para o componente AtividadeAnormal. Uma requisição Reiniciar transfere o controle de volta para o componente AtividadeNormal e é enviado pelo componente AtividadeAnormal quando a recuperação de erro é mal sucedida e uma exceção de defeito é enviada pelo iC2C.

Os conectores do iC2C (Figura 7) são conectores C2 especializados e reutilizáveis, com as seguintes responsabilidades:

1. O conector iC2C_baixo conecta o iC2C com componentes clientes, situados abaixo dele na configuração C2. Esse conector serializa as Requisições de Serviços recebidas. Uma vez aceita uma requisição, outras requisições recebidas são enfileiradas até que o processamento daquela primeira requisição seja completado. O processamento de uma requisição termina com o envio de uma notificação, que pode ser uma Resposta Normal, uma Exceção de Interface ou uma Exceção de Defeito.
2. O conector iC2C_interno controla o fluxo de mensagens no interior do iC2C, selecionando o destino de cada mensagem recebida baseado na origem da mensagem, no seu tipo e no estado operacional do iC2C (normal ou anormal).
3. O conector iC2C_cima conecta o iC2C com componentes servidores, situados acima dele na configuração C2. Esse conector sincroniza cada Requisição de Serviço enviada para um componente servidor com sua notificação correspondente, que pode ser uma Resposta Normal, uma Exceção de Interface ou uma Exceção de Defeito.

A estrutura geral do iC2C o torna plenamente conforme com as regras do estilo arquitetural C2 para seus componentes. Isso permite que o iC2C seja integrado em qualquer configuração C2 para interagir com outros componentes de um sistema maior. Quando essas interações definem uma cadeia de componentes baseados no iC2C, as exceções externas levantadas por um desses componentes podem ser tratadas por outro componente situado mais abaixo na arquitetura, permitindo uma estruturação hierárquica das atividades de recuperação de erros. Um componente iC2C pode interagir também com componentes C2 elementares, tanto provendo como requisitando serviços.

### 3.1.1. Estruturação do Componente AtividadeNormal

O componente AtividadeNormal pode ser construído "do zero", especialmente para sua integração num iC2C, ou a partir de um componente já existente, sem o conhecimento das regras definidas para o iC2C. Essa seção detalha esse segundo caso.

Como já mencionado, o componente AtividadeNormal é responsável pela: (i) implementação do comportamento normal do iC2C; e (ii) detecção de erros que possam afetar esse comportamento normal. Quando o componente AtividadeNormal é construído a partir de um componente já existente, que ofereça apenas a implementação do comportamento normal, é necessária a adição de detecção de erros a esse componente. Para isso, um solução arquitetural para estruturação do componente AtividadeNormal é apresentada na Figura 8 e descrita a seguir.

O componente NormalBásico é um componente C2 já existente que implementa os serviços providos pelo iC2C para outros componentes clientes e que definem o seu comportamento normal. O componente ComponenteColaborador é um outro componente C2 que implementa funções de detecção de erros que podem ser requeridas pelo iC2C. Esses dois componentes são envolvidos por um par de conectores (normal_cima e normal_baixo).

O conector normal_baixo coordena a colaboração entre os componentes NormalBásico e ComponenteColaborador, provendo a capacidade de detecção de erros ao componente AtividadeNormal. A detecção de erros é baseada em teste de pré-condições, pós-condições e invariantes associadas aos vários serviços providos pelo componente NormalBásico. A detecção de erros pode exigir que o componente AtividadeNormal interaja com outros componentes fora do escopo local do iC2C. Nesse caso, o componente ComponenteColaborador estaria situado acima do iC2C, na configuração C2, e o conector normal_cima atuaria como um representante desse componente no contexto do componente AtividadeNormal.

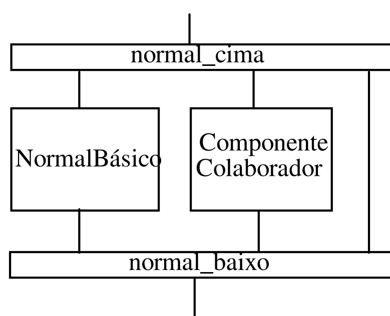


**Fig. 8.** Decomposição do Componente AtividadeNormal.

### 3.1.2. Estruturação do componente AtividadeAnormal

O componente AtividadeAnormal é responsável pelo tratamento dos erros detectados, o que inclui: (i) diagnosticar o tipo e a extensão do erro detectado; e (ii) reparar o estado do sistema, o que depende do tipo do erro detectado. Dependendendo da complexidade dessas tarefas, pode ser desejável decompor esse componente em subcomponentes especializados, conforme ilustrado na Figura 9. Nessa solução, o componente AnalisadorDeErros é responsável por diagnosticar o erro e coordenar a sua recuperação ativando um (ou mais) TratadorDeErro especializado(s).

### 3.1.3. Integração do iC2C em Configurações C2

Nessa seção é descrito como um componente do tipo iC2C pode interagir com outros componentes numa configuração arquitetônica do estilo C2. É assumido que os tipos das notificações de um sistema permitem distinguir entre notificações de exceção, emitidas por componentes do tipo iC2C, e notificações de respostas normais, emitidas por componentes C2 regulares ou do tipo iC2C. Feita essa distinção, uma notificação de exceção segue as mesmas regras de uma notificação normal, ou seja: identifica a operação da interface do componente que foi executada, os parâmetros utilizados e o valor do resultado obtido. No caso de uma notificação de exceção o resultado obtido é um objeto que descreve a condição excepcional ocorrida. É assumido também que um componente C2 regular (rC2C) não emite notificações de exceção e não reconhece notificações de exceção como sinalização de condições excepcionais. Para um rC2C, uma notificação de exceção emitida por um iC2C é reconhecida apenas como uma notificação de um tipo desconhecido. Essa suposição é válida para componentes C2 construídos

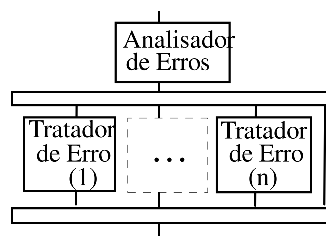


**Fig. 9.** Decomposição do componente AtividadeAnormal.

sem o conhecimento das regras definidas para o iC2C. Neste trabalho não consideramos casos especiais, como um componente C2 capaz de enviar falsas exceções de notificação ou que reconhece exceções de notificação, mas não se comportam como um iC2C. Em seguida são analisadas as diferentes formas de interação entre um iC2C e outros componentes iC2C ou rC2C.

**Caso 1. Um iC2C cliente requisitando serviço de outro iC2C (servidor).** Uma requisição de serviço pode resultar numa notificação normal ou numa notificação de exceção. No caso de uma exceção de notificação o iC2C cliente deve ser capaz de tratá-la de maneira previsível.

**Caso 2. Um iC2C requisitando serviços de um rC2C.** O iC2C deve ser capaz de reconhecer os diferentes tipos de notificação emitidos pelo rC2C, incluindo aqueles utilizados pelo rC2C para sinalizar alguma condição excepcional. O conector iC2C_topo do iC2C cliente é responsável pela tradução das notificações recebidas para o domínio do iC2C, o que inclui transformar aquelas notificações emitidas pelo rC2C para sinalizar alguma condição excepcional em notificações de exceção reconhecidas pelo conector iC2C_interno e pelo componente AtividadeAnormal. O componente AtividadeNormal é responsável por detectar erros causados por qualquer outra condição anormal que não seja devidamente sinalizada pelo rC2C.

**Caso 3. Um rC2C requisitando serviços de um iC2C.** Esse caso se desdobra em dois casos distintos:

*Caso 3.1. O iC2C não emite notificações de exceções.* O iC2C é capaz de mascarar qualquer condição excepcional. Qualquer requisição recebida resulta numa notificação normal que é reconhecida corretamente pelo rC2C.

*Caso 3.2. O iC2C emite notificações de exceção.* Nesse caso, o rC2C irá se comportar de maneira imprevisível quando emitida uma notificação de exceção pelo iC2C. Para resolver esse problema, há duas soluções possíveis: (i) promover o rC2C a um iC2C, procedendo então como no caso 1; ou (ii) elevar a tolerância a falhas do iC2C, para que

seja capaz de mascarar todos as condições excepcionais e, então, proceder como no caso 3.1. Essas soluções são detalhadas em seguida.

Para se promover um rC2C a um iC2C, devemos: (i) construir um componente AtividadeNormal estruturado como mostra a Figura 8 e utilizando o rC2C em lugar do componente NormalBásico; (ii) construir um componente AtividadeAnormal que é responsável pelo tratamento das notificações de exceção emitidas pelo componente AtividadeNormal ou por outros componentes servidores; (iii) integração dos componentes AtividadeNormal e AtividadeAnormal num novo iC2C, estruturado como mostra a Figura 7.

A tolerância a falhas de um iC2C pode ser elevada de duas formas:

1. Inserindo o iC2C original num iC2C envolvente (Figura 10). O iC2C original é inserido em lugar do componente AtividadeNormal do iC2C envolvente. O componente AtividadeAnormal do iC2C envolvente é responsável por tratar as notificações excepcionais emitidas pelo iC2C inserido como AtividadeNormal.
2. Interceptando a interface de baixo do iC2C através de um iC2C adaptador (Figura 11). O componente AtividadeAnormal do iC2C adaptador trata as notificações de exceção emitidas pelo iC2C original. O componente AtividadeNormal do iC2C adaptador delega a execução das requisições de serviços ao iC2C original.

Ambas alternativas acima produzem uma nova interface, provida pelo iC2C envolvente ou adaptador, que é uma especialização da interface do iC2C original, como numa relação de subtipo.

No estilo arquitetural C2, uma configuração de componentes e conectores pode ser tratada como um componente composto. Dessa forma, podemos usar o iC2C para construir uma "configuração C2 ideal tolerante a falhas", onde a configuração C2 original é inserida em lugar do componente AtividadeNormal.

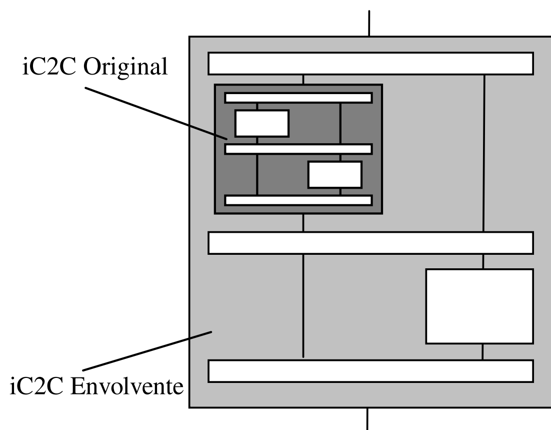


**Fig. 10.** Composição com um iC2C envolvente.

Nessa configuração ideal, o componente AtividadeAnormal é responsável por tratar as condições excepcionais que não podem ser tratadas na configuração original, incluindo as chamadas exceções de configurações [46] que são condições excepcionais cujo tratamento exige a reconfiguração da arquitetura.

### 3.1.4. Modelo Formal do iC2C

Foi desenvolvido um modelo formal do iC2C que permite a especificação precisa e a verificação dos protocolos utilizados pelos elementos internos do iC2C. Para desenvolvimento desse modelo foi utilizada a ferramenta UPPAAL [15]. Um modelo UPPAAL é formado por um conjunto de

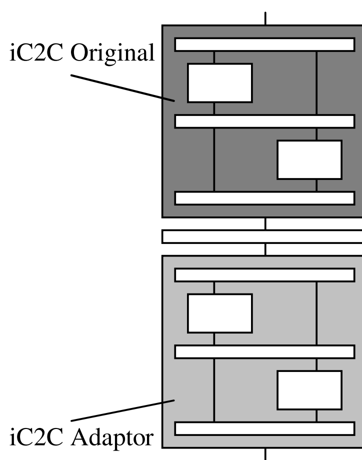


**Fig. 11.** Composição com um iC2C adaptador.

autômatos que se comunicam através de canais síncronos ou assíncronos. Cada autômato modela um processo que é descrito através de um grafo. Os nós do grafo representam os diferentes estados do processo. Os arcos do grafo representam transições de estado, e podem estar associados a condições de guarda, sinais de sincronização entre processos e atribuições de variáveis.

O primeiro modelo desenvolvido (Figura 12) foi uma representação de alto nível com 3 processos: (i) o processo iC2C, que representa um componente do tipo iC2C; (ii) o processo Client, que representa o conjunto de componentes e conectores conectados à interface de baixo do iC2C; e (iii) o processo Server, que representa conjunto de componentes e conectores aos quais o iC2C se conecta através da sua interface de cima.

No processo Client (Figura 12(a)) estão modeladas as requisições de serviços (r_in) enviadas ao iC2C e as notificações correspondentes recebidas, que podem ser notificações normais (n_out), exceções de interface (ie_out) ou exceções de defeito (fe_out).

A Figura 12(b) mostra o processo iC2C com 5 estados: (i) S0 - é o estado inicial do iC2C, quando

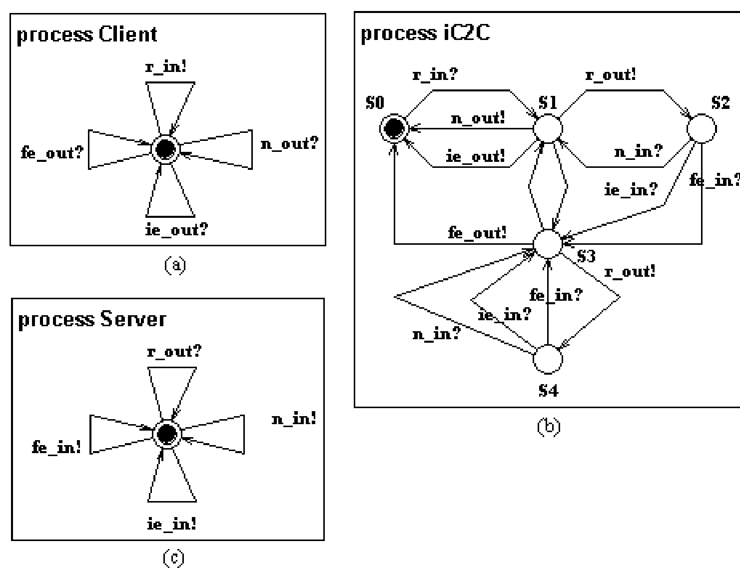


**Fig. 12.** Modelo UPPAAL de mais alto nível do iC2C.

está inativo e pronto para aceitar uma nova requisição de serviço (r_in); (ii) S1 - quando o iC2C está em sua atividade normal, processando uma requisição de serviço; (iii) S2 - quando o iC2C está aguardando uma resposta a uma requisição de serviço enviada (r_out) durante sua atividade normal, resposta essa que pode ser uma notificação normal (n_in), uma notificação de exceção de interface (ie_in) ou uma notificação de exceção de defeito (fe_in); (iv) S3 - quando o iC2C está em sua atividade anormal, recuperando de uma exceção; e (v) S4 - quando o iC2C está aguardando uma resposta a uma requisição de serviço enviada (r_out) durante sua atividade anormal.

A Figura 12(c) mostra o processo Server que recebe requisições de serviços enviadas pelo iC2C e envia notificações de volta, que podem ser notificações normais (n_in) ou de exceção (ie_in e fe_in).

Nesse modelo de mais alto nível estão representadas todas as mensagens que atravessam as fronteiras mais externas do iC2C. Essas mensagens foram modeladas como eventos síncronos e correspondem às mensagens enviadas ou recebidas pelo iC2C por meio de uma das suas portas mais externas. As transições de S1 para S3 e de S3 para S1 representam, respectivamente, uma exceção interna e a volta ao estado normal.

Posteriormente esse modelo de mais alto nível foi refinado em dois passos. Num modelo intermediário o processo iC2C foi decomposto em três processos representando, respectivamente, o componente AtividadeNormal, o componente AtividadeAnormal e o conector iC2C_cima. Nesse modelo intermediário foram abstraídos os papéis dos conectores iC2C_baixo e iC2C_interno, com os processos AtividadeNormal, AtividadeAnormal e Client se comunicando diretamente e também através de mensagens síncronas. Finalmente, um modelo de baixo nível foi desenvolvido, adicionando dois outros processos modelando, respectivamente, os conectores iC2C_baixo e iC2C_interno. A Figura 13 mostra o modelo final do processo AtividadeNormal nesse nível mais baixo.

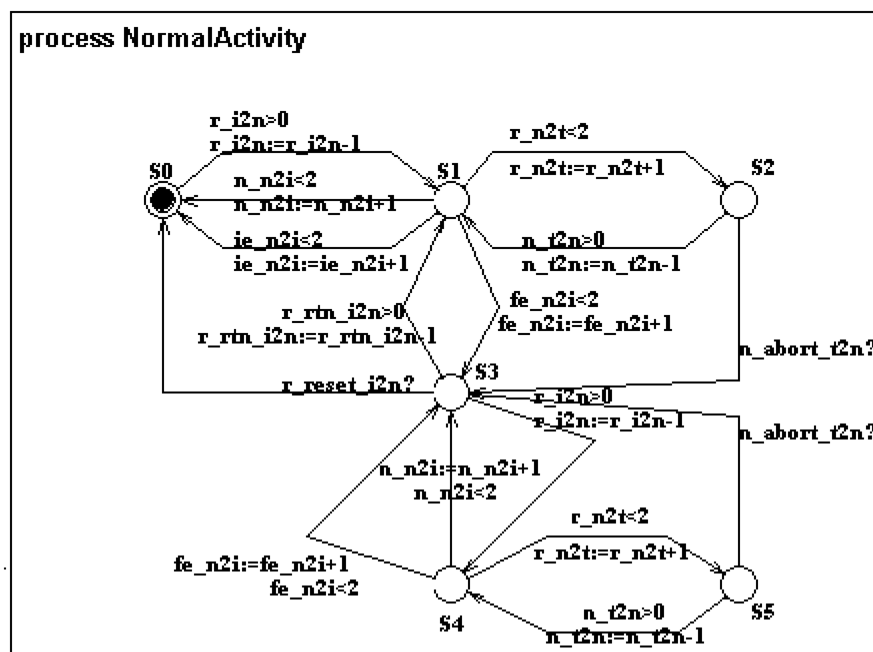


**Fig. 13.** Modelo refinado do processo AtividadeNormal.

Com exceção das mensagens Reiniciar e Abortar, todas as demais mensagens que fluem dentro do iC2C foram modeladas como eventos assíncronos, usando variáveis inteiras que representam filas de mensagens entre uma porta de saída e outra de entrada de dois elementos que se comunicam. O envio e o recebimento de uma mensagem através dessas filas são representados através de operações de, respectivamente, incremento e decremento da variável correspondente. Os nomes atribuídos a essas variáveis foram compostos da seguinte forma: (i) um prefixo indicando o tipo da mensagem, que pode ser "r_" para requisições de serviços, "n_" para notificações normais, "fe_" para notificações de exceções internas, "ie_" para notificações de exceções de interface lançadas pelo iC2C, "eie_" para notificações de exceções de interface externas (recebidas pelo iC2C), e "efe_" para exceções de defeito externas (recebidas pelo iC2C); (ii) um sufixo indicando o emissor e o destinatário da mensagem, na forma "_x2y" onde **x** e **y** podem ser uma das letras **a**, **b**, **i**, **n** or **t** para, respectivamente, o componente AtividadeAnormal, o conector iC2C_baixo, o conector iC2C_interno, o componente AtividadeNormal e o conector iC2C_cima; e (iii) um subtipo

da mensagem, opcional, como "rtn" (***r**eturn-**t**o-**n**ormal* ou VoltarAoNormal), "reset" (Reiniciar) ou "abort" (Abortar).

As propriedades verificadas para o modelo desenvolvido foram: (i) não há *deadlock*; (ii) sempre que o iC2C aceita uma nova requisição de serviços todos os seus elementos internos estão nos seus respectivos estados iniciais; (iii) quando o componente AtividadeNormal está em atividade normal, processando uma requisição de serviços (estados S1 ou S2 da Figura 13) então o componente AtividadeAnormal está inativo, no seu estado inicial; e (iv) quando o componente AtividadeAnormal está ativo então o componente AtividadeNormal não está em atividade normal. Através do recurso de simulação presente na ferramenta UPPAAL foi possível verificar também a propriedade de alcançabilidade, garantindo-se que todos os estados do iC2C são alcançáveis a partir do seu estado inicial.

O modelo UPPAAL completo está apresentado no Anexo 1.

### 3.2. Um Componente Ideal Baseado em COTS (iCOTS)

Nessa seção é descrita uma solução arquitetural para transformar componentes de software de prateleira, usualmente denominados COTS, em componentes ideais tolerantes a falhas. O componente COTS ideal tolerante a falhas (iCOTS) é uma solução arquitetural que encapsula um componente COTS adicionando tolerância a falhas, para permitir sua integração em sistemas confiáveis. A tolerância a falhas é obtida através de invólucros protetores [71], ou simplesmente **protetores**, que são responsáveis pelas atividades de detecção e recuperação de erros.

Durante a integração de um sistema baseado em componentes COTS, os detalhes internos de projeto e implementação desses componentes permanecem ocultos do integrador do sistema. Um componente COTS é, em geral, fornecido apenas na sua forma executável, não sendo permitido o acesso ao seu código fonte e, conseqüentemente, impedindo a sua análise ou modificação. As especificações fornecidas pelos seus desenvolvedores são, geralmente, descritas de maneira informal, dando margem a imprecisões e omissões, o que dificulta o projeto de integração do sistema e o torna mais vulnerável a falhas. O integrador de um sistema pode ser forçado a

substituir um componente já integrado ao sistema por uma nova versão, que pode se tornar uma nova fonte de falhas para o sistema. Por tudo isso, componentes COTS são, essencialmente, componentes pouco confiáveis e devem ser tratados como possíveis fontes de falhas. Um sistema baseado em COTS deve, portanto, ser capaz de tolerar esse tipo de falha sem poder inspecionar ou modificar o seu estado interno ou o seu comportamento.

### 3.2.1. Conceito de Invólucros Protetores

Um invólucro é uma parte inserida em torno de um componente e que é capaz de interceptar qualquer forma de comunicação entre esse componente e o seu ambiente externo. Dessa forma, o invólucro pode modificar o comportamento visível externamente de um componente, sem alterar a estrutura ou o comportamento interno do componente. O uso de invólucros de software é uma técnica de estruturação que se mostra eficaz na solução de diversos problemas em DBC. Invólucros podem ser empregados, por exemplo, para simplificar a interface de um componente ou adicionar segurança ao sistema por meio de criptografia das mensagens enviadas e recebidas por um componente. Um protetor [71] é um invólucro de software cuja finalidade é elevar a confiabilidade de um componente de software de prateleira (COTS). Isso é obtido através de dois tipos de proteção: do restante do sistema (ROS[28]) contra um possível comportamento errôneo do COTS e, simetricamente, o COTS contra um possível comportamento errôneo do ROS. Por esse duplo papel do protetor, chamamos a interface entre o COTS e o ROS de **interface protegida**. Um protetor pode ser visto, de forma simplificada, como uma parte de software redundante que detecta erros ou atividades suspeitas numa interface protegida e executa ações de recuperação apropriadas. O desenvolvimento de um protetor requer, portanto, a especificação rigorosa do comportamento considerado correto e seguro para o sistema, que seja observável na interface protegida pelo mesmo. Essa especificação é formalizada através de um conjunto de assertivas, chamadas **restrições do comportamento permitido**. A violação de uma dessas é uma condição de erro a ser tratada pelo protetor.

[28] do inglês: *Rest Of the System.*

### 3.2.2. Passos para Desenvolvimento de iCOTS

Nas seções seguintes é descrito o método proposto neste trabalho para o desenvolvimento dos invólucros protetores. Esse método pode ser aplicado no desenvolvimento de um sistema de software baseado em componentes de prateleira. Os passos apresentados a seguir aplicam-se aos estágios de provisionamento e montagem de um processo de desenvolvimento orientado a DBC (Seção 2.1.5). É assumido que os seguintes artefatos já foram produzidos no estágio de especificação: (i) o projeto do sistema descrevendo sua arquitetura de software inicial e as especificações relativas aos aspectos de segurança no seu funcionamento[29]; e (ii) projetos dos componentes do sistema, especificando as interfaces dos componentes e suas respectivas especificações de segurança no funcionamento.

### 3.2.3. Passos do Estágio de Provisionamento

*Passo 1.* *Desenvolver um plano de testes básico para o componente.* Esse plano de teste deve ser baseado no perfil operacional [44] (Seção 2.4.4) esperado para o componente quando integrado no sistema em desenvolvimento.

*Passo 2.* *Listar os componentes COTS candidatos.* Deve-se obter uma lista de componentes COTS que possam prover as interfaces especificadas no projeto do componente. Para cada COTS candidato executar os passos 3 a 6 descritos a seguir.

*Passo 3.* *Consolidar especificações do COTS.* Deve-se obter do fornecedor do COTS (ou seu desenvolvedor) as seguintes informações, a serem consolidadas num documento de especificação do COTS.

a) Especificações das interfaces providas pelo COTS, comumente denominadas Interfaces de Programação, ou API[30].

b) Especificações das interfaces requeridas pelo COTS, comumente encontradas sob o título de Requisitos do Sistema.

---

[29] conceito relacionado, na terminologia inglesa, a *safety*.

[30] do inglês: Application Programming Interface.

c) Falhas de software ainda remanescentes no código do COTS, muitas vezes encontradas em seções sob o título "Limitações e Problemas Conhecidos".

d) Qualquer informação que possa dar uma visão, ainda que parcial, do projeto interno e implementação do COTS (caixa-cinza) [69]. Esse tipo de informação pode ser encontrada em artigos técnicos e outras publicações produzidas pelos desenvolvedores e usuários do COTS.

*Passo 4.* *Testar o COTS.* Aplicar o plano de testes básico à instância do COTS. Os resultados desses testes devem ser documentados com informações coletadas sobre:

a) O subconjunto das interfaces do COTS (providas e requeridas) que foram ativadas durante os testes.

b) O domínio das entradas coberto pelos testes.

c) As condições de erro detectadas pelos testes e o comportamento do COTS observado nessas condições.

d) Discrepâncias observadas entre o comportamento especificado para o COTS e o que foi observado durante os testes.

