Classificação e busca de componentes com tratamento de exceções

Luciana Setsuko Gakiya

Dissertação apresentada
ao
Instituto de Matemática e Estatística
da
Universidade de São Paulo
para
obtenção do título
de
Mestre em Ciências

Programa: Ciência da Computação
Orientador: Profa. Dra. Ana Cristina Vieira de Melo

São Paulo, julho de 2008

Agradecimentos

Em primeiro lugar agradeço a Deus pelo desenvolvimento desse trabalho.

Ao IME pela oportunidade de aprendizado e à minha orientadora Ana Cristina Vieira de Melo pela paciência nos meus momentos de crise e por acreditar no meu trabalho.

À minha irmã Kátia Asakawa e Leonardo Asakawa pelo apoio e incentivo durante os quatro anos de mestrado. À minha mãe Seiko Oba, à minha irmã Irina Bressan e ao meu cunhado Adauto Bressan pelo carinho e compreensão nos momentos em que fiquei ausente para me dedicar às pesquisas.

Ao meu irmão, Nelson, por assumir a responsabilidade de cuidar de meus pais quando ficaram doentes enquanto eu estudava.

Ao presidente da empresa onde trabalho, Sr Henrique Ribeiro, e ao meu chefe, José Thomé Filho, por me liberarem nos dias em que precisei ir ao IME.

Aos meus colegas de equipe Tessie Ikemori, Fabio Kimura e Adriano Lopes Pereira pelos conselhos, compreensão e pelo otimismo que me transmitiram nos momentos de estresse e desânimo.

Aos meus amigos Rodrigo Mendes Leme e Clarice Saito por estarem sempre me apoiando, ajudando e torcendo por mim.

Às amigas Cleonice Cavalcante e Luciana Hokazono pelo ombro amigo de todos os momentos.

E a todos aqueles que direta ou indiretamente contribuíram com informações, sugestões, esclarecimentos e conselhos técnicos.

Resumo

Em tempos onde a alta produtividade e competitividade são valores fundamentais para garantir a estabilidade no mercado de *software*, o desenvolvimento de sistemas baseado em componentes reutilizáveis tornou-se um paradigma bastante utilizado. Os sistemas atuais resultantes da integração dos componentes executam serviços cada vez mais complexos, e o número de situações de lançamento e tratamento de exceções também cresce. O desenvolvimento baseado em componentes (DBC) baseia-se na idéia de gerar um sistema a partir do uso de diversos componentes de *software* já existentes. Sendo assim, os componentes podem vir de fabricantes diferentes e, muitas vezes, quem os utiliza não necessariamente é quem os criou.

A falta de informações sobre o comportamento das exceções dos componentes dificulta o tratamento das mesmas no sistema resultante. De um lado, os desenvolvedores não possuem conhecimento completo dos diferentes ambientes operacionais nos quais os componentes serão integrados. Por outro lado, os componentes são fornecidos como "caixas-pretas" aos integradores que muitas vezes não têm acesso aos detalhes do comportamento anormal do componente.

Além disso, o mecanismo de tratamento de exceções freqüentemente é implementado de forma incorreta. Para um melhor gerenciamento, existem os repositórios onde os componentes após classificados são armazenados para posterior recuperação. Tipicamente, as informações armazenadas estão associadas às suas, funcionalidades, aos serviços oferecidos de forma que o desenvolvedor, durante a busca, não obtém informações a respeito das exceções e de como são ou devem ser tratadas. Tal situação pode comprometer a utilização adequada do componente. Dentro desse contexto, o trabalho a seguir mostra uma solução que minimiza a falta de informações sobre as exceções no processo de classificação e busca de componentes Java.

Palavras-chave: Exceção, Componentes, Repositório.

Abstract

In times when high productivity and competitiveness are fundamental values to ensure stability in the software market, the development of systems based on reusable components has become an widely used paradigm. The current systems, resulting from the integration of the components, perform increasingly complex services, and the number of cases of launching and management of exceptions grows also. Componentes based development (DBC) is based on the idea of building a system from the usage of many already existing software components. Meanwhile, the components may come from different sources and often those who use them are not necessarily those who created them.

The lack of information about the behavior of the exceptions of components is low. On one hand, developers of the components do not have full knowledge of the different operating environments in which the components will be integrated. On the other, these components are provided as "black-box"to integrators, who often have no access to details of the abnormal behavior of the component.

In addition, the mechanism for dealing with exceptions are often implemented in an improper manner. For a better management of the components, there are repositories where components, after being classified are stored for later recovery. Tipically, the details of the stored components are related to features and to the services offered by them, so the developer, during the search, gets no information about the exceptions thrown by the component nether how they are or whether they should be treated or not. Such situation may affect the proper usage of the component. In this context, the work below shows a solution that minimizes the lack of information about the exceptions in the process of classification and searching of Java components.

Keywords: Exception, Component, Repository.

Sumário

Lista de Abreviaturas **xiii**

Lista de Figuras **xv**

Lista de Tabelas **xvii**

1 Introdução **1**

1.1 Diretrizes 4

1.2 Objetivos 4

1.3 Organização do Trabalho 5

2 Classificação e Busca de Componentes **7**

2.1 Desenvolvimento Baseado em Componentes - DBC 7

2.1.1 Componentes 8

2.1.2 Modelo de Componentes 9

2.1.3 Arcabouço de Componentes 9

2.1.4 Especificação, Implementação e Instanciação 9

2.1.5 Principais Fases do DBC 10

2.1.6 Arquitetura de Sistemas Baseados em Componentes 10

2.2 Repositório de Componentes 11

2.3 Componentes EJB - Enterprise Java Beans 15
2.3.1 Estrutura e Implementação 16
2.4 Classificação e Busca de Componentes 17
2.4.1 Classificação por Facetas 17
2.5 Outros Métodos de Classificação 18
2.5.1 Classificação de Componentes de Negócio (*Business Component*) 18
2.6 Busca e recuperação de Componentes 19
2.6.1 Ontologias 22
2.7 Conclusão 24

3 Tratamento de Exceções **27**
3.1 Exceções 27
3.2 Tratamento de Exceções em Java 28
3.2.1 Principais classes de Exceções 29
3.2.2 Lançamento e Captura de Exceções 29
3.3 As más práticas no tratamento de exceções 32
3.4 Tratamento de Exceções em DBC 34
3.4.1 Tratamento de Exceções no Componente 34
3.4.2 Tratamento de Exceções na Arquitetura 35
3.5 Conclusão 37

4 Modelo de Classificação e Busca com Tratamento de Exceções **39**
4.1 O Contexto 39
4.2 O Modelo de Classificação de Componentes 41
4.2.1 Comportamento Normal 41

4.2.2 Comportamento Excepcional 42
4.3 O modelo de facetas, atributos e ontologia 42
4.3.1 As Facetas e os Atributos 43
4.4 O Método de Classificação 45
4.5 A Busca 45
4.5.1 Ontologia de Exceção 45
4.6 Conclusão 47

5 O Sistema Reuse++ 49

5.1 Introdução 49
5.2 O Funcionamento geral 49
5.3 O Projeto arquitetônico 50
5.3.1 Plataforma Arquitetural 52
5.3.2 Camadas Lógicas 52
5.3.3 Integração com outras aplicações 53
5.4 Reuse++ como aplicação J2EE 54
5.4.1 Componentes Web 55
5.4.2 Componentes EJB 55
5.5 Autenticação e Tela Principal 55
5.6 O mecanismo de Classificação 56
5.6.1 Análise das informações de exceções do Código-fonte 58
5.6.2 Exemplo de Cadastro e Classificação 59
5.6.3 O mecanismo de armazenamento 60
5.7 Mecanismos de Busca e Recuperação 60
5.7.1 Exemplo de Busca 63

5.7.2 Ontologia 64
5.8 Visualização das informações do Componente 66
5.9 Avaliação do Modelo e Método 67
5.9.1 A Coleção de Testes e as Consultas 67
5.9.2 Medidas de Avaliação 69
5.9.3 Resultados 69
5.10 Conclusão 70

6 Conclusões **73**
6.1 Contribuições 74
6.2 Trabalhos Futuros 74

Referências Bibliográficas **77**

Lista de Abreviaturas

DBC	Desenvolvimento Baseado em Componentes.
EJB	Enterprise Java Bean.
J2EE	Java 2 Entreprise Edition
XML	eXtensible Markup Language

Lista de Figuras

2.1 Relação entre a hierarquia e o nível de abstração de um componente [Padmal Vitharana, 2003] . . . 20

3.1 Hierarquia de Exceções em Java . . . 30

3.2 Estrutura de tratamento de exceções em Java . . . 31

3.3 Tratamento de Exceções em Java usando finally . . . 31

3.4 Relançamento de Exceções em Java . . . 32

3.5 Componente tolerante a falhas [Ferreira, 2001] . . . 35

4.1 Método de Classificação para a extração das informações . . . 46

4.2 Ontologia de Exceção . . . 47

5.1 Funcionamento Geral do REUSE++. . . . 51

5.2 Arquitetura J2EE do Reuse++ . . . 52

5.3 Arquitetura MVC do Reuse++ . . . 53

5.4 Diagrama UML de Implantação do REUSE++ . . . 54

5.5 Tela de Login do Reuse++ . . . 56

5.6 Tela Principal do Reuse++ . . . 57

5.7 Extração das informações de exceções para as facetas e atributos do modelo . . . 59

5.8 Tela de Cadastro de Componente . . . 60

5.9 Tela de Classificação com as novas facetas incluídas 61
5.10 Tabelas do Reuse++ para os dados de exceções. 62
5.11 Campos para preenchimento no mecanismo de busca. 63
5.12 Exemplo de Busca por exceção no Reuse++ 64
5.13 Resultado de Busca utilizando ontologia. 65
5.14 Informações do Componente no Reuse++ 66
5.15 Informações das Exceções do Componente no Reuse++ 68
5.16 Curva com a cobertura e precisão das consultas submetidas no Reuse++ 71

Lista de Tabelas

2.1 Atributos do Modelo de Classificação de Vitharana [Padmal Vitharana, 2003] para Componentes de Negócio 20

2.2 Facetas do Modelo de Classificacão de Vitharana [Padmal Vitharana, 2003] para Componentes de Negócio 21

4.1 Facetas do Modelo com Exceções 44

4.2 Atributos do Modelo com Exceções 44

Capítulo 1

Introdução

Em meio a alta produtividade e competitividade no mercado de *software*, as fortes restrições aos prazos e custos passaram a ser fatores comuns no desenvolvimento de sistemas. O Desenvolvimento Baseado em Componentes surge como uma nova perspectiva de desenvolvimento de *software* caracterizada pela composição de partes já existentes. Construir novas soluções pela combinação de componentes prontos aumenta a qualidade e proporciona um rápido desenvolvimento, diminuindo o tempo de entrega do produto a o mercado. Os sistemas resultantes permitem que sejam adicionadas, removidas e substituídas partes do sistema sem a necessidade de sua completa substituição [Szyperski, 2002].

Porém, criar um sistema a partir de um conjunto de componentes pré-existentes exige um processo de escolha cuidadoso. Repositórios ganharam importância ao armazenar diversos componentes e ainda, mecanismos de classificação e busca foram inseridos para facilitar a atividade de seleção do componente.

Um dos métodos de classificação mais tradicionais é a classificação baseada em facetas. Ranganathan foi o primeiro a introduzir ao mundo o termo **faceta** na ciência da computação e na biblioteconomia e também foi o primeiro a desenvolver a teoria de análise de facetas [Ranganathan, 1960]. Originalmente facetas são definidas como aspectos, propriedades ou características dentro de uma classe ou tópico específico definidos claramente. Por exemplo, dentro do contexto de componentes, podemos ter a faceta "função"com os valores correspondentes às funcionalidades assumidas pelo componente e assim, permitir a classificação e busca por funcionalidade.

Atualmente pesquisas têm sido feitas na área de recuperação da informação sugerindo o uso de ontologias no processo de busca de documentos [Ancelmo Maciel Nunes, 2006]. Uma ontologia é

uma especificação explícita e formal de uma conceitualização compartilhada [Gruber, 1995]. Uma ontologia inclui conceitos que descrevem o domínio de conhecimento, relacionamentos entre esses conceitos, instâncias desses conceitos, relacionamentos entre instâncias e axiomas. O uso de ontologias se justifica porque elas constituem uma forma de compartilhar a mesma estrutura de informação entre pessoas e agentes de *software*, permitindo o reuso do conhecimento do domínio [Ancelmo Maciel Nunes, 2006].

Os desenvolvedores passaram a enfrentar o desafio de buscar nos repositórios componentes que executem as funcionalidades desejadas e ainda, tenha um comportamento esperado em situações de falhas.

A introdução de tolerância a falhas em um sistema exige técnicas de estruturação que minimize a interferência negativa do código de detecção e recuperação de erros com o código resposável pelas funções normais do *software*. Nesse contexto, os mecanismos de tratamento de exceções existentes em liguagens como Java e C++, são fundamentais, pois permitem a separação do código de recuperação de erro do código normal. Entretanto, não raro, o mecanismo de tratamento de exceções é utilizado de forma errada ou simplesmente sua utilização é ignorada.

O tema tratamento de exceções ganhou importância e atualmente existem várias pesquisas, a maioria delas tem como foco prever e tratar a existência das exceções de forma antecipada, ou seja, durante todas as fases do processo de desenvolvimento. Há também trabalhos referentes à documentação e modelagem das exceções, bem como estudos de arquiteturas que valorizem as exceções dos componentes.

As pesquisas caminham para a criação de ferramentas e solução de problemas como o mau uso dos mecanismos de tratamento de exceções durante a codificação. Assim, dentre vários trabalhos temos :

- [Brito, 2005] apresenta um método de modelagem de exceções em Sistemas Baseados em Componentes chamado MDCE+. MDCE+ é uma versão melhorada do MDCE que é um método genérico que apresenta diretrizes para a especificação do comportamento excepcional em todas as fases de desenvolvimento: requisitos, análise, projeto e implementação. MDCE+ combina as duas abordagens de desenvolvimento: Bottom Up e Top Down priorizando a arquitetura do sistema. A ênfase dada na arquitetura desde a fase de análise permite a descoberta antecipada de falhas.

- [Guerra, 2004] apresenta uma abordagem arquitetural para embutir tolerância a falhas em sistemas baseados em componentes. No seu trabalho são apresentados 4 elementos: a arquitetura de um sistema baseada em componentes ideais, uma solução arquitetural para transformar componentes de prateleira em componentes ideais, uma estratégia geral para tratamento de exceções em DBC e ainda uma proposta de um ambiente integrado para DBC centrado na arquitetura.

- [And, 2004] apresenta um *framework* de tratamento de exceções gerenciado pelo container. O *framework* CMEH permite que as exceções sejam tratadas de forma simples e modular dentro do contexto específico da aplicação. A base do CMEH é um modelo de eventos de exceções que permite ao desenvolvedor implantar código de tratamento de exceções, de forma que uma vez registrados os trechos de código suspeitos, os mesmos são observados para eventos de Exceções específicos. Daí durante a execução do método do componente, tais Exceções são lançadas permitindo ao desenvolvedor lidar melhor com o comportamento excepcional do componente.

- [Pereira, 2007] em seu trabalho apresenta um *framework* para a especificação do tratamento de exceções em nível arquitetural, baseando-se no modelo conceitual de ações atômicas coordenadas e utilizando a linguagem orientada a eventos CSP(Communication Sequential Processes). Sua principal característica é prover um protocolo padronizado para a coordenação do tratamento de exceções, envolvendo a cooperação dos componentes do sistema.