*Passo 5.* *Melhorar a cobertura dos testes.* O plano de testes do componente deve ser revisado e o procedimento de testes repetido até que se obtenha uma cobertura considerada adequada. A cobertura dos testes influencia a confiabilidade, já que uma cobertura maior permite que se descubra um maior número de falhas de software a serem tratadas pelo sistema [77]. Os testes finais devem detectar todos as falhas de software do COTS já antecipadas na sua documentação e que possam ser ativadas pelo perfil operacional esperado para o COTS no sistema em desenvolvimento. Esse plano de testes deve incluir também casos de teste baseados na visão "caixa-cinza" disponível para o COTS (Seção 2.1.4).

*Passo 6.* *Anotar a documentação do COTS.* A documentação anotada do COTS consolida a informação obtida sobre o comportamento real do COTS. Essa documentação anotada é

baseada na documentação já existente, nas especificações de segurança no funcionamento do componente no sistema em desenvolvimento e nos resultados dos testes, incluindo:

a) Especificações detalhadas do comportamento real das interfaces ativadas durante os testes, tanto nas condições normais como em condições excepcionais.

b) Especificação de condições potencialmente perigosas associadas com as interfaces que não foram ativadas durante os testes.

c) Outras informações adicionais que possam ser obtidas de usos anteriores da mesma instância do COTS.

*Passo 7.* *Selecionar uma instância do COTS.* Se, para um mesmo componente especificado pelo sistema, houver dois ou mais COTS candidatos sendo considerados, selecionar aquele que melhor se adapte ao sistema em desenvolvimento. Essa seleção é baseada na informação contida na especificação do sistema e na documentação anotada dos vários COTS. Para essa seleção, pode ser necessário desenvolver versões alternativas para o projeto de integração do sistema, adaptadas às limitações e requisitos específicos de cada instância de COTS. O resultado desse passo é uma especificação do sistema revisada e com a arquitetura de software refinada para incluir a instância COTS selecionada envolta por um protetor, a ser desenvolvido no estágio de montagem.

*Passo 8.* *Decidir pela integração com COTS.* Nesse ponto deve-se decidir entre prosseguir na integração do sistema usando a instância COTS selecionada ou, alternativamente, usando um novo componente a ser desenvolvido especificamente para o sistema.

### 3.2.4. Passos do Estágio de Montagem

*Passo 9.* *Classificar condições errôneas.* Deve-se definir um conjunto de condições genéricas de erros que podem surgir na interface protegida. As condições errôneas (ou perigosas) especificadas na documentação anotada do COTS (Passo 6) são analisadas diante das especificações de segurança no funcionamento do sistema e classificadas de acordo com sua gravidade e a extensão do seu impacto nos requisitos de confiabilidade do sistema. Cada classe obtida define uma condição excepcional genérica.

*Passo 10. Especificar as restrições do comportamento permitido (RCP) associados às condições errôneas.* Essa especificação é baseada na informação contida na documentação anotada do COTS. As RCP podem incluir assertivas sobre:

a) O domínio dos parâmetros e resultados das requisições de serviço que fluem entre o COTS e o restante do sistema (ROS).
b) O histórico de mensagens trocadas através da interface protegida.
c) Partes do estado interno do sistema que podem ser inspecionadas através de funções que não produzem efeitos colaterais.

*Passo 11. Especificar o comportamento excepcional desejado para o sistema.* Essa especificação define os objetivos da recuperação de erro, que dependem do tipo e gravidade dos erros detectados. A principal fonte para essa especificação é a especificação de segurança no uso do sistema.

*Passo 12. Atribuir responsabilidades pela recuperação de erros.* O comportamento excepcional especificado no passo anterior é decomposto num conjunto de ações de recuperação que são atribuídas a componentes determinados da arquitetura de software. Parte dessas responsabilidades serão alocadas ao protetor associado à instância do COTS integrada ao sistema (Passo 7). Essas ações de recuperação são especificadas neste mesmo passo.

*Passo 13. Refinar a arquitetura de software.* Esse refinamento decompõe os componentes envolvidos com o processamento de erros em novos elementos arquiteturais, com responsabilidades específicas de detecção ou tratamento de erros.

*Passo 14. Implementar detectores de erros.* As RCP especificadas (Passo 10) são implementadas através de assertivas executáveis que ficam encapsuladas em dois detectores de erros que atuam como filtros de mensagens. O primeiro detector de erros intercepta e monitora as requisições de serviços que fluem do ROS para o COTS e os resultados que fluem de volta para o ROS. O segundo detector de erros intercepta e monitora as requisições de serviços que fluem do COTS para o ROS e os resultados

correspondentes que fluem de volta para o COTS. Quando um detector de erro verifica uma condição que viola uma RCP é levantada uma exceção de um tipo específico associado àquela RCP. A exceção levantada contém a mensagem que originou a exceção e que, nesse caso, não é entregue ao seu destinatário original. Mensagens que não violam as RCP são entregues ao destinatário sem qualquer modificação.

*Passo 15. Implementar tratadores de erros.* As ações de recuperação especificadas (Passo 12) são implementadas por tratadores de erros associados aos vários tipos de exceções. Esses tratadores de erros ficam associados a um ou mais componentes arquiteturais, conforme o escopo das suas ações de recuperação.

*Passo 16. Integrar os protetores.* Nesse passo, as instâncias de COTS são integradas com os detectores de erros (Passo 14) e os tratadores de erros (Passo 15), como especificado pela arquitetura de software refinada no Passo 13. O resultado desse passo é um conjunto de componentes do tipo iCOTS, estruturados de acordo com o iC2C.

*Passo 17. Integrar o sistema.* Nesse passo, o sistema final é construído através da interligação dos componentes iCOTS, que encapsulam as instâncias de COTS, e das instâncias domésticas.

### 3.2.5. Arquitetura Geral do iCOTS

O iCOTS é uma especialização do iC2C que utiliza invólucros protetores para encapsular um componente COTS. Nessa abordagem, o componente COTS é conectado a um par de conectores especializados que atuam como detectores de erros (Figura 14) para compor o componente AtividadeNormal do iCOTS. Esses detectores são responsáveis por verificar se as mensagens que fluem para (ou do) componente COTS violam alguma restrição de comportamento especificada para o sistema.

O detector_inferior inspeciona as requisições de serviços que chegam ao COTS e as respostas que ele envia (notificações C2). O detector_superior inspeciona as requisições de serviços enviadas pelo COTS e as respostas que ele recebe. O conector iC2C_baixo conecta o iCOTS com os componentes mais abaixo na arquitetura C2, que são os clientes do COTS, serializando suas

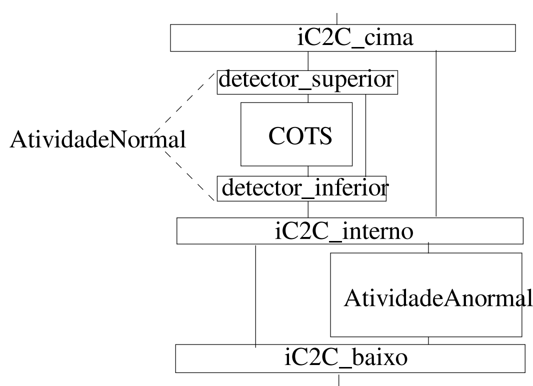


**Fig. 14.** Estrutura Geral do iCOTS.

requisições de serviços. O conector iC2C_interno controla o fluxo de mensagens no interior do iCOTS. O conector iC2C_cima conecta o iCOTS com os componentes mais acima na arquitetura C2, que são os servidores do COTS.

Quando um detector verifica uma violação de alguma restrição de comportamento especificada, ele envia uma notificação de exceção de defeito correspondente. Essa exceção é tratada pelo componente AtividadeAnormal, de acordo com as regras definidas para o iC2C. Um detector pode ser decomposto num conjunto de detectores de erros mais especializados que, por sua vez, são também envolvidos por outro par de conectores. A Figura 15 mostra um detector de erro decomposto em dois detectores mais especializados. O conector detector_baixo coordena a detecção de erro e o conector detector_cima liga o detector ao COTS ou ao conector iC2C_cima. O componente AtividadeAnormal é responsável pelo diagnóstico do erro e recuperação do erro, podendo também ser decomposto em subcomponentes mais especializados como ilustrado na Figura 9.

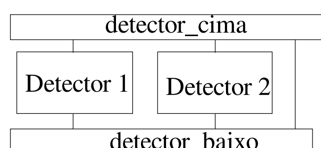


**Fig. 15.** Decomposição de um detector de erro.

## 3.3. Considerações Finais

Neste Capítulo, é apresentada uma abordagem para estruturação de arquiteturas de software tolerantes a falhas, para sistemas baseados em componentes. São propostos dois modelos para composição de componentes denominados, respectivamente, iC2C e iCOTS. Esses modelos definem, essencialmente, gabaritos de software abstratos (Seção 2.2.2) ou topologias [58] (Seção 2.4.2) tolerantes a falhas baseadas no conceito do componente ideal tolerante a falhas [72] (Seção 2.3.10).

Num iC2C as responsabilidades pela atividade normal e excepcional são separadas em elementos arquiteturais distintos. Na instanciação de um iC2C, a responsabilidade pela atividade normal pode ser atribuída a um componente já existente, não-tolerante a falhas. A comunicação entre os elementos internos de um iC2C é feita usando um protocolo, ou contrato de interação [25], definido formalmente (Anexo 1). O iCOTS é uma especialização do iC2C que adiciona invólucros protetores [71] a um componente de prateleira que implementa a atividade normal do componente composto. O Capítulo 6 apresenta, no seu primeiro estudo de caso, um exemplo de aplicação da abordagem proposta, no contexto de um sistema de controle de caldeira a vapor.

Ainda no contexto de arquiteturas de software tolerantes a falhas, o autor colaborou na orientação de pesquisa de mestrado paralela que desenvolveu uma arquitetura de software para composição de sistemas-de-sistemas, denominada Arquitetura FT-SC3 (*Fault-Tolerant System Composition using Coordination Contracts*) [85]. Esse trabalho conjunto resultou em duas publicações em co-autoria [84] [86].

As técnicas discutidas neste Capítulo visam o desenvolvimento de arquiteturas de software tolerantes a falhas, porém não garantem que a implementação final do sistema apresente essa propriedade. Um dos problemas no desenvolvimento de software centrado na arquitetura é garantir a conformidade entre a implementação final do sistema e a arquitetura do software.

A implementação correta de uma arquitetura tolerante a falhas depende de mecanismos de tratamento de exceções apropriados, que permitam a separação efetiva do código responsável pela recuperação do erro do código normal. Com esse objetivo, o Capítulo seguinte trata da

implementação de tratamento de exceções (Seção 2.3.9) em sistemas construídos de acordo com o paradigma de desenvolvimento baseado em componentes.

# Capítulo 4

# Uma Abordagem Arquitetural para Tratamento de Exceções em DBC

Neste capítulo é proposta uma estratégia arquitetural para tratamento de exceções em sistemas de software baseados em componentes. Essa estratégia é baseada numa hierarquia de tipos de exceções e num conjunto de tipos de tratadores de exceções. Essa hierarquia de tipos de exceções estabelece uma semântica precisa que é empregada para mapear exceções levantadas pelos diferentes componentes em tipos de exceções genéricos que podem ser tratados consistentemente numa estratégia para tratamento de exceções definida globalmente para um sistema. Os diferentes tipos de tratadores de exceções simplificam a divisão de responsabilidades de tratamento de exceções entre os vários elementos arquiteturais, que são os seus componentes e conectores (Seção 2.2). Para implementação da estratégia proposta é utilizado o modelo COSMOS [88] que é aqui estendido com os elementos básicos necessários para tratamento de exceções em sistemas DBC. O modelo COSMOS estendido permite a implementação de maneira sistemática de arquiteturas de software tolerantes a falhas baseada em componentes, tais como as baseadas nos modelos do iC2C e do iCOTS (Capítulo 3), também propostas neste trabalho.

## 4.1. Descrição do Problema

As linguagens de programação mais modernas, tais como C++[93], Java[94], e C#[61], incluem sofisticados mecanismos de tratamento de exceções (Seção 2.3.9) que se destinam a auxiliar a estruturação da atividade excepcional de um sistema de software. Entretanto, esses mecanismos freqüentemente são subutilizados ou empregados de forma incorreta [73] [83], mesmo em sistemas de software convencionais não baseados em componentes. A utilização eficaz desses

mecanismos requer uma estratégia sistêmica para tratamento das exceções que seja fundamentada nos requisitos de confiabilidade e hipóteses de falhas (Seção 2.3.4), que são específicos de cada sistema de software. O projeto desse tipo de estratégia é uma tarefa difícil para os desenvolvedores das aplicações, que tendem a se concentrar no projeto da atividade normal do sistema, relacionada com o atendimento aos seus requisitos funcionais. Freqüentemente, o projeto da atividade excepcional é negligenciado também em decorrência dos curtos prazos de desenvolvimento.

A estruturação da atividade excepcional em sistemas baseados em componentes reutilizáveis é ainda mais complexa do que em sistemas convencionais, não baseados em componentes, pelos motivos expostos a seguir. Componentes reutilizáveis são desenvolvidos sem o conhecimento de todos os contextos dos sistemas onde serão integrados. Sendo assim, as hipóteses de falhas assumidas no desenvolvimento de um componente reutilizável podem não ser válidas no contexto de um novo sistema. Ainda mais freqüentemente, a especificação do componente não explicita a estratégia para tratamento de exceções empregada, resultando num comportamento excepcional "ad hoc" e inteiramente dependente da implementação do componente.

Conseqüentemente, na integração de um sistema baseado em componentes reutilizáveis, podem surgir conflitos entre (i) o comportamento excepcional dos componentes reutilizados, e (ii) as hipóteses de falhas do novo sistema e a estratégia para tratamento de exceções planejada para o mesmo. Como exemplos desses conflitos, podemos citar: (i) incompatibilidade entre tipos de exceções, quando um componente lança uma exceção de um tipo diferente daquele esperado por um outro componente responsável pelo tratamento da condição excepcional sinalizada por aquela exceção; e (ii) condições excepcionais não antecipadas, quando surge, no contexto de um novo sistema, uma condição excepcional que não foi prevista no desenvolvimento de um componente reutilizado; e (iii) tratamento excepcional inadequado, quando o tratamento de uma exceção por um componente reutilizado é incompatível com a estratégia para tratamento de exceções planejada para o sistema.

No desenvolvimento de sistemas convencionais, quando se tem pleno controle sobre a especificação e implementação de todos os seus componentes, conflitos desses tipos podem ser evitados através de um processo de refinamento do sistema que uniformize as hipóteses de falhas e as estratégias para tratamento de exceções de todos os componentes do sistema. O sistema assim produzido torna-se, porém, fortemente acoplado às implementações específicas de seus componentes.

No desenvolvimento de sistemas baseados em componentes reutilizáveis, essa uniformização geralmente não é possível. Um sistema baseado em componentes deve ser capaz de operar corretamente usando qualquer implementação de uma mesma especificação de componente. Diferentes implementações de uma mesma especificação podem implicar em diferentes hipóteses de falhas e, eventualmente, diferentes estratégias para tratamento de exceções. Por isso mesmo, o desenvolvimento de sistemas tolerantes a falhas, baseados em componentes, requer uma estratégia para tratamento de exceções que permita integrar componentes desenvolvidos sob diferentes hipóteses de falhas e que implementem diferentes estratégias para tratamento de exceções.

## 4.2. Um Modelo para Tratamento de Exceções

### 4.2.1. As Abordagens de Flaviu Cristian e Bertrand Meyer

A proposta para tratamento de exceções desse trabalho se baseia em duas abordagens distintas para tratamento de exceções. A primeira abordagem, apresentada por Flaviu Cristian em artigo clássico sobre tratamento de exceções [20], é uma formalização do chamado modelo de terminação para tratamento de exceções em programas seqüenciais. Os mecanismos de tratamento de exceções das linguagens orientadas a objetos modernas, tais como C++, Java e C#, são consistentes com essa abordagem. A segunda abordagem, de Bertrand Meyer, é adotada no método de desenvolvimento de software orientado a objetos denominado Projeto por Contrato[31] [59] e na linguagem e ambiente de desenvolvimento Eiffel [27].

**Síntese da Abordagem de Cristian**

[31] do inglês: *Design by Contract*.

O foco principal da abordagem de Cristian é a robustez do programa, ou seja, o desenvolvimento de programas cujo comportamento seja previsível para todas as possíveis entradas fornecidas. Essa abordagem, com foco em robustez, é um meio para tolerância a falhas, pois inclui as condições excepcionais decorrentes de falhas entre as entradas previstas pelo programa. Segundo Cristian, um programa robusto deve estar preparado para tratar qualquer entrada possível e sempre se comportar de acordo com sua especificação. A especificação de um programa define um ou mais **pontos de saída**[32], que sinalizam o término de uma execução, da seguinte forma:

1. É definido sempre um ponto de saída padrão e um número qualquer (inclusive nenhum) de pontos de saída excepcionais declarados na especificação.
2. O ponto de saída padrão sinaliza o término de uma execução normal, quando o programa é capaz de retornar o resultado padrão esperado.
3. Um ponto de saída excepcional sinaliza o término de uma execução em que o programa não é capaz de retornar o resultado padrão esperado.
4. Um ponto de saída excepcional que é declarado na especificação do programa corresponde a uma condição excepcional que é antecipada no desenvolvimento do programa.

Um programa correto deve terminar sempre no ponto de saída padrão ou, excepcionalmente, num ponto de saída excepcional declarado na sua especificação. Entretanto, pode haver também pontos de saída excepcionais não declarados na especificação do programa, que correspondem a condições excepcionais que não foram previstas antecipadamente na especificação do programa. Na abordagem de Cristian, essas condições excepcionais não previstas são falhas de projeto (da especificação do programa).

A Figura 16 ilustra a abordagem de Cristian através de dois diagramas de estados que representam, respectivamente, a especificação de um programa P e uma implementação dessa especificação. Na especificação do programa, representada na Figura 16(a), é prevista uma

[32] do inglês: *exit point*.

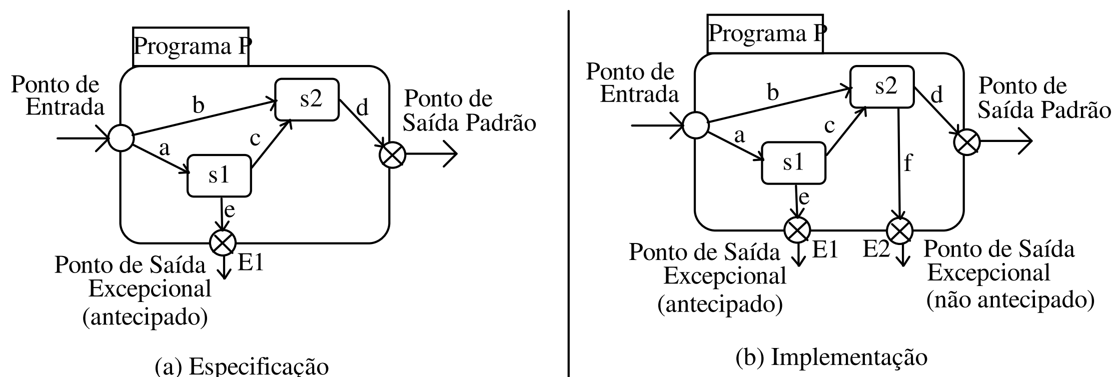


**Fig. 16.** Exemplo de programa com pontos de saída excepcionais.

condição excepcional, causada por um evento e, que implica no término da execução do programa num ponto de saída excepcional antecipado E1. A implementação desse programa, representada na Figura 16(b), está sujeita também a um evento f, decorrente de uma falha de software ou de uma restrição de implementação, que causa uma condição excepcional não prevista na especificação e, conseqüentemente, o término do programa num ponto de saída excepcional não antecipado E2.

**Síntese da Abordagem de Bertrand Meyer**

O principal objetivo da abordagem denominada Projeto por Contrato é a corretude do programa (ou *rotina*). Nessa abordagem, a corretude de um programa é relativa à sua especificação: um programa correto é aquele que funciona em conformidade com sua especificação, ou seja, que não contém falhas de software. Esse foco na corretude visa, sobretudo, a prevenção de falhas e caracteriza-se por:

1. Uma rotina não deve, necessariamente, estar preparada para tratar qualquer entrada possível, mas apenas aquelas especificadas pelas pré-condições estabelecidas no seu contrato.
2. Uma rotina tem um único ponto de saída especificado, que é alcançado sempre que a execução da rotina termina com sucesso, ou seja: as pós-condições estabelecidas no seu contrato foram alcançadas.

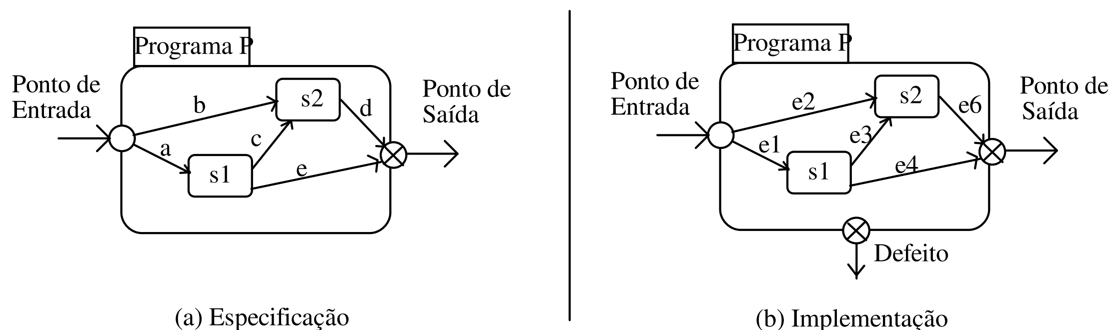


**Fig. 17.** Pontos de saída na abordagem de Projeto por Contrato.

3. Qualquer condição que impeça a rotina de cumprir o contrato é considerada uma falha de projeto e sinalizada através de uma exceção. As exceções estão sempre associadas a quebras de condições estabelecida no contrato, que podem ser da responsabilidade do cliente da rotina (pré-condições) ou da rotina que fornece o serviço (pós-condições).

Há, portanto, um único ponto de saída excepcional na implementação de um componente, sempre associado a defeito provocado por uma falha de software (Figura 17).

A seguir é descrito o modelo geral para tratamento de exceções proposto neste trabalho, que é fundamentado nas abordagens acima descritas.

### 4.2.2. Condições Excepcionais Antecipadas

A especificação de um componente é composta por uma **especificação normal** e, opcionalmente, uma **especificação excepcional**. A especificação normal define o comportamento esperado do componente sob uma **hipótese de ausência de falhas**, ou seja, supondo que nunca ocorre qualquer falha que possa interferir na execução dos serviços providos pelo componente. A hipótese de ausência de falhas raramente pode ser garantida em sistemas reais. Sistemas reais estão sujeitos a falhas de diversos tipos, que podem introduzir erros no componente ou no seu ambiente (Seção 2.3.3). As ocorrências desses erros constituem **condições excepcionais**, que não são previstas na especificação normal. Para um componente desenvolvido sob essa hipótese exclusivamente, qualquer condição excepcional invalida as garantias oferecidas pela especificação do componente e, conseqüentemente, torna o seu comportamento imprevisível.

Uma especificação excepcional complementa a especificação normal do componente adicionando uma **hipótese de presença de falhas**. Essa hipótese especifica determinados tipos de falhas e um conjunto de **condições excepcionais antecipadas**. A especificação excepcional define o comportamento esperado do componente nessas condições excepcionais antecipadas. Num sistema tolerante a falhas, esse comportamento excepcional visa mascarar o erro que provocou aquela condição excepcional (Seção 2.3.6) ou, quando isso não é possível, terminar a execução de forma ordenada, apesar do seu insucesso, sinalizando a condição excepcional que provocou o defeito.

### 4.2.3. Condições Excepcionais Não Antecipadas

Sistemas de software reais são sempre sujeitos a falhas e, conseqüentemente, a erros que não são previstos na sua especificação excepcional, por dois motivos:

1. A complexidade e os custos adicionais introduzidos no software para o tratamento de condições excepcionais limitam, na prática, a abrangência das hipóteses de presença de falhas utilizadas. Seja, por exemplo, uma hipótese de falha num circuito de memória, que possa introduzir erros no código executável do sistema. A inclusão dessa hipótese de falha no projeto dos mecanismos de tolerância a falhas do sistema de software, exige uma arquitetura de software complexa, possivelmente com módulos redundantes distribuídos em vários processadores. Por outro lado, essa mesma hipótese de falha pode ser mais facilmente tratada na fase de prevenção de falhas, utilizando um sistema de hardware baseado em módulos de memória com detecção e correção automática de erros.
2. Uma implementação de um componente possui um conjunto de propriedades, chamadas **interfaces vinculadas**[33] [5], que são específicas daquela implementação em particular. A quantidade de memória utilizada é um exemplo comum desse tipo de propriedade que, em geral, não é definida na sua especificação. As interfaces vinculadas de um componente introduzem restrições e dependências que podem implicar em novas hipóteses de falhas,

[33] do inglês: *bound interfaces*.

não previstas na sua especificação. Por exemplo: a execução do componente num computador excessivamente carregado pode resultar numa falha por insuficiência de memória que pode não estar prevista na especificação excepcional do componente.

O comportamento de um componente numa condição excepcional não prevista na sua especificação é inteiramente dependente da sua implementação. Uma implementação do componente pode, por exemplo, prosseguir na execução das suas operações sem que o erro seja detectado e tratado de alguma forma. Esse tipo de comportamento resulta, invariavelmente, na **propagação do erro** para outras partes do sistema, com o fornecimento de resultados incorretos para os clientes das operações e a introdução de novos erros no sistema.

Para evitar esse tipo de defeito, a implementação de um componente pode incluir a detecção de alguns tipos de erros que não são previstos na sua especificação, que são chamadas **condições excepcionais não-antecipadas** na especificação. Sendo assim, o comportamento excepcional de um componente inclui o tratamento dessas condições excepcionais não antecipadas que, quando não puderem ser mascaradas, irão provocar o insucesso da execução de suas operações.