- [da Silva Xavier, 2008] utililiza em seu trabalho uma técnica denominada verificação de modelos e como ponto de entrada, define um conjunto de propriedades relacionadas ao tratamento de exceções. É apresentado um ambiente de testes que permite o uso dessas propriedades bem como a exibição das estatísticas de cobertura dos testes de acordo com o critério selecionado.

Estudos também estão sendo feitos a fim de melhorar os mecanismos de classificação e busca de componentes como é mostrado a seguir:

- [Melo, 2006] apresenta um modelo de classificação e busca de componentes de negócio. Nesse trabalho a classificação é semi-automática e baseada em um conjunto de facetas e atributos. A partir da análise do código-fonte e diretivas Javadoc as informações relacionadas à funcionalidade do componente são extraídas automaticamente. As demais informações como domínio, regras de negócio e papéis que o componente pode exercer são inseridas manualmente pelo classificador. O protótipo, Reuse+, foi criado para viabilizar o modelo e método propostos.

No paradigma DBC, os componentes selecionados comunicam-se através de interfaces bem definidas e que podem ser implementadas independentemente [Alan W. Brown, 1998]. Infra-estruturas de componentes J2EE e .NET são exemplos de tecnologias de DBC amplamente utilizadas no desenvolvimento de *software* atuais.

No trabalho [Melo, 2006], o REUSE+, só considera e manipula classes Java e ainda, ignora a existência das interfaces do componente. Além disso, durante a classificação e recuperação não armazena nem exibe informações a respeito das exceções lançadas ou tratadas pelo componente.

O fato do mecanismo de exceções ser, várias vezes, empregado de forma incorreta motivou esse trabalho a coletar também medidas de exceções (Seção 3.3) existentes no código. Essa coleta permitirá avaliar a qualidade do componente selecionado.

Assim, a fim de oferecer uma descrição mais completa e, ainda, melhorar o mecanismo de busca, o modelo foi implementado na ferramenta Reuse+ de [Melo, 2006] e algumas partes foram modificadas. O novo modelo de facetas e ontologia com informações de exceções foi agregado, dando origem ao sistema Reuse++. Nele, os componentes manipulados deixam de ser classes Java e passam a ser componentes EJB, com suas respectivas interfaces.

1.1 Diretrizes

O desenvolvimento desse trabalho teve as seguintes diretrizes:

- Baixo Custo de Especificação Excepcional Manual - o modelo de classificação das exceções dos componentes deve ser leve e simples. O objetivo da proposta é extrair automaticamente o máximo de informações do código-fonte, excluindo totalmente a especificação manual das Exceções do Componente.
- Alta Expressividade - o máximo de informações pertinentes de exceções devem ser representadas de forma organizada e legível.
- Facilidade de Manutenção - a ferramenta não deve sofrer grandes modificações com as sucessivas atualizações dos Componentes.

1.2 Objetivos

Os objetivos desse trabalho de mestrado são:

- Apresentar um novo modelo de Classificação baseado em facetas e ontologia que considere também informações relacionadas ao comportamento excepcional do componente.
- Apresentar um método de classificação que faça a extração automática das principais informações de exceções.
- Apresentar o protótipo REUSE++ com o modelo e método propostos.
- Avaliar a viabilidade do modelo e método propostos através de métricas de cobertura e precisão.

1.3 Organização do Trabalho

- Capítulo 2: são apresentados os conceitos necessários para a compreensão desse trabalho. Serão explicados de forma breve os seguintes conceitos: Componente, Desenvolvimento Baseado em Componentes, Repositório, Classificação e Busca de Componentes, Ontologia.
- Capítulo 3: é mostrado o recurso de tratamento de exceções. Todas as informações relacionadas ao funcionamento das exceções nos sistemas e componentes serão vistos.
- O Capítulo 4: diz respeito ao modelo de classificação e busca com tratamento de exceções. Neste capítulo é apresentado também o método de extração das informações de exceções dos componentes.
- Capítulo 5: é apresentada a ferramenta Reuse++ que implementa o modelo e o método mostrados. Sua estrutura, funcionamento e telas principais estão contidos nesse capítulo. Finaliza mostrando os resultados de um conjunto de experimentos efetuados na ferramenta.
- Capítulo 6: serão apresentadas as conclusões obtidas no desenvolvimento desse trabalho.

Capítulo 2

Classificação e Busca de Componentes

Pesquisar por documentos a partir de ferramentas de busca é uma atividade essencial em muitos domínios. Vários sistemas de recuperação de informação (RI) têm sido desenvolvidos para facilitar a pesquisa e recuperação de informação por parte do usuário. Esses sistemas tipicamente realizam buscas utilizando palavras-chave fornecidas pelo usuário. Normalmente obtém-se como resposta um grande número de itens, o que força o usuário a dedicar uma quantidade significativa de tempo na análise das informações até encontrar aquelas que realmente são relevantes. Nesse sentido, pesquisas vêm sendo desenvolvidas com o intuito de incrementar o processo de classificação da informação e melhorar a velocidade de recuperação e relevância dos itens retornados em uma pesquisa [Pasi, 2002, Pereira, 2007].

No contexto de reutilização de componentes, a utilização de boas técnicas de classificação de componentes é essencial para que eles sejam descritos de forma completa e sintética, facilitando e melhorando o processo de seleção [Mittermeir et al., 1998].

Neste capítulo são explicados conceitos que facilitarão a compreensão do tema classificação e busca de componentes.

2.1 Desenvolvimento Baseado em Componentes - DBC

Os Componentes de *software* surgiram no final da década de 60 e foram enfatizados em 1968 por McIlroy [McIlroy, 1968] que defendia a produção de *software* em larga escala através da criação de uma indústria de componentes de *software* semelhante à indústria de componentes de *hardware*. Assim, desenvolvimento baseado em componentes pode ser definido como a construção de sistemas a partir da composição planejada de componentes previamente especificados, construídos e testados [Alan

W. Brown, 1998].

DBC permite alto grau de reutilização trazendo redução de custos, de tempo de desenvolvimento e o fato dos componentes já terem sido utilizados em outros contextos proporciona uma alta qualidade no sistema resultante.

2.1.1 Componentes

Segundo Szyperski [Szyperski, 2002], um componente é uma unidade de composição com interfaces contratualmente especificadas e dependências de contexto explícitas, podendo ser usado independentemente e ser combinado por outras partes. De acordo com essa definição, o componente é um elemento de *software* que pode ser associado a outros componentes através de suas interfaces, as quais servem para esconder os detalhes de implementação.

As interfaces representam pontos de comunicação entre um componente e o seu ambiente. Podem ser subdivididas em 2 grupos: **interface requerida** que identifica um ponto de acesso a serviços que o componente necessita do ambiente e **interface provida** que identifica um ponto de acesso a serviços que o componente é capaz de fornecer ao ambiente [D. Garlan, 2000].

Segundo Sametinger [Johannes, 1997] existem fatores que exercem grande influência na qualidade do componente e também no seu grau de reutilização. São eles:

- **Autocontido**: o componente não deve depender de outros componentes, ele sozinho deve ser capaz de ser reutilizado.
- **Identificação**: o componente deve estar localizado em um único local, não pode estar misturado com outros artefatos de forma a ser claramente identificável.
- **Funcionalidades**: o componente deve apresentar funcionalidades, executar e fornecer serviços.
- **Interfaces**: o componente deve possuir interfaces bem definidas que escondam detalhes de implementação e permitam a interação do componente com outros componentes.
- **Documentação**: o componente precisa apresentar uma documentação clara com informações a respeito dos serviços fornecidos pelo componente.
- **Condição de reuso**: o componente deve apresentar também informações de como reutilizar o componente e direcionar ao contato apropriado no caso de ocorrência de problemas ou dúvidas.

2.1.2 Modelo de Componentes

Um modelo de componentes especifica os padrões e convenções impostas aos desenvolvedores de componentes e os pressupostos básicos que podem ser assumidos a respeito do ambiente. Os tipos de componentes, formas de interação e a definição de recursos são especificações esperadas no modelo de Componentes segundo Bachmann [Bachmann et al., 2000]. Entre os exemplos de modelos de componentes temos: CCM da OMG, DCOM e COM/COM+ da Microsoft e JavaBeans e EJB da Sun MicroSystems.

2.1.3 Arcabouço de Componentes

O arcabouço (*Framework*) de componentes representa a infra-estrutura dos componentes, ou seja, a base sobre as quais estes padrões e convenções do modelo de componentes são empregados. Com isso, arcabouço e modelo de componentes são conceitos complementares e fortemente relacionados. O arcabouço deve dar suporte às definições estabelecidas pelo modelo de componentes. Além disso, o seu uso possibilita que os desenvolvedores e aplicações não se preocupem com a implementação de serviços relacionados com a troca de mensagens, passagem de dados e conectividade dos componentes deixando-os sob responsabilidade do arcabouço.

O arcabouço é responsável por pelo menos uma das seguintes categorias de serviços: empacotamento, distribuição, segurança, gerenciamento de transações e comunicação assíncrona [Bachmann et al., 2000]. Por exemplo, dentre os arcabouços do modelo EJB, existem produtos comerciais como Websphere Application Server da IBM e BEA WebLogic da BEA Systems que são proprietários e os de domínio público como JBoss e Jonas da Evidian.

2.1.4 Especificação, Implementação e Instanciação

A Especificação de Componente define o comportamento observável externamente do componente com abrangência e precisão necessários para a sua integração em diferentes sistemas, porém abstraindo detalhes de qualquer implementação específica. Uma Implementação de componente realiza a especificação fornecendo um modelo concreto para a instanciação do Componente em diferentes ambientes. Uma instanciação do Componente só ocorre no momento da execução do sistema onde o Componente está integrado [Guerra, 2004].

2.1.5 Principais Fases do DBC

Segundo *Brown* [Alan W. Brown, 1996] as principais fases de desenvolvimento baseado em componentes são:

1. **Seleção**: consiste em pesquisar e selecionar componentes que possuem potencial de serem usados no desenvolvimento do sistema. Envolve um processo de investigação das qualidades e propriedades dos Componentes. O objetivo dessa atividade é gerar uma relação de componentes candidatos.

2. **Qualificação**: consiste em garantir que o Componente candidato execute as funcionalidades necessárias, o ajuste dele na arquitetura definida para o sistema e se apresenta as qualidades necessárias à aplicação sob ponto de vista de usabilidade, desempenho e confiabilidade. É feita uma análise detalhada de toda a documentação disponível e testes do componente em diferentes cenários.

3. **Adaptação**: consiste na correção das potencias fontes de conflitos entre os componentes selecionados e qualificados para compor o sistema.

4. **Composição**: consiste em juntar os componentes em uma infra-estrutura comum. A infra-estrutura deve prover a ligação dos componentes e um conjunto de convenções conceituais compartilhadas pelos componentes.

5. **Atualização**: consiste na substituição dos Componentes antigos por outros com comportamentos similares.

2.1.6 Arquitetura de Sistemas Baseados em Componentes

A arquitetura de um sistema mostra como um sistema é realizado por um conjunto de componentes arquiteturais e as interações entre eles. O objetivo é mostrar 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 [Mary Shaw, 1996].

Os componentes são abstrações que encapsulam a computação parcial de um sistema (aspecto computacional), enquanto que conectores são abstrações que encapsulam a interação entre componentes (aspecto composicional). As interações entre componentes na arquitetura correspondem,

em geral, a protocolos relativamente complexos que requerem meios mais expressivos (para a sua descrição) que os mecanismos primitivos oferecidos por linguagens de programação convencionais.

Linguagens para Descrição de Arquiteturas (ADL) são linguagens especialmente projetadas para a descrição de componentes e interações entre componentes na arquitetura. Arquiteturas descritas em ADL podem ser analisadas por ferramentas com o objetivo de verificar qual arquitetura satisfaz as propriedades desejadas do sistema final. Uma das características marcantes de tais linguagens é a descrição das interações entre componentes através do uso de conectores explícitos, que têm situação comparável ao de componentes, sendo tratados como entidades com tipo, que podem ser explicitamente descritas, analisadas e reutilizadas [Paul Clements, 1996].

2.2 Repositório de Componentes

Com a crescente importância atribuída ao reuso de *software* e informações, muito se tem discutido a respeito da organização de centralização desses valores. O repositório surge como uma solução para se resolver um paradoxo observado nas organizações: sabe-se que o compartilhamento de experiências e informações pode proporcionar grandes vantagens competitivas, entretanto muitas empresas ficam impossibilitadas de aproveitar tais benefícios porque não conseguem lidar com a variedade e a distribuição ineficiente das informações desejadas e, muitas vezes, sequer tem conhecimento das informações disponíveis nos repositórios [Henninger, 1997].

Os repositórios de componentes construídos com foco no reuso são considerados sistemas de informação com um propósito especial, exigindo um poderoso suporte a modelagem semântica e flexível recuperação de informações [Panos Constantoupolos, 1993]. Dessa forma, é possível armazenar e recuperar Componentes durante as fases de análise, projeto e implementação de um sistema. Podem armazenar também descrições de interfaces, propriedades não-funcionais, implementações e casos de teste.

O repositório exerce um papel importante na decisão de construção de um novo Componente. Se o programador vai reutilizar um componente ao invés de criar um novo, depende essencialmente da disponibilidade desse componente no repositório, do mecanismo de busca para encontrá-lo e da habilidade do programador em pesquisar no repositório [Johannes, 1997].

A documentação dos Componentes fornecem informações adicionais para ajudar o potencial usuário a medir o esforço requerido para fazer reuso do mesmo. Repositórios devem armazenar informações sobre propriedades de um componente que facilitem acesso a ele. Estas propriedades

devem ser armazenadas em um formato pesquisável, para que componentes possam ser facilmente encontrados. Ainda, usuários devem ser capazes de colocar e retirar componentes do repositório consistentemente [Redolph et al., 2004].

Segundo [Sanchez, 2005], as principais vantagens no uso de um repositório centralizado são:

- Lugar único para armazenar os componentes bem como pesquisar por componentes existentes;
- Forma padronizada de documentar, pesquisar e enumerar os componentes;
- Forma padronizada de controlar mudanças e melhorias nos componentes.

As características dos componentes permitem uma melhor compreensão do componente e viabiliza a sua classificação no repositório. O objetivo é que uma vez classificados, os componentes possam ser facilmente recuperados por um mecanismo de busca. A literatura divide as características de um Componente em grupos envolvendo abordagens de vários autores. Esses grupos referem-se a identificação, uso, maturidade, documentação, tecnologia, alterações e controle de qualidade do componente. De acordo com [Redolph et al., 2004] temos:

1. Identificação
    - Nome - corresponde ao nome do Componente. No mecanismo de armazenamento esse nome segue uma padronização de atribuição de nomes de forma a garantir a unicidade dos nomes dos componentes no repositório.
    - Origem - identifica o responsável pelo desenvolvimento do componente.
2. Uso
    - Propósito - define quais são as funcionalidades do sistema.
    - Domínio de Aplicação - refere-se às situações, ao contexto em que o componente se aplica.
    - Componentes Similares - lista outros componentes com funcionalidades semelhantes.
    - Tipo - informa a forma de disponibilidade do componente: documentação e diagramas, código-fonte, executável.
    - Fase de Integração - informa o momento em que o componente deve ser inserido ou executado no processo de desenvolvimento da aplicação.

- Granularidade - identifica o tamanho do componente segundo estratégias como quantidade de funcionalidades, número de casos de uso, fases de integração.

3. Maturidade

- Nível de reuso - informa o número de vezes em que o Componente foi usado para o desenvolvimento de diferentes aplicações.
- Versões - define um histórico com as diferentes versões que o componente assumiu ao longo do tempo.

4. Documentação

- Modelo de Componentes - informa os padrões e as convenções usadas pelo componente em relação à sua estrutura e formas interação.
- Especificação das Interfaces - corresponde a definição das operações , parâmetros e tipos esperados pelas interfaces do Componente.

5. Tecnologia

- Infraestrutura - informa como deve ser a infra-estrutura onde o Componente será executado, os serviços necessários para a interação/comunicação com outros Componetes.
- Portabilidade - corresponde à plataforma de atuação bem como informações relacionadas ao seu uso em outras plataformas.
- Ferramenta de desenvolvimento - corresponde à linguagem de programação, ambiente de desenvolvimento, compiladores e bibliotecas utilizados no desenvolvimento do Componente.
- Interoperabilidade - informações que especificam ou restrinjam a comunicação com os componentes de diferentes tecnologias.
- Interface gráfica - corresponde a definição de alguma interface gráfica do Componente se existir.
- Característica não-funcionais - informam a respeito de questões de desempenho, segurança, confiabilidade, manipulação de erros e concorrência.
- Restrições - definem as limitações do Componente.

- Distribuição - característica que define a localização do Componente e a possibilidade de ser distribuído.

6. Alterações

   - Formas de modificações - define os meios de como modificar o Componentes.
   - Acesso ao código-fonte - informa a possibilidade de acesso ao código-fonte do Componente.

7. Controle de Qualidade

   - Métricas - informa quais são as métricas para avaliar o uso, qualidade e maturidade do Componente.
   - Teste - define casos de teste ou outros mecanismos que permitem testar as funcionalidades do Componente individualmente ou quando está integrado a determinadas aplicações.

Lucrédio et al. [Lucredio et al., 2004] apresentam um conjunto de requisitos para um eficiente engenho de busca e recuperação de componentes. São eles:

1. **Elevada precisão e recuperação**. Elevada precisão significa que a maioria dos componentes recuperados são relevantes. Elevada recuperação significa que poucos elementos relevantes são identificados, sem ser recuperados.

2. **Segurança**. Em um mercado de componentes global, a segurança deve ser considerada uma característica primordial, já que existe a possibilidade de que pessoas não autorizadas possam acessar o repositório.

3. **Formulação de consultas**. Há uma natural perda de informação quando os usuários formulam consultas. Segundo [Henninger, 1997], existe uma lacuna conceitual entre o problema e a solução, já que componentes são descritos em termos da sua funcionalidade e as consultas em termos da solução. Assim, um engenho de busca deve prover meios de auxiliar o usuário na formulação das consultas, buscando reduzir esta lacuna.

4. **Descrição do componente**. O engenho de busca é responsável por identificar os componentes relevantes para o usuário de acordo com a consulta formulada e executada em cima das descrições dos componentes.

5. **Familiaridade no repositório**. O reuso ocorre mais freqüentemente através de componentes bem conhecidos [Ye and Fischer, 2002]. Entretanto, um engenho de busca deve auxiliar usuário a explorar e recuperar componentes familiares ao que era o alvo inicial, facilitando futuras buscas e estimulando a concorrência entre os fornecedores de componentes.

6. **Interoperabilidade**. Em um cenário envolvendo repositórios distribuídos, é inevitável não pensar em interoperabilidade. Deste modo, um engenho de busca que funciona neste cenário deve ser baseado em tecnologias que facilitem sua futura expansão e integração com outros sistemas e repositórios.

7. **Desempenho**. Desempenho é usualmente medida em termos de tempo de resposta. Sistemas centralizados envolvem variáveis relacionadas ao poder de processamento e algoritmos de busca. Já em sistemas distribuídos outras variáveis devem ser consideradas, como, por exemplo, controle de tráfego da rede, distância geográfica e, claro, o número de componentes disponíveis.

2.3 Componentes EJB - Enterprise Java Beans

Os componentes EJB pertencem à especificação J2EE definida plea Sun Microsystems. São utilizados em aplicações corporativas que exigem maior suporte de serviços relacionados a escalabilidade e segurança.

EJB também caracteriza-se como uma arquitetura de componentes para o desenvolvimento e utilização de aplicações corporativas baseadas em componentes distribuídos. Em uma arquitetura cliente/servidor, são considerados componentes distribuídos posicionados do lado servidor que podem ser acessados por diferentes aplicações cliente. São desenvolvidos principalmente pra executar tarefas de lógica de negócio no lado servidor e são capazes de coordenar e executar grandes volumes de transações [Microsystems, 2008].

O Conteiner EJB é o local onde os beans são disponibilizados para serem utilizados e tem por funçãoo dar suporte a vários serviços ao EJB durante sua execução. Esses serviços são: acesso remoto, segurança, persistência, transações, concorrência e acesso a recursos. Um exemplo de Container EJB é o servidor *JBoss*.

Os componentes EJB, de acordo com a tarefa a ser executada, podem ser de três tipos [Microsystems, 2008]:

1. **Session Beans**: componentes que encapsulam a lógica de negócios, oferecendo vários serviços

a seus clientes a través de suas interfaces.

2. **Entity Beans**: componente de dados que realiza serviços de armazenamento permanentemente dos dados em uma estrutura secundária, normalmente Banco de Dados.

3. **Message Driven Beans**: componentes que tem a função de consumir mensagens de uma fila e executar algum processamento.

2.3.1 Estrutura e Implementação

O desenvolvimento de componentes EJB envolve a codificação de vários elementos. São eles:

- **Classe Enterprise Bean**: corresponde à classe Java que representa concretamente o componente com as suas funcionalidades implementadas. Dependendo do seu tipo (*Entity*, *Session* ou *MessageDriven*) deve implementar as interfaces *javax.ejb.SessionBean*, *javax.ejb.EntityBean* ou javax.ejb.MessageDrivenBean.

- **Home Interface**: define as operações relativas ao ciclo de vida de um EJB, possui os métodos para a criação, localização e remoção dos EJBs. É caracterizada por estender a interface *javax.ejb.EJBHome*.

- **Remote Interface**: através desta interface são mostradas as operações disponíveis ao acesso de aplicações externas ao container EJB. É caracterizada por estender *javax.ejb.EJBObject*.

- **Local Interface**: seu uso é opcional e possui papel equivalente a *Remote Interface*, apenas com o diferencial que trata apenas componentes locais a um mesmo container. É caracterizada por estender *javax.ejb.EJBLocalHome*.

Para informar o container quais EJB's serão gerenciados e suas respectivas características são utilizados arquivos *deployment descriptor* no formato XML(*Extensible Markup Language*). Atualmente, existem ferramentas que criam automaticamente esses arquivos de configuração. Dessa forma, toda vez que o *Container* for inicializado, as informações dos EJB's serão carregadas.

A finalização do desenvolvimento de um EJB ocorre com o empacotamento do conjunto de interfaces e classes criados em um arquivo JAR (*Java Archive*). Os arquivos com extensão JAR representam um conjunto de arquivos compactados similares aos arquivos de extensão ZIP.

2.4 Classificação e Busca de Componentes

O processo de organização de componentes com o objetivo de recuperá-los para reuso em um determinado contexto é denominado classificação. A classificação é usada para agrupar componentes similares, que compartilham características que os demais grupos não têm [Johannes, 1997].

2.4.1 Classificação por Facetas

Facetas são consideradas como pontos de vista, perspectivas ou dimensões, organizadas e analisadas dentro de termos básicos de um domínio específico. Por exemplo, no domínio de cinema os filmes podem ser classificados pela faceta: categoria (comédia, drama e aventura seriam os possíveis valores) e pela faceta faixa etária, permitindo assim uma busca de filmes por categoria ou faixa etária.

A classificação por facetas consiste na análise do assunto e na síntese das idéias nele contidas enquadrando-as em uma das facetas (grupos ou classes) do domínio previamente estabelecido [Buchanan, 1979]. Por esse motivo os esquemas de classificação facetados são também denominados esquemas analíticos-sintéticos. Sua essência está na organização de termos de um determinado campo de conhecimento em facetas mutuamente exclusivas.

No ambiente computacional, o conjunto de facetas não necessita ser fechado, durante a classificação ou a recuperação da informação por buscadores. Os criadores de uma base de dados compartilhada podem adicionar uma nova faceta a qualquer hora, e os usuários podem selecionar elementos de quantas facetas quiserem.

Unidades de informação podem ser associadas com elementos de diferentes hierarquias de facetas. Desta forma, buscadores de informação podem passar por diversas hierarquias para ver quais informações estão associadas com determinado elemento. Elas simplesmente representam uma perspectiva de interesse de pelo menos um espectador.

Esse tipo de classificação apresenta vantagens, pois facetas podem ser combinadas e a alteração individual de uma faceta não afeta outras ligações com ela [Vickery, 1960].

Segundo [Prieto-Diaz, 1991], as principais vantagens da classificação por facetas são:

- Facilidade de classificação do artefato;
- Simplicidade de representação;

- Armazenamento das descrições;
- Uniformidade de atributos da classificação;
- Facilidade da automatização.

2.5 Outros Métodos de Classificação

- **Classificação por texto livre**: exige apenas uma rigorosa documentação textual acoplada ao componente reutilizável, buscando-os através do acesso aos textos completos com as informações sobre o componente. O usuário deve ter familiaridade com o domínio da busca e saber quais palavras são usadas na classificação dos componentes, o que pode não abranger todas as possibilidades de uso de um componente.
- **Classificação por palavra-chave**: atribui-se palavras-chaves ao componente, as quais podem ser características de componente. O usuário informa a palavra-chave para a busca que é comparada a palavra chave atribuída ao componente no repositório. Redolph [Redolph et al., 2004] defende que os métodos baseados em palavras-chaves como, por exemplo, os encontrados na Web, são limitados e recuperam uma grande quantidade de informação que muitas vezes são desnecessárias. Outra questão é a possibilidade de existirem termos diferentes para descrever a mesma informação sobre o componente num mesmo contexto, resultando numa imprecisão ainda maior na busca por um componente. Esse alto grau de inexatidão, pode resultar na recuperação de componentes inadequados.
- **Classificação Enumerada**: classifica componentes como categorias hierárquicas divididas em subcategorias. É fácil de ser entendida e utilizada, porém é pouco flexível e pode causar ambigüidades, pois um componente pode pertencer a várias categorias. Novos elementos só podem ser inseridos em níveis mais baixos da hierarquia sob pena de reorganização total da hierarquia. Se a hierarquia for grande e complexa a busca pode ser frustrante e cansativa.

2.5.1 Classificação de Componentes de Negócio (*Business Component*)

O que diferencia um componente de negócio de um componente simples é a sua capacidade de oferecer uma funcionalidade de negócio. Por exemplo, um componente que faz processamento de cartão de crédito. É possível compor vários componentes de negócio afim de oferecer uma funcionalidade mais complexa ao sistema, seguindo a mesma abordagem de construção baseada em componentes que Mcllroy propôs [McIlroy, 1968].

Vitharana [Padmal Vitharana, 2003], propôs um esquema efetivo de classificação para componentes de negócio considerando vários níveis de detalhe e abstração, além de um repositório para armazenar e recuperar tais componentes. Um componente de negócio pode ser visto como a implementação de um único conceito autônomo de negócio.

Avanços recentes na área de *Web Services* e a adoçãoo de padrões como SOAP (Simple Object Access Protocol), WSDL (Web Service Description Language) e UDDI (Universal Description, Discovery Integration) deram suporte ao desenvolvimento de componentes de negócio e aplicações baseadas nesses componentes. O diferencial da técnica é a preocupação em capturar informações que o componente de negócio tem, tornando a classificação mais completa e abrangente. Como mostra a Figura 2.1, um componente de negócio pode ser descrito em vários níveis de abstração ao longo de uma hierarquia. Nela, o componente de negócio representa o mais alto nível de abstração, enquanto as interfaces, tipos de dados e métodos representam os níveis de detalhe subsequentes. Particularmente, em orientação a objetos, a hierarquia de abstração de um componente consiste em uma ou mais interfaces que representam as funcionalidades providas pelo componente [Padmal Vitharana, 2003].

Para prover informações suficientes sobre um componente de negócio levando em conta diferentes níveis de abstração, Vitharana [Padmal Vitharana, 2003] definiram dois tipos de informação: estruturadas e semi-estruturadas. As informações estruturadas são todas as características bem definidas que descrevem um componente (como versão e plataforma) e as semi-estruturadas que são aquelas que podem assumir diversas formas (palavras-chave, texto ou nula, quando não se aplicarem). Atributos são utilizados para armazenar os identificadores estruturados e facetas são utilizadas para descrever as características semi-estruturadas. Ambos são utilizados para classificar os diferentes níveis da hierarquia de abstração de um componente. Melo [Melo, 2006] em seu trabalho utiliza usa esse modelo de facetas e atributos e ainda, executa um método para extração semi-automática de tais informações a partir do código-fonte e diretivas *javadoc* do componente.

2.6 Busca e recuperação de Componentes

Na medida que cresce a adoção de políticas de reutilização, o volume de componentes armazenado nos repositórios cresce também. Uma busca eficiente dos componentes torna-se uma tarefa difícil. No contexto de reutilização o termo documento, freqüentemente utilizado em RI, pode ser entendido como componente de *software* que será classificado e posteriormente

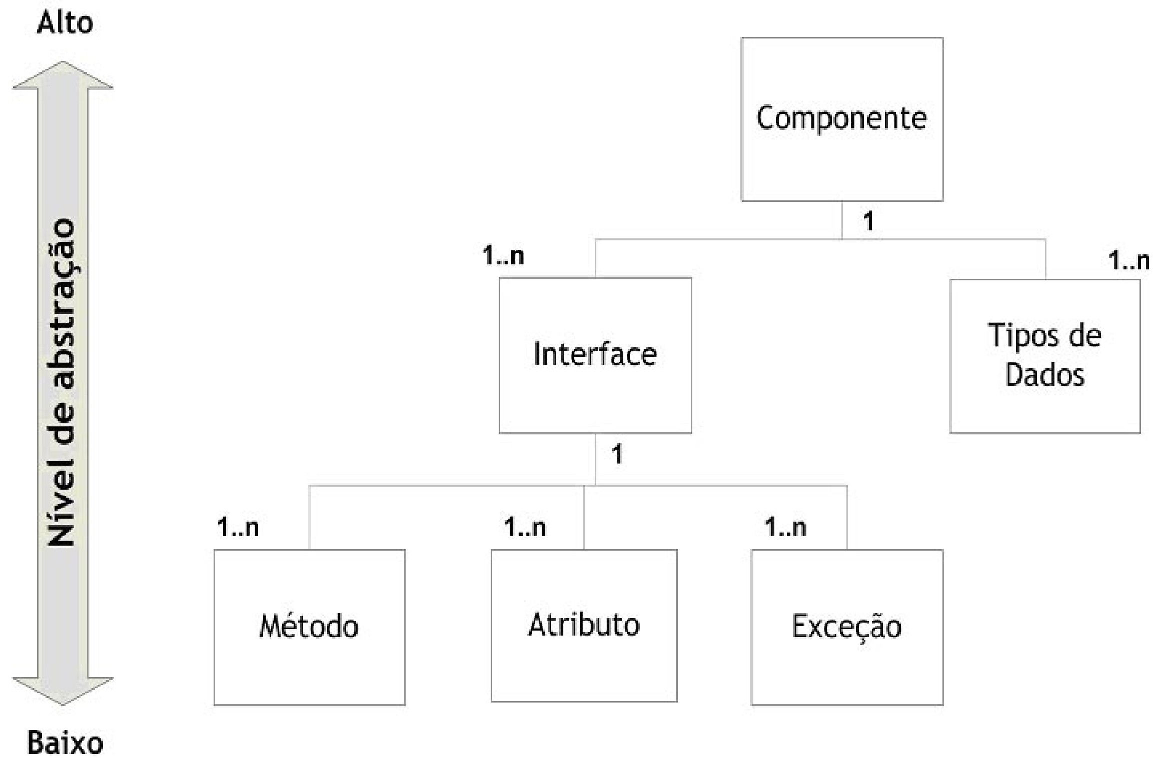

Figura 2.1: Relação entre a hierarquia e o nível de abstração de um componente [Padmal Vitharana, 2003]

Atributo	Descrição
Identification(Id, Name)	Identificdor único, Nome do Componente
Type (System, Algorithmic, Application)	Tipo de sistema quanto ao escopo e nível de especialização
Management (owner, contact, version)	Informações sobre o autor do componente, meio de contato, versão entregue
Industry-Specific Application	Tipo específico de indústria onde o componente pode ser usado
Across Industry Application	Tipo geral de indústria onde o componente pode ser usado
OperationSystem	Sistema Operacional onde o componente pode ser instalado
Implementation Language	Linguagem de Programação em que o componente foi consruído

Tabela 2.1: Atributos do Modelo de Classificação de Vitharana [Padmal Vitharana, 2003] para Componentes de Negócio

recuperado.