### 4.2.4. Pontos de Saída de uma Operação

O sucesso ou fracasso da execução de uma operação deve ser sempre corretamente sinalizado para o cliente que requisitou o serviço. Isso é feito através do **ponto de saída** em que a execução termina. Uma execução bem sucedida deve sempre terminar no **ponto de saída padrão** enquanto que uma execução **mal sucedida** deve sempre terminar num **ponto de saída excepcional**. No ponto de saída padrão o controle do fluxo de execução é devolvido imediatamente para o cliente da operação, junto com o seu **resultado** (quando especificado). O resultado de uma operação é um valor produzido pela sua execução, cujo domínio determina o **tipo da operação**. Um ponto de saída excepcional termina a execução da operação indicando a **condição excepcional** que causou o fracasso da execução da operação, através do **lançamento de uma exceção**. O **tipo da exceção** especifica um ponto de saída excepcional associado a uma classe de condições excepcionais. As condições excepcionais antecipadas são sinalizadas por meio de exceções de **tipos de exceções declarados** na especificação. Por outro lado, as condições excepcionais não

antecipadas são sinalizadas através de exceções de **tipos não-declarados** na especificação. O **valor da exceção** descreve uma condição excepcional específica, que determinou o insucesso da execução da operação.

### 4.2.5. Semântica do Defeito

O lançamento de uma exceção por um componente sinaliza um **defeito** na sua interface provida, em conseqüência de alguma falha ocorrida. Esse defeito, por sua vez, significa uma nova falha no sistema onde o componente está integrado. Para as demais partes do sistema, que dependem do componente que apresentou o defeito, a exceção lançada é uma nova condição excepcional a ser tratada.

Para possibilitar um tratamento subseqüente adequado das exceções lançadas por um componente, a sua especificação excepcional inclui a definição de **pós-condições excepcionais**. Uma pós-condição excepcional define as garantias oferecidas pelo componente ao término de uma execução mal sucedida, quando é lançada uma exceção de um tipo declarado na sua especificação. Essa pós-condição excepcional pode ser:

1. Idêntica à pré-condição da operação, definida na especificação normal. Isso significa que a execução da operação termina sem produzir qualquer efeito sobre o estado do componente ou no seu ambiente.
2. Uma condição especificada que é diferente tanto da pré-condição como da pós-condição, definidas na especificação normal. Essa condição especifica um **resultado degradado** da execução da operação.
3. Indeterminada. Na ausência de restrições de pós-condições, nenhuma garantia é dada pelo componente quanto aos efeitos que possam ter sido produzidos sobre o estado do componente ou no seu ambiente. Essa situação deve ser evitada, pois pode impossibilitar o tratamento das exceções de forma a permitir a continuidade da execução do sistema.

Como as garantias oferecidas pelas especificações do componente não se aplicam às condições excepcionais não-antecipadas, nenhuma pós-condição é especificada para os casos em que a

execução termina com o lançamento de uma exceção de um tipo não declarado na sua especificação.

## 4.3. Solução Proposta

A estratégia proposta neste trabalho se destina a elevar o grau de tolerância a falhas em sistemas baseados em componentes através de um tratamento sistemático dos possíveis conflitos entre as hipóteses de falhas dos componentes reutilizados.

Essa estratégia é composta por duas partes complementares: uma estratégia intra-componente e uma estratégia inter-componente voltadas, respectivamente, para os estágios de provisionamento e integração de um processo de desenvolvimento orientado a DBC (Seção 2.1.5). A estratégia intra-componente é aplicada ao desenvolvimento de novos componentes de software e à adaptação de componentes já existentes. A estratégia inter-componente é aplicada a configurações de componentes e conectores. Para que essas duas estratégias funcionem de forma efetiva é definida uma hierarquia de tipos de exceções, que é o ponto de união das duas estratégias.

### 4.3.1. Hierarquia de Tipos de Exceções Proposta

A Figura 18 mostra a hierarquia de tipos de exceções proposta, que tem a sua raiz no supertipo mais genérico Exception. Imediatamente abaixo desse supertipo mais genérico, são definidos dois tipos básicos: DeclaredException e UndeclaredException. DeclaredException é a raiz da hierarquia de tipos de exceções declarados nas especificações dos componentes. Para esses tipos de exceções, a semântica de defeito correspondente é definida pela especificação excepcional do componente.

O supertipo UndeclaredException é a raiz de uma hierarquia utilizada pela implementação de um componente para expressar a semântica de defeito das exceções cujos tipos não são declarados nas especificações. Essa hierarquia se divide em dois outros supertipos diretos RejectedRequestException e FailureException.

Exceções do tipo RejectedRequestException são utilizadas para sinalizar violações de pré-condições que resultem na rejeição das requisição de serviços correspondentes, sem que seja produzido qualquer efeito sobre o estado do componente ou no seu ambiente. Essa classe de exceções corresponde às exceções de interface definidas para um componente ideal tolerante a falhas (Seção 2.3.10).

O supertipo FailureException abrange as exceções que sinalizam um fracasso na execução de uma requisição válida, que representa um defeito no componente ou em alguma parte do sistema da qual ele depende. Essa classe de exceções corresponde às exceções de defeito definidas para um componente ideal tolerante a falhas (Seção 2.3.10) e se divide em dois outros subtipos: RecoveredFailureException e UnrecoveredFailureException. Exceções do tipo RecoveredFailureException são utilizadas quando, apesar do defeito, a implementação do componente garante que nenhum efeito foi produzido sobre o estado do componente ou no seu ambiente. Em outras palavras, uma exceção do tipo RecoveredFailureException indica que, apesar da falha, a operação não produziu qualquer efeito e as suas pré-condições foram mantidas. Exceções do tipo UnrecoveredFailureException são utilizadas quando a implementação do

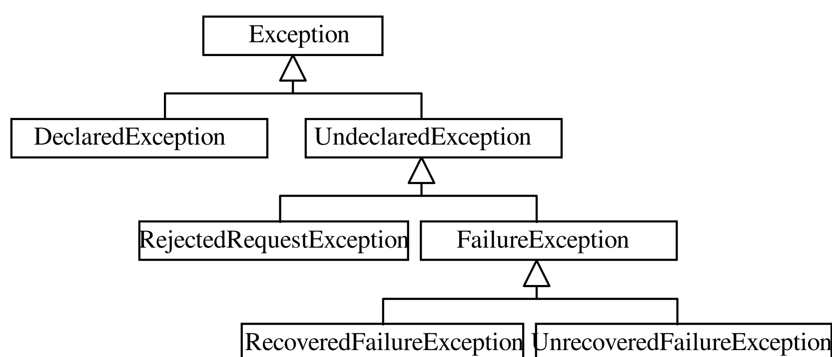


**Fig. 18.** Hierarquia Comum de Tipos Abstratos de Exceções

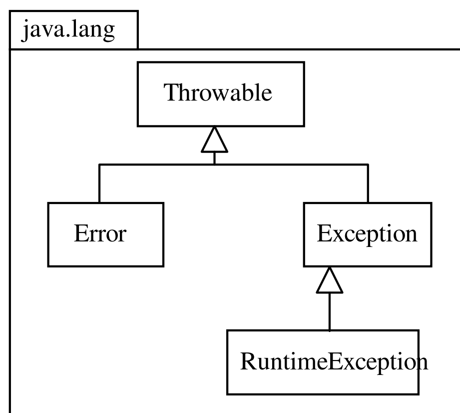


**Fig. 19.** Hierarquia de Tipos de Exceções Java.

componente fracassa na execução da operação e não pode garantir a ausência de efeitos colaterais, produzidos pela execução da operação, sobre o estado interno do componente ou no seu ambiente.

### 4.3.2. Implementação Java da Hierarquia de Tipos de Exceções Proposta

O mecanismo de tratamento de exceções da linguagem Java é baseado numa hierarquia de classes, definida no pacote java.lang, que especifica os tipos dos objetos que podem ser lançados pelo sistema ou capturados pelos tratadores de exceções. A raiz dessa hierarquia (Figura 19) é a classe Throwable, que se divide em duas subclasses: Exception e Error. Os tipos definidos pela hierarquia de Error representam condições excepcionais associadas a erros no ambiente de execução do programa que impedem o prosseguimento da execução do interpretador de Java, que é a máquina virtual de Java (ou JVM). Esses tipos de erros são detectados pela JVM e, como regra geral, não devem ser capturados pelo programa de aplicação, que depende do funcionamento correto da JVM. Todas as classes da hierarquia Error definem exceções ditas **não-verificáveis**[34], pois o compilador Java não verifica se o código da aplicação contém tratadores apropriados para esses tipos de exceções. Em princípio, qualquer instrução Java executada pode resultar no lançamento, pela JVM, de uma exceção desse tipo.

[34] do inglês: *unchecked*.

Os tipos definidos pela hierarquia de Exception são utilizados para representar as condições excepcionais associados a erros no programa de aplicação. A classe Exception possui uma subclasse especial, chamada RuntimeException, que é usada pela JVM e classes das bibliotecas nativas de Java, para representar condições excepcionais associadas a erros no software que podem ser facilmente evitados ou detectados pelo programa de aplicação. Um exemplo comum desse tipo de erro é a passagem de uma referência de objeto nula na chamada de uma operação da biblioteca de classes de Java, que pode ser evitado através de um simples teste do parâmetro antes da chamada da operação. Esse tipo de erro geralmente causa o lançamento de uma exceção do tipo NullPointerException, que é filha de RuntimeException.

Todas as classes da hierarquia RuntimeException definem exceções não-verificáveis. As demais classes da hierarquia Exception definem exceções verificáveis, ou seja: o compilador Java obriga, para todo método em que possa ocorrer uma exceção verificável de um tipo V, que esse método contenha um tratador de exceção para o tipo V ou que o tipo V esteja incluído nos tipos de exceções declarados na assinatura do método.

Uma possível implementação, em linguagem Java, da hierarquia de tipos de exceções descrita na seção anterior [18] consiste em:

1. Mapear o supertipo mais genérico (Exception) para a classe java.lang.Exception, que é a superclasse de todas as exceções em Java.
2. Mapear o supertipo DeclaredException para uma nova subclasse derivada de java.lang.Exception. Isso obriga a inclusão, nas assinaturas das operações, dos tipos de exceções declarados nas especificações do componente.
3. Mapear o supertipo UndeclaredException para uma nova subclasse de java.lang.RuntimeException.

### 4.3.3. A Estratégia Intra-componente

A estratégia intra-componente define as responsabilidades locais de um componente de software em condições excepcionais. O objetivo dessa estratégia é facilitar a integração desses componentes em diferentes contextos de sistemas, que podem incluir diferentes hipóteses de

falhas. Para isso, o integrador do sistema deve ser capaz de prever o comportamento normal e excepcional do componente com base apenas na sua especificação.

A estratégia intra-componente é aplicada durante o estágio de provisionamento de um processo de desenvolvimento baseado em componentes (Seção 2.1.5). Nesse estágio, os componentes abstratos, especificados na arquitetura do software, são instanciados concretamente. Essa instanciação pode ser feita de duas formas: reutilizando um componente já existente ou desenvolvendo um novo componente. Quando reutilizamos um componente já existente, pode ser necessário algum tipo de adaptação desse componente para adequá-lo ao contexto do novo sistema. Os novos componentes desenvolvidos devem poder ser reutilizados em outros sistemas. A estratégia intra-componente aqui descrita se aplica, portanto, tanto à adaptação de um componente já existente, para reutilização num novo sistema, como ao desenvolvimento de um novo componente reutilizável.

As possibilidades de reutilização de um componente de software são limitadas pelas hipóteses assumidas no seu desenvolvimento, a respeito dos contextos onde serão empregados. As hipóteses de falhas assumidas na especificação e implementação de um componente representam, portanto, restrições para sua reutilização. A estratégia intra-componente aqui proposta visa, essencialmente, elevar as possibilidades de reutilização de um componente de software através das seguintes táticas:

1. Com relação às interfaces requeridas do componente (Seção 2.1.1) :

a) Declarar explicitamente, na descrição dessas interfaces, todos os tipos de exceções previstos antecipadamente na especificação. Qualquer implementação do componente deve prover, obrigatoriamente, tratadores para todos esses tipos de exceções. Esses tratadores implementam um comportamento em conformidade com o previsto na especificação excepcional do componente.

b) A implementação do componente pode incluir, opcionalmente, tratadores para exceções do supertipo UndeclaredException (Figura 18), que possam ser recebidas através dessas interfaces. Essas exceções sinalizam condições excepcionais não-antecipadas para as quais

a implementação da interface pode oferecer alguma garantia quanto aos efeitos produzidos pela execução da operação. Essas exceções devem ser tratadas considerando a semântica especificada para a hierarquia de tipos de exceções da Seção 4.3.1.

c) A implementação do componente pode incluir, opcionalmente, um tratador genérico para outros tipos de exceções não declarados nessas interfaces e que não sejam do supertipo UndeclaredException. Essas exceções sinalizam condições excepcionais não-antecipadas para as quais a implementação da interface não oferece qualquer garantia. Exceções desses tipos devem ser consideradas não recuperadas, com a mesma semântica de UnrecoveredFailureException.

d) Qualquer exceção resultante de uma requisição de serviço emitida pelo componente e para a qual não haja, na implementação do componente, um tratador apropriado, deve provocar o fracasso da operação que está sendo executada pelo componente que depende dessa interface requerida.

2. Com relação às interfaces providas do componente:

e) Declarar explicitamente, na definição dessas interfaces, todos os tipos de exceções que o componente pode lançar em conseqüência das condições excepcionais antecipadas na sua especificação. Qualquer implementação do componente deve ser capaz de detectar e sinalizar corretamente essas condições excepcionais antecipadas.

f) Outras condições excepcionais não-antecipadas, que provoquem o fracasso de uma operação, devem ser sinalizadas pela implementação do componente através de uma exceção de um dos supertipos RejectedRequestException, RecoveredFailureException ou UnrecoveredFailureException, considerando a semântica especificada na hierarquia de tipos de exceções da Figura 18.

Para aplicação da estratégia intra-componente são consideradas duas formas de instanciação de um componente especificado: através da reutilização de um componente já existente ou através do desenvolvimento de um novo componente, a partir "do zero". As seções seguintes discutem a aplicação dessa estratégia em cada caso.

### 4.3.4. Aplicação da Estratégia Construindo um Componente "do Zero"

No desenvolvimento de um componente do zero, adotamos o modelo de implementação COSMOS [88], que possui os seguintes elementos básicos (Figura 20):

1. A implementação de um componente possui um ponto de acesso único (Seção 2.1.1) para cada operação provida pelo componente. Esses vários pontos de acesso são organizados em uma ou mais **interfaces providas**.
2. O acesso a uma interface provida e suas operações só é permitido através de requisições de serviços enviadas a objetos do tipo Fachada[35] [31], que são as suas **fachadas providas**. Um objeto Fachada materializa, em tempo de execução, uma interface de um componente. Uma fachada provida expõe, para o ambiente externo do componente, apenas a especificação de uma interface provida, ocultando todos os detalhes de implementação das suas operações.
3. A implementação das operações providas pelo componente é delegada, pelas fachadas providas, a uma ou mais *classes de implementação*. Essas classes de implementação são protegidas contra qualquer outra forma de acesso que não seja através de uma requisição de serviço enviada através de uma fachada provida.
4. Um componente pode requerer serviços de outros componentes. A implementação de um componente possui um ponto de acesso único para cada operação requerida pelo componente. Esses vários pontos de acesso são organizados em uma ou mais **interfaces requeridas**.
5. O acesso a uma interface requerida é feito sempre através de objetos do tipo Fachada, providos pelo ambiente, e que são suas **fachadas requeridas**.

[35] do inglês: *Façade*.

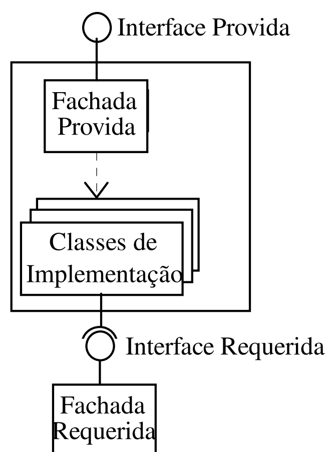


**Fig. 20.** Elementos básicos da implementação de um componente.

A estratégia proposta adiciona a esse modelo dois tipos de tratadores de exceções (Figura 21): (i) tratadores associados às classes de implementação, que são chamados tratadores da aplicação (ou tratador ALE[36]); e (iii) tratadores associados às fachadas do componente, que são chamados tratadores de fronteira (ou tratador BLE[37]).

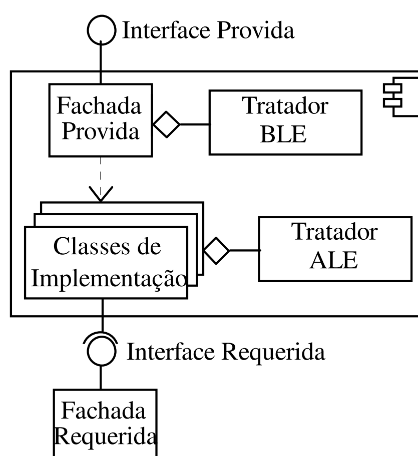


**Fig. 21.** Implementação de componente com tratadores de exceções.

A responsabilidade pelo comportamento excepcional do componente é dividida entre esses elementos da seguinte forma:

[36] do inglês: ***A**pplication-**L**evel **E**xception handler*.
[37] do inglês: ***B**oundary-**L**evel **E**xception handler*.

1. As classes de implementação são responsáveis por detectar todas as condições excepcionais antecipadas na especificação do componente.
2. As fachadas do componente podem, a critério da implementação, detectar condições excepcionais não antecipadas que impliquem na violação das pré-condições e pós-condições especificadas para as operações.
3. As classes de implementação podem, também a critério da implementação, detectar qualquer tipo de condição excepcional não antecipada que possa interferir no comportamento normal do componente.
4. As condições excepcionais que não possam ser mascaradas são sinalizadas para os clientes das operações pela própria classe de implementação ou fachada que detecta a condição.
5. As condições excepcionais que podem ser mascaradas são sinalizadas para um tratador ALE ou BLE, do próprio componente, através do lançamento de uma exceção interna, que é uma exceção de um tipo definido internamente ao componente.
6. Os tratadores ALE e BLE são responsáveis pelo tratamento das exceções internas lançadas, respectivamente, pelas classes de implementação e fachadas do componente.
7. Os tratadores ALE são responsáveis também pelo tratamento das exceções externas, que são as exceções lançadas pelas fachadas externas do componente.

### 4.3.5. Aplicação da Estratégia a Componentes Reutilizáveis

Nesse caso, pode ser necessária a adaptação das interfaces excepcionais do componente a ser reutilizado à estratégia aqui proposta. Essa adaptação é feita através de um invólucro, criado em torno do componente reutilizado, com os seguintes novos elementos (Figura 22): (i) interceptadores das interfaces providas do componente, e (ii) tratadores de exceções associados a esses interceptadores, que são os tratadores BLE.

As responsabilidades atribuídas aos elementos de um novo componente (Seção 4.3.4) passam a ser distribuídas entre os elementos do componente adaptado (Figura 22) da seguinte forma:

1. A implementação do componente reutilizado assume as responsabilidades atribuídas às classes de implementação e aos tratadores ALE.
2. Os interceptadores de fachada e os tratadores BLE correspondentes são responsáveis por adaptar os tipos das exceções lançadas pela implementação do componente à hierarquia comum de tipos abstratos de exceções. Nesse processo de adaptação, as exceções lançadas que não sejam dos tipos declarados na especificação das suas interfaces são convertidas para um subtipo de UndeclaredException.
3. Os interceptadores de fachada e os tratadores BLE assumem, adicionalmente, as demais responsabilidades atribuídas, no caso anterior, às fachadas do componente e aos tratadores BLE, respectivamente.

### 4.3.6. A Estratégia Inter-Componente

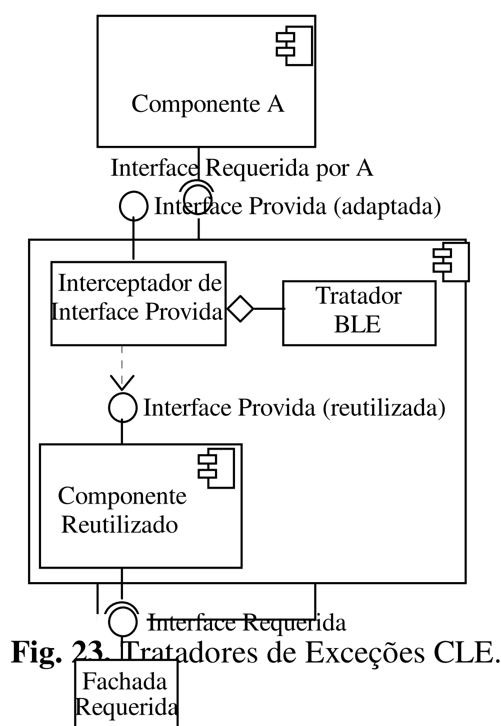


**Fig. 22.** Adaptação das interfaces excepcionais de um componente já existente.

A estratégia inter-componente cuida da integração de componentes já existentes num novo sistema. Essa estratégia é baseada em tratadores de exceções associados aos conectores, que são específicos de cada configuração (Figura 23).

Esse tratadores, chamados tratadores CLE[38] são responsáveis por:

a) Tratar exceções de configuração [46], que não podem ser tratados no contexto de um componente isoladamente. Um exemplo desse tipo de tratamento de erro é a utilização de configurações arquiteturais com componentes redundantes.
b) Adaptar as interfaces excepcionais dos componentes, convertendo os tipos das exceções lançadas por um componente servidor para exceções de tipos declarados nas interfaces requeridas dos seus clientes.
c) Resolver conflitos arquiteturais entre as hipóteses de falhas de dois ou mais componentes que se comuniquem através do conector. Esse tipo de conflito ocorre quando, em resposta a uma requisição de serviço, a implementação do componente que provê o serviço lança uma exceção sinalizando uma condição excepcional que não é prevista na especificação do componente que requisitou o serviço.

Um conector deve prover tratadores apropriados para exceções de qualquer tipo que possa ser lançado através das interfaces providas dos componentes. Isso inclui tratadores para: (i) os tipos de exceções declarados nas especificações dessas interfaces, que são subtipos de DeclaredException; (ii) os tipos de exceções não declarados nessas especificações, mas que podem ser previstos pela implementação, que são os tipos abstratos RejectedRequestException, RecoveredFailureException, e UnrecoveredFailureException; e (iii) outros tipos de exceções que possam ser lançadas por uma implementação, em condições não previstas pela mesma.

Os tratadores CLE implementam o comportamento excepcional especificado para o sistema no seu nível arquitetural, que é independente de implementações específicas dos seus componentes. Um tratador CLE pode ser capaz, em determinados casos, de mascarar a exceção, restabelecendo

[38] do inglês: ***C**onnector-**L**evel **E**xception handler*.

uma condição normal de operação e enviando um resultado normal para o cliente da operação. Caso não seja possível mascarar a exceção, o tratador CLE lança uma nova exceção para o cliente da operação, adaptando o tipo da exceção lançada às especificações da interface requerida desse cliente. A Tabela 1 define as regras gerais para determinação do tipo da exceção a ser lançada pelo tratador CLE.

**Tabela 1.** Regras de tradução de tipos de exceções por um tratador CLE.

| **Tipo da exceção lançada pelo servidor** | **Tipo da exceção propagada para o cliente** |
|---|---|
| Um tipo E1 declarado nas especificações de ambas interfaces, a provida pelo servidor e a requerida pelo cliente. | E1 (a exceção pode ser propagada automaticamente) |
| Um tipo E1, declarado na especificação da interface provida pelo servidor, para o qual existe um tipo E2 correspondente declarado na especificação da interface requerida pelo cliente (com a mesma semântica de defeito). Por exemplo: E2 é um supertipo de E1. | E2 |
| Um tipo E1, declarado na especificação da interface provida pelo servidor, para o qual não existe nenhum tipo correspondente declarado na especificação da interface requerida pelo cliente (com a mesma semântica de defeito). | Um subtipo de um dos tipos abstratos RejectedRequestException, RecoveredFailureException ou UnrecoveredFailureException, a ser escolhido de acordo com a semântica de defeito especificada para o tipo E1. |
| Um tipo E1 que é um subtipo de UndeclaredException | E1 (a exceção pode ser propagada automaticamente) |

| | |
|---|---|
| Qualquer outro tipo não declarado na especificação da interface provida pelo servidor e que não é um subtipo de UndeclaredException | Um subtipo de UnrecoveredFailureException |

## 4.4. Considerações Finais

Uma grande parcela da complexidade de sistemas de software tolerantes a falhas é associada ao comportamento do sistema em condições excepcionais. A implementação de mecanismos de tolerância a falhas num sistema requer a estruturação do comportamento excepcional das várias partes do software de acordo com uma estratégia sistêmica, que considere as hipótese de falhas e os requisitos de confiabilidade específicos do sistema. No desenvolvimento de software baseado em componentes (DBC), a implementação desse tipo de estratégia é dificultada pelos conflitos que podem surgir entre o comportamento excepcional dos componentes reutilizados e as hipóteses de falhas e requisitos de confiabilidade do novo sistema.

Neste Capítulo é proposta uma abordagem para tratamento de exceções em DBC que se divide em duas estratégias: (i) uma estratégia intra-componente aplicada localmente no desenvolvimento de um novo componente ou na adaptação de um componente reutilizável; e (ii) uma estratégia inter-componente aplicada na integração do sistema. Essas duas estratégias compartilham de uma hieraquia de tipos de exceções com uma semântica definida. Essa hierarquia de tipos de exceções possibilita a especificação do comportamento excepcional dos componentes abstratos de um sistema (Seção 2.1.5) mantendo um nível de abstração que permite a construção do sistema com base em instâncias desses componentes abstratos que possam ter sido desenvolvidas sob diferentes hipóteses de falhas.

A abordagem proposta prevê a materialização dos conectores especificados na arquitetura do software na implementação do sistema. A responsabilidade pela estratégia inter-componente é atribuída a esses conectores, o que inclui o tratamento de exceções no nível de configurações arquitetônicas, envolvendo dois ou mais componentes (Seção 2.2). Essa abordagem inclui também a extensão do modelo de componentes COSMOS [88], para inclusão do tratamento de

exceções. O Capítulo 6 descreve, no seu segundo estudo de caso, a aplicação do modelo COSMOS assim estendido a um sistema de automação bancária.

O autor colaborou na orientação de duas pesquisas de mestrado paralelas relacionadas com a estratégia aqui proposta. A primeira pesquisa [88] teve como resultado o desenvolvimento do modelo COSMOS, sobre o qual foi produzida uma publicação em co-autoria [87]. A segunda pesquisa [68], em fase de conclusão, visa a aplicação ao processo UML Components de uma metodologia para desenvolvimento do comportamento excepcional denominada MDCE [29], que é fruto de pesquisas anteriores do mesmo grupo de pesquisa. Nesse segunda pesquisa, a abordagem proposta no presente capítulo foi aplicada a um estudo de caso de um sistema de informações rodoviárias e contribuiu para o aperfeiçoamento das estratégias aqui descritas.