Um sistema de recuperação de informação deve representar o conteúdo dos documentos do *corpus* (conjunto de documentos do sistema), apresentando-os de maneira adequada a cada consulta do usuário, possibilitando uma rápida seleção dos itens de interesse . Tais consultas,

Faceta	Descrição
Synonym	Outros possíveis nomes para o componente
Role	Papéis que o componente poderia exercer em aplicações
Business Rules	Regras importantes de alto nível apliáveis ao componente
Function/Task	O que o componente faz
Element/Part	Elementos ou partes associadas ao componente
Action/Event	Ações e eventos relacionados ao componente
User	Pessoa/coisa que usa o componente

Tabela 2.2: Facetas do Modelo de Classificacão de Vitharana [Padmal Vitharana, 2003] para Componentes de Negócio

em alguns sistemas de RI podem ser expandidas, agregando novos termos aos originalmente inseridos. O objetivo dessa expansão é obter uma classificação mais apurada para os documentos recuperados em termos de relevância [Romanovsky et al., 2005].

Nos sistemas RI existem duas fases:

1. **Processo de Indexação**: a coleção de documentos é indexada para melhorar o desempenho da recuperação;
2. **Processo de Recuperação**: dada uma consulta, os documentos que apresentarem uma maior similaridade serão recuperados.

O principal elemento do processo RI é a função de busca, que compara as representações internas dos documentos com as expressões de busca do usuário e recupera os itens que supostamente fornecem a informação desejada.

Segundo Vitharana [Padmal Vitharana, 2003], o construtor dos componentes pode classificar, codificar e armazenar componentes em um repositório de componentes baseado em conhecimento (CKBR), e pode consultar o CKBR para encontrar componentes que combinem determinados requisitos durante o processo de integração. Desta forma, pode-se buscar componentes através de identificadores estruturados tais como, nome, indústria e ambiente de desenvolvimento, reduzindo a escolha a um número menor de componentes candidatos.

Para alguns autores, um banco de dados é eficiente para armazenar, pesquisar e recuperar identificadores estruturados de componentes e, dessa forma, as buscas podem ser feitas por uma lista de opções de comandos. Porém, essa busca pode ser refinada usando o conhecimento semi-estruturado, codificado como facetas de componentes, por exemplo, regras, papéis e funcionalidades. Um descritor de facetas pode conter palavras-chaves, pequenos textos descritivos

ou pode assumir valores nulos. Uma linguagem de marcação como XML pode ser apropriada para codificar conhecimento baseado em facetas, pois possibilita manutenção de esquemas para codificar uma base semi-estruturada de conhecimento do componente e suportar a busca de componentes.

A literatura apresenta alguns esforços mais recentes na busca de um mecanismo eficiente para recuperar componentes armazenados em um repositório. O método apresentado em Guo [Guo, 2000], usa o conceito de aspectos de componentes para indexar e recuperar componentes baseados em suas características, o qual possibilita que consultas sejam construídas de forma automática, baseadas no contexto de reuso do componente e novos componentes podem ser adicionados ao repositório com indexação automática, gerada a partir do um alto nível de caracterização dos aspectos de um componente. Aspectos descrevem os serviços requeridos e fornecidos de um componente e as restrições não-funcionais como, o fornecimento de interface de usuário, distribuição e gerenciamento de persistência, suporte ao trabalho colaborativo, processo de segurança e transação, gerenciamento das relações entre componentes.

O Odyssey Mediator Layer(OML) apresentado por Braga [Regina Braga, 2001], adota ontologias para possibilitar uma busca mais exata das informações sobre componentes. Essa abordagem visa fornecer flexibilidade, transparência e exatidão no processo de recuperação. As ontologias serão explicadas resumidamente na seção a seguir.

2.6.1 Ontologias

Muitos trabalhos na área de inteligência artificial vêm explorando o uso de ontologias como uma maneira formal para especificar informações conceituais de um domínio, compartilhar e reusar conhecimento. O potencial do uso de ontologias para lidar com o problema da semântica de recursos de informação sobretudo quando há grandes volumes de informação, tem sido largamente explorado pelas áreas de pesquisa da Web Semântica e da gerência de Conhecimento [Van Harmelen, 2003].

Para Grubber [Gruber, 1995] uma ontologia é uma especificação explícita dos objetos, conceitos e outras entidades que assumimos existirem em uma área de interesse, além das relações entre esses conceitos e restrições expressados através de axiomas. Essa especificação é dita formal, explícita e compartilhada. O termo formal significa que pode ser processado por computador; o caráter explícito denota que os conceitos utilizados e as restrições a ele aplicadas são prévia e explicitamente definidos; e, por fim, compartilhamento refere-se a um conhecimento consensual,

utilizado por mais de um indivíduo e aceito por um grupo.

Segundo [Gómez-Pérez and Benjamins, 1999], os princípios básicos para o para o desenvolvimento de uma ontologia são:

- **Clareza e objetividade**: os termos devem ser acompanhados de definições objetivas e também de documentação em linguagem natural;
- **Completeza**: uma definição deve expressar as condições necessárias e suficientes para expressar um termo, indo além das necessidades circunstanciais de uma aplicação;
- **Coerência**: para permitir derivar inferências que sejam consistentes com as definições;
- **Extensibilidade monôtonica**: para permitir a inclusão de novos termos sem revisão das definições existentes;
- **Mínimo compromisso ontológico**: para permitir que sejam definidas tão poucas suposições quanto possíveis sobre o mundo a ser modelado, permitindo que as especializações e instanciações da ontologia sejam definidas com liberdade;
- **Princípio da distinção ontológica**: as classes definidas na ontologia devem ser disjuntas, sem superposição de conceitos;
- **Diversificação das hierarquias**: para aproveitar ao máximo os mecanismos de herança múltipla;
- **Modularidade**: para minimizar o acoplamento entre os módulos;
- **Minimização da distância semântica entre conceitos similares**: para agrupá-los e representá-los utilizando as mesmas primitivas;
- **Padronização**: adoção da padronização dos nomes sempre que possível.

Sistemas baseados em conhecimento que possuem a formalização de seu conhecimento representado através de uma ontologia, podem optar ou não por mapeá-la para o esquema de banco de dados, uma vez que o nível de conhecimento é totalmente independente do nível de implementação. Para Guarino [Guarino, 1997], existe uma similaridade entre ontologia e modelo conceitual em banco de dados convencionais, pois pode-se fazer uso da ontologia para a representação do esquema relacional através de uma ontologia que servirá como esquema do banco de dados.

Segundo Novello [Novello, 2003], com a utilização de ontologias é possível definir uma infra-estrutura para integrar sistemas inteligentes no nível do conhecimento, trazendo grandes vantagens, como:

- **Colaboração**: possibilitam o compartilhamento do conhecimento entre os membros interdisciplinares de uma equipe;
- **Interoperação**: facilitam a integração da informação, especialmente em aplicações distribuídas;
- **Informação**: podem ser usadas como fonte de consulta e de referência do domínio;
- **Modelagem**: as ontologias são representadas por blocos estruturados que podem ser reusáveis na modelagem de sistemas no nível de conhecimento.
- **Busca baseada em ontologia**: recuperar recursos desejados em bases de informações estruturadas por meio de ontologias. Dessa forma, a busca torna-se mais precisa e rápida, pois quando não é encontrada uma resposta exata à consulta, a estrutura semântica da ontologia possibilita, ao sistema, retornar respostas próximas à especificação da consulta.

Na busca e recuperação de componentes, o uso de semântica permite a resolução de alguns problemas existentes nos métodos de classificação tradicionais e ainda, permite o uso de linguagens mais naturais. Sugumaram [Vijayan Sugumaran, 2003] propõe uma abordagem que faz uso de domínios, processos, ações, atores e ontologias de termos de domínios de aplicações. Com essa abordagem é possível criar consultas usando linguagem natural através de uma interface Web. Segundo essa abordagem, as consultas são processadas fazendo uso de ontologias de domínio e as buscas são feitas em repositórios de componentes.

2.7 Conclusão

Neste capítulo foram apresentados os principais conceitos envolvidos no tema classificação e busca de componentes.

Um dos maiores desafios da área de engenharia de *software* é produzir sistemas corporativos dentro de prazos curtos, orçamentos limitados e que sejam necessariamente flexíveis e tolerantes a falhas. Como resposta às crescentes pressões, novos métodos e tecnologias estão sendo criados. As pesquisas caminham na direção de criação de componentes com alto grau de reutilização, técnicas que facilitem sua integração nos sistemas e estudos de mecanismos que facilitem e tornem eficientes o processo de classificação e busca.

Foi mostrado também a importância das interfaces no processo de integração e comunicação dos componentes. Entretanto, não foi encontrada na literatura nenhuma pesquisa que apresentasse uma solução de recuperação que manipulasse componentes Java (incluindo suas interfaces). Este trabalho cobre essa deficiência e apresenta uma ferramenta que faz classificação e busca manipulando componentes EJB.

No próximo capítulo serão explicados os termos relacionados ao mecanismo de tratamento de exceções.

Capítulo 3

Tratamento de Exceções

Durante a execução do sistema situações inesperadas podem ocorrer fazendo com que o sistema assuma um comportamento anormal ou até mesmo tenha seu funcionamento interrompido. Um sistema confiável precisa saber lidar com os problemas que ocorrerem, mantendo assim o seu funcionamento correto. Este capítulo vai explicar conceitos relacionados a tratamento de exceções, seu funcionamento na linguagem Java , os problemas existentes dentro do contexto de componentes, citando várias pesquisas que estão sendo feitas na área para minimização de tais problemas.

3.1 Exceções

Exceções é um mecanismo usado por várias linguagens de Programação para descrever o que deve ser feito quando algo inesperado ocorrer durante a execução do programa. Quando algum problema ocorre o programa lança uma Exceção a qual é capturada e tratada, o fluxo de execução é redirecionado de forma a tratar o problema que ocorreu e depois, se possível, continuar.

Exemplos comuns de Exceções são:

– Índice de uma lista (Array) fora do intervalo permitido.

– Problemas em operações aritméticas, tais como "overflows"e divisões por zero.

– Argumentos inválidos numa chamada a um método.

– Uso de uma referência que não aponta para nenhum objeto.

– Falta de memória

- Falha na conexão com o Banco de Dados

O Tratamento de Exceções fornece ao sistema uma melhor tolerância às falhas. Durante a codificação é possível isolar o código responsável pelo tratamento do erro em blocos separados, deixando o código mais limpo e fácil de manter. Portanto, o objetivo dos mecanismos de Tratamento de Exceções é garantir robustez com o isolamento do código e uma vez que as Exceções em uma aplicação possuem tratamento adequado, garantir que o sistema não será encerrado inesperadamente sem informar qual o real motivo de sua finalização.

3.2 Tratamento de Exceções em Java

Na linguagem de programação Java as exceções podem ser subdivididas em [Microsystems, 2002, da Silva Xavier, 2008]:

- **Síncronas**: ocorrem em pontos bem determinados do programa e são geradas através da avaliação de expressões, chamadas a métodos ou através da instrução *throw*. Esta categoria de exceções são classificada em diversos tipos:
  * Verificadas (*checked*): são verificadas pelo compilador e devem ser tratadas pelo programador. Isto significa que o compilador garante que para cada exceção lançada existe uma estrutura que faz a sua captura. São geradas a partir de condições externas durante o funcionamento do programa. Falha na rede, tentativa de abrir arquivo inexistente são exemplos desse tipo de exceção.
  * Não-Verificadas (*unchecked*): não são verificadas pelo compilador. Isto significa que estas exceções podem ser lançadas no código sem que seja obrigatória a criação de uma estrutura para a sua captura. São geradas em contextos que representam *bugs* do sistema ou em situações consideradas fatais, muito difícil de ser tratada pelo programador.
  * Implícitas: são aquelas lançadas por uma chamada a uma sub-rotina ou pelo ambiente de execução.
  * Explícitas: lançadas pelo desenvolvedor através do comando *throw*.
- **Assíncronas**: ocorrem em pontos não determinísticos do programa, geradas pela JVM ou por outras *threads* em processos concorrentes através da operação *stop()*.

3.2.1 Principais classes de Exceções

De acordo com [Microsystems, 2002] as principais classes que representam as exceções em Java são:

- *java.lang.Exception* - é a classe básica que representa as exceções verificadas, todos os tipos de exceções que devem ser tratadas derivam dessa classe. Exemplo: *NumberFormatException*.
- *Error* - é classe das exceções não verificadas que representa os erros sérios que ocorrerem. Não se espera que essas exceções sejam tratadas e na maioria das vezes quando esses erros ocorrem é necessário a finalização do programa. Exemplo: problemas com a jvm (java virtual machine).
- *RuntimeException* - é a classe de exceções não-verificadas que representam *bugs* do sistema. Quando essas exceções ocorrem o código do programa deve ser revisto e o problema consertado pelo programador. Exemplo: *ArrayIndexOutOfBoundsException*. Todas as classes acima derivam da classe *Throwable* a qual captura a mensagem de erro e imprime os métodos envolvidos.

3.2.2 Lançamento e Captura de Exceções

Quando uma exceção ocorre no programa o método que encontra a exceção pode tratá-la ou lançá-la de volta ao chamador desse método que por sua vez pode relançar a exceção ou trata-la. Se a exceção não for tratada em nenhum ponto do programa, ela chegará ao método principal *main*() e o programa será finalizado de forma anormal sendo exibida então a mensagem de que uma determinada exceção foi gerada.

Esse mecanismo de relançar ou tratar a Exceção permite que o programador possa escolher fazer o tratamento da exceção na parte em que achar mais conveniente. Através da API do Java, o programador tem conhecimento de quais exceções são lançadas pelos métodos das classes. Para tratar as exceções é utilizado o bloco *try* juntamente com o(s) bloco(s) *catch*, existindo um bloco *catch* para cada exceção a ser tratada. É possível ter um bloco *try* com vários blocos *catch*, cada um tratando um tipo diferente de exceção. As instruções dentro do bloco *catch* somente serão executadas se a exceção lançada coincidir com aquela definida no *catch*. A Figura 3.2 mostra a estrutura *try/catch*.

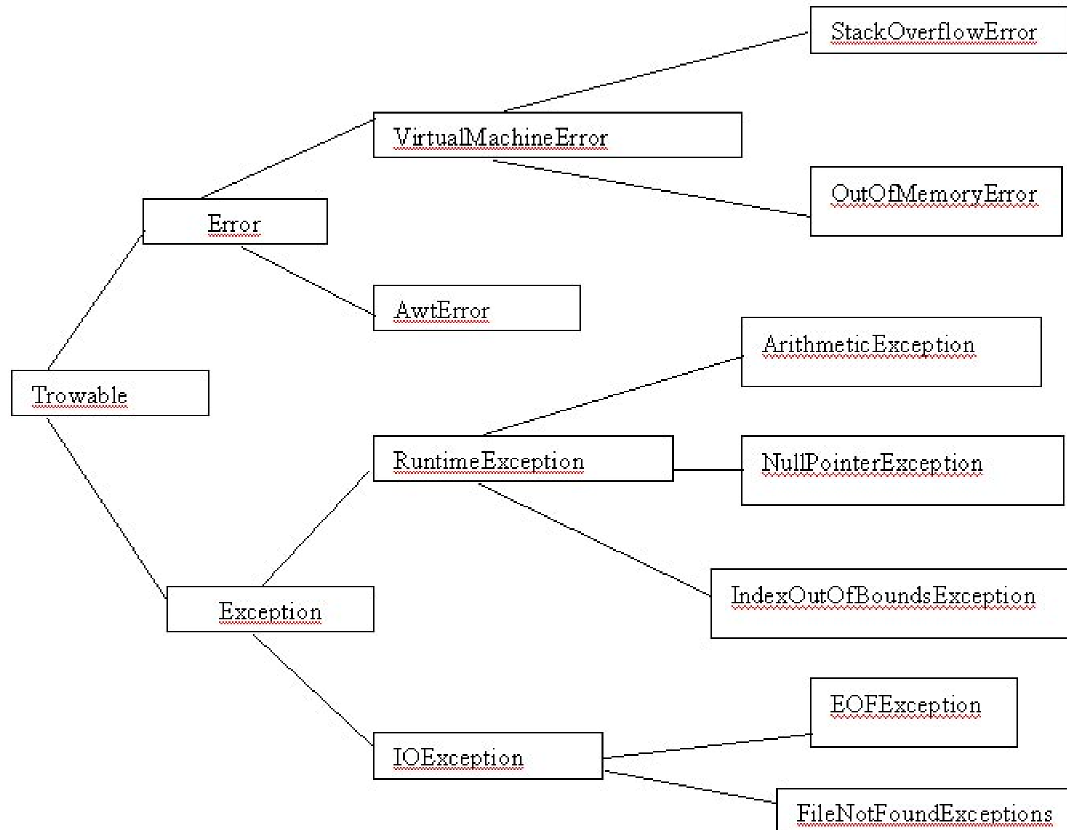

Figura 3.1: Hierarquia de Exceções em Java

Opcionalmente, existe ainda a cláusula *finally* que define um bloco de código que sempre é executado independente se a exceção foi tratada ou não. Isso permite que alguns procedimentos como fechamento de conexões e arquivos sejam executados antes do programa finalizar. O sistema de hierarquia das Exceções permite que ao declarar uma exceção na cláusula *catch* todas as Exceções descendentes sejam pegas também. Por exemplo, se for declarado no *catch* o *IOException*, as suas sublasses *EOFException* e *FileNotFoundException* também serão tratadas. Outra opção fornecida pela linguagem Java é relançar as exceções, nesse caso o tratamento através do *try/catch* não é necessário e teríamos: Segundo Malayeri [Donna Malayeri, 2005] o mecanismo de tratamento de exceções se por um lado, traz a vantagem de separação do

```
public void calcular() {
    try{
            //Código que pode gerar Exceções do tipo A e B
    }catch(ExceptionA e1){
            //Código para tratar a Exceção A
    }catch(ExceptionB e2){
            //Código para tratar a Exceção B
    }
}
```

Figura 3.2: Estrutura de tratamento de exceções em Java

```
public void calcular() {
    try{
            //Código que pode gerar Exceções do tipo A e B
        }catch(ExceptionA e1){
            //Código para tratar a Exceção A
        }catch(ExceptionB e2){
            //Código para tratar a Exceção B
    }finally{
            //Código que deve ser executado independente
            // se o bloco try ou catch foi executado.
    }
}
```

Figura 3.3: Tratamento de Exceções em Java usando finally

```
public void calcular() throws ExceptionA, ExceptionsB {
    //Código que pode gerar Exceções do tipo A e B
}
```

Figura 3.4: Relançamento de Exceções em Java

código funcional do código de tratamento de erros, traz também como conseqüência inúmeros problemas. As declarações das exceções em Java através do *throws* são de muito baixo nível e políticas de exceções somente podem ser especificadas em nível de método. Esse fato traz uma sobrecarga na fase de especificação e aborrecimentos na escrita e manutenção das declarações. Modificações simples no código, por exemplo, um método lançando um novo tipo de exceção ou mover um tratador de um método para outro, poderia resultar na atualização das declarações de toda uma cadeia de chamadas àquele método alterado.

Conseqüentemente, uma prática comum é os programadores declararem o método com *throws Exception* , pois através do polimorfismo o novo tipo de exceção não precisaria ser declarado. É comum também deixarem vazios os locais de tratamento da nova exceção (blocos do *catch*). Essas práticas devem ser desencorajadas porque tornam o código ruim.

Algumas ferramentas de desenvolvimento como, por exemplo, o Eclipse, possuem mecanismos que servem para atualizar as declarações *throws* ou mesmo, obrigar o desenvolvedor a criar estruturas de tratamento das exceções, mas esses procedimentos são feitos individualmente por método e, não raro, os blocos de tratamento *catch* se encontram vazios.

3.3 As más práticas no tratamento de exceções

Um sistema que utilize um tratamento de condições excepcionais eficiente, tem sua manutenibilidade, robustez e confiabilidade melhoradas. Por isso, pesquisas têm sido feitas com o objetivo de detectar propriedades e más práticas relacionadas a exceções dentro do código-fonte. [da Silva Xavier, 2008] em seus estudos, mostra um conjunto de práticas relativas à falta de tratamento de exceções que dificultam a manutenção do sistema:

– **lançar *java.lang.Exception* na assinatura do método** - essa exceção é muito genérica, pertence ao início da hierarquia das exceções (3.1). Por isso, não fornece informações úteis sobre os erros que podem ocorrer no processamento do método.

- **lançar um grande número de exceções na assinatura do método** - essa prática polui muito o código e só é válida no caso onde cada exceção representa uma falha diferente, cada uma necessitando assim de um tratamento diferenciado.
- **declarar uma exceção na assinatura do método que nunca é lançada** - só é válida na definição de métodos abstratos. Tal prática obriga o desenvolvedor a capturar exceções que nunca serão lançadas.
- **ignorar a exceção** - torna o código menos robusto, pois exceções não-tratadas podem provocar o término anormal da execução sem a possibilidade de recuperação oferecida pelo mecanismo de tratamento de exceções. Essa prática ocorre quando todos os métodos onde possa ocorrer uma determinada exceção têm declarado na assinatura o relançamento da mesma através do termo *throws*. As exceções do tipo *RuntimeException*, por não serem veficadas pelo compilador, normalmente são ignoradas.
- **capturar e ignorar *InterruptedException*** - essa exceção costuma estar relacionada a código de programas concorrentes e serve para notificar a thread de que ela deve cessar o seu processamento atual. Ignorar tal exceção pode fazer com que a thread fique em execução quando deveria parar.
- **registrar no arquivo de Log e relançar a exceção** - pode gerar inúmeras mensagens de erro para a mesma exceção, dificultando o trabalho de manutenção.
- **registrar no arquivo de Log e retornar null** - não permite diferenciar quando o valor retornado pelo método é legítimo ou resultado de uma exceção.
- **encapsulamento destrutivo** - ocorre quando a exceção é relançada mudando o tipo, sem armazenamento da exceção original. Os dados da exceção original são perdidos.
- **capturar *java.lang.Exception*** - essa prática fornece um tratamento padrão para um conjunto de exceções (*java.lang.Exception* e suas exceções descendentes) e pode gerar problemas se o método onde tal exceção é tratada for modificado para lançar novas exceções as quais devem ser manipuladas de modo diferenciado. O trecho de código tratador não ficará ciente de que existe esta nova exceção.
- **lançar exceções dentro do bloco *finally*** - uma exceção lançada no bloco *finally* pode sobrescrever uma exceção gerada originalmente, perdendo os seus dados.

3.4 Tratamento de Exceções em DBC

O modelo de abstração do paradigma orientado a objetos e do DBC são diferentes, em vista disso, o DBC possui algumas particularidades que não são totalmente satisfeitas pela linguagem de programação orientada a objetos [Brito, 2005]. Serão abordados nas seções a seguir o tratamento de exceções dentro do contexto de componente e dentro do contexto de arquitetura.

3.4.1 Tratamento de Exceções no Componente

Uma das dificuldades na construção de sistemas tolerantes a falhas baseados em componentes é a falta de separação entre o tratamento excepcional e a implementação das funcionalidades do sistema, prejudicando o seu entendimento e a sua reutilização [Chris Leer, 2001].

Nos componentes tolerantes a falhas que implementam mecanismos internos para o tratamento de exceções, as respostas são divididas em duas categorias distintas: normais e excepcionais. Respostas normais são produzidas quando o componente realiza corretamente o serviço requisitado. Respostas excepcionais são aquelas produzidas quando algum evento anormal ocorre durante a execução do serviço. Ou seja, as respostas excepcionais de um componente correspondem às exceções [Ferreira, 2001, Pereira, 2007].

O processamento do componente tolerante a falhas deve ser particionado em atividade normal e atividade excepcional como mostra a Figura 3.5. A atividade normal implementa os serviços especificados para o componente. A atividade excepcional implementa as rotinas de tratamento para as exceções lançadas. Dessa forma, o tratamento de exceções é considerado uma técnica essencial para a estruturação interna do componente tolerante a falhas porque introduz uma separação clara entre a execução normal do código e o tratamento das condições excepcionais. As exceções dos componentes podem ser classificadas como [Ferreira, 2001, Pereira, 2007]:

- **Exceções Internas**: As exceções internas ocorrem quando o componente detecta uma situação inesperada durante a atividade normal. Após o lançamento de uma exceção interna, o próprio componente realiza o seu tratamento, retornando à atividade normal logo em seguida.
- **Exceções de Defeito**: Caso não seja possível a recuperação da exceção interna, o componente lança uma exceção de defeito indicando que não foi possível realizar o serviço solicitado. Este tipo de exceção também é denominado de exceção externa pois as atividades relacionadas ao tratamento da mesma são externas ao componente que a lançou.

– **Exceções de Interface**: São lançadas por um componente quando ele recebe solicitações que não estão em conformidade com sua especificação. Normalmente, este tipo de exceção é provocado por uma falha na combinação entre os componentes na montagem do sistema.

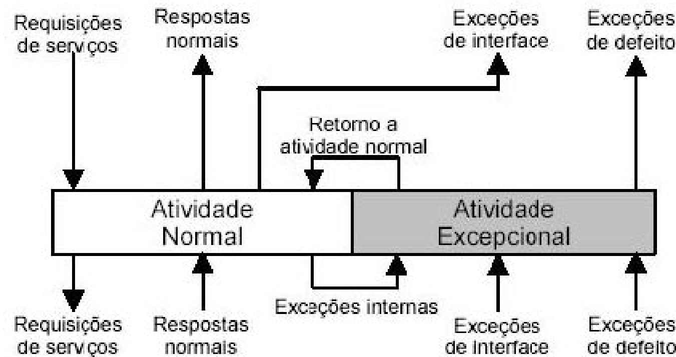

Figura 3.5: Componente tolerante a falhas [Ferreira, 2001]

3.4.2 Tratamento de Exceções na Arquitetura

Além da separação da parte excepcional e normal do componente, uma outra necessidade é a preocupação com a propagação de exceções de acordo com o fluxo interativo entre os componentes dentro da arquitetura do sistema. Por exemplo, pode ser necessário converter os tipos de algumas exceções entre componentes por estes adotarem modelos de falhas distintos [Ferreira, 2001, Souchon et al., 2003].

É possível que componentes distribuídos que façam parte de um sistema, lancem exceções de maneira autônoma, o que pode significar um estado errôneo diferenciado, que não é equivalente à soma do tratamento das exceções lançadas inicialmente [Ferreira, 2001, Souchon et al., 2003]. Assim, se faz necessário a execução de uma função que a partir de um conjunto de exceções lançadas, possibilite a descoberta do seu tratador equivalente.

Pesquisas caminham no sentido de especificar, tratar e coordenar as exceções no nível arquitetural.

Guerra [Guerra, 2004] defende que tanto a conversão dos tipos de exceções que muitas vezes é necessária quanto o tratamento que envolve análise do fluxo excepcional de mais de um

componente devem ser feitas na arquitetura do sistema. Daí, a importância dos conectores arquiteturais, pois através deles é possível:

1. Gerenciar o fluxo excepcional dos componentes
2. Implementar heurísticas para melhorar a escolha do tratador apropriado
3. Possibilitar o uso de componentes de *software* de prateleira, também chamados de COTS (*Common Of The Shelf*) através das possíveis conversões entre os tipos de dados.
4. Implementar o tratamento de falhas através do isolamento dos componentes defeituosos.

Brito [Brito, 2005] em seu trabalho, define alguns requisitos como sendo essenciais no projeto do comportamento excepcional de qualquer sistema baseado em componentes. São eles:

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

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

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

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

Considerando como base o mecanismo de tratamento de exceções da linguagem Java, Malayeri [Donna Malayeri, 2005] sugere definir as informações de exceções usando dois níveis de abordagem. A primeira seria no nível do componente, definindo um conjunto limitado de exceções genéricas que podem ser geradas em cada componente. O componente ficaria restrito a lançar apenas esse conjunto limitado de exceções. Reduzir o número de exceções lançadas é importante na medida em que se diminui também o número e a complexidade dos tratadores dessas exceções no processo de integração do componente na arquitetura.

A segunda abordagem abrange o nível intra-componente, a proposta seria simplificar o desenho das exceções existentes nos componentes. Isso seria efetuado através de um re-mapeamento significativo das exceções internas para as exceções genéricas.

Contudo, a aplicação dessas abordagens podem trazer os seguintes problemas: ambiguidade na semântica da exceção, confusão no uso das exceções definidas pela abordagem, ilimitação das exceções não verificadas. Tais problemas decorrem de dois fatores principais segundo Malayeri [Donna Malayeri, 2005]: o fato do tratamento das exceções ser um fenômeno global e o fato de ser difícil e custoso antecipar todas as categorias de problemas que podem ocorrer e como eles podem ser tratados durante a fase de design do Componente.

3.5 Conclusão

Neste capítulo foram apresentados os conceitos relacionados ao mecanismo de tratamento de exceções e seu funcionamento na linguagem Java. Foram mostrados os problemas de exceções existentes no contexto do paradigma DBC e a necessidade de medidas e técnicas que verifiquem a qualidade no tratamento das situações de exceções dos componentes.

Apesar do mecanismo de exceções fornecer a base para a estruturação do código-fonte tratador das situações de funcionamento anormal, ainda hoje, apenas uma pequena quantidade de informações está disponível no componente para guiar o desenvolvedor a usar as exceções de uma forma apropriada.