O Capítulo seguinte apresenta uma proposta de um ambiente integrado de desenvolvimento que visa facilitar a aplicação das abordagens aqui descritas.

# Capítulo 5
# Proposta de um Ambiente de Desenvolvimento de Software Confiável Baseado em Componentes

A adoção de um processo de desenvolvimento de software baseado em componentes é dificultada pela escassez de ferramentas mais especializadas. Os ambientes integrados de desenvolvimento de software hoje mais difundidos, como o Eclipse e Rational Rose, integram ferramentas para modelagem UML e implementação de sistemas usando linguagens orientadas a objetos, como Java e C++. A maior parte dessas ferramentas ainda são baseadas no padrão UML 1.4, cujas deficiências limitam a aplicação dessas ferramentas na modelagem de arquiteturas de software [104] e no desenvolvimento baseado em componentes.

Neste capítulo é proposto um ambiente integrado voltado para o desenvolvimento de sistemas confiáveis baseados em componentes, oferecendo recursos adequados para modelagem de arquiteturas de software e desenvolvimento baseado em componentes. O desenvolvimento desse ambiente integrado é objeto de uma dissertação de mestrado [99], iniciada recentemente no Instituto de Computação. Esse projeto está sendo desenvolvido em parceira com a empresa Autbank.

## 5.1. Requisitos Básicos do Ambiente Integrado Proposto

Nessa seção são descritos os requisitos considerados mais importantes para um ambiente de desenvolvimento de sistemas tolerantes a falhas baseados em componentes de software:

**Modelagem de Arquiteturas de Software.**

O ambiente deve apoiar a modelagem de arquiteturas de software em, pelo menos, dois níveis de abstração:

1. Modelagem de arquiteturas de software genéricas, para uma linha de produtos, com a especificação de um conjunto de tipos de componentes, tipos de conectores e restrições de comunicação. Um exemplo desse nível de abstração é o utilizado na especificação das camadas de uma arquitetura, como ilustrado na Figura 35 (Arquitetura de software em camadas do sistema Boleto de Ativos).
2. Modelagem de arquiteturas de software específicas de um sistema, com a especificação de configurações arquitetônicas formada por componentes e conectores abstratos. Um exemplo desse nível de abstração é o utilizado na especificação das camadas de uma arquitetura, como ilustrado na Figura 38 (Diagrama parcial de componentes do Sistema Boleto de Ativos).

**Especificação Normal e Excepcional de Componentes e Conectores**

O ambiente deve apoiar a especificação das interfaces externas dos componentes de uma arquitetura de software, abrangendo os quatro níveis de especificação das suas interfaces: sintático, comportamental, sincronização e qualidade de serviço (Seção 2.1.2). Essas especificações, com exceção do nível de qualidade de serviço, devem ser formais.

O ambiente deve apoiar também a especificação de conectores, associando interfaces providas e requeridas, através da especificação do protocolo utilizado na conexão dessas interfaces e das adaptações necessárias, atribuídas aos conectores.

O ambiente deve apoiar ainda a especificação do comportamento excepcional dos componentes e conectores, através da especificação de hipóteses de falhas, condições excepcionais, tipos de exceções e tratadores de exceções.

**Implementação de Componentes e Conectores**

O ambiente deve apoiar a modelagem do projeto interno de componetes e conectores e a geração automática de código, aplicando o modelo de implementação COSMOS com as extensões de tratamento de exceções (Capítulo 4).

O ambiente deve permitir o desenvolvimento das instâncias concretas de componentes e conectores por desenvolvedores independentes, geograficamente distribuídos, sem requerer a comunicação permanente entre esses grupos de trabalho. O ambiente deve oferecer também apoio para os testes de componentes e conectores.

**Repositórios de Componentes**

O ambiente deve oferecer apoio para a criação, manutenção, versionamento e busca de todos os tipos de artefatos produzidos, incluindo: especificações e implementações de interfaces, componentes, conectores e configurações. O ambiente deve permitir a utilização simultânea de repositórios distribuídos.

**Integração de Sistemas**

O ambiente deve oferecer apoio à integração de sistemas, a partir de componentes e conectores armazenados nos repositórios de componentes. O ambiente deve oferecer apoio para testes de integração de configurações de componentes e conectores.

## 5.2. Diretrizes Gerais do Projeto de Desenvolvimento do Ambiente

Foram definidas, inicialmente, as seguintes diretrizes gerais para o projeto de desenvolvimento do ambiente:

1. O ambiente deverá dar suporte ao desenvolvimento de sistemas usando a tecnologia Java, visando a maior portabilidade das aplicações desenvolvidas. No desenvolvimento do ambiente também será dada preferência à linguagem Java e ferramentas multi-plataformas.
2. O desenvolvimento do ambiente adotará o modelo de software aberto. O software produzido durante esse desenvolvimento poderá ser usado livremente, inclusive para fins comerciais, de acordo com os termos da licença *Common Public License* [66].

3. No desenvolvimento do ambiente serão utilizados, sempre que possível, componentes e ferramentas já existentes. Será priorizada a integração do ambiente com a plataforma Eclipse [26].
4. O processo de desenvolvimento do ambiente será incremental e também baseado em componentes de software, visando a liberação de versões intermediárias em curtos intervalos de tempo.
5. O ambiente será centrado no processo de desenvolvimento [1], tomando-se por base o processo denominado UML Components [17]. O ambiente deverá permitir a sua adaptação e extensão de forma a dar suporte a outros processos de desenvolvimento baseados em componentes.

## 5.3. Concepção Inicial do Ambiente Integrado

A Figura 24 apresenta a arquitetura de software de mais alto nível do ambiente proposto, com os seguintes módulos principais:

1. Controle do Processo de Desenvolvimento. Módulo para especificação e gerência do processo de desenvolvimento. Sua principal função no ambiente é guiar os desenvolvedores durante o processo de desenvolvimento escolhido, auxiliando na seleção das ferramentas a serem utilizadas em cada etapa do processo.
2. Plataforma Eclipse. É a plataforma de desenvolvimento central do ambiente, que já inclui diversas ferramentas para modelagem UML e para desenvolvimento de programas Java.
3. Novas Ferramentas. Conjunto de novas ferramentas, a serem desenvolvidas de forma integrada à plataforma Eclipse. Essas novas ferramentas oferecem funcionalidades adicionais requeridas por um processo de desenvolvimento baseado em componentes e centrado na arquitetura do software. Inicialmente, são previstas as seguintes novas ferramentas:

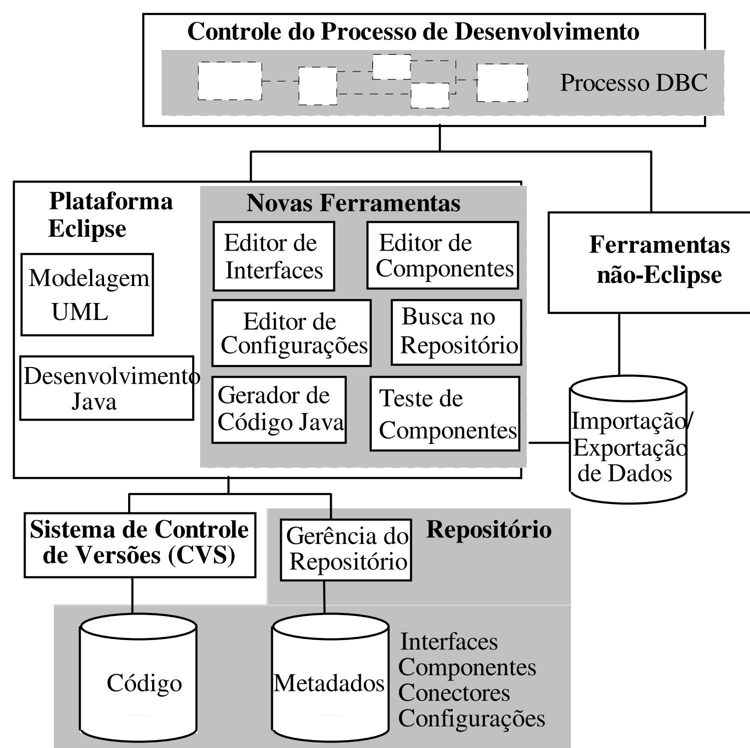


**Fig. 24.** Arquitetura de software do ambiente integrado proposto.

a) Editor de Interfaces. Ferramenta para especificação de interfaces, abrangendo os quatro níveis de especificação das interfaces: sintático, comportamental, sincronização e qualidade de serviço (Seção 2.1.2).

b) Editor de Componentes. Ferramenta para especificação de componentes, associando ao componente uma ou mais interfaces providas ou requeridas.

c) Editor de Configurações. Ferramenta para modelagem de arquiteturas de software, com a especificação de configurações de componentes e de conectores. Inclui facilidades para especificação de arquiteturas tolerantes a falhas, baseadas nos modelos do iCOTS e do iC2C, propostos no Capítulo 3, e na estratégia para tratamento de exceções proposta no Capítulo 4.

d) Gerador de Código Java. Ferramenta para geração de código Java a partir das especificações de componentes, conectores e configurações, baseando-se no modelo de implementação COSMOS [88] e nas extensões propostas no Capítulo 4.

e) Busca no Repositório. Ferramenta para recuperação de especificações e implementações armazenadas no Repositório do ambiente.

f) Teste de Componentes. Ferramenta para testes de componentes.

4. Ferramentas não-Eclipse. O ambiente poderá incluir outras ferramentas, não integradas à plataforma Eclipse, que complementem as suas funcionalidades.

5. Sistema de Controle de Versões (CVS). Módulo responsável pelo controle de versões dos artefatos de software produzidos, integrado à plataforma Eclipse.

6. Repositório. No repositório do ambiente são armazenados os diferentes artefatos produzidos pelo ambiente, tais como: (i) especificações de interfaces produzidas pelo Editor de Interfaces; (ii) especificações de componentes produzidas pelo Editor de Componentes; (iii) especificações de configurações e de conectores, produzidas pelo Editor de Configurações; (iv) implementação de componentes, configurações e conectores, produzidas pelo Gerador de Código Java. O código desses artefatos é armazenado na mesma base de dados controlada pelo Sistema de Controle de Versões. Além dessa base de dados de código, o Repositório mantém uma base de dados com metadados necessários para a recuperação dos artefatos armazenados, pela ferramenta de Busca no Repositório.

## 5.4. Situação Atual do Desenvolvimento do Ambiente

A primeira etapa do projeto consistiu na análise de ferramentas já existentes que pudessem ser integradas ao ambiente proposto ou auxiliar no desenvolvimento do ambiente. Como resultado dessa análise, foram selecionadas três ferramentas: Eclipse UML plugin [65], OpenFlow e GME [53].

1. O Eclipse UML plugin é uma ferramenta para modelagem UML e geração automática de código Java, totalmente integrada à plataforma Eclipse e ao seu repositório. O modelo UML e o código Java são mantidos automaticamente sincronizados, possibilitando a

edição de ambas visões do projeto. Embora seja um projeto comercial, é oferecida uma versão gratuita para a comunidade de desenvolvedores de software.

2. O OpenFlow [67] é uma ferramenta para gerência de processos (*workflow*), multiplataforma, integrada ao projeto de software aberto do servidor Zope [105]. É utilizada, no ambiente, para suprir as funções do módulo de Controle do Processo de Desenvolvimento.
3. O GME, ou *Generic Modeling Environment*, é um ambiente para modelagem de sistemas, configurável, desenvolvido pelo *Institute for Software Integrated Systems*, da *Vanderbilt University*. A configuração do ambiente é feita através de metamodelos que especificam o paradigma de modelagem do domínio das aplicações. A especificação dos metamodelos é feita através de diagramas de classe UML. A principal limitação do GME é sua dependência à plataforma Windows. No projeto do ambiente integrado, o GME é utilizado como ferramenta para especificação e prototipagem do ambiente.

A seguir estão descritas, sucintamente, as atividades em desenvolvimento ou já completadas e os resultados obtidos até a presente data.

**Atividade 1 - Modelagem do Processo de Desenvolvimento.** Consistiu na elaboração de um conjunto de diagramas de atividades UML com o mapeamento de todas as etapas e atividades do processo de desenvolvimento UML Components, a partir da descrição constante na sua bibliografia de referência [17]. A Figura 25 apresenta um dos diagramas obtidos, que modela a atividade de refinamento das especificações dos componentes e da arquitetura.

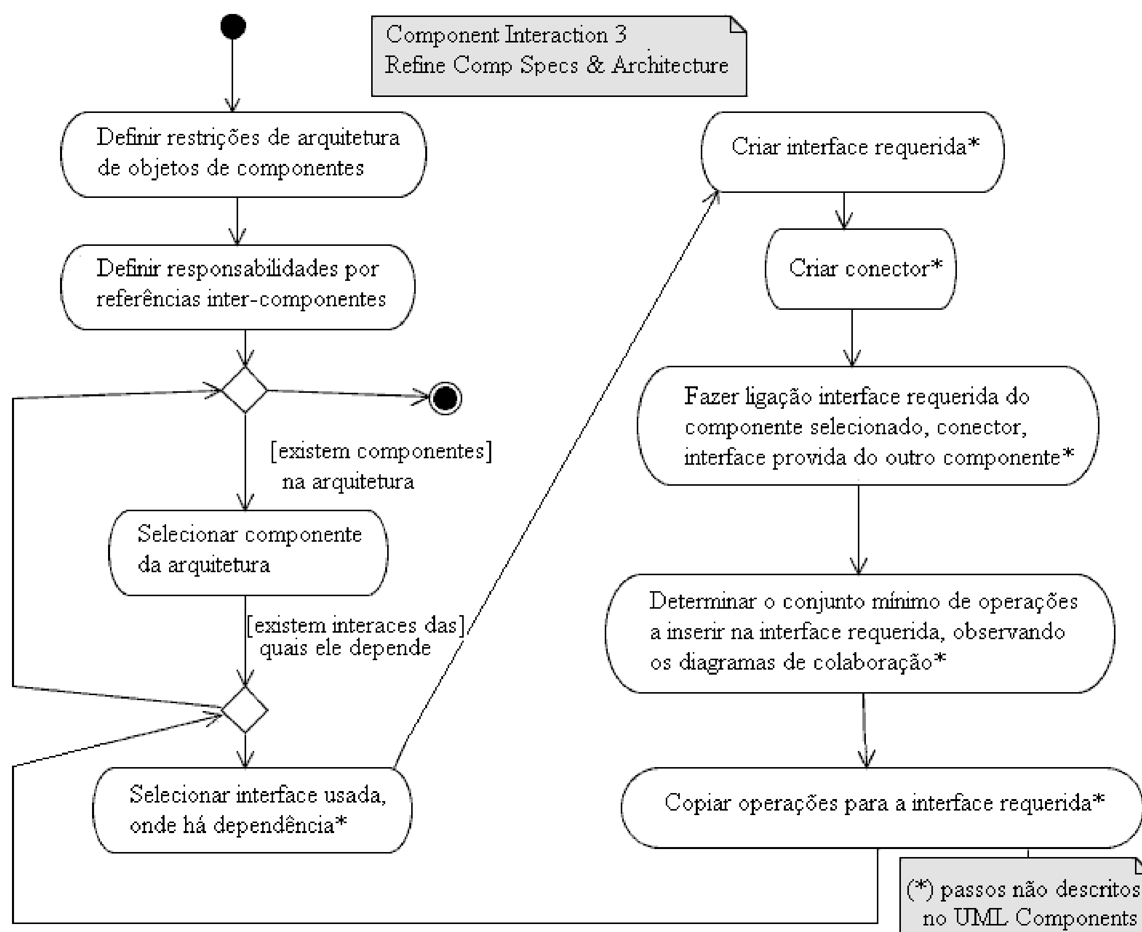


**Fig. 25.** Exemplo de diagrama de atividades do processo UML Components.

**Atividade 2 - Modelagem do Editor de Configurações.** Consistiu no desenvolvimento de um novo paradigma de modelagem para o GME, que possibilite a especificação de configurações de componentes e conectores. O metamodelo produzido está apresentado na Figura 26. O GME, configurado por esse metamodelo desenvolvido, vem sendo utilizado como um protótipo do Editor de Configurações, no desenvolvimento de aplicações reais.

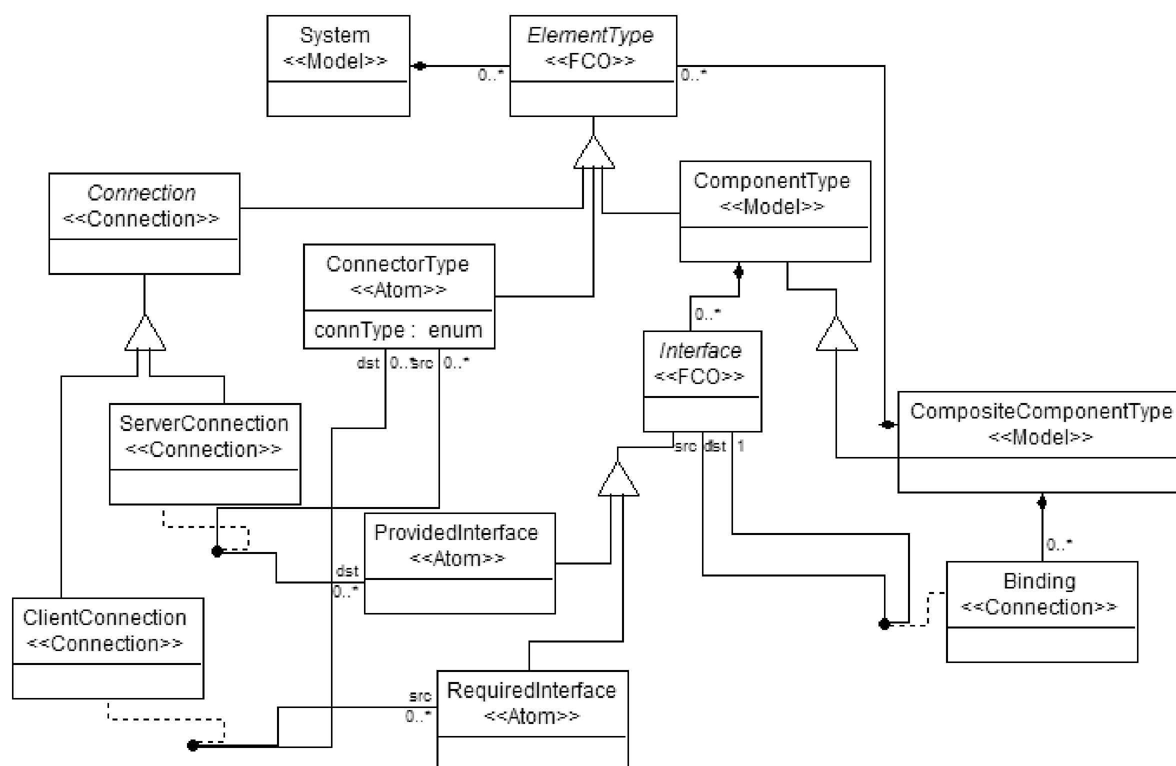


**Fig. 26.** Metamodelo GME para o Editor de Configurações.

**Atividade 3 - Especificação da Ferramenta de Teste de Componentes.** Consiste no desenvolvimento de um protótipo dessa ferramenta, ainda não integrada à plataforma Eclipse. A Figura 27 apresenta a tela principal do protótipo desenvolvido. Esse protótipo encontra-se ainda em desenvolvimento.

Os ambientes integrados de desenvolvimento de software (IDE) hoje existentes oferecem, de forma generalizada, um amplo leque de ferramentas de apoio a modelagem e implementação de sistemas de software segundo o paradigma de orientação a objetos. Por outro lado, os ambientes integrados atuais ainda são bastante imaturos no que diz respeito às necessidades mais específicas do desenvolvimento baseado em componentes e centrado na arquitetura de software. A modelagem de arquiteturas de software e a gerência de repositórios de componentes reutilizáveis são exemplos de tarefas para as quais esses ambientes geralmente não oferecem um apoio adequado de ferramentas integradas.

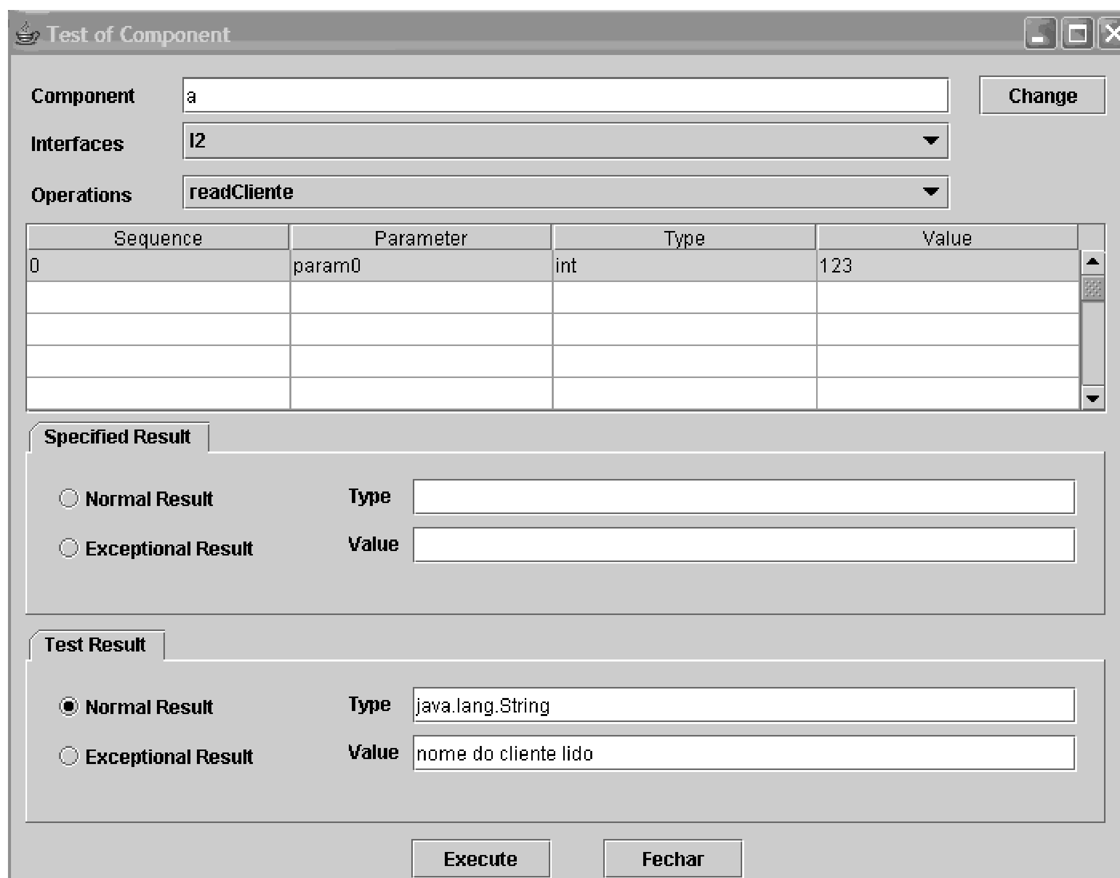


**Fig. 27.** Tela principal da ferramenta de Teste de Componentes.

## 5.5. Considerações Finais

Esse Capítulo apresenta uma proposta de um ambiente integrado, centrado no processo de desenvolvimento, e que oferece funcionalidades consideradas mais importantes para o desenvolvimento de sistemas tolerantes a falhas baseados em componentes de software. O autor do presente trabalho colabora na orientação de uma pesquisa de mestrado [99], recentemente iniciada, que se dedica ao desenvolvimento do ambiente integrado proposto.

Capítulo 5 Proposta de um Ambiente de Desenvolvimento de Software Confiável Baseado em Componentes

# Capítulo 6

# Estudos de Casos

Neste capítulo são descritos dois estudos de casos em que foram aplicadas as abordagens propostas neste trabalho. O primeiro estudo de caso é um sistema de controle de uma caldeira a vapor, descrito na literatura [3], que integra três controladores internos que são instâncias de um mesmo componente de prateleira. O objetivo desse estudo de caso foi aplicar as abordagens do iC2C e iCOTS (Capítulo 3) no desenvolvimento de uma versão tolerante a falhas desse sistema. O segundo estudo de caso é um sistema de automação bancária, desenvolvido por uma empresa independente, baseado em componentes e utilizando o modelo COSMOS [88]. O objetivo desse segundo estudo de caso foi aplicar a abordagem descrita no Capítulo 4 na reestruturação do tratamento de exceções de um sistema já existente.

## 6.1. Estudo de Caso 1: Sistema de Controle de Caldeira a Vapor

Esse sistema foi descrito num estudo de caso anterior [3] sobre projeto de invólucros protetores [71]. Esse estudo anterior foi baseado num modelo simulado desse sistema, desenvolvido usando a ferramenta de modelagem e simulação de sistemas matemáticos Simulink [56]. O presente estudo de caso partiu desse modelo de simulação, de onde foram extraídos os requisitos do sistema, para o desenvolvimento de uma versão tolerante a falhas desse sistema, baseada em componentes e utilizando os modelos do iC2C (Seção 3.1) e do iCOTS (Seção 3.2). Nesse sistema a caldeira é controlada por um sistema baseado em controladores PID[39] de prateleira. Os protetores foram utilizados para permitir a detecção e recuperação de erros típicos causados pela ausência de algum sinal, valores fora de limites pré-estabelecidos e oscilações nos valores. As principais partes desse sistema são (Figura 28):

[39] do inglês: ***P**roportional, **I**ntegral and **D**erivative controller*.

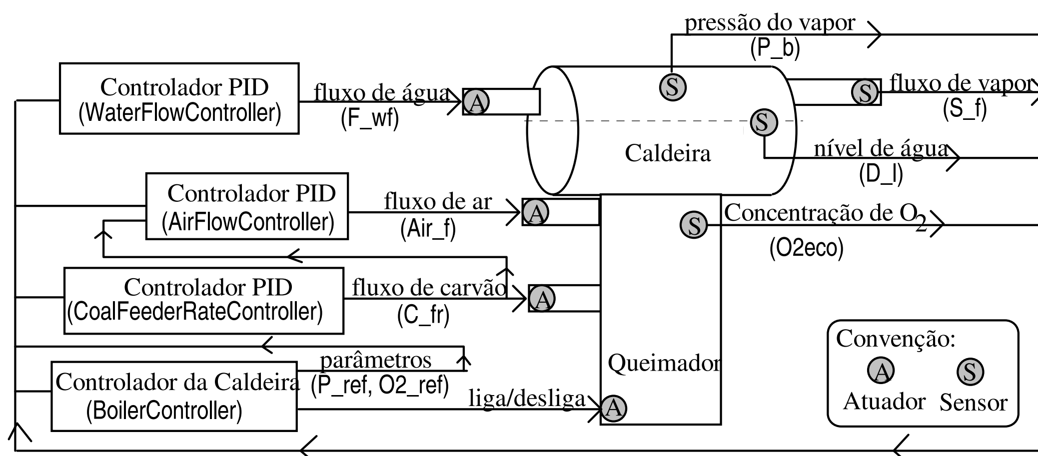


**Fig. 28.** Partes Componentes do Sistema de Controle da Caldeira.

1. Três instâncias do mesmo controlador PID de prateleira (WaterFlowController, AirFlowController e CoalFeederRateController) que controlam, respectivamente, os fluxos de entrada de água (F_wf), de ar (Air_f) e de carvão (C_fr);
2. Quatro sensores inteligentes (**S**), que monitoram o comportamento da caldeira através das seguintes variáveis: a pressão do vapor (P_b), o fluxo de vapor (S_f), o nível de água no reservatório (D_l) e a concentração de oxigênio (O2eco);
3. Quatro acionadores (**A**) que regulam o funcionamento da caldeira, permitindo ligar ou desligar a caldeira e ajustar, de acordo com os valores produzidos pelos controladores PID, os fluxos de entrada de água, de ar e de carvão.
4. O controlador da caldeira (BoilerControler), que permite ao operador configurar o sistema para diferentes níveis de operação, fixando a carga de vapor desejada (P_ref) e a qualidade do carvão utilizado (O2_ref).

As seções seguintes descrevem o desenvolvimento desse sistema de controle seguindo os passos descritos nas Seções 3.2.3 e 3.2.4.

### 6.1.1. Estágio de Provisionamento

Nesse estudo de caso, é assumido que o estágio de provisionamento foi concluído com a seleção de um controlador PID de prateleira (controlador PID COTS), como descrito no Passo 7

(Selecionar uma instância do COTS) da Seção 3.2.3 (Estágio de Provisionamento). Anderson et. al. [3] resumem as informações disponíveis que descrevem o comportamento desse componente COTS e que são usadas no desenvolvimento dos protetores. Essas informações constituem a documentação anotada do COTS, mencionada no Passo 6 (Anotar a documentação do COTS) da Seção 3.2.3.

A Figura 29 mostra a arquitetura inicial do software, que é parte da especificação do sistema. Essa arquitetura é baseada no estilo arquitetural C2 e organizada em quatro camadas, com os seguintes respectivos componentes: (i) o controlador da caldeira (BoilerController), localizado na camada 1; (ii) os controladores dos fluxos de ar (WaterFlowController) e de carvão (CoalFeederController), localizados na camada 2; (iii) o controlador do fluxo de ar (AirFlowController), localizado na camada 3, que recebe como entrada a taxa de alimentação de carvão gerada pelo CoalFeederController; e (iv) os sensores e atuadores, localizados na camada 4. A Tabela 2 especifica as principais operações providas por esses componentes.

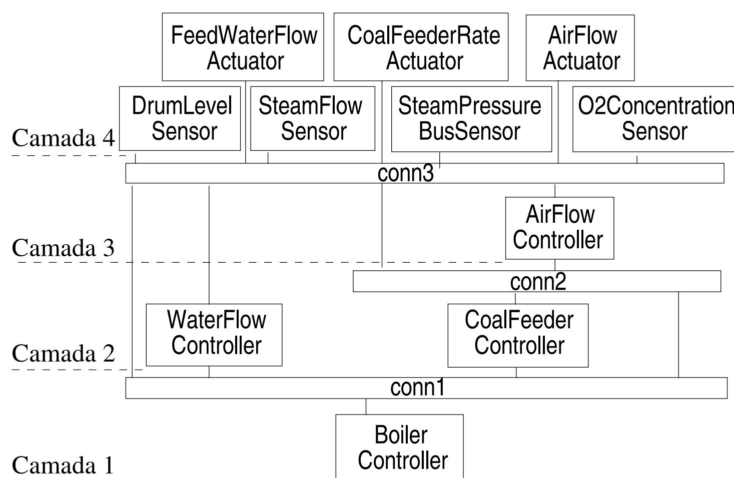


**Fig. 29.** Arquitetura Inicial do Sistema de Controle da Caldeira.

**Tabela 2.** Lista de Operações do Sistema de Controle da Caldeira.

| Operação | Componente Servidor |
|---|---|
| readDrumLevel() : D_l | DrumLevelSensor |
| readSteamFlow() : S_f | SteamFlowSensor |
| readBusPressure() : P_b | SteamPressureBusSensor |
| readO2Concentration() : O2eco | O2ConcentrationSensor |
| setFeedWaterFlow(F_wf) | FeedWaterFlowActuator |
| setCoalFeedRate(C_fr) | CoolFeederRateActuator<br>AirFlowController |
| setAirFlow(Air_f) | AirFlowActuator |
| setConfiguration(P_ref, O2_ref) | CoalFeederController<br>AirFlowController |

### 6.1.2. Estágio de Montagem

A seguir são descritos os vários passos, iniciando a partir do Passo 9 da Seção 3.2.4, para a montagem do sistema com o desenvolvimento de protetores.

*Passo 9.* *Classificar condições errôneas.* Foram considerados três tipos genéricos de condições errôneas: (i) indisponibilidade de um valor de entrada ou saída de um COTS PID; (ii) violação dos limites especificados para uma variável monitorada; e (iii) oscilações nos valores de uma variável monitorada.

*Passo 10.* *Especificar as restrições do comportamento permitido associadas às condições errôneas.* As restrições de comportamento permitido especificadas para a interface entre o AirFlowController e o restante do sistema estão apresentadas na segunda coluna da Tabela 3.

*Passo 11.* *Especificar o comportamento excepcional desejado para o sistema.* Foram definidos dois tipos de ações de recuperação, dependendo da gravidade do erro: (i) soar um alarme, e (ii) parar a caldeira.

*Passo 12.* *Atribuir responsabilidades pela recuperação de erros.* Os protetores associados aos controladores AirFlowController, WaterFlowController e CoalFeederController são responsáveis por detectar violações nas restrições de comportamento permitido. O controlador BoilerController é responsável pela recuperação, que pode ser soar o alarme ou parar o sistema.

**Tabela 3.** Especificações de detecção de erros para o AirFlowController.

| Tipo da Mensagem | Restrições de Comportamento Permitido | Notificação de Exceção |
|---|---|---|
| **Detector Superior** | | |
| Request setConfiguration (P_ref, O2_ref) | 0 <= O2_ref <= 0.1 | InvalidConfigurationSetpoint |
| | corresponding notification must be received within a specified time interval | PIDTimeout |
| Request setCoalFeeder(C_fr) | 0 <= C_fr <= 1 | InvalidCoalFeederRate |
| | *check_oscillate(Air_f)* | CoalFeederRateOscillating |
| | corresponding notification must be received within a specified time interval | PIDTimeout |
| **Detector Inferior** | | |
| Request setAirFlow(Air_f) | 0 <= Air_f <= 0.1 | InvalidAirFlowRate |
| | check_oscillate(Air_f) | AirFlowRateOscillating |
| | corresponding notification must be received within a specified time interval | AirFlowActuatorTimeout |
| Notification from readO2Concentration() | 0 <= O2eco <=1 | InvalidO2Concentration |

*Passo 13. Refinar a arquitetura de software.* A solução proposta aplica os conceitos do iCOTS e iC2C para estruturar os quatro controladores. O AirFlowController, o WaterFlowController e o CoalFeederController são estruturados como componentes iCOTS que encapsulam instâncias do controlador COTS (PID Controller) envolvidas por protetores. O BoilerController é estruturado como um iC2C desenvolvido especificamente para o sistema. A seguir é descrito o desenvolvimento do iCOTS associado ao AirFlowController. A Figura 30 mostra a estrutura interna desse iCOTS, seguindo os modelos descritos na Seção 3.2. Essa mesma solução se aplica ao desenvolvimento dos iCOTS associados aos controladores WaterFlowController e o CoalFeederController.

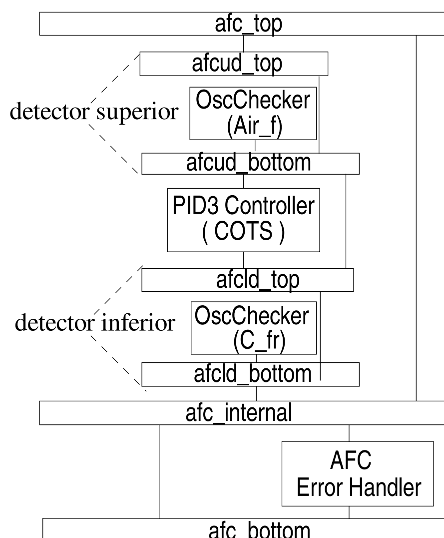


**Fig. 30.** Decomposição do Componente AirFlowController.

O controlador PID COTS (PID3 Controller) envolvido por um par de detectores de erros (detector superior e detector inferior) é inserido no lugar do componente AtividadeNormal de um iCOTS (Figura 14). Ambos detectores usam instâncias do componente OscChecker, que verifica se o valor de uma variável tende a se estabilizar dentro de um número máximo de oscilações aceitável.

A Tabela 3 especifica, para cada um desses detectores: os tipos de mensagens a serem inspecionadas, as assertivas que garantem as restrições do comportamento permitido e o tipo da notificação de exceção que deve ser gerada quando uma restrição é violada. A Tabela 4 resume os tipos das notificações de exceção geradas, classificadas de acordo com os tipos genéricos definidos no Passo 9. Os tipos de exceções InvalidConfigurationSetpoint e InvalidCoalFeederRate são exceções de interface, enviadas diretamente para a camada inferior da arquitetura de software. Os demais tipos de exceções são exceções internas ao AirFlowController e, portanto, tratados pelo componente AFCErrorHandler, inserido no lugar do componente AtividadeAnormal do iCOTS.

**Tabela 4.** Resumo das notificações excepcionais do sistema de controle da caldeira.

| Notificação de Exceção | Tipo Genérico de Exceção |
|---|---|
| PIDTimeout | NoResponse (indisponibilidade de valore de entrada/saida de Controlador PID) |
| AirFlowActuatorTimeout | |
| InvalidConfigurationSetpoint* | OutOfRange (violação de limites de variável monitorada) |
| InvalidCoalFeederRate* | |
| InvalidO2Concentration | |
| InvalidAirFlowRate | |
| CoalFeederRateOscillating | Oscillation (oscilação em variável monitorada) |
| AirFlowRateOscillating | |
| * Exceções de interface. | |

De acordo com o especificado no Passo 12, o AFCErrorHandler apenas propaga essas exceções internas como notificações de exceções de defeito, do tipo genérico correspondente especificado na Tabela 4. Uma exceção interna PIDTimeout, por exemplo, gera uma exceção de defeito do tipo NoResponse.

O componente BoilerController é responsável por:

1. Configurar o sistema de controle da caldeira, enviando requisições do tipo setConfiguration() quando necessário.
2. Tratar as notificações de exceções de interface que podem ser geradas em resposta a uma requisição setConfiguration().
3. Tratar as notificações de exceções de defeito do tipos genéricos especificados na Tabela 4, que podem ser geradas por qualquer dos três controladores iCOTS (WaterFlowController, CoalFeederController ou AirFlowController).

O componente BoilerController foi estruturado como um iC2C para lidar com as responsabilidades de tolerância a falhas especificadas nos itens 2 e 3 acima, além das responsabilidades funcionais especificadas no item 1 acima.

A Figura 31 mostra a arquitetura de software final obtida, que é um refinamento da arquitetura de software inicial (Figura 29) com a inclusão de configurações tolerantes a falhas para os diversos

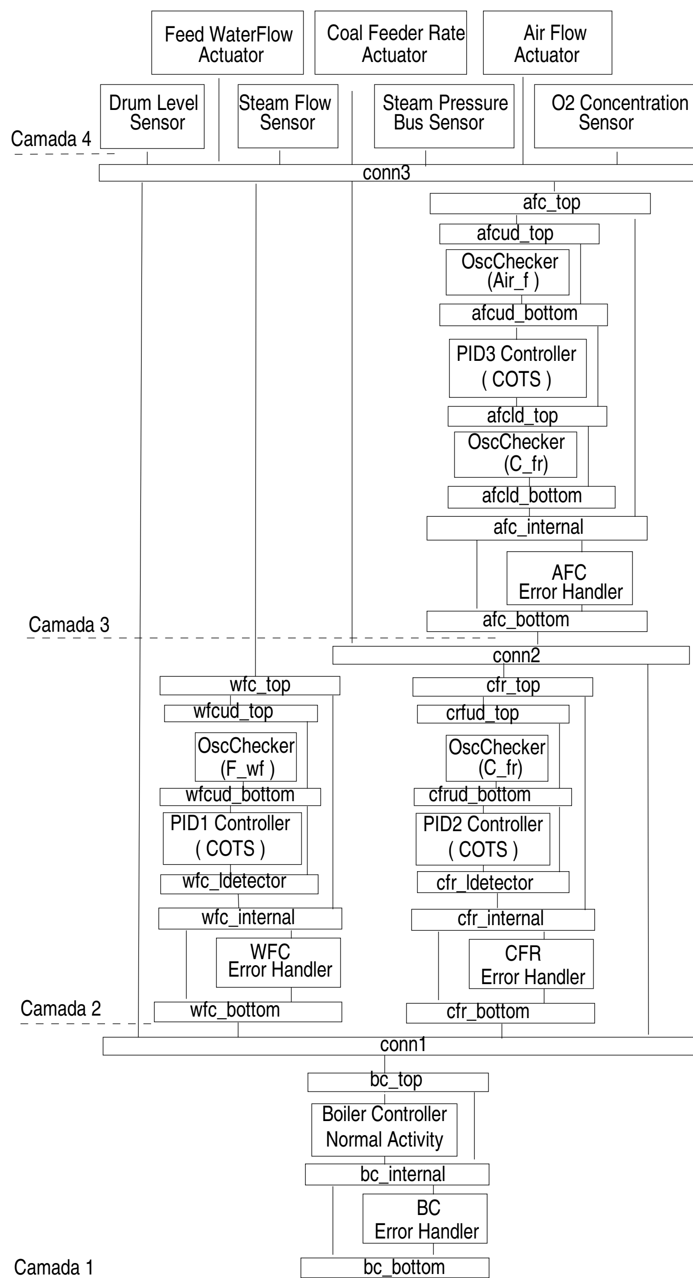


**Fig. 31.** Arquitetura de Software Final do Sistema de Controle da Caldeira.

controladores. Assume-se que os componentes que implementam os sensores e atuadores, assim como os diversos conectores, são confiáveis, ou seja, não estão sujeitos a falhas.

A Figura 32 mostra um diagrama de seqüência em UML ilustrando o fluxo de mensagens entre os componentes do sistema quando o controlador PID associado ao AirFlowController (PID3Controller)

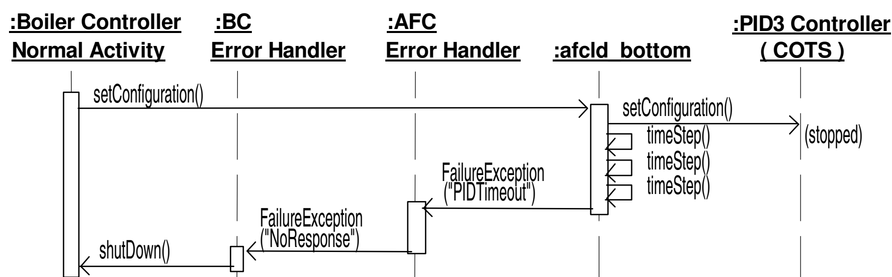


**Fig. 32.** Diagrama de seqüência UML para uma exceção PIDTimeout.

deixa de responder durante o processamento de uma requisição setConfiguration() enviada pelo BoilerController. Quando o detector inferior do AirFlowController (afcld_bottom) detecta que o controlador PID não está respondendo, é lançada uma exceção interna do tipo PIDTimeout. Como o AFCErrorHandler não é capaz de mascarar esse tipo de exceção, ele lança uma nova exceção, do tipo NoResponse, que é tratada pelo componente BCErrorHandler que envia a mensagem shutDown() para o BoilerController, que desliga o sistema.

*Passo 14. Implementar detectores de erros.* Durante esse passo e os subseqüentes foi utilizado um arcabouço orientado a objetos[40] que provê uma implementação genérica para as abstrações principais definidas para o iC2C e o iCOTS. Usando esse arcabouço os detectores superior e inferior de um iCOTS são implementados como subclasses de LowerDetectorAbst e UpperDetectorAbst, respectivamente. A Figura 33 mostra o código parcial da classe AfcLowerDetector, que implementa o detector inferior associado ao AirFlowController. Nesse exemplo, os métodos setConfiguration() e setCoalFeedRate() interceptam as requisições enviadas para o AirFlowController e verificam as restrições de comportamento permitido associadas a essas requisições, conforme especificado na Tabela 3. Quando uma restrição de comportamento permitido é violada, uma exceção é lançada através do método raiseException(), que é implementado pelas classes abstratas do

[40] do inglês: *object-oriented framework*.

arcabouço. Requisições válidas são entregues ao AirFlowController que é conectado à interface wrappedNormal do detector.

*Passo 15. Implementar tratadores de erros.* Usando o mesmo arcabouço já mencionado, os tratadores de erros são implementados como subclasses de AbnormalActivityAbst. A Figura 34 mostra o código da classe AFCErrorHandler, que implementa o tratador de erros associado ao AirFlowController. Nesse exemplo, o método handle() é chamado quando uma notificação de exceção de defeito é lançada por um dos detectores de erro que envolvem o AirFlowController. Essas exceções são propagadas após a conversão do seu tipo para o tipo genérico correspondente, conforme especificado na Tabela 4.

```
1  public class AfcLowerDetector extends LowerDetectorAbst
2              implements IConnector, IAirFlowController {
3   private IAirFlowController wrappedNormal;
4   private OscillatorChecker oscillatorChecker;
5   public void setConfiguration
6                (double P_ref, double O2_ref) {
7     if (O2_ref<0 || O2_ref>0.1)
8       raiseException (new InvalidConfigurationSetpoint(O2_ref));
9     try {wrappedNormal.setConfiguration(P_ref, O2_ref);
10    } catch (TimeoutException e) {
11      raiseException (new PIDTimeout("setConfiguration()"));
12    } catch (AbortException e) { aborted(); }
13  }
14  public void setCoalFeedRate(double C_fr) {
15    if (C_fr<0 || C_fr>1)
16      raiseException(new InvalidCoalFeederRate(C_fr));
17    if (oscillatorChecker.check_oscillate(C_fr))
18      raiseException (new CoalFeederRateOscillating(C_fr));
19    try {
20      wrappedNormal.setCoalFeedRate(C_fr);
21    } catch (TimeoutException e) {
22        raiseException (new PIDTimeout("setCoalFeedRate()"));
23    } catch (AbortException e) { aborted(); }
24  }
```

**Fig. 33.** Implementação parcial Java do detector inferior do AirFlowController.

```
1  public class AFCErrorHandler
2   extends AbnormalActivityAbst implements IComponent {
3   public void handle(Exception exception) {
4     try { throw exception; }
5       catch (PIDTimeout e) {
6       throw new NoResponse(e);
7     } catch (AirFlowActuatorTimeout e) {
8       throw new NoResponse(e);
9     } catch (InvalidO2Concentration e) {
10      throw new OutOfRange(e);
11    } catch (InvalidAirFlowRate e) {
12      throw new OutOfRange(e);
13    } catch (CoalFeederRateOscillating e) {
14      throw new Oscillation(e);
15    } catch (AirFlowRateOscillating e) {
16      throw new Oscillation(e);
17    } catch (Exception e) {
18      throw new FailureException(e); }
19  }
```

**Fig. 34.** Implementação do tratador de erros do AirFlowController.

*Passo 16. Integrar os protetores.* O trecho de código abaixo cria um componente afc composto que encapsula o AirFlowController e os detectores de erros e tratador de erros associados ao mesmo, estruturado como um iCOTS (Figura 30), que é um tipo especializado de um iC2C. O AirFlowController, envolvido pelos dois detectores de erro, atua como o componente atividade normal do iC2C e o tratador de erro atua como o atividade anormal desse iC2C.

```
Icomponent afc=new iC2C( new AfcWrappedNormal
                            ( new AirFlowController(),
                              new AfcLowerDetector(),
                              new AfcUpperDetector() ),
                            new AFCErrorHandler() );
```

*Passo 17. Integrar o sistema.* A integração dos vários componentes e conectores que compõem o sistema é codificada no método main() da classe StartUp, baseando-se na configuração apresentada na Figura 31. O trecho de código abaixo ilustra esse método com: (i) a instanciação do componente bc como um iC2C composto pelo componente BoilerController e seu tratador de erro (BCErrorHandler); (ii) a instanciação de um conector conn1; e (iii) a conexão entre as interfaces de cima do componente bc e de baixo do conector conn1.

```
IComponent bc=new iC2C
       (new BoilerController(), new BCErrorHandler());
IConnector c1=new Conn1();
bc.connectTop(c1);
```

### 6.1.3. Discussão do Estudo de Caso 1

O sistema resultante foi testado num ambiente de simulação que permitiu a injeção de diferentes falhas no sistema. O sistema final foi bem sucedido em todos esses testes, comportando-se como especificado mesmo na presença de falhas. Uma limitação desse caso de estudo é o fato de ser baseado numa simulação do sistema de caldeira, o que não possibilitou uma análise de desempenho objetiva, em condições reais de operação. Durante a execução do sistema, o código dos protetores só é executado quando uma requisição de serviço é recebida (ou emitida) pelo componente COTS envolvido pelo protetor. O acréscimo no tempo de execução devido a um protetor é proporcional ao número e complexidade das restrições encapsuladas no protetor. Se assumirmos que essa complexidade deve ser inferior à complexidade dos serviços providos pelos componentes COTS envolvidos pelos protetores, podemos deduzir que o impacto no desempenho do sistema será relativamente baixa.

## 6.2. Estudo de Caso 2: Sistema de Automação Bancária

### 6.2.1. Objetivo

O objetivo desse estudo de caso foi aplicar, a um caso real, a estratégia proposta no Capítulo 4, para tratamento de exceções em sistemas baseados em componentes. O estudo foi realizado em parceria com uma empresa de desenvolvimento de software que atua há aproximadamente vinte anos no mercado de automação bancária nacional. Situada em São Paulo, essa empresa possui um corpo técnico de aproximadamente sessenta pessoas e é líder no seu segmento de mercado, que são bancos de pequeno porte vinculados a grandes grupos empresariais, aos quais prestam serviços financeiros variados, como contas correntes, captação de recursos e empréstimos.

Para esse estudo de caso, foi escolhido o sistema denominado Boleto de Ativos, para o qual já existia uma versão inicial, em fase de desenvolvimento, com uma arquitetura de software baseada em componentes. O tratamento de exceções, nessa versão inicial, era feito de forma não-sistemática. Durante esse estudo de caso foi desenvolvida uma nova versão do mesmo sistema,

implementando um tratamento sistemático das exceções, ainda de forma parcial, seguindo a estratégia descrita no Capítulo 4.

### 6.2.2. Descrição do Sistema Boleto de Ativos

A função principal desse sistema é captar propostas de financiamento, tais como de *leasing* de equipamentos, com informações a respeito das necessidades dos clientes que solicitam esse tipo de operação. A arquitetura de software em camadas desse sistema está ilustrada na Figura 35. A camada de Apresentação é implementada através de páginas HTML dinâmicas que são controladas por um componente reutilizável que é chamado PortalWeb. A camada de Sistema contém os componentes que implementam as operações requeridas pelos diversos casos de uso do sistema, chamados componentes SB. A camada de Entidades contém os componentes que implementam as entidades fundamentais do negócio como, por exemplo, uma conta corrente, e que são chamados componentes EB. A camada de Persistência contém os componentes que oferecem serviços de acesso a um banco de dados relacional que é utilizado para persistir o estado dos componentes EB.

A implementação do sistema utiliza a linguagem Java, seguindo o modelo COSMOS [88]. O

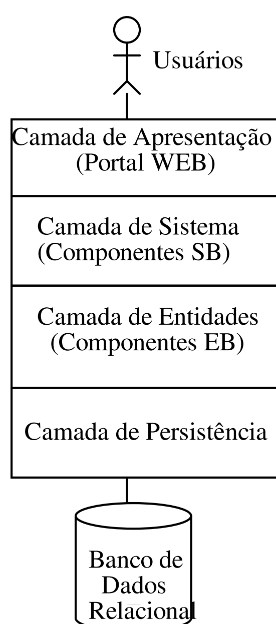


**Fig. 35.** Arquitetura de software em camadas do sistema Boleto de Ativos.

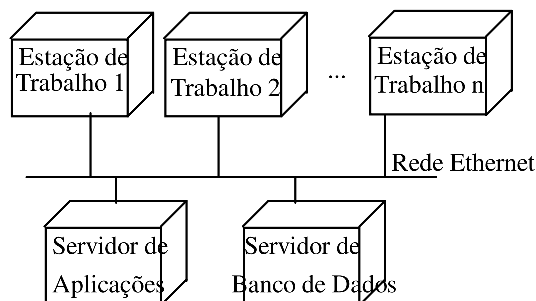


**Fig. 36.** Visão física do Sistema Boleto de Ativos.

sistema é instalado numa rede de computadores formada por estações de trabalho, um servidor de aplicações e um servidor de banco de dados (Figura 36). As estações de trabalho são operadas através de navegadores Web padrões, tais como Internet Explorer e Netscape Navigator. O componente PortalWeb e os componentes das camadas de Sistema e de Entidades são instalados no servidor de aplicações. O gerenciador de banco de dados é instalado no servidor de banco de dados.

Para esse estudo de caso foi escolhido o caso de uso "Cadastramento de Propostas de Leasing sem Garantias ou Garantidores" (Figura 37). Os passos 4, 9, 24 e 25 desse caso de uso requerem operações de componentes da camada de Sistema. Os demais passos desse caso de uso são realizados através de operações providas diretamente pelo componente PortalWeb, da camada de Apresentação, ou por componentes de outros sistemas relacionados. A Figura 38 apresenta um diagrama parcial do Sistema BA com os componentes e conectores requeridos por esse caso de uso.