No próximo capítulo é apresentado um modelo de classificação e busca de componentes que supre tais necessidades.

Capítulo 4

Modelo de Classificação e Busca com Tratamento de Exceções

A Classificação e Busca nos repositórios pode ser enriquecida com a inserção e extração de informações relacionadas ao comportamento excepcional dos componentes. Informações precisas, legíveis e simples a respeito das exceções exercem papel fundamental no processo de escolha e uso adequado do componente. O objetivo deste capítulo é apresentar o modelo de classificação e busca baseado em facetas e ontologia para as informações de exceções dos componentes EJB armazenados em um repositório.

4.1 O Contexto

Os sofisticados mecanismos de tratamento de exceções das linguagens de programação modernas como Java, C++, C freqüentemente são empregados de forma incorreta [Donna Malayeri, 2005].

O fato dos desenvolvedores concentrarem seus esforços principalmente no projeto do comportamento normal do sistema que atendam os requisitos funcionais e os curtos prazos de entrega favorecem a negligência do comportamento excepcional dos componentes. Além disso, é preciso também considerar duas situações que dificultam a estruturação do comportamento excepcional em DBC:

- Componentes reutilizáveis são desenvolvidos sem se conhecer todos os contextos nos quais serão integrados. Dessa forma, as hipóteses de falhas assumidas durante a criação de um Componente reutilizável podem não ser válidas dentro de um determinado contexto de um novo sistema.

– Normalmente, a especificação de um componente não descreve a estratégia adotada no tratamento de exceções do componente, resultando em um comportamento excepcional "*ad hoc*"e totalmente dependente da codificação do Componente.

Por isso, durante o processo de integração dos Componentes no sistema é comum surgir conflitos:

1. Entre o comportamento excepcional dos componentes reutilizados
2. Entre as hipóteses de falhas do novo sistema e a estratégia para tratamento de exceções planejada para o mesmo.

Alguns exemplos desses conflitos:

– **Incompatibilidade dos tipos de exceções lançados pelos componentes**: um componente A pode lançar uma exceção de um tipo diferente daquele que é esperado por um componente B responsável pelo tratamento da condição excepcional sinalizada por A.
– **Existência de condições excepcionais não antecipadas**: ou seja, no contexto de um novo sistema, uma condição excepcional que não foi prevista no desenvolvimento daquele componente que está sendo reutilizado.
– **Tratamento de exceções inadequado**: quando o tratamento de uma exceção do componente reutilizado é incompatível com a estratégia adotada no sistema em construção.

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 [Guerra, 2004].

Porém, 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 [Guerra, 2004]. Sendo assim, é importante para o desenvolvedor conhecer detalhes do comportamento excepcional antes do uso do componente.

4.2 O Modelo de Classificação de Componentes

4.2.1 Comportamento Normal

O modelo apresentado por [Melo, 2006] está relacionado à atividade normal do componente e consiste nas seguintes facetas e atributos:

1. **Facetas**:
   - *Application Domain*: possíveis contextos de utilização/integração de um componente. (Preenchimento manual)
   - *Role*: papéis que o componente pode (potencialmente) assumir uma variedade de aplicações. (Preenchimento manual)
   - *Rule*: Processos de negócio ou funções em um domínio particular que o componente oferece suporte. (Preenchimento manual)
   - *Function*: Funções ou tarefas executadas pelo componente e seus métodos. (Preenchimento automático)
   - *Element*: Representam parte do domínio de aplicação, sendo objetos manipulados nele. (Preenchimento automático)
   - *Action*: Ações e eventos que o componente provê. Preenchimento automático.
   - *User*: No contexto de negócio, o usuário pode ser caracterizado como o usuário final de uma aplicação baseada em componentes. (Preenchimento manual)
   - *Test*: Indica a presença de testes para componente (unitários ou de integração) por meio de um mecanismo de inclusão. (Preenchimento manual)
   - *Quality*: Como mecanismo de garantia/medição da qualidade do componente, essa faceta permite a descrição de métricas de qualidade de software que estejam disponíveis. (Preenchimento manual)
2. **Atributos**: (todos de preenchimento manual)
   - *Identification(ID, name)*: o ID é utilizado para garantir a unicidade do componente no repositório.
   - *Management(owner, contact, version, version date)*: identifica o responsável pelo desenvolvimento do componente, versão e forma de contato.

  - *Environment(Operation System, plattform, programming language/tools)*: informações sobre o ambiente são importantes para descrever aspectos não-funcionais, como portabilidade ou restrições de compilação.
  - *Source Code Access*: Indica se o código-fonte do componente classificado será disponibilizado.
  - *Maturity (age, number of reuses)*: A idade do componente e o número de reusos dão a noção de estabilidade ou maturidade do componente.

Como visto acima no modelo de [Melo, 2006], a maioria das informações de classificação são inseridas manualmente. Foi observado também que melhorias poderiam ser feitas em alguns pontos e assim, elas foram inseridas na solução desse trabalho. São elas:

- **alteração para manipular componentes EJB**: o modelo antigo considera classes Java como componente, ou seja, limita-se a classificar e recuperar somente classes Java. Isso não é muito bom, pois se um usuário deseja reutilizar um componente real, irá precisar do pacote completo, incluindo as interfaces.
- **extração automática de um maior conjunto de informações**: serão extraídas dos componentes outras informações também: interface provida, interface requerida, data de registro do componente e pacote em que pertence.

4.2.2 Comportamento Excepcional

No mecanismo para tratamento de exceções [Cristian, 1995], a detecção de um erro durante a execução do código normal é informada através do lançamento de uma exceção e a recuperação do erro é feita por um determinado tratador de exceção. Assim, enquanto o código normal descreve o comportamento normal do sistema na ausência de falhas, o conjunto dos tratadores de exceções descrevem o comportamento excepcional do mesmo.

A solução deste trabalho tem como foco as informações de exceções dos componentes, porém, o modelo analisa e armazena também trechos de código normal do componente, pois eles representam o contexto de lançamento das exceções.

4.3 O modelo de facetas, atributos e ontologia

Após várias pesquisas na literatura de classificação e busca de componentes não foi encontrado nenhum modelo que considerasse também o comportamento do componente em situações de

falhas.

Sendo assim, um modelo novo baseado em facetas, atributos e ontologia foi criado com o intuito de minimizar os conflitos de exceções citados anteriormente na Seção 4.1

4.3.1 As Facetas e os Atributos

A classificação facetada permite que um mesmo objeto possua diversas classificações, permitindo assim a busca e a navegação por classes que formam grupos de objetos. Ela difere de uma classificação tradicional, pois a mesma não atribui entalhes fixos aos assuntos em seqüência, mas utiliza uma definição clara, mútua exclusividade, e aspectos coletivos, propriedades ou características de uma classe ou assunto específico. Tais aspectos, propriedades, ou características são chamados de facetas de uma classe ou de um assunto, um termo introduzido na teoria da classificação e dado este significado novo pelo bibliotecário e pelo classificador indiano S.R. Ranganathan e usado primeiramente em sua classificação no início dos anos 30 [Ranganathan, 1960].

O modelo é formado por duas facetas e seis atributos. As informações que preencherão as facetas e atributos serão extraídas automaticamente durante a classificação do componente. O desenvolvedor pode querer recuperar apenas componentes que tratem ou lancem determinadas exceções. Dessa forma, as facetas novas permitirão que sejam feitas buscas de componentes por exceção lançada ou tratada. São elas:

O atributo ***context*** apresentará informações a respeito dos trechos de códigos responsáveis pelo lançamento das exceções. Corresponde a blocos de código que representam a atividade normal do componente.

O atributo ***action handling*** conterá as instruções de código de tratamento das exceções. Corresponderá aos trechos de código que representam a atividade excepcional do componente.

Durante a escolha do componente é importante conhecer também a forma como foi implementado o mecanismo de exceções.

O mau uso do mecanismo de exceções afeta a robustez e dificulta a manutenção do componente, por isso, baseando-se nas más práticas definidas na Seção 3.3 o modelo utiliza alguns atributos para armazenar a contabilização de práticas consideradas ruins no uso de exceções.

Os atributos a seguir servirão de medidas que permitirão avaliar a qualidade do código do componente em relação às suas exceções.

- **catch vazios** - essa implementação é colocada, muitas vezes, para permitir que o código compile trechos de códigos que contém operações que lançam exceções. É equivalente a não colocar nenhum tratamento à exceção. Torna o código ruim, pois se ocorrer alguma falha nenhum procedimento alternativo ou de correção será executado.
- **finally vazios** - essa declaração não afeta em nada a execução do código, por isso, se possível, deve ser removida.
- **catch Exception** - essa declaração fará com que todas as exceções sejam tratadas de uma forma padrão. É uma prática pouco recomendada porque dificulta a manutenção do código e corre sério risco de tratar uma determinada exceção de forma errada.
- **throws Exception** - essa declaração em uma operação permite que todas as exceções que aparecerem sejam relançadas para o código chamador da operação. É colocada quando não se quer tratar nenhum tipo de exceção. Esse procedimento não é recomendado porque torna a aplicação menos robusta.

Com esses atributos será possível avaliar a qualidade do código do componente. Um componente de alta qualidade, normalmente, possuirá o valor desses atributos tendendo a zero.

As Tabelas 4.1 e 4.2 mostram as facetas e os atributos do modelo.

Faceta	Descrição
Catch Exceptions List	Lista das Exceções Tratadas pelo Componente
Throws Exceptions List	Lista das Exceções Lançadas pelo Componente

Tabela 4.1: Facetas do Modelo com Exceções

Atributo	Descrição
Action Handling	Ação de Tratamento para exceção capturada
Context	Contexto de lançamento da exceção
Empty Catch	Quantidade de estruturas *catch* vazias
Empty Finally	Quantidadade de estruturas *finally* vazias
Catch Exception	Quantidade de declarações catch Exception
Throws Exception	Quantidade de declarações *throws Exception*

Tabela 4.2: Atributos do Modelo com Exceções

4.4 O Método de Classificação

Foram encontradas poucas informações a respeito de um padrão de especificação de situações de falhas e pelo fato de nem sempre existir comentários *Javadoc* a respeito das exceções, optou-se pela análise de código-fonte do componente para a extração das informações.

A linguagem de programação dos componentes é a linguagem Java da Sun Microsystems e a classificação considera o código-fonte de componentes que seguem o modelo EJB. O método de classificação para a extração das informações de exceções é formado pelas seguintes atividades:

1. **Extração das informações** - é feita através da análise do código-fonte do EJB 2.3 pelo módulo Java Parser (estendido) e pelo módulo Java Parser Exceptions.
2. **Representação das informações** - as informações são processadas e agrupadas em arquivos xml e em dados inseridos em um banco de dados.
3. **Indexação** - as informações extraídas são indexadas pelo *framework* Lucene [Foundation, 2006].
4. **Armazenamento do componente** - o componente EJB é armazenado em um diretório.

A Figura 4.1 mostra as atividades do método de classificação.

4.5 A Busca

As pesquisas atuais em busca e recuperação de componentes têm se concentrado em aspectos e requisitos chaves para os mercados de componentes, que buscam promover o reuso em larga escala [Lucredio et al., 2004]. Contudo, não foi encontrado nenhum trabalho que considere durante a recuperação informações relacionadas às exceções.

Assim, com o objetivo de inserir semântica ao modelo de busca e também melhorar a eficiência, uma ontologia de exceção foi criada e agregada. Utilizando esse modelo o repositório poderá trazer componentes alternativos em situações de insucesso de uma busca exata.

4.5.1 Ontologia de Exceção

Como visto na seção 2.6.1, as ontologias através de propriedades, conceitos e relações, fornecem um meio de lidar com a representação de recursos de informação: o modelo de domínio descrito por uma ontologia pode ser usado como uma estrutura unificadora para dar semântica e uma representação comum à informação [Van Harmelen, 2003].

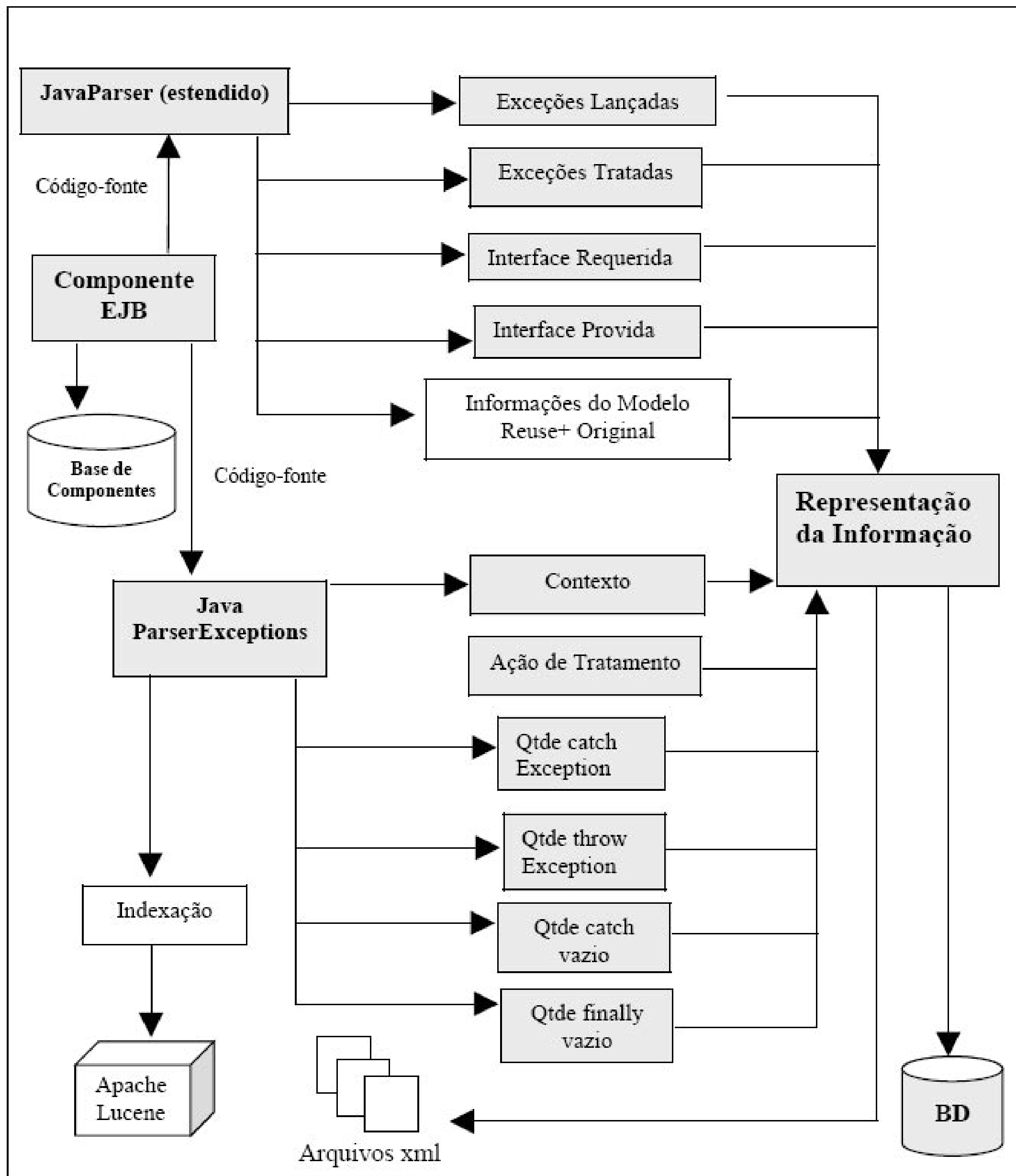

Figura 4.1: Método de Classificação para a extração das informações

A ontologia de exceção foi mapeada para o modelo relacional possibilitando o uso de um Banco de Dados para o armazenamento e posterior recuperação das informações de exceções. O uso desta ontologia permitirá que buscas alternativas sejam efetuadas no caso de insucesso da busca por palavra-chave.