Caso de Uso: Cadastramento de Propostas de Leasing sem Garantias ou Garantidores

Objetivo: Inserir nova proposta no sistema

Passos:

1. usuário informa nome do cliente
2. sistema busca cliente no banco de dados do Sistema Infobank (IB) e retorna: nome do cliente, código, cpf/cnpj, cod. do gerente e nome do gerente
3. usuário inicia busca do produto-boletagem[41]
4. sistema exibe lista de produtos-boletagem cadastrados (*)
5. usuário seleciona produto-boletagem
6. sistema retorna código do produto boletagem, código do produto no IB, tipo de leasing e origem de recursos
7. usuário informa cód. da empresa, código da agência, data de início, prazo total, quant. de parcelas e periodicidade
8. usuário inicia busca de tipo de vencimento
9. sistema exibe lista de tipos de vencimento cadastrados (*)
10. usuário seleciona tipo de vencimento
11. informa valor principal, tipo de residual, quantidade parcelas de residual e percentual do residual, taxa básica, periodicidade, forma de cálculo, indexador
12. usuário inicia inclusão de bem
13. usuário informa código de garantia do BA
14. usuário inicia busca de bens fabricados
15. sistema lista bens fabricados cadastrados
16. usuário seleciona o bem fabricado
17. sistema recupera dados do bem que constam do cadastro
18. informa n. de série, data de aquisição
19. usuário informa grupo de bem
20. usuário informa vida útil
21. confirma inclusão do bem
22. confirma inclusão da proposta
23. sistema valida a proposta
24. sistema requisita do Sistema de Leasing (LE) o cálculo do valor residual total, valor das parcelas de residual, valor das parcelas de aluguel, juros principal e saldo devedor (*)
25. sistema inclui a proposta no banco de dados do BA (*)

(*) Passos em que há interação com algum componente da camada SB do Sistema BA.

**Fig. 37.** Caso de uso Cadastramento de Proposta de Leasing.

[41] tipo de produto cadastrado no Sistema Infobank com parâmetros associados.

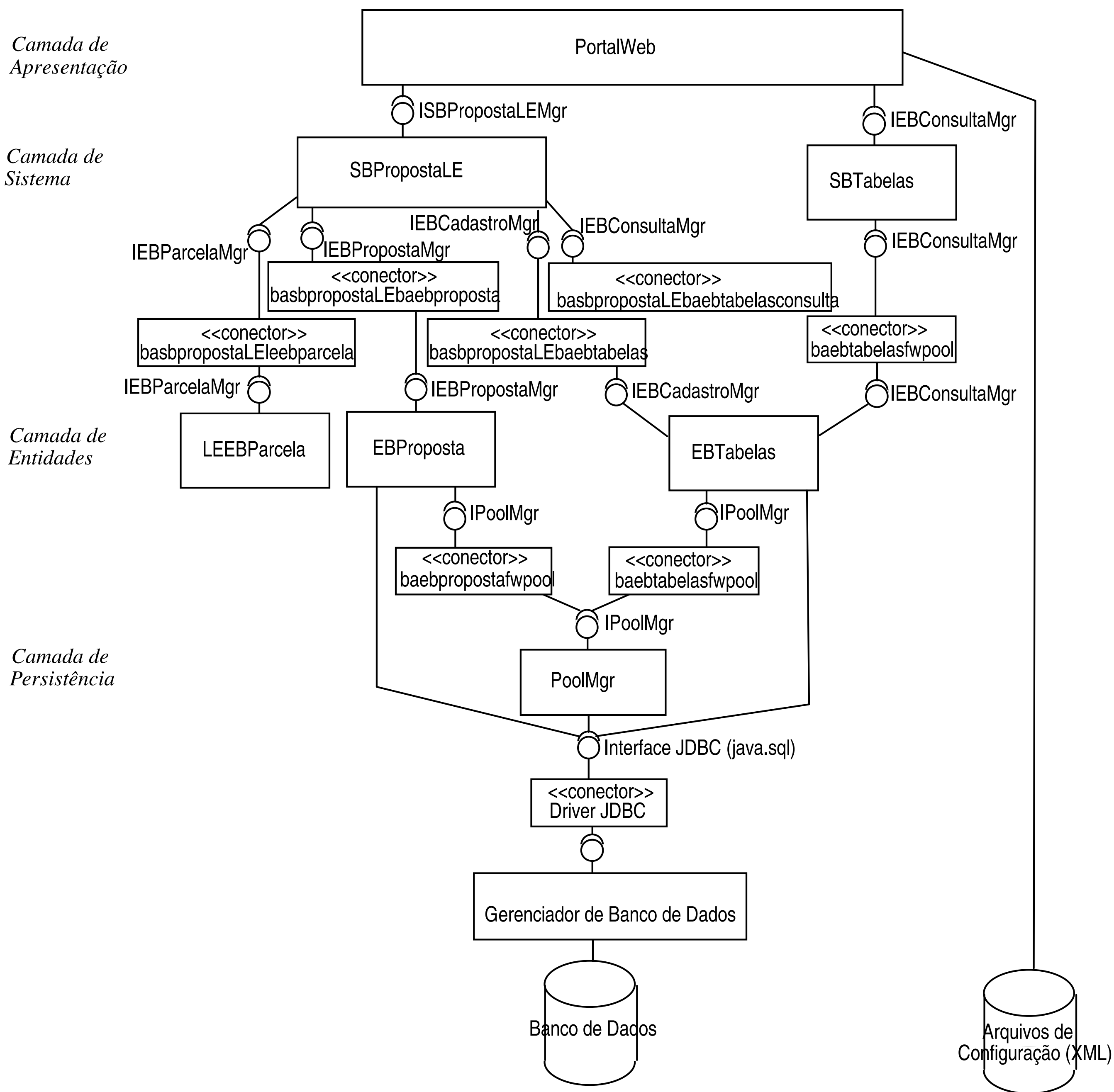


**Fig. 38.** Diagrama parcial de componentes do Sistema Boleto de Ativos.

Após o levantamento dessas informações sobre a funcionalidade e estrutura do sistema, foi selecionado o passo de número 4 (*"sistema exibe lista de produtos-boletagem cadastrados"*) da Figura 37, como exemplo típico das operações realizadas pelo sistema e que envolve componentes das várias camadas. Esse passo é implementado através da operação listaProdutos()

provida pelo componente SBTabelas, através da interface IEBConsultaMgr, e depende também dos componentes EBTabelas, Pool e do Gerenciador de Banco de Dados (Figura 38).

Para essa operação foi desenvolvido o seguinte plano de atividades:

Atividade 1. Especificação das hipóteses de falhas a serem assumidas no projeto do comportamento excepcional do sistema.

Atividade 2. Projeto do comportamento excepcional, de acordo com as hipóteses de falhas definidas na Atividade 1 e no modelo de tratamento de exceções descrito no Capítulo 4.

Atividade 3. Alteração do código da versão inicial do sistema, produzindo uma nova versão em conformidade com o projeto de comportamento excepcional elaborado na Atividade 2.

Atividade 4. Testes das versões inicial e final do sistema, com injeção de falhas que permitam exercitar o modelo de falhas especificado.

Atividade 6. Avaliação final, com a comparação dos resultados dos testes e dos códigos das versões inicial e final do sistema.

A seguir estão descritos os resultados obtidos no desenvolvimento dessas atividades.

### 6.2.3. Especificação das Hipóteses de Falhas

A partir de uma análise do código da versão inicial do sistema foram levantados: (i) um diagrama de seqüência, em UML, com o detalhamento da operação listaProdutos() provida pelo componente SBTabelas (Figura 39); e (ii) uma lista das condições excepcionais possíveis de ocorrer durante a execução dessa operação (Tabela 5).

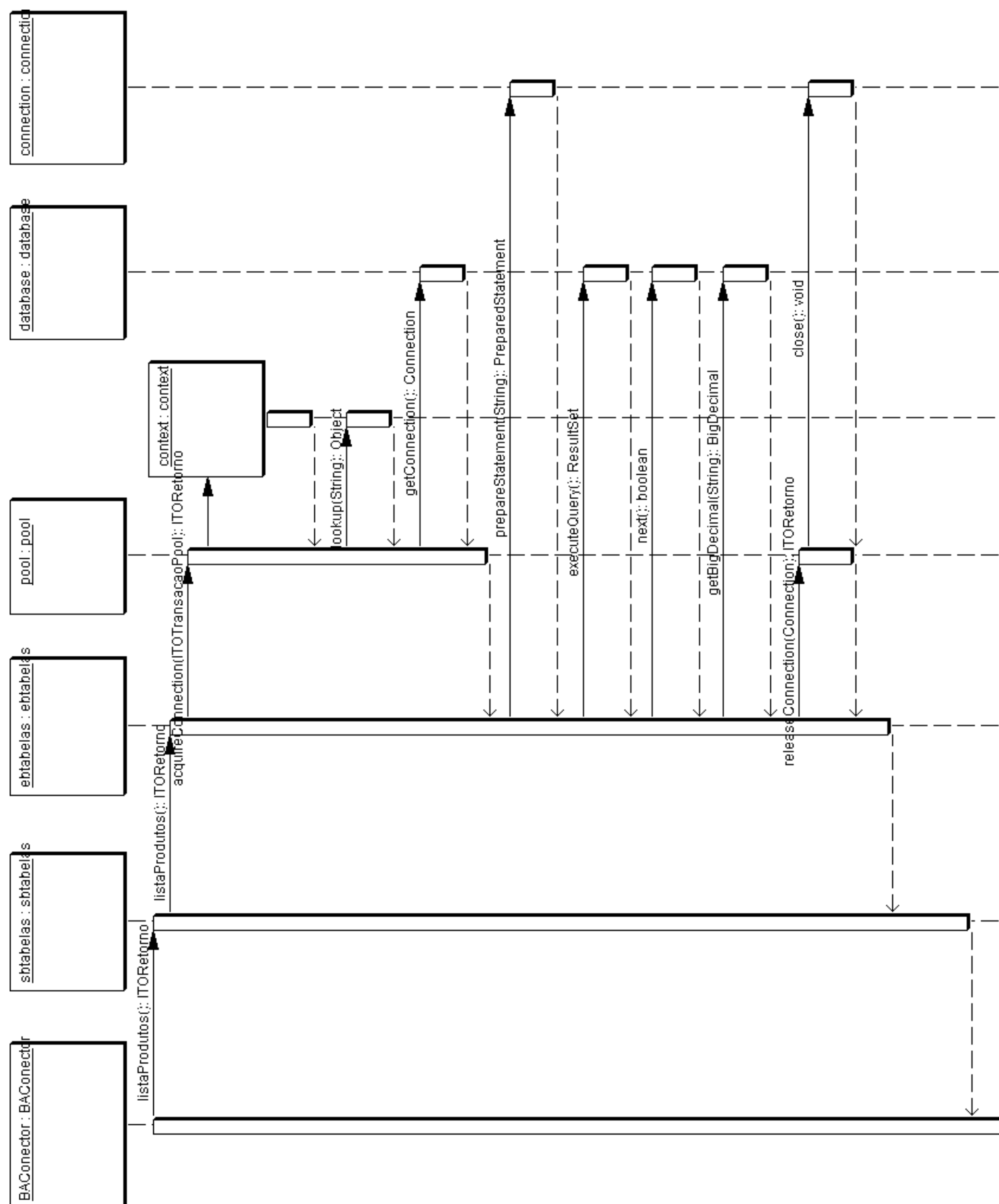


**Fig. 39.** Diagrama de seqüência da operação listaProduto().

**Tabela 5.** Tipos de condições excepcionais ao listar produtos.

| Operação Executada | Tipo da Exceção | Origem da Falha | Falha Provável |
|---|---|---|---|
| Context::lookup() | NamingException | Componente Pool | passou nome do recurso inválido |
| | | Configuração do sistema (*resource file*) | nome do recurso errado |
| DataSource::getConnection() | SQLException | Resource file | nome do usuário, senha ou nome do banco errados |
| | | | número de conexões simultâneas inadequado |
| | | Banco de Dados | não disponível |
| | | | usuário/senha não autorizado |
| | | | corrompido |
| | | Sistema BA | conexões retidas indevidamente |
| | | Rede | rede indisponível |
| Connection::prepareStatement() | SQLException | Componente Pool | conexão já fechada |
| | | Rede | rede indisponível |
| PreparedStatement::executeQuery() | SQLException | Componente EBTabelas | erro no SQL passado no prepareStatement() |
| | | Rede | rede indisponível |
| ResultSet::next() | SQLException | Não determinada | |
| ResultSet::getBigDecimal() | SQLException | Componente EBTabelas | nome da coluna inválido |
| | | Administração do Banco de Dados | esquema do banco desatualizado |
| | ArrayIndexOutOfBounds | Componente EBTabelas | recuperando campo float s/ atribuição da escala |
| Connection::close() | SQLException | Driver JDBC | |

A análise dessas condições excepcionais resultou na definição dos seguintes tipos de falhas a serem considerados:

Grupo 1. Falhas de software no Sistema de Boleto de Ativos como, por exemplo, falhas na implementação no componente SBTabelas, da camada de Sistema.

Grupo 2. Falhas no Banco de Dados como, por exemplo, Gerenciador do Banco de Dados indisponível.

Grupo 3. Falhas no Driver JDBC como, por exemplo, uma conexão com o Banco de Dados liberada indevidamente.

Grupo 4. Falhas nos Arquivos de Configuração do sistema como, por exemplo, senha de acesso ao Banco de Dados incorretos.

Grupo 5. Falhas na administração de Banco de Dados como, por exemplo, esquema do Banco de Dados desatualizado.

Grupo 6. Falhas na rede (rede indisponível).

Foi assumido também que: (i) a ocorrência de qualquer dessas falhas causa o lançamento de uma exceção, de um dos tipos especificados na Tabela 5; e (ii) num dado momento o sistema apresenta no máximo uma falha.

### 6.2.4. Projeto do Comportamento Excepcional do Sistema

O projeto do comportamento excepcional do sistema limitou-se às camadas de Sistema, Entidades e Persistência (Figura 35). A camada de apresentação não foi incluida nesse estudo.

Com base nas hipóteses de falhas definidas na Seção 6.2.3, foi especificado o seguinte comportamento excepcional esperado para o sistema. Como regra geral em caso de falha, o sistema deve interromper a operação onde o erro foi detectado e enviar mensagem para o usuário do sistema relatando a ocorrência do erro. Falhas de software no sistema Boleto de Ativos (Grupo 1) não devem impedir que o usuário continue a operar o sistema ou que tente executar novamente a operação que falhou. Para qualquer outro tipo de falha, a execução do sistema deve ser interrompida. No caso de falhas na rede (Grupo 6), o sistema deve fazer uma segunda tentativa de executar a operação, antes de interromper a operação.

#### 6.2.4.1. A nova interface IPoolMgr

Na versão inicial do sistema, a interface IPoolMgr continha apenas duas operações: acquireConnection() e releaseConnection(). O resultado da operação acquireConnection() era um objeto do tipo java.sql.Connection. Através desse objeto os componentes da camada de Entidades, tais como o EBTabelas, acessavam o banco de dados diretamente, através da interface JDBC (Figura 38) e sem o controle do componente Pool. No projeto do comportamento excepcional,

decidiu-se eliminar essa comunicação direta entre os componentes da camada de Entidades e o banco de dados. Para isso, a nova interface IPoolMgr (Figura 40) inclui operações que, na versão inicial do sistema, eram providas pela interface java.sql.Connection, tais como as operações prepareStatement() e excecuteQuery(). Dessa forma, na nova versão, a interface IPoolMgr é o único ponto de acesso ao banco de dados, passando apenas objetos imutáveis como parâmetros e resultados dessas operações.

```
1  public Interface IPoolMgr {
2
3   ITOConnection acquireConnection
4         (ITOTransacaoPool dadosTransacao, ObjectFactory objectFactory)
5     throws SimboloSistemaInvalidoException, ConexaoIndisponivelException,
6         RecursoInvalidoException, FalhaJdbcDriverException;
7
8   void releaseConnection(ITOConnection idcon)
9     throws ConexaoNulaException, FalhaJdbcDriverException;
10
11  ITOPreparedStatement prepareStatement
12        (ITOConnection idcon, String sql,  ObjectFactory objectFactory)
13    throws SqlInvalidoException;
14
15  ITOResultSet executeQuery
16        (ITOPreparedStatement idpst,   ObjectFactory objectFactory)
17    throws SqlInvalidoException;
18
19  boolean next(ITOResultSet idrst)
20    throws SqlInvalidoException;
21
22  String getString(ITOResultSet idrst, String fieldName)
23    throws SqlInvalidoException;
24
25  int getInt(ITOResultSet idrst, String fieldName)
26    throws SqlInvalidoException;
27
28  Timestamp getTimestamp(ITOResultSet idrst, String fieldName, Calendar cal)
29    throws SqlInvalidoException;
30
31  Date getDate(ITOResultSet idrst, String fieldName, Calendar cal)
32    throws SqlInvalidoException;
33
34  BigDecimal getBigDecimal(ITOResultSet idrst, String fieldName, int scale)
35    throws SqlInvalidoException;
36 }
```

**Fig. 40.** Nova versão da interface IPoolMgr.

#### 6.2.4.2. Sinalização das Condições Excepcionais

Na implementação original, os resultados das operações eram passados numa estrutura do tipo ITORetorno, com dois atributos: tipo do resultado, obtido através da operação getTipo(), e valor do resultado, obtido através da operação getConteudo(). O tipo do resultado é um inteiro cujo valor indica se a operação foi concluída normalmente, com sucesso, ou por alguma condição excepcional. O valor do resultado é um objeto do tipo Object que contém, no caso de uma operação bem sucedida, o resultado normal da operação. No projeto do comportamento excepcional decidiu-se eliminar essa convenção e sinalizar as condições excepcionais através do lançamento de exceções Java, de acordo com a hierarquia de tipos de exceções definida na Seção 4.3.1 (Figura 18).

#### 6.2.4.3. Detecção das Condições Excepcionais

O componente Pool é responsável pela detecção de todas as condições excepcionais relacionadas com o acesso ao banco de dados, incluindo os erros causados por falhas na rede que conecta os servidores de aplicações e de banco de dados (Figura 36). A sinalização dessas condições excepcionais é feita através das exceções declaradas na nova interface pool.spec.prov.IPoolMgr (Figura 40), que são: SimboloSistemaInvalidoException, ConexaoIndisponivelException, RecursoInvalidoException, FalhaJdbcDriverException, ConexaoNulaException e SqlInvalidoException.

O componente EBTabelas detecta uma única condição excepcional, que é a ausência de produtos cadastrados no sistema. Essa condição é sinalizada através de uma exceção do tipo ProdutosInexistentesException, declarado na interface ebtabelas.spec.prov.IEBTabelasMgr (provida pelo componente EBTabelas).

#### 6.2.4.4. Tratamento das Condições Excepcionais

O componente Pool trata apenas as exceções internas do tipo java.sql.SQLException que possam ser causadas por falhas na rede, efetuando uma segunda tentativa de executar a operação antes de sinalizar uma exceção externa do tipo SqlInvalidoException. O componente SBTabelas trata apenas as exceções do tipo ProdutosInexistentesException, enviando para o usuário uma tela apropriada, informando que não existe nenhum produto cadastrado. Exceções de outros tipos são tratadas

pela camada de apresentação. Na implementação do componente PortalWeb, esse tratamento consiste apenas em exibir a mensagem com a descrição da exceção, o que foi mantido no presente estudo.

Como os tipos de exceções lançadas pelo componente Pool e declaradas na interface pool.spec.prov.IPoolMgr não são tratadas pelo componente EBTabelas, é necessária a adaptação entre a interfaces IPoolMgr provida pelo componente Pool e a interface IPoolMgr requerida pelo componente EBTabelas. Essa adaptação é responsabilidade do conector baebtabelasfwpool, que converte as exceções dos tipos declarados na interface provida do componente Pool em exceções do tipo RecoveredFailureException. Essas exceções são propagadas pelos componentes das camadas de Entidades e Sistemas até o componente PortalWeb, da camada de Apresentação.

### 6.2.5. Alterações no Código do Sistema

As Figuras 41 e 42 ilustram as alterações efetuadas no código dos componentes mostrando, respectivamente, o código original e o novo código do método listaProdutos(), do componente EBTabelas. No código original havia um tratador de exceções, do tipo mais genérico Exception, que foi eliminado.

O tratamento excepcional de caráter mais genérico foi transferido para os conectores, passando a ser feito de maneira sistemática como descrito na Seção 4.3.6. A Figura 43 ilustra a adaptação das interfaces excepcionais dos componentes, através da nova implementação do método acquireConnection() do conector baebtabelasfwpool.

```
1   public ITORetorno listaProdutos
2          (ObjectFactory objectFactory, IPoolMgr facade) {
3    try {ITORetorno retorno=facade.acquireConnection
4               (objectFactory.getITOTransacaoPool("BA"));
5         if (retorno.getTipo() != ITORetorno.SUCESSO) {return retorno;}
6         Connection con = (Connection) retorno.getConteudo();
7         try {PreparedStatement p = con.prepareStatement(SQLTODOS);
8              ResultSet rslt = p.executeQuery();
9              ArrayList listaProdutos = new ArrayList();
10             while (rslt.next()) {
11              retorno = preencher(objectFactory, rslt);
12              ITOProdutos produto = (ITOProdutos) retorno.getConteudo();
13              listaProdutos.add(produto);
14             }
15             if (listaProdutos.size() > 0) {
16               return objectFactory.getITORetorno  (ITORetorno.SUCESSO,
17               (ITOProdutos[])listaProdutos.toArray( new ITOProdutos[0]));
18             } else {
19               return objectFactory.getITORetorno(ITORetorno.ERRONEGOCIO,
20                new Exception(IGenericBAException.PRODUTOS_INEXISTENTES));
21             }
22        } finally {   facade.releaseConnection(con);
23        }
24   } catch (Exception e) {
25        return objectFactory.getITORetorno(ITORetorno.ERROSISTEMA, e);
26   }
27  }
```

**Fig. 41.** Versão inicial do método listaProdutos() do componente EBTabelas.

```
1   public ITOProdutos[] listaProdutos
2          (ObjectFactory objectFactory, IPoolEBMgr facade) {
3    boolean Conectado = false;
4    ITOConnection con = null;
5    try {con = facade.acquireConnection
6                 (objectFactory.getITOTransacaoPool("BA"));
7         Conectado = true;
8         ITOPreparedStatement ps = facade.prepareStatement(con,SQLTODOS);
9         ITOResultSet rs = facade.executeQuery(ps);
10        ArrayList listaProdutos = new ArrayList();
11        while (facade.next(rs)) {
12         listaProdutos.add(preencherITOProduto(objectFactory, rs, facade));
13        }
14        if (listaProdutos.size() > 0) {
15          return (ITOProdutos[]) listaProdutos.toArray(new ITOProdutos[0]);
16        } else {
17          throw new ProdutosInexistentesException
18                      (IGenericBAException.PRODUTOS_INEXISTENTES);
19        }
20   } finally { if (Conectado) {
21                 facade.releaseConnection(con);
22             }
23   }
24  }
```

**Fig. 42.** Nova versão do método listaProdutos() do componente EBTabelas.

```
1   public ITOConnection acquireConnection(ITOTransacaoPool dadosTransacao) {
2    try { return poolMgr().acquireConnection(dadosTransacao);
3        } catch (UndeclaredException e) {
4          throw e;
5        } catch (RuntimeException e) {
6          throw new RecoveredFailureException("acquireConnection()", e);
7        } catch (SimboloSistemaInvalidoException e) {
8          throw new RecoveredFailureException("acquireConnection()", e);
9        } catch (ConexaoIndisponivelException e) {
10         throw new RecoveredFailureException("acquireConnection()", e);
11       } catch (FalhaJdbcDriverException e) {
12         throw new RecoveredFailureException("acquireConnection()", e);
13       } catch (RecursoInvalidoException e) {
14         throw new RecoveredFailureException("acquireConnection()", e);
15       }
16  }
```

**Fig. 43.** Nova versão do conector baebtabelasfwpool.

### 6.2.6. Testes do Comportamento Excepcional do Sistema

As duas versões do sistema, a versão inicial e a nova versão produzida durante o estudo de caso, foram submetidas a uma mesma seqüência de testes onde foram injetadas falhas dos seis tipos especificados no modelo de falhas. A Tabela 6 apresenta os casos de teste incluídos nessa seqüência. Em todos os casos, o resultado observado foi a mensagem descritiva da condição excepcional, exibida pelo componente PortalWeb. O critério de aceitação do teste consistiu em verificar se a mensagem exibida continha informações suficientes para permitir o diagnóstico da condição excepcional ocorrida e a semântica do defeito causado por essa condição.

No caso de teste 1.1, por exemplo, foram obtidos os seguintes resultados:

Mensagem exibida pela versão inicial:

```
Exceção: java.lang.ArithmeticException
Mensagem: / by zero
Linha: br.com.autbank.ba.eb.tabelas.impl.DBProdutos.listaProdutos
           (DBProdutos.java:95)
```

Mensagem exibida pela nova versão:

```
Exceção: br.com.autbank.ba.co.basbtabelasbaebtabelasconsulta.
           impl.Facade$RecoveredFailureException
Mensagem: br.com.autbank.ba.co.baebtabelasfwpoolteste.impl::
           listaProdutos() mensagem de erro original
           java.lang.ArithmeticException: / by zero
Linha: br.com.autbank.ba.co.basbtabelasbaebtabelasconsulta.impl.
           Facade.listaProdutos(Facade.java:212)
```

A principal diferença entre esses dois resultados é o tipo da exceção, que é exibida na primeira linha da mensagem. Essa informação de tipo é obtida da exceção recebida pelo componente

PortalWeb, no caso de insucesso de uma operação requisitada do sistema. Na versão inicial é recebida uma exceção do tipo java.lang.ArithmeticException, que não fornece qualquer informação a respeito da semântica do defeito. Na nova versão, é recebida uma exceção do tipo RecoveredFailureException indicando, de acordo com a semântica de defeito definida pela hierarquia de tipos de exceções (Seção 4.3.1), que nenhum efeito foi produzido sobre o estado do sistema ou no seu ambiente.

Resultados semelhantes foram obtidos nos demais casos de teste, com exceção dos casos 5.1, 5.2 e 6.1. Nesses três casos de teste, o comportamento da nova versão foi idêntico ao da versão inicial. A análise posterior desses resultados permitiu concluir que, nesses três casos de teste, a execução do sistema é interrompida por uma exceção lançada num momento anterior à chamada, pelo componente PortalWeb, da operação listaProdutos(), na qual o estudo de caso se concentrou.

**Tabela 6.** Casos de Testes de Condições Excepcionais.

| Caso de Teste | Descrição |
|---|---|
| *Grupo 1* | *Falhas de projeto no software do Sistema BA* |
| 1.1 | Exceção interna não prevista pelo componente EBTabelas.<br>Incluido comando `'int i = 1/0;'` na primeira linha do método listaProdutos(). |
| 1.2 | Exceção interna não prevista pelo componente SBTabelas.<br>Incluido comando `'int i = 1/0;'` na primeira linha do método listaProdutos() do |
| 1.3 | Exceção interna não prevista pelo componente Pool.<br>Incluido comando `'int i = 1/0;'` na primeira linha do método acquireConnection(). |
| *Grupo 2* | *Falhas no Banco de Dados* |
| 2.1 | Banco de dados incorreto.<br>Componente Pool emite requisição acquireConnection() fornecendo nome de sistema não registrado no B.D. |
| 2.2 | Esquema do B.D. incorreto.<br>Componente Pool emite requisição executeQuery() fornecendo nome de tabela não registrado no B.D. |
| *Grupo 3* | *Falha no driver JDBC* |
| 3.1 | Conexão liberada indevidamente.<br>Componente Pool libera uma conexão com um releaseConnection() imediatamente após o acquireConnection(). |
| *Grupo 4* | *Falha no arquivo de configuração do Banco de Dados* |
| 4.1 | B.D. incompatível com arquivos de configuração xml.<br>Alterar codigo do sistema no autbank.xml sem alterar web.xml |
| 4.2 | B.D. incompatível com arquivos de configuração xml.<br>Alterar codigo do sistema na autbank.xml e no web.xml |
| *Grupo 5* | *Falha na administração de Banco de Dados* |
| 5.1 | Usuário não cadastrado no B.D. |
| 5.2 | Senha do usuário inválida para o B.D. |
| *Grupo 6* | *Falha na rede* |
| 6.1 | Indisponibilidade da rede.<br>Desconectar a rede antes de iniciar o caso de uso |
| 6.2 | Indisponibilidade da rede.<br>Desconectar a rede no momento da chamada do método listaProdutos() do componente SBTabelas |

# Capítulo 7

# Conclusões e Trabalhos Futuros

*Nós só sabemos precisamente quando sabemos pouco;*

*com o conhecimento as dúvidas aumentam.*

*Johann Wolfgang von Goethe*

## 7.1. Objetivos e Contribuições Principais

Sistemas de automação bancária e de comércio eletrônico são exemplos de uma nova geração de sistemas computacionais, intensivos em software, e de cujo funcionamento correto dependem, cada vez mais, as organizações e indivíduos da nossa sociedade atual. As técnicas empregadas no desenvolvimento desses sistemas, aqui denominados sistemas críticos "modernos", concentram-se quase que exclusivamente ao atendimento dos seus requisitos funcionais. Os requisitos não-funcionais desses sistemas críticos "modernos", que incluem a confiabilidade, ainda são tratados de maneira não-sistemática, como um subproduto do processo de desenvolvimento. Como resultado, a confiabilidade desses sistemas é, em geral, insatisfatória.

Para abordarmos adequadamente a questão da confiabilidade dos sistemas críticos "modernos" devemos, necessariamente, considerar duas importantes características: a complexidade do sistema de software e a intensa reutilização de componentes de software de prateleira, como forma de reduzir custos e prazos de desenvolvimento. Essas características limitam severamente: (i) a confiabilidade que o integrador do sistema pode atribuir aos componentes de software integrados ao sistema; (ii) o acesso, pelo integrador do sistema, ao projeto interno e ao código fonte da implementação de componentes do tipo caixa-preta; e (ii) a prevenção de falhas nos sistemas assim construídos. Tolerância a falhas torna-se, nesse contexto, um meio essencial para se obter confiabilidade. As restrições impostas pela utilização de componentes do tipo caixa-preta

e a constante evolução dos componentes do sistema, de uma forma geral, limitam o grau de confiabilidade que se pode atribuir aos componentes do sistema, individualmente. Dessa forma, a tolerância a falhas deve ser provida pela arquitetura do software. O foco do presente trabalho é, portanto, o desenvolvimento de arquiteturas de software tolerantes a falhas e baseadas em componentes.

Os principais resultados desse trabalho são: (i) componente Ideal C2 Tolerante a Falhas (iC2C), que é uma abordagem para estruturação de sistemas baseados em componentes arquiteturais ideais tolerantes a falhas, utilizando o estilo arquitetural C2; (ii) componente COTS Ideal Tolerante a Falhas (iCOTS), que é uma solução arquitetural para transformar componentes de software comerciais de prateleira (COTS) em componentes ideais tolerantes a falhas, adicionando aos mesmos invólucros protetores[42]; (iii) uma estratégia geral para tratamento de exceções em sistemas confiáveis baseados em componentes de software; e (iv) proposta de um ambiente integrado de desenvolvimento de sistemas de software confiáveis, baseado em componentes.

## 7.2. Lista de Publicações

Os resultados dessa pesquisa foram objeto das seguintes publicações:

1. Structuring Exception Handling for Dependable Component-Based Software Systems. Artigo aceito para o *Workshop on Component Models for Dependable Systems (part of the EUROMICRO 2004 Conference)*. Rennes, France. A ser realizado em 09/2004.
2. A Dependable Architecture for COTS-Based Software Systems using Protective Wrappers. Capítulo do livro *Architecting Dependable Systems Vol. 2*. (Editado por Rogério de Lemos, Cristina Gacek e Alexander Romanovsky). No prelo.
3. Desenvolvimento Baseado em Componentes e Confiabilidade. Capítulo do livro *Desenvolvimento Baseado em Componentes*. Editora da Universidade Estadual de Maringá. No prelo.

[42] do inglês, *protective wrappers*.

4. Especificação Formal do Componente C2 Ideal Tolerante a Falhas. Relatório Técnico do Instituto de Computação, Unicamp. A ser publicado.
5. A Fault-Tolerant Software Architecture for Component-Based Software Systems. Capítulo do livro *Architecting Dependable Systems*. (Editado por Rogério de Lemos, Cristina Gacek e Alexander Romanovsky). 2003.
6. A Fault-Tolerant Software Architecture for COTS-based Software Systems. Poster apresentado na *Joint Conferences 9th European Software Engineering Conference and 11th ACM SIGSOFT Symposium on Foundations of Software Engineering (ESEC/FSE'2003)*. Helsinki, Finland. 09/2003.
7. Integrating COTS Software Components into Dependable Software Architectures. Artigo curto apresentado na *6th IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC'2003)*. Hokkaido, JAPAO. 05/2003.
8. An Idealized Fault-Tolerant Architectural Component. Artigo apresentado no *ICSE 2002 Workshop on Architecting Dependable Systems (24th ACM International Conference on Software Engineering)*. Orlando, USA. Maio, 2002.
9. *Integrating COTS Software Components Into Dependable Software Architectures*. Relatório Técnico do Department of Computing Science, University of Newcastle upon Tyne (UK). 2002.
10. *An Architectural Approach to Fault-Tolerant Component Composition based on Exception Handling*. Co-autoria em relatório técnico do Instituto de Computação, UNICAMP. 2004.
11. An Architectural-level Exception Handling System for Component-based Applications. Co-autoria em artigo apresentado no *LADC 2003 - 1st Latin-American Symposium on Dependable Computing*. São Paulo, Brasil. 10/2003.
12. A Java Component Model for Evolving Software Systems. Co-autoria em poster apresentado na *18th IEEE International Symposium on Automated Software Engineering*, 10/2003. Montreal, CANADA.

13. Component-Based Development Using Model Driven Architecture. Co-autoria em artigo apresentado no *I3E2003 Research Coloquium (realizado em conjunto com o 3rd IFIP Conference on E-Commerce, E-business, and E-government)*. São Paulo, Brasil. 09/2003.
14. Component Integration Using Composition Contracts with Exception Handling. Co-autoria em artigo apresentado no *Workshop on Exception Handling in Object-Oriented Systems - ECOOP'2003*. Darmstad, Alemanha. 07/2003.
15. FaTC2: An Object-Oriented Framework for Developing Fault-Tolerant Component-based Systems. Co-autoria em artigo apresentado no *ICSE 2003 Workshop on Software Architectures for Dependable Systems*. Portland, USA. 05/2003.

### 7.3. Dificuldades Encontradas

O presente trabalho se desenvolveu na confluência de três grandes áreas de pesquisa, que são: tolerância a falhas, desenvolvimento de software baseado em componentes (DBC) e arquiteturas de software. A área de tolerância a falhas consolida décadas de pesquisas e aplicações práticas bem sucedidas. DBC e arquiteturas de software, ao contrário, são áreas de pesquisa em franco desenvolvimento, com diversos problemas fundamentais ainda em aberto. A especificação de interfaces em suas múltiplas dimensões [8] é um exemplo desses problemas. Tanto as linguagens de definição de interfaces (IDL[43]) como as linguagens de descrição de arquiteturas (ADL[44]) disponíveis atualmente limitam-se à especificação sintática das operações. Os aspectos semânticos, de sincronização e de qualidade de serviço das interfaces são simplesmente ignorados ou tratados de maneira excessivamente simplificada.

Desenvolvimento baseado em componentes (DBC) e arquitetura de software são áreas complementares que, em conjunto, permitem aplicarmos ao desenvolvimento de software uma estratégia de "divisão e conquista". O DBC nos ajuda a dividir o problema em partes menores, mais fáceis de desenvolver, administrar e reutilizar. A arquitetura de software, por sua vez, nos

[43] do inglês, *Interface Definition Language*.
[44] do inglês, *Architecture Description Language*.

ajuda a integrar diversas partes num sistema com um propósito específico. Apesar disso, as duas áreas evoluem de forma pouco coordenada. A tecnologia de DBC evolui rapidamente e impulsionada principalmente pela indústria, como é o caso das infra-estruturas de componentes J2EE e .NET. Por outro lado, inexiste uma tecnologia de arquitetura de software propriamente dita, de ampla aceitação. Arquitetura de software ainda permanece, em grande parte, restrita à pesquisa acadêmica. Esse descompasso dificulta a integração dessas duas áreas, que deveria ser natural.

Uma outra dificuldade, que afeta tanto o trabalho em DBC como em arquitetura de software, é a distância entre os conceitos introduzidos por esses novos paradigmas e as abstrações das linguagens de programação existentes. Linguagens orientadas a objetos, por exemplo, em geral não incluem abstrações para conceitos como componentes e conectores. Isso torna necessário, no desenvolvimento do software, um mapeamento explícito das abstrações utilizadas no projeto arquitetural, ou de componentes, para as abstrações disponíveis na linguagem, como classes e objetos, por exemplo. Esse mapeamento, quando feito de maneira não sistemática, além de ser mais uma fonte de erros dificulta enormemente a manutenção do sistema em conformidade com sua arquitetura. Parte do presente trabalho de pesquisa foi, assim, a definição de um modelo de componentes, independente de tecnologia, e o seu mapeamento para linguagens orientadas a objetos atuais [87][88].

## 7.4. Limitações da Pesquisa

Os resultados obtidos foram aplicados em estudos de casos que servem como indicativos da efetividade das soluções propostas. Esses estudos de casos incluíram o desenvolvimento, por empresas privadas com equipes independentes, de sistemas para as áreas de automação bancária e bioinformática, com resultados considerados satisfatórios pelos seus dirigentes. A principal limitação do presente trabalho é a ausência de experimentos práticos, em condições controladas, que permitam aferir com precisão a eficácia das soluções propostas e o seu impacto nos custo e prazos de desenvolvimento, aqui considerados essenciais.

Duas das soluções propostas, que são o iC2C e o iCOTS, são baseados no estilo arquitetural C2. Essa decisão de projeto foi motivada pela independência entre os componentes do sistema e facilidade de integração de componentes heterogêneos, que são característica desse estilo arquitetural. No entanto, essa flexibilidade é obtida à custa de regras bastante restritivas para comunicação entre os componentes do sistema, o que se mostrou inadequado a sistemas reais de maior complexidade. Embora pareça evidente que esses resultados possam ser generalizadas para outros estilos arquiteturais menos restritivos, como o estilo cliente/servidor, essa tarefa é deixada como trabalho futuro.

O presente trabalho considera apenas aspectos de projeto, ao nível arquitetural, dos mecanismos de tolerância a falhas e sua implementação utilizando uma linguagem de programação orientada a objetos. Não foram tratadas as questões relacionados com a especificação dos requisitos de tolerância a falhas, que antecede ao projeto daqueles mecanismos. Essa importante questão é abordada em [29].

O presente trabalho não considera restrições de tempo ou de recursos, tais como memória e processador. As soluções propostas não são, portanto, eficientes do ponto de vista dos recursos computacionais. Por esse motivo, essas soluções não se aplicam a sistemas de tempo-real ou sistemas embarcados.

## 7.5. Trabalhos Futuros

Mais do que resultados finais, o presente trabalho aponta para lacunas importantes na pesquisa de técnicas, métodos e ferramentas para construção de sistemas confiáveis baseados em componentes de software. Dando continuidade ao tema desse trabalho, duas dissertações de mestrado e uma tese de doutorado se iniciaram recentemente no Instituto de Computação. Os temas dessas pesquisas já em andamento são: (i) projeto detalhado e implementação do ambiente integrado de desenvolvimento proposto, como uma extensão do ambiente integrado de desenvolvimento Eclipse [26]; (ii) extensão do método de desenvolvimento UML Components para incorporação das atividades de especificação, projeto e implementação de mecanismos de

tolerância a falhas; (iii) projeto de uma arquitetura de software que visa garantir propriedades relativas à confiabilidade de sistemas construídos a partir da composição de componentes COTS não-confiáveis.

Outros temas de pesquisa propostos, a serem abordados em trabalhos futuros, são: (i) transposição das propostas do iC2C e iCOTS para o estilo arquitetural cliente/servidor; (ii) separação de interesses, no nível arquitetural, para permitir a inclusão dos mecanismos de tolerância a falhas num sistema como um "aspecto" da arquitetura de software; (iii) ferramentas para análise da confiabilidade de uma arquitetura de software com base em diagramas de colaboração entre seus componentes, possibilitando a determinação de componentes críticos para a aplicação e auxiliando no projeto dos mecanismos de tolerância a falhas; e (iv) realização de experimentos práticos que permitam aferir quantitativamente o impacto das técnicas propostas na qualidade final do sistemas desenvolvidos.

# Referências Bibliográficas


[1] AMBRIOLA, Vincenzo, CONRADI, Reidar, FUGGETA, Alfonso. Assessing Process-Centered Software Engineering Environments. In: *ACM Transactions on Software Engineering and Methodology*, v.6, n.3, pp.283-328, Julho 1997.

[2] ANDERSON, T., LEE, P. A. *Fault Tolerance: Principles and Practice*. Prentice-Hall, 1981.

[3] ANDERSON, T., FENG, M., RIDDLE, S., ROMANOVSKY, A. Protective Wrapper Development: A Case Study. In: *Proc. 2nd Int. Conference on COTS-Based Software Systems (ICCBSS 2003), Lecture Notes in Computer Science Volume 2580.* Springer-Verlag. 2003. pp.1-14.

[4] ATKINSON, C. et. al. *Component-Based Produt Line Engineering with UML*. Addison-Wesley, 2001.

[5] BACHMANN, Felix et. al. *Volume II: Technical Concepts of Component-Based Software Engineering*. Relatório Técnico CMU/SEI-2000-TR-008. Software Engineering Institute, Carnegie Mellon. Abril, 2000.

[6] BAELEN, Stefan Van et. al. Toward a Unified Terminology for Component-Based Development. In: *Proc. WCOP Workshop, ECOOP 2000. Cannes, France.* 2000.

[7] BARBACCI, Mario R. et. al. *Steps in Architeture Tradeoff Analysis Method: Quality Attribute Models and Analysis*. Relatório técnico CMU/SEI-97-TR-029. Software Engineering Institute, Carnegie Mellon. Maio, 1998.

[8] BASS, Len, KAZMAN, Rick. Architecture-Based Development. Relatório técnico CMU/SEI-99-TR-007. Software Engineering Institute, Carnegie Mellon. Abril, 1999.

[9] BEUGNARD, Antoine, JEZEQUEL, Jean-Marc, PLOUZEAU, Noël, WATKINS, Damien. Making Components Contract Aware. In: *IEEE Computer*. Julho, 1999. pp.38-45.

[10] BRITO, Patrick Henrique da Silva. *Um Método para Tratamento de Exceções em Desenvolvimento Baseado em Componentes*. Proposta de Mestrado. Instituto de Computação, Unicamp. 2004. Em elaboração.

[11] BROWN, A. W. and WALLNAU, K. C. The current state of CBSE. *IEEE Software*, v.15, n.5, pp.37-46, Setembro/Outubro 1998.

[12] BUSCHMANN, F., MEUNIER, R., ROHNERT, H., SOMMERLAD, P., STAL, M. *Pattern-Oriented Software Architecture - A System of Patterns*. John Wiley & Sons, 1996.

[13] BUZATO, Luiz E., RUBIRA, Cecília M.F. *Construção de Sistemas Orientados a Objetos Confiáveis*. UFRJ. 1998.

[14] CASTOR FILHO, Fernando José, GUERRA, Paulo Asterio de Castro, FISCHER RUBIRA, Cecília Mary. An Architectural-level Exception Handling System for Component-based Applications. In: *LADC 2003 - 1st Latin-American Symposium on Dependable Computing*. São Paulo, SP, BRASIL. LNCS 2847, Springer-Verlag, Outubro 2003. pp.321-340.

[15] CASTOR FILHO, Fernando José, GUERRA, Paulo Asterio de Castro, FISCHER RUBIRA, Cecília Mary, FaTC2: An Object-Oriented Framework for Developing Fault-Tolerant Component-based Systems. In: *ICSE 2003 Workshop on Software Architectures for Dependable Systems*, 05/2003. Portland, Oregon, EUA. Maio, 2003. pp.13-18,

[16] CASTOR FILHO, Fernando José. Proposta de Doutorado. Instituto de Computação, UNICAMP. Em elaboração.

[17] CHEESMAN, J., DANIELS, J. *UML Components: A Simple Process for Specifying Component-Based Software*. Addison-Wesley, 2000.

[18] CLEMENTS, Paul C., NOTHROP, Linda M. *Software Architecture: An Executive Overview*. Relatório técnico CMU/SEI-96-TR-003. Fevereiro, 1996.

[19] COOK, Jonhathan E., DAGE, Jerey A. Dage. Highly reliable upgrading of components. In: *21st International Conference on Software Engineering - ICSE'99*, New York, NY, May 1999. ACM Press. pp.203-212.

[20] CRISTIAN, Flaviu. Exception Handling. In: ANDERSTON, T. (Ed.) *Dependability of Resilient Computers*. BSP Professional Books, UK. 1989. pp.68-97.

[21] DASHOFY, Eric M., HOEK, André Van der, TAYLOR, Richard N. A Highly-Extensible, XML-Based Architecture Description Language. In: *Proceedings of the Working IEEE/IFIP Conference on Software Architectures (WICSA 2001)*. 2001.

[22] DE LEMOS, Rogério. Describing Evolving Dependable Systems Using Co-Operative Software Architectures. In: *Proc. IEEE International Conference on Software Maintenance (ICSM'01)*. 2001. pp.320-329.

[23] DE LEMOS, Rogério, GACEK, Cristina, ROMANOVSKY, Alexander (Eds.). *Architecting Dependable Systems*. Lecture Notes in Computer Science. v. 2677. Springer-Verlag. 2003.

[24] DELLAROCAS, Chrysanthos. Toward Exception Handling Infrastructures for Component-Based Software. In: *1998 International Workshop on Component-Based Software Engineering*. 1998.

[25] D'SOUZA, D., WILLS, A. C. *Objects, Components and Frameworks with UML: The Catalysis Approach*. Addison-Wesley, 1999.

[26] ECLIPSE FOUNDATION. *Eclipse.org Main Page*. Disponível em http://www.eclipse.org (acessada em 28/05/04).

[27] EIFFEL SOFTWARE INC. *Eiffel Software - The Home of EiffelStudio and Eiffel ENVisioN!*. Disponível em http://www.eiffel.com (acessada em 28/05/04).

[28] FERREIRA, Gisele Rodrigues de Mesquita, RUBIRA, Cecília Mary Fischer, De LEMOS, Rogerio. In: *Proc. Of the 6th IEEE International High-Assurance Systems Engineering Symposium (HASE 2001)*. 2001.

[29] FERREIRA, Gisele Rodrigues de Mesquita. *Tratamento de Exceções no Desenvolvimento de Sistemas Confiáveis Baseados em Componentes*. Dissertação de Mestrado. Instituto de Computação, UNICAMP. 2001.

[30] FOX, Greg, LANTNER, Karen, MARCOM, Steven. A Software Development Process for COTS-Based Information System Infrastructure. In: *Proc. Fifth International Symposium on Assessment of Software Tools and Technologies*. Junho, 1997. pp.133-142.

[31] GAMMA, E., HELM, R., JOHNSON, R., VLISSIDES, J. *Design Patterns*. Addison-Wesley, 1995.

[32] GARLAN, David, ALLEN, Robert, OCKERBLOOM, John. Architectural Mismatch: Why Reuse Is So Hard. In: *IEEE Software*. v.12, n.6, pp.17-26. Novembro, 1995.

[33] GARLAN, Garlan, MONROE, Robert T., WILE, David. Acme: Architectural Description of Component-Based Systems. In: LEAVENS, G.T., SITARAMAN, M. (Eds.) *Foundations of Component Based Systems*. Cambridge University Press, Cambridge, UK. 2000. Cap. 3, pp.47-67.

[34] GARTINER, Felix C. Fundamentals of Fault-Tolerant Distributed Computing in Asynchronous Environments. In: *ACM Computing Surveys*, v.31, n.1. Março, 1999. pp.1-26.

[35] GIBSON, David S., WEIDE, Bruce W., PIKE, Scott M., EDWARDS, Stephen H. Toward a normative theory for component-based system design and analysis. In: LEAVENS, G.T., SITARAMAN, M. (Eds.) *Foundations of Component Based Systems*. Cambridge University Press, Cambridge, UK. 2000. Cap. 10, pp.211-230.

[36] GUERRA, Paulo Asterio de Castro, RUBIRA, Cecília Mary Fischer, ROMANOVSKY, Alexander, DE LEMOS, Rogério *Integrating COTS Software Components Into Dependable Software Architectures*. Relatório Técnico CS-TR 780. Department of Computing Science, University of Newcastle upon Tyne (UK). 2002. Disponível em http://www.cs.ncl.ac.uk/research/pubs/trs/papers/780.pdf (acessada em 28/05/04).

[37] GUERRA, Paulo Asterio de Castro, RUBIRA, Cecília Mary Fischer, DE LEMOS, Rogério. An Idealized Fault-Tolerant Architectural Component. In: *ICSE 2002 Workshop on Architecting Dependable Systems, (24th ACM International Conference on Software Engineering)*. Florida, ESTADOS UNIDOS, 2002. pp.15-19. Disponível em http://www.cs.kent.ac.uk/events/conf/2002/wads/Proceedings/guerra.pdf (acessada em 28/05/04).

[38] GUERRA, Paulo Asterio de Castro, RUBIRA, Cecília Mary Fischer, DE LEMOS, Rogério. A Fault-Tolerant Software Architecture for Component-Based Software Systems. In: *Architecting Dependable Systems*. Lecture Notes in Computer Science. Vol. 2677. 2003.

[39] GUERRA, Paulo Asterio de Castro, DE LEMOS, Rogério, ROMANOVSKY, Alexander, FISCHER RUBIRA, Cecília Mary, A Fault-Tolerant Software Architecture for COTS-based Software Systems. In: *Joint Conferences 9th European Software Engineering Conference and 11th ACM SIGSOFT Symposium on Foundations of Software Engineering (ESEC/FSE'2003)*, 09/2003. Helsinki, FINLANDIA. pp.375-378,

[40] GUERRA, Paulo Asterio de Castro, ROMANOVSKY, Alexander, DE LEMOS, Rogério, RUBIRA, Cecília Mary Fischer. Integrating COTS Software Components into Dependable Software Architectures. In: *6th IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC'2003)*, 05/2003. Hokkaido, JAPAO, 2003. pp.139-142.

[41] GUERRA, Paulo Asterio de Castro, ROMANOVSKY, Alexander, DE LEMOS, Rogério, FISCHER RUBIRA, Cecília Mary. A Dependable Architecture for COTS-Based Software Systems using Protective Wrappers. In: *Architecting Dependable Systems Vol. 2*. No prelo.

[42] GUERRA, Paulo Asterio de Castro, CASTOR FILHO, Fernando, Castor Filho, PAGANO, Vinicius, RUBIRA, Cecília Mary Fischer. Structuring Exception Handling for Dependable Component-Based Software Systems. In: Proc. *Workshop on Component Models for Dependable Systems (part of the EUROMICRO 2004 Conference)*. 2004. A ser publicado.

[43] GUERRAOUI, Rachid, SCHIPER, André. Software-Based Replication for Fault Tolerance. *IEEE Computer*, v.30, n.4, April 1997. pp.68-74.

[44] HAMLET, Dick, MASON, Dave, WOIT, Denise. Theory of system reliability based on components. In: *2000 International Workshop on Component-Based Software Engineering*. Carnegie Mellon Software Engineering Institute, 2000.

[45] IBM CORPORATION. *IBM WebSphere - e-business on demand, middleware, application server, portal, business*. Disponível em http://www.ibm.com/websphere (acessada em 28/05/04).

[46] ISSARNY, Valérie, BANATRE, Jean-Pierre. Architecture-Based Exception Handling. In: *Proc. 34th Annual Hawaii International Conference on System Sciences (HICSS'34)*. 2001.

[47] JACOBSON, Ivar, BOOCH, Grady, RUMBAUGH, James. *The Unified Software Development Process*. Addison-Wesley, 1999.

[48] JBOSS INC. *JBoss 4.0 Vision*. Disponível em http://www.jboss.org/products/jbossas (acessada em 28/05/04).

[49] KAÂNICHE, Mohamed, LAPRIE, Jean-Claude, BLANQUART, Jean-Paul Blanquart. Dependability Engineering of Complex Computing Systems. In: *Proc. 6th IEEE International Conference on Complex Computer Systems (ICECCS´00).* 2000.