A Figura 4.2 descreve a estrutura da ontologia de exceção adotada no modelo de busca.

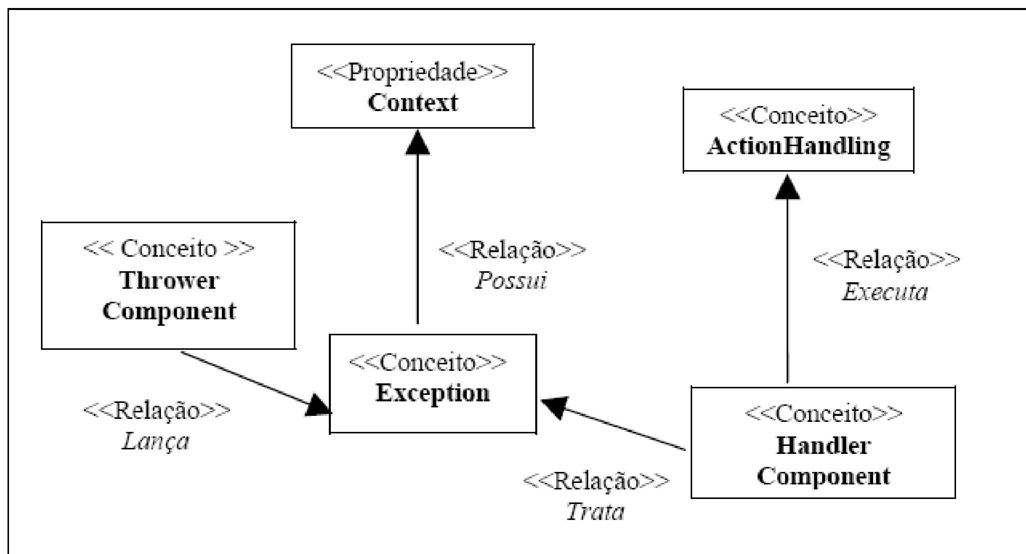

Figura 4.2: Ontologia de Exceção

4.6 Conclusão

O modelo apresentado supre várias necessidades existentes na literatura de componentes:

- armazenamento e exibição das informações de exceções
- busca de componentes por exceções lançadas e tratadas
- manipulação de componentes EJB que é um padrão no mercado
- utilização de semântica relacionada a exceções durante a busca de componentes
- captura de medidas que possibilitem avaliar o código de tratamento de exceções de componentes

Assim, neste capítulo foi apresentado uma solução de classificação e busca que valoriza também as exceções do componente. Foi mostrado o contexto de surgimento do modelo, suas facetas e atributos, a ontologia de exceção e ainda, o método para a extração das informações.

No capítulo a seguir será mostrada a ferramenta Reuse++ que implementou essa solução.

Capítulo 5

O Sistema Reuse++

Nos capítulos anteriores foram apresentados um novo modelo de classificação e um método para a extração automática das informações de exceções que preencherão as facetas e atributos do modelo. Nesse capítulo, será apresentado o protótipo Reuse++ que utiliza o modelo e o método mostrados anteriormente.

5.1 Introdução

O Sistema Reuse++ utiliza um modelo baseado em um conjunto de facetas e atributos referentes a exceções. O armazenamento das informações é feito em uma base de conhecimento representada por arquivos xml. Essas informações são indexadas e depois recuperadas com a ajuda do *framework* Apache Lucene [Foundation, 2006]. A consulta aos componentes pode ser expandida com a ajuda do sistema léxico WordNet [Miller, 1998] que buscará sinônimos para cada termo, realizando com eles novas consultas à base de componentes.

Contudo, para a utilização adequada do componente é importante que o repositório mostre o máximo de informações possíveis não só sobre o funcionamento padrão dos componentes, mas também como são seus comportamentos em situações de falhas.

5.2 O Funcionamento geral

A implementação dos mecanismos de classificação e busca a serem explicados nas próximas seções visam trazer melhorias no processo DBC. Uma vez que a ferramenta passe a fornecer vários detalhes a respeito do comportamento dos componentes nas situações de exceções, espera-se que os usuários consigam fazer também uma escolha mais consciente do componente mais adequado às suas necessidades.

Os componentes armazenados pelo Reuse++ seguem agora o modelo EJB e todas as suas informações de exceções são armazenadas e indexadas também. Durante a classificação são extraídas automaticamente do código-fonte: exceções lançadas, exceções tratadas,o contexto de lançamento das exceções e as ações de tratamento. Após o preenchimento das facetas e atributos do modelo, uma parte das informações de exceções é indexada e armazenada em arquivos xml e a outra parte é transformada em dados que são inseridos no banco de dados MySql.

A tela que exibe os detalhes do componente passou a ter uma referência para outra tela que mostrará todas as informações de exceções extraídas. Além disso, cada exceção pertencente à API do java conterá uma referência para o respectivo javadoc disponibilizado no site da Sun. Assim, se o usuário desejar conhecer mais detalhes de alguma exceção será automaticamente redirecionado ao javadoc dessa exceção.

O processo de busca da ferramenta foi alterado para utilizar também a ontologia de exceção apresentada no capítulo anterior 4.5.1. Com essa modificação, será possível agora recuperar componentes baseando-se também em semântica e assim oferecer componentes alternativos no resultado da busca. Para calcular a relevância dos componentes selecionados, o Reuse++ manteve a mesma estratégia do Reuse+, ou seja, utiliza os seguintes mecanismos:

1. Remoção das palavras com menor valor de busca (normalmente preposições e artigos).
2. Remoção de afixos (stemming). Esse processo tem como objetivo extrair os radicais das palavras. Com isso é garantido que palavras diferentes originadas de um mesmo radical tenham o mesmo tratamento durante a recuperação da informação.
3. Cálculo da similaridade entre a consulta e cada componente classificado, obtendo assim a relevância do mesmo em relação à consulta.
4. Ordenação dos resultados.

A Figura 5.1 mostra o funcionamento geral do Reuse++.

5.3 O Projeto arquitetônico

Como visto na seção 2.1.6, a arquitetura mostra como é o desenho do sistema a partir de um conjunto de componentes arquiteturais e as interações entre eles. Segue abaixo um breve comentário a respeito de algumas decisões relacionadas à arquitetura do projeto.

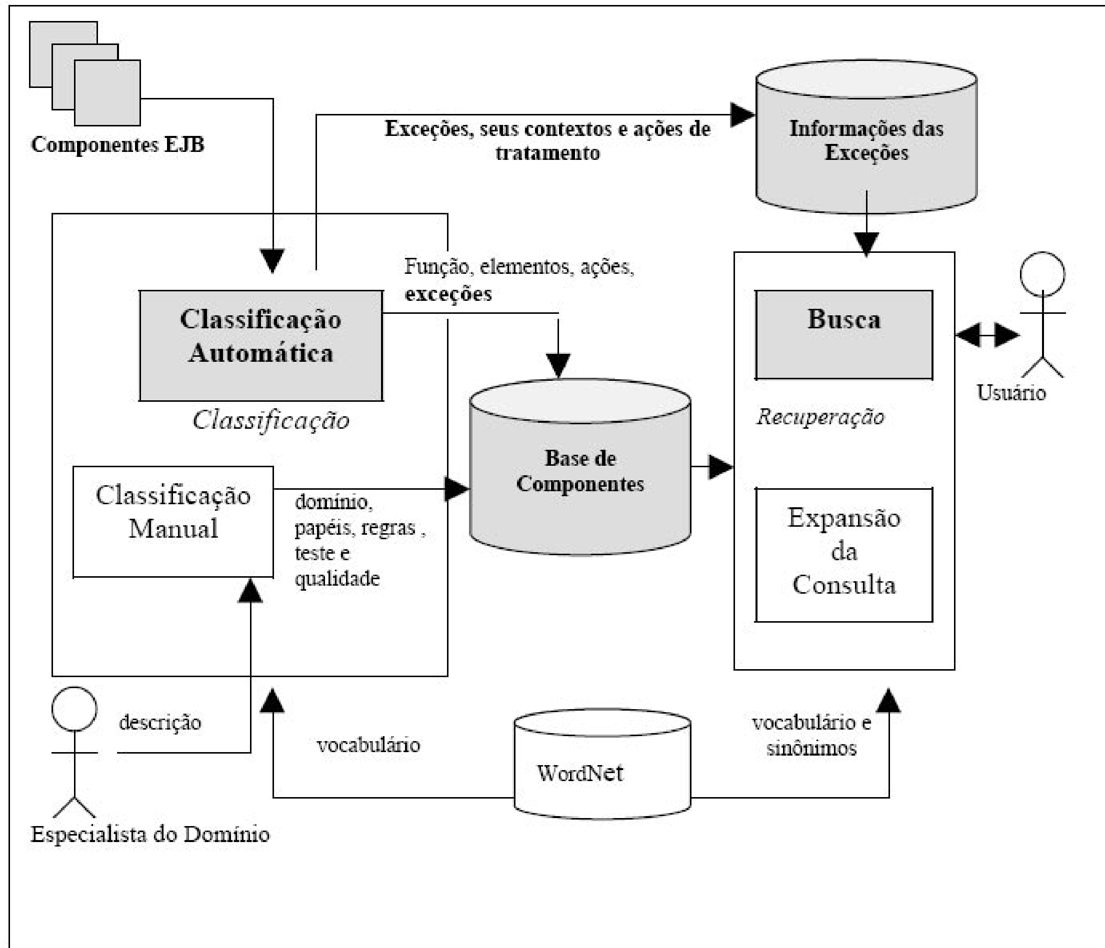

Figura 5.1: Funcionamento Geral do REUSE++.

5.3.1 Plataforma Arquitetural

A escolha da plataforma J2EE foi feita decorrente à grande quantidade de recursos e benefícios que o padrão J2EE oferece para o desenvolvimento de aplicações multicamadas. Dentre eles : integração com sistemas legados, controle transacional, segurança, escalabilidade e independência de plataforma. O servidor escolhido que implementa o padrão J2EE, JBoss versão 4.0, foi utilizado no projeto. A Figura 5.2 mostra a arquitetura do Reuse++.

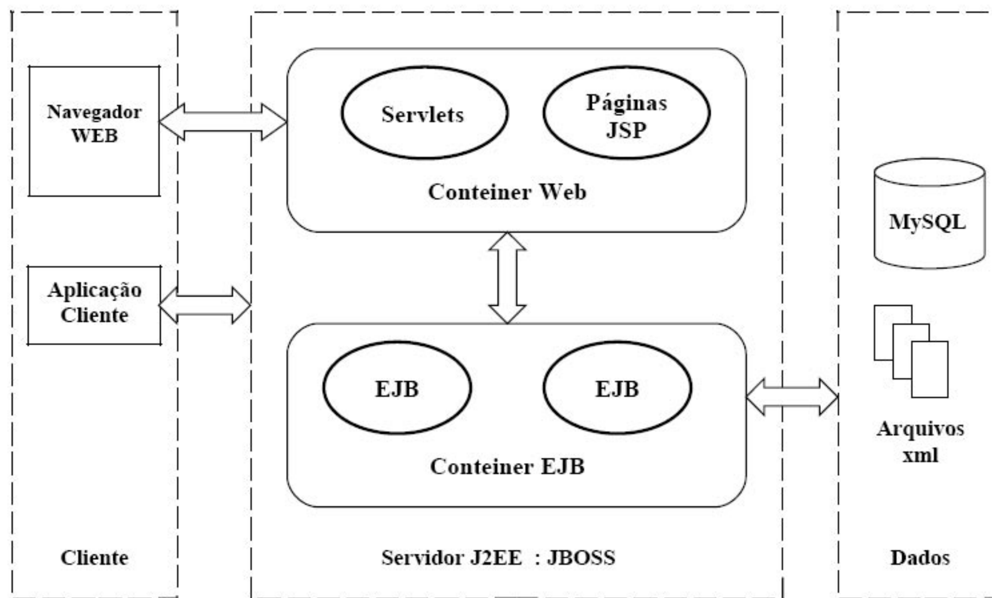

Figura 5.2: Arquitetura J2EE do Reuse++

5.3.2 Camadas Lógicas

O Reuse++ precisa estar preparado para sofrer mudanças em trabalhos futuros e a divisão em camadas vai facilitar essa situação. O padrão MVC(Modelo Visualização Controlador) defende a idéia de organizar o código da aplicação em três camadas de acordo com as responsabilidades.

A camada Modelo agrupa os componentes responsáveis lógica de negócios, a camada Visualização contém os componentes que formam a interface gráfica acessada pelos usuários e o

Controlador é responsável por gerenciar as ações executadas durante o funcionamento do sistema. Esse padrão de projeto facilita a manutenção da aplicação, traz independência entre as camadas e aumenta o grau de reutilização dos componentes envolvidos. Em vista dessas vantagens e por se adequar bem à arquitetura web baseada em requisições e respostas, esse padrão de projeto foi não só mantido, mas melhorado também. A Figura 5.3 mostra o modelo MVC aplicado no Reuse++.

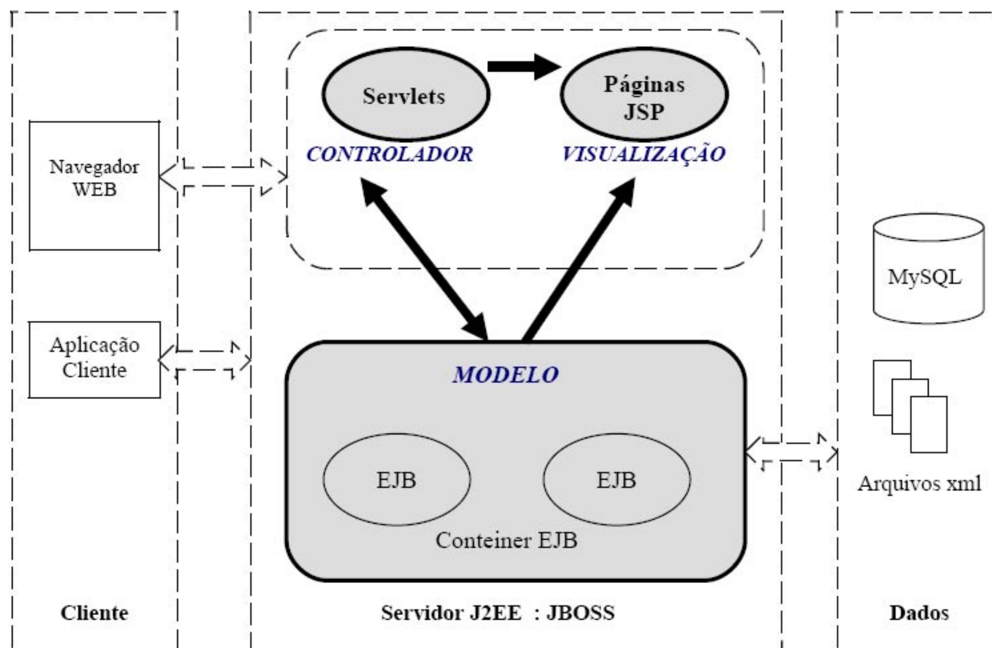

Figura 5.3: Arquitetura MVC do Reuse++

5.3.3 Integração com outras aplicações

O modelo e método propostos neste trabalho não fizeram uso de nenhuma outra aplicação já existente. O Reuse++ faz uso dos *frameworks* Lucene [Foundation, 2006] e Wordnet [Miller, 1998].