[50] KOOPMAN, Philip, DE VALE, John. The exception handling effectiveness of POSIX operating systems. *IEEE Transactions on Software Engineering*. v.26, n.9. Setembro, 2000.

[51] KRUCHTEN, P. The 4+1 View Model of Software Architecture. In: *IEEE Software*. Novembro, 1995. pp.42-50.

[52] LAPRIE, Jean-Claude. Dependable Computing and Fault Tolerance: Concepts and Terminology. In: *Resilient Computing Systems*. Collins and Wiley, 1987. pp. 2-11.

[53] LEDECZI A., MAROTI M., BAKAY A., KARSAI G., GARRETT J., THOMASON IV C., NORDSTROM G., SPRINKLE J., VOLGYESI P.: The Generic Modeling Environment, *Workshop on Intelligent Signal Processing*, Budapest, Hungary, May, 2001. Disponível em http://www.isis.vanderbilt.edu/publications/archive/Ledeczi_A_5_17_2001_The_Generi.pdf (acessada em 28/05/04).

[54] LEE, P. A., ANDERSON, T. *Fault Tolerance: Principles and Practice*. 2.ed. Springer-Verlag. 1990.

[55] LIONS, Jacques-Louis. *ARIANE 5 501 failure report by the inquiry board*. Relatório técnico, CNES, 1996.

[56] MATHWORKS. *Using Simulink: reference guide*. Disponível em http://www.mathworks.com/access/helpdesk/help/pdf_doc/simulink/sl_using.pdf (acessada em 28/05/04).

[57] McILROY, M.D. Mass-Produced Software Components. In: *Software Engineering*. Petrocelli/Charter, Brussels, Belgium. 1976. pp.88-98.

[58] MEDVIDOVIC, Nenad, ROSENBLUM, David S., TAYLOR, Richard N. A language and environment for architecture-based software development and evolution. In: *21st International Conference on Software Engineering - ICSE'99*, New York, NY, May 1999. ACM Press. pp.44-53

[59] MEYER, Bertrand. *Object-Oriented Software Construction*. Prentice-Hall. 1988.

[60] MICROSOFT CORPORATION. *Microsoft .Net Information*. Disponível em http://www.microsoft.com/net (acessada em 28/05/04).

[61] MICROSOFT CORPORATION. *The C# language specification*. Disponível em http://msdn.microsoft.com/library/en-us/csspec/html/CSharpSpecStart.asp (acessada em 28/05/04).

[62] MONDAY, Paul, CAREY, James, DANGLER, Mary. *SanFrancisco Component Framework - An Introduction*. Addison Wesley. 1999.

[63] OBJECT MANAGEMENT GROUP. Model Driven Architecture (MDA). Relatório Técnico ORMSC/2001-07-01. 2001.

[64] OBJECT MANAGEMENT GROUP. Unified Modeling Language: Superstructure Version 2.0. Relatório Técnico ptc/03-07-06. Julho, 2003.

[65] OMONDO. *Eclipse UML plugin*. Disponível em http://www.omondo.com (acessada em 28/05/04).

[66] OPEN SOURCE INITIATIVE. *Common Public License Version 1.0*. Disponível em http://www.opensource.org/licenses/cpl.php (acessada em 28/05/04).

[67] OPENFLOW. *OpenFlow - A Free Software Workflow Management System*. Disponível em http://www.openflow.it/Overview/zopeproduct_html (acessada em 28/05/04).

[68] PAGANO, Vinicius A. *Uma Abordagem Arquitetural com Tratamento de Exceções para Projeto de Sistemas de Software Baseado em Componentes.* Trabalho Final de Mestrado Profissional. A ser publicado.

[69] PLASIL, F., VISNOVSKY, S. Behavior Protocols for Software Components. In: *IEEE Transactions on Software Engineering*, v.28, n.11. 2002. pp.1056-1076.

[70] PRESIDENT'S INFORMATION TECHNOLOGY ADVISORY COMMITEE, USA (PITAC). *Interim Report to the President*. Relatório técnico, National Coordination Office for Computing, Information, and Communications, 1998.

[71] POPOV, P., RIDDLE, S., ROMANOVSKY, A., STRIGINI, L. On Systematic Design of Protectors for Employing OTS Items. In: *27th Euromicro conference*. September, 2001. Warsaw, Poland. pp. 22-29.

[72] RANDELL, Brian, XU, Jie. The Evolution of the Recovery Block Concept. In: *Software Fault Tolerance*. John Wiley & Sons Ltd. 1995.

[73] REIMER, D., SRINIVASAN, H. Analyzing exception usage in large java applications. In: *Proc. ECOOP'2003 -Workshop on Exception Handling for Object-Oriented Systems*, Darmstadt, Germany, Julho 2003. pp.10-19.

[74] ROMANOVSKY, A. Exception Handling in Component-Based System Development. In: *Proc. 15th Int. Computer Software and Application Conference, COMPSAC 2001*. 2001.

[75] ROMANOVSKY, Alexander, DONY, Christophe, KNUDSEN, Jorgen Lindskov, TRIPATHI, Anand. *Proceedings of the ECOOP 2003 Workshop on Exception Handling in Object-Oriented Systems: Towards Emerging Application Areas and New Programming Paradigms*. Relatório Técnico 03-028. University of Minnesota - Computer Science and Engineering. 2003.

[76] RUBIRA, Cecília Mary Fischer, GUERRA, Paulo Asterio de Castro. Desenvolvimento Baseado em Componentes e Confiabilidade. In: *Desenvolvimento Baseado em Componentes*. EDUEM (Editora da Universidade Estadual de Maringá), 2003. pp.10-34.

[77] SEDIGH-ALI, S., GHAFOOR, A., PAUL, R. A. Metrics and Models for Cost and Quality of Component-Based Software. In: *Proc. 6th IEEE International Symposium on Object-Oriented Real-Time Distributed Computing*. Hokkaido, Japão, 2003.

[78] SHA, Lui, RAJKUMAR, R., GAGLIARDI, M. Evolving dependable real-time systems. In: *1996 IEEE Aerospace Applications Conference Proceedings*, Aspen, CO, February, 1996. IEEE New York, NY, USA. pp.335-346.

[79] SHAW, Mary et. al. Abstractions for Software Architecture and Tools to Support Them. In: *IEEE Transactions on Software Engineering*. v.21, n.4, Abril, 1995, pp.314-335.

[80] SHAW, Mary, DeLINE, Robert, ZELESNIK, Gregory. Abstractions and Implementations for Architectural Connections. In: *Third International Conference on Configurable Distributed Systems*. 1996.

[81] SHAW, Mary. Truth vs knowledge: The difference between what a component does and what we know it does. In: *Proc. 8th International Workshop on Software Specification and Design*. Março, 1996.

[82] SHAW, Mary, CLEMENTS, Paul. A Filed Guide to Boxology: Preliminary Classification of Architectural Styles for Software Systems. In: *Proceedings of the COMPSAC97, First International Computer Software and Applications Conference*. Abril, 1997.

[83] SIEDERSLEBEN, J. Errors and exceptions - rights and responsibilities. In: *Proc. ECOOP'2003 - Workshop on Exception Handling for Object-Oriented Systems*, Darmstadt, Alemanha. Julho, 2003. pp.2-9.

[84] SILVA, Ricardo de Mendonça, GUERRA, Paulo Asterio de Castro, RUBIRA, Cecília Mary Fischer. Component Integration Using Composition Contracts with Exception Handling. In: *Workshop on Exception Handling in Object-Oriented Systems - ECOOP'2003*. Darmstadt, Alemanha. Julho, 2003. pp. 1-20.

[85] SILVA, Ricardo de Mendonça. *Técnicas de Estruturação de Software No Desenvolvimento de Sistemas-de-Sistemas Confiáveis*. Dissertação de Mestrado. Instituto de Computação, UNICAMP. 2003.

[86] SILVA, Ricardo de Mendonça, CASTOR FILHO, Fernando José, GUERRA, Paulo Asterio de Castro, FISCHER RUBIRA, Cecília Mary. *An Architectural Approach for Fault-Tolerant Component Composition based on Exception Handling*. Relatório técnico IC-04-02, Instituto de Computação, UNICAMP, 2004.

[87] SILVA JUNIOR, Moacir Caetano, GUERRA, Paulo Asterio de Castro, FISCHER RUBIRA, Cecília Mary. A Java Component Model for Evolving Software Systems. In: *18th IEEE International Symposium on Automated Software Engineering*, 10/2003. Montreal, CANADA. pp.327-330.

[88] SILVA JÚNIOR, Moacir Caetano. *COSMOS - Um Modelo de Estruturação de Componentes para Sistemas Orientados a Objetos*. Disssertação de Mestrado. Instituto de Computação, UNICAMP. 2003.

[89] SOTIROVSKI, Drasko. Towards Fault-Tolerant Software Architectures. In: *Working IEEE/IFIP Conference on Software Architecture*. 2001.

[90] SOUZA, Milton Cesar Fraga, GUERRA, Paulo Asterio de Castro, RUBIRA, Cecília Mary Fischer. Component-Based Development Using Model Driven Architecture. In: *I3E2003 Research Coloquium (realizado em conjunto com o 3rd IFIP Conference on E-Commerce, E-business, and E-government)*. São Paulo, Brasil. Setembro, 2003.

[91] SOUZA, Milton Cesar Fraga. *Um Processo de Desenvolvimento Baseado em Componentes Adaptado ao Model Driven Architecture*. Trabalho Final de Mestrado Profissional. Instituto de Computação, UNICAMP. 2004.

[92] STAVRIDOU, Victoria. Provably Dependable Software Architectures. In: *Proceedings of the 18th Digital Avionics Systems Conference*. 1999.

[93] STROUSTRUP, B. An overview of C++. In: *ACM SIGPLAN Notices*. v.23, n.10. October, 1986. pp. 7-18.

[94] SUN MICROSYSTEM. *Java 2 Platform, Standard Edition (J2SE)*. Disponível em http://java.sun.com/j2se/index.jsp (acessada em 28/05/04).

[95] SUN MICROSYSTEM. *Java 2 Platform, Enterprise Edition (J2EE)*. Disponível em http://java.sun.com/j2ee/index.jsp (acessada em 28/05/04).

[96] SZYPERSKI, Clemens. *Component Software: Beyond Object-Oriented Programming*. Addison-Wesley, 1998.

[97] SZYPERSKI, Clemens. Component Software and the Way Ahead. In: LEAVENS, G.T., SITARAMAN, M. (Eds.) *Foundations of Component Based Systems*. Cambridge University Press, Cambridge, UK. 2000. Cap. 1, pp.1-20.

[98] TAYLOR, R. N. et. al. A component- and message-based architectural style for GUI software. *IEEE Transactions on Software Engineering*, v.22, n.6, pp.390-406, June 1996.

[99] TOMITA, Rodrigo Teruo. *COMBINE: Um Ambiente para Desenvolvimento de Sistemas Baseados em Componentes*. Proposta de Mestrado. Instituto de Computação, Unicamp. 2004. Não publicado.

[100] UCI. ArchStudio 3 - *An Architecture-Based Software Development Environment*. Disponível em http://www.isr.uci.edu/projects/archstudio (acessada em 28/05/04).

[101] VERÍSSIMO, P., LEMOS, R. *Confiança no Funcionamento: Proposta para uma Terminologia em Português.* Publicação conjunta INESC e LCMI/UFSC. 1989.

[102] VITHARANA, Padmal, ZAHEDI, Fatemeh “Mariam”, JAIN, Hemant. Knowledge-Based Repository Scheme for Storing and Retrieving Business Components: A Theoretical Design and an Empirical Analysis. In: *IEEE Transactions on Software Engineering*. v.29, n.7, pp.649-664, Julho 2003.

[103] VOAS, Jeffrey M. The Challenges of Using COTS Software in Component-Based Development. *IEEE Computer*. v.31, n.6, pp.44-45. Junho, 1998.

[104] YPHISE. *Software Assessment Report: UML Modeling of Projects and Information Systems (EXECUTIVE Volume)*. Novembro, 2002. Disponível em http://yphise.fr/Gifs2/uml_exec_us.pdf (acessada em 28/05/04).

[105] ZOPE ORG. *Welcome to Zope.org*. Disponível em http://www.zope.org (acessada em 28/05/04).

# Anexo 1

# Especificação Formal do iC2C

A seguir estão listados os arquivos iC2C.ta e iC2C.q contendo, respectivamente, a especificação formal do iC2C (Seção 3.1) e as propriedades que foram verificadas para esse modelo. Essas especificações foram produzidas através do programa UPPAAL2K versão 3.2.2, de propriedade da Uppsala University e Aalborg University e que pode ser obtido através do endereço da Internet http://www.it.uu.se/research/group/darts (acessado em 28/05/04)).

### Arquivo iC2C.ta

```
1    //Insert declarations of global clocks, variables, constants and
      channels.
2    chan startA, stopA, resetA, abortN, resetN, fe, rtn;
3    int eie_t2i:=0, eie_i2a:=0, efe_t2i:=0, efe_i2a:=0;
4
5    // External signs from (to) Client components to (from) the iC2C
6    chan r_in; // a new request is accepted by the iC2C_bottom connector
7    chan n_out; // a normal notification is sent by the iC2C_bottom connector
8    chan ie_out;  // an interface exception is sent by the iC2C_bottom
      connector
9    chan fe_out;  // a failure exception is sent by the iC2C_bottom connector
10   // External signs from (to) Server components to (from) the iC2C
11   chan r_out; // a request is sent by the iC2C_top connector
12   chan n_in; // a notification arrives at the iC2C_top connector
13   chan ie_in; // an interface exception arrives at the iC2C_top connector
14   chan fe_in; // a failure exception arrives at the iC2C_top connector
15
16   // Requests sent by the iC2C_bottom connector to the iC2C_internal
17   int r_b2i:=0; // a request received from a Client
18   chan Failed, r_sync_b2i; // a control request to synchronize initial
      states
19   // sent after a failure exception is raised by the AbnormalActivity
20
21   // Requests sent by the iC2C_internal connector
22   int r_i2n:=0; // a request to the NormalActivity
23   // it may be a request received from a Client or
24   // from the AbnormalActivity, during error recovery
25   int r_rtn_i2n:=0; // a return to normal message to the NormalActivity
26   // from the AbnormalActivity after a successfull recovery
27   int r_i2t:=0; // a request to the iC2C_top
28   // from the AbnormalAcitivty, during error recovery
29
30   // Notifications sent by the iC2C_internal connector
31   int n_i2b:=0; // a normal notification to the iC2C_bottom
```

```
32   // from the NormalActivity in response of a successfull request
33   int n_i2a:=0; // a notification to the AbnormalActivity
34   // it may be a notification received the NormalActivity or
35   // from a server component, during error recovery
36   int ie_i2b:=0; // an interface exception to the iC2C_bottom
37   // from the NormalActivity in response of a denied request
38   int fe_i2a:=0; // a failure exception to the AbnormalActivity
39   // from the NormalActitivy after an error is detected
40   int ee_i2a:=0; // an external exception to the AbnormalActivity
41   // from the iC2C_top connector
42
43   // Control signs sent by the iC2C_internal connector
44   chan Recover, r_start_i2t; // control request to unblock the iC2C_top
      connector
45   // when the iC2C is entering in abnormal processing mode
46   chan r_reset_i2t; // control request to reset iC2C_top
47   // when the iC2C is returning to normal processing mode
48   chan r_reset_i2n; // control request to reset NormalActivity
49   // when the iC2C is returning to normal processing mode
50
51   // Messages sent by the NormalActivity component
52   int r_n2t:=0; // a request to the iC2C_top
53   // a service request to a Server component
54   int n_n2i:=0; // a normal notification to the iC2C_internal
55   // it may be a normal notifications to be sent to the iC2C Clients or
56   // a normal / exceptional notification to the AbnormalActivity,
57   // during error recovery
58   int ie_n2i:=0; // an interface exception to the iC2C_internal
59   // to be sent to the iC2C Clients
60   int fe_n2i:=0; // a failure exception to the iC2C_internal
61   // to be sent to the AbnormalActivity
62
63   // Control signs sent by the NormalActivity component
64   chan NormalBegin, ReturnToNormal, r_start_n2t;
65   // control request to unblock the iC2C_top connector
66   // when the iC2C is entering in normal processing mode
67   chan NormalResponse, InterfaceException, r_reset_n2t;
68   // a control request to reset iC2C_top
69   // upon completion of the normal processing of a request
70   chan InternalException, r_abort_n2t; // a control request to stop the
      iC2C_top
71   // when an internal exception occurs
72
73   // Messages sent by the AbnormalActivity
74   int r_a2i:=0; // a request to the iC2C_internal
75   int r_rtn_a2i:=0; // a return to normal message to the iC2C_internal
76   int fe_a2b:=0; // a failure exception to the iC2C_bottom
77   // to be sent to the iC2C clients
78   chan RecoveryEnd, r_reset_a2i; // a control request to reset
      iC2C_internal
79   // when error recovery is unsuccessfull
80
81   // Notifications sent by the iC2C_top connector
82   int n_t2n:=0; // a normal notification to the NormalActivity
83   // from a Server component
84   int n_t2i:=0; // a notification to the iC2C_internal
85   // from a Server component, during error recovery
86   int ee_t2i:=0; // an external exception to the iC2C_internal
87   // from a Server component
```

```
88   chan ExternalException, n_abort_t2n; // a control notification to the
      NormalActivity
89   // to switch to abnormal state, after an external exception
90
91   process clients{
92   state S0;
93   init S0;
94   trans S0 -> S0{sync r_in!; },
95   S0 -> S0{sync n_out?; },
96   S0 -> S0{sync fe_out?; },
97   S0 -> S0{sync ie_out?; };
98   }
99   process iC2C_bottom{
100  state S0, S1;
101  init S0;
102  trans S0 -> S1{guard r_b2i<2; sync r_in?; assign r_b2i:=1; },
103  S1 -> S0{guard n_i2b>0; sync n_out!; assign n_i2b:=n_i2b-1; },
104  S1 -> S0{guard ie_i2b>0; sync ie_out!; assign ie_i2b:=ie_i2b-1; },
105  S1 -> S0{guard fe_a2b>0; sync fe_out!; assign fe_a2b:=fe_a2b-1; };
106  }
107  process AbnormalActivity{
108  state S0, S1, S2, S3;
109  init S0;
110  trans S0 -> S1{guard fe_i2a>0; assign fe_i2a:=fe_i2a-1; },
111  S1 -> S2{guard r_a2i<2; assign r_a2i:=r_a2i+1; },
112  S2 -> S1{guard eie_i2a>0; assign eie_i2a:=eie_i2a-1; },
113  S2 -> S1{guard n_i2a>0; assign n_i2a:=n_i2a-1; },
114  S2 -> S1{guard efe_i2a>0; assign efe_i2a:=efe_i2a-1; },
115  S1 -> S0{guard r_rtn_a2i<2; assign r_rtn_a2i:=r_rtn_a2i+1; },
116  S1 -> S3{sync r_reset_a2i!; },
117  S3 -> S0{guard fe_a2b<2; assign fe_a2b:=fe_a2b+1; },
118  S0 -> S1{guard efe_i2a>0; assign efe_i2a:=efe_i2a-1; },
119  S0 -> S1{guard eie_i2a>0; assign eie_i2a:=eie_i2a-1; },
120  S2 -> S1{guard fe_i2a>0; assign fe_i2a:=fe_i2a-1; };
121  }
122  process iC2C_internal{
123  state S3, S4, S1, S0, S2;
124  commit S4, S4;
125  init S0;
126  trans S0 -> S1{guard r_b2i>0, r_i2n<2; assign r_b2i:=r_b2i-
      1,r_i2n:=r_i2n+1; },
127  S1 -> S0{guard n_n2i>0,n_i2b<2; assign n_n2i:=n_n2i-1,
128  n_i2b:=n_i2b+1; },
129  S1 -> S0{guard ie_n2i>0, ie_i2b<2; assign ie_n2i:=ie_n2i-1,
130  ie_i2b:=ie_i2b+1; },
131  S3 -> S1{guard eie_t2i>0,eie_i2a<2; assign eie_t2i:=eie_t2i-1,
132  eie_i2a:=eie_i2a+1; },
133  S1 -> S3{guard r_a2i>0,r_i2t<2; assign r_a2i:=r_a2i-1,
134  r_i2t:=r_i2t+1; },
135  S3 -> S1{guard n_t2i>0,n_i2a<2; assign n_t2i:=n_t2i-1,n_i2a:=n_i2a+1; },
136  S3 -> S1{guard efe_t2i>0,efe_i2a<2; assign efe_t2i:=efe_t2i-1,
137  efe_i2a:=efe_i2a+1; },
138  S2 -> S1{guard fe_n2i>0,fe_i2a<2; assign fe_n2i:=fe_n2i-1,
139  fe_i2a:=fe_i2a+1; },
140  S1 -> S1{guard r_rtn_a2i>0,r_rtn_i2n<2; assign r_rtn_a2i:=r_rtn_a2i-1,
141  r_rtn_i2n:=r_rtn_i2n+1; },
142  S1 -> S4{sync r_reset_a2i?; },
143  S4 -> S0{sync r_reset_i2n!; },
144  S1 -> S2{guard r_a2i>0,r_i2n<2; assign r_a2i:=r_a2i-1,r_i2n:=r_i2n+1; },
145  S2 -> S1{guard n_n2i>0,n_i2a<2; assign n_n2i:=n_n2i-1,n_i2a:=n_i2a+1; },
```

```
146  S1 -> S1{guard fe_n2i>0,fe_i2a<2; assign fe_n2i:=fe_n2i-
      1,fe_i2a:=fe_i2a+1; },
147  S1 -> S1{guard eie_t2i>0,eie_i2a<2; assign eie_t2i:=eie_t2i-
      1,eie_i2a:=eie_i2a+1; },
148  S1 -> S1{guard efe_t2i>0,efe_i2a<2; assign efe_t2i:=efe_t2i-
      1,efe_i2a:=efe_i2a+1; },
149  S2 -> S1{guard eie_t2i>0,eie_i2a<2; assign eie_t2i:=eie_t2i-
      1,eie_i2a:=eie_i2a+1; },
150  S2 -> S1{guard efe_t2i>0,efe_i2a<2; assign efe_t2i:=efe_t2i-
      1,efe_i2a:=efe_i2a+1; };
151  }
152  process NormalActivity{
153  state S0, S1, S2, S3, S4, S5;
154  init S0;
155  trans S0 -> S1{guard r_i2n>0; assign r_i2n:=r_i2n-1; },
156  S1 -> S2{guard r_n2t<2; assign r_n2t:=r_n2t+1; },
157  S2 -> S1{guard n_t2n>0; assign n_t2n:=n_t2n-1; },
158  S1 -> S3{guard fe_n2i<2; assign fe_n2i:=fe_n2i+1; },
159  S3 -> S1{guard r_rtn_i2n>0; assign r_rtn_i2n:=r_rtn_i2n-1; },
160  S3 -> S0{sync r_reset_i2n?; },
161  S1 -> S0{guard n_n2i<2; assign n_n2i:=n_n2i+1; },
162  S1 -> S0{guard ie_n2i<2; assign ie_n2i:=ie_n2i+1; },
163  S3 -> S4{guard r_i2n>0; assign r_i2n:=r_i2n-1; },
164  S4 -> S3{guard n_n2i<2; assign n_n2i:=n_n2i+1; },
165  S4 -> S3{guard fe_n2i<2; assign fe_n2i:=fe_n2i+1; },
166  S4 -> S5{guard r_n2t<2; assign r_n2t:=r_n2t+1; },
167  S5 -> S4{guard n_t2n>0; assign n_t2n:=n_t2n-1; },
168  S5 -> S3{sync n_abort_t2n?; },
169  S2 -> S3{sync n_abort_t2n?; };
170  }
171  process iC2C_top{
172  state S0, S2, S1, S3;
173  commit S3;
174  init S0;
175  trans S0 -> S1{guard r_n2t>0; sync r_out!; assign r_n2t:=r_n2t-1; },
176  S1 -> S0{guard n_t2n<2; sync n_in?; assign n_t2n:=n_t2n+1; },
177  S1 -> S3{guard eie_t2i<2; sync ie_in?; assign eie_t2i:=eie_t2i+1; },
178  S1 -> S3{guard efe_t2i<2; sync fe_in?; assign efe_t2i:=efe_t2i+1; },
179  S0 -> S2{guard r_i2t>0; sync r_out!; assign r_i2t:=r_i2t-1; },
180  S2 -> S0{guard n_t2i<2; sync n_in?; assign n_t2i:=n_t2i+1; },
181  S2 -> S0{guard eie_t2i<2; sync ie_in?; assign eie_t2i:=eie_t2i+1; },
182  S2 -> S0{guard efe_t2i<2; sync fe_in?; assign efe_t2i:=efe_t2i+1; },
183  S3 -> S0{sync n_abort_t2n!; };
184  }
185  process servers{
186  state S0;
187  init S0;
188  trans S0 -> S0{sync r_out?; },
189  S0 -> S0{sync n_in!; },
190  S0 -> S0{sync ie_in!; },
191  S0 -> S0{sync fe_in!; };
192  }
193  //Edit system definition.
194  //system LOWER, MIDDLE, UPPER;
195  //system LOWER, iC2C_internal, NormalActivity, UPPER;
196  //system clients, iC2C_bottom, AbnormalActivity, MIDDLE, UPPER;
197  //system LOWER, MIDDLE, iC2C_top, servers
198  //system iC2C, clients, iC2C_bottom, AbnormalActivity, iC2C_internal,
      NormalActivity, iC2C_top, servers;
```

```
199  system clients, iC2C_bottom, AbnormalActivity, iC2C_internal,
      NormalActivity, iC2C_top, servers;
```

## Arquivo iC2C.q

```
1    //This file was generated from UPPAAL 3.2.2, November 2001
2    /*
3    */
4    ] not deadlock
5    /*
6    */
7    ] (AbnormalActivity.S1 or AbnormalActivity.S2) imply (not
      NormalActivity.S1 and not NormalActivity.S2)
8    /*
9    */
10   ] iC2C_bottom.S0 imply (AbnormalActivity.S0 and iC2C_internal.S0 and
      NormalActivity.S0 and iC2C_top.S0)
11   /*
12   */
13   ] (NormalActivity.S1 or NormalActivity.S2) imply (not AbnormalActivity.S1
      and not AbnormalActivity.S2)
```