A Figura 5.4 mostra em notação UML a topologia do Reuse++ (diagrama de implantação).

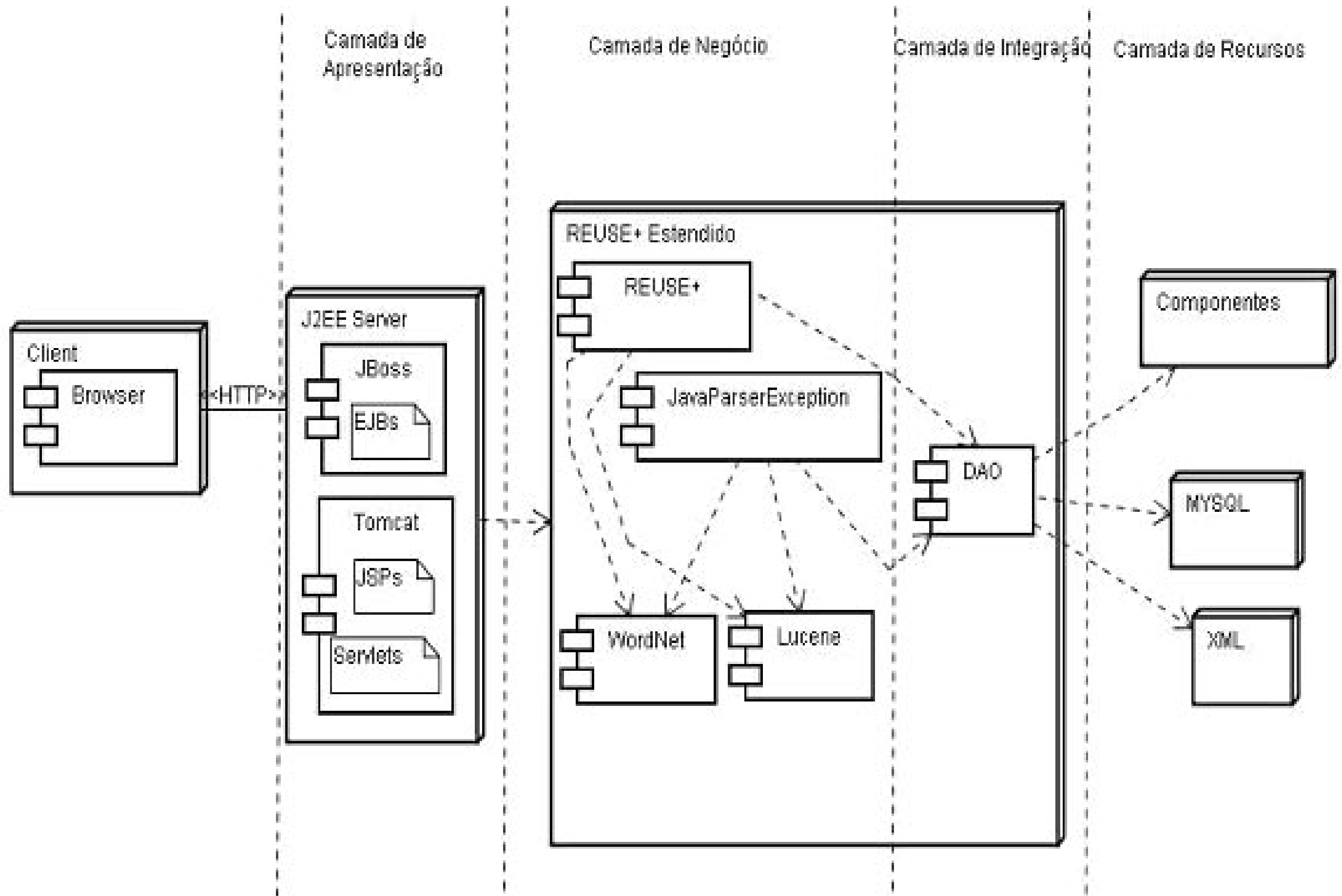

Figura 5.4: Diagrama UML de Implantação do REUSE++

5.4 Reuse++ como aplicação J2EE

Como mostrado na seção anterior o Reuse ++ segue o padrão arquitetural J2EE, portanto, é constituído de componentes Web e componentes EJB que irão interagir no processo de classificação e na realização das buscas. A Figura 5.2 mostra como é a comunicação entre esses componentes. Para um melhor entendimento segue abaixo uma descrição desses componentes:

5.4.1 Componentes Web

Os componentes Web possuem a responsabilidade de executar as requisições proveniente dos usuários. Um conteiner Web aloca e gerencia os componentes Web de forma que cada requisição que chega é direcionada ao componente web adequado. Sendo assim, todas as requisições para o cadastro e busca de componentes do Reuse++ passarão pelo conteiner e em seguida por componentes Web. Podem ser de dois tipos:

1. **Servlets**: são componentes java que irão receber, tratar e responder as requisições do cliente Web. No Reuse++ existem quatro Servlets: ClassifyServlet para gerenciar as requisições do processo de classificação, SearchServlet para gerenciar as requisições de Busca, UploadServlet para a requisição do cadastro do Componente e LoginServlet para a autenticação do usuário.
2. **Java Server Pages(JSP)**: esses componentes são responsáveis principalmente pela visualização dos dados que seguem para o navegador do usuário. Um arquivo no formato jsp pode conter código java, html, xml e imagens. Todas as telas do Reuse++ que serão mostradas nas próximas seções são componentes jsp.

5.4.2 Componentes EJB

Como foi mostrado na seção 2.3, componentes EJB (Enterprise Java Bean) são resposáveis principalmente pela lógica de negócios da aplicação. Existem três tipos de componentes EJB, porém para o desenvolvimento do Reuse++ foram criados apenas EJB do tipo *SessionBean*:

- **LoginUsuarioBean**: que faz a autenticação do usuário. e o
- **ComponenteInfoBean**: que oferece os serviços de consulta, inserção, alteração e listagem dos componentes.
- **ExcecaoInfoBean**: que oferece os serviços de consulta, inserção, alteração e a listagem das exceções lançadas e tratadas.

5.5 Autenticação e Tela Principal

Para ter acesso aos serviços oferecidos pelo Reuse++, o usuário precisa ser cadastrado no sistema. A autenticação utilizada é simples e não faz uso de criptografia de senhas. A Figura 5.5 mostra como é a tela de login. Após a autenticação, o usuário é redirecionado para a tela

principal do sistema mostrada na Figura 5.6. A tela principal contém um *link* para o cadastro e classificação do componente e também campos de preenchimento para a opção de busca.

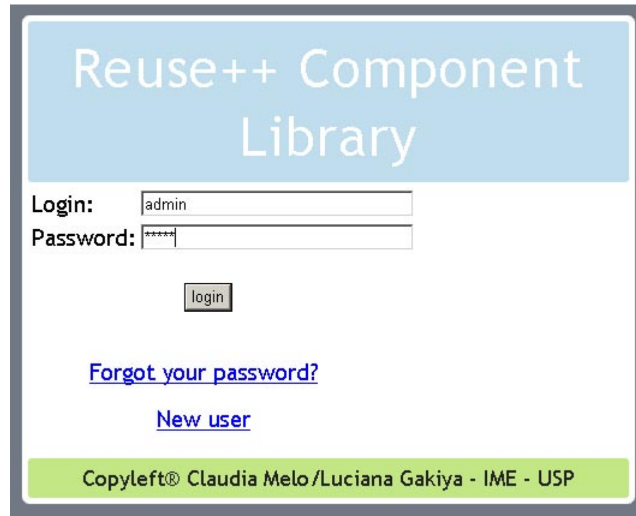

Figura 5.5: Tela de Login do Reuse++

5.6 O mecanismo de Classificação

A classificação é efetuada apenas através da análise do código-fonte do componente. O mecanismo tem início no momento em que é feito o *upload* do componente, representado através de um pacote compactado no formato *zip*. A seguir são extraídas as informações de interface e pacote.

Os componentes devem ter uma funcionalidade clara e específica do que realizam e/ou descrevem, devem possuir também interface clara, que indica como podem ser reusados e conectados a outros componentes, e devem ocultar os detalhes que não são necessários para reuso. Szyperski [Szyperski, 2002] define uma interface como um conjunto de assinaturas de operações que podem ser invocadas por um cliente.

Considerando o modelo de componente EJB explicado na seção 2.3, o conteúdo é analisado pelo *parser*, proveniente do trabalho de [Melo, 2006], alterado para essa finalidade. São extraídas as seguintes informações:

- **interface provida**: é a interface disponibilizada para comunicação com o ambiente. A

Figura 5.6: Tela Principal do Reuse++

solução considerou como interface provida a *interface* Java que estende *javax.ejb.EJBObject*.

- **interface requerida**: é a interface necessária para que o componente possa ser acessado. A solução considerou como interface requerida a *interface* Java que estende *javax.ejb.EJBHome*.
- **pacote**: em Java, o termo pacote é equivalente a diretório e é utilizado para organizar as *classes*. Essa informação é capturada através da declaração *package* localizada na primeira linha do arquivo que representa o *EJB*.

O passo seguinte do mecanismo é obter as informações de exceções. O objetivo é minimizar a

classificação manual durante o registro do componente, por isso, os esforços caminharam em direção à extração totalmente automática.

5.6.1 Análise das informações de exceções do Código-fonte

A extração automática de todas as informações foi possível devido ao mecanismo de tratamento de exceções na linguagem de programação Java possuir uma estrutura padronizada. Sendo assim, um *parser* foi criado especificamente para percorrer o código-fonte do componente e extrair as seguintes informações de exceções:

- **Conjunto de exceções lançadas pelo componente** : montado com as exceções que acompanham os termos *throw* e *throws*. Preencherá a **faceta *Thrown Exceptions*** do modelo.
- **Conjunto de exceções tratadas pelo componente** : montado com as exceções que acompanham o termo *catch*. Preencherá a **faceta *Caught Exceptions*** do modelo.
- **Conjunto com os contextos de lançamento de todas as exceções lançadas** : cada contexto de uma exceção corresponde a todo código existente entre a estrutura *try/catch*. Sendo assim, cada exceção desse conjunto pode estar associada a vários contextos de lançamento dentro do componente. Preencherá o **atributo *Context*** de cada exceção lançada.
- **Conjunto com as ações de tratamento de todas as exceções capturadas** : cada ação de tratamento de uma exceção corresponde ao código existente na estrutura do termo *catch* correspondente. Preencherá o *atributo Action Handling* de cada exceção.
- **Quantidade de throws Exception** : número de vezes em que o termo *throws Exception* foi encontrado no componente. Preencherá o **atributo *ThrowsException*** do modelo.
- **Quantidade de catch Exception** : número de vezes em que o termo *catch Exception* foi encontrado no componente. Preencherá o **atributo *CatchException*** do modelo.
- **Quantidade de finally vazios** : número de vezes em que a estrutura *finally* foi encontrada vazia no componente. Preencherá o **atributo *EmptyFinally*** do modelo.
- **Quantidade de catch vazios** : número de vezes em que a estrutura *catch* de qualquer exceção foi encontrada vazia no componente. Preencherá o **atributo *EmptyCatch*** do modelo.

A Figura 5.7 mostra um exemplo de código-fonte com as informações que são capturadas e armazenadas pelo parser de exceções.

```
111  /**
112   * Stores the EJBean in the persistent storage.
113   *
114   * @exception              javax.ejb.NoSuchEntityException
115   *                         if the bean is not found in the database
116   * @exception              javax.ejb.EJBException
117   *                         if there is a communications or systems failure
118   */
     public void ejbStore() {
120    log("ejbStore (" + id() + ")");
121
122    Connection con = null;
123    PreparedStatement ps = null;
124
125    try {
126
127      con = getConnection();
128      ps = con.prepareStatement("update ejbAccounts set bal = ? where id = ?");
129      ps.setDouble(1, balance);
130      ps.setString(2, accountId);
131      if (!(ps.executeUpdate() > 0)) {
132        String error = "ejbStore: AccountBean (" + accountId + ") not updated";
133        log(error);
134
135        throw new NoSuchEntityException (error);
136
137      }
138
139    } catch(SQLException sqe ) {
140
141      log("SQLException:  " + sqe);
142      throw new EJBException (sqe);
143
144    } finally {
145      cleanup(con, ps);
146    }
147  }
```

Contexto da Exceção SQLException

Exceção Lançada

Exceção Tratada

Ação de Tratamento

Exceção Lançada

Figura 5.7: Extração das informações de exceções para as facetas e atributos do modelo

5.6.2 Exemplo de Cadastro e Classificação

Uma vez autenticado, se o usuário deseja cadastrar um componente é preciso ir na tela de *upload* clicando-se em *upload Component*. Depois deve selecionar o componente EJB empacotado no

formato .zip e em seguida preencher os campos de classificação manual. As Figuras 5.8 e 5.9 mostram esses passos no Reuse++.

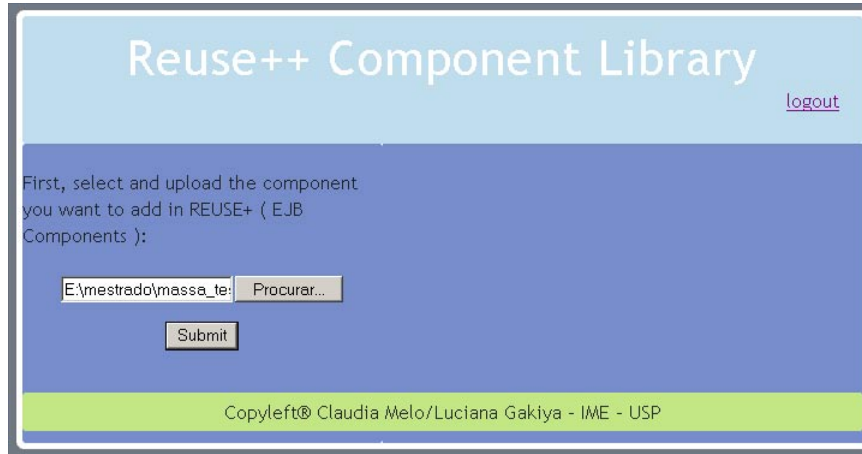

Figura 5.8: Tela de Cadastro de Componente

5.6.3 O mecanismo de armazenamento

O armazenamento das informações de exceções é feito durante o processo de cadastro e classificação do componente. O Reuse++ cria para cada componente um arquivo xml e armazena-o pra que possa ser utilizado depois pelo *framework* Lucene durante a busca. A indexação das informações às respectivas facetas é feita e armazenada em um arquivo com formato próprio do Lucene.

O Reuse++ faz o armazenamento utilizando o pacote JDBC(Java Database Connectivity) e o padrão DAO(Data Access Object). Como foi visto anteriormente, a busca utiliza uma ontologia de exceção que é facilmente mapeada para o modelo relacional. Para isso, algumas tabelas foram criadas e alteradas no MySql. Essas tabelas são populadas com os dados de exceções provenientes do código-fonte do componente. A Figura 5.10 mostra as principais tabelas do Reuse++.

5.7 Mecanismos de Busca e Recuperação

Após classificação, indexação e armazenamento descritos nas seções anteriores, o componente estará qualificado para participar do processo de busca. O Sistema Reuse++ efetua a busca

Figura 5.9: Tela de Classificação com as novas facetas incluídas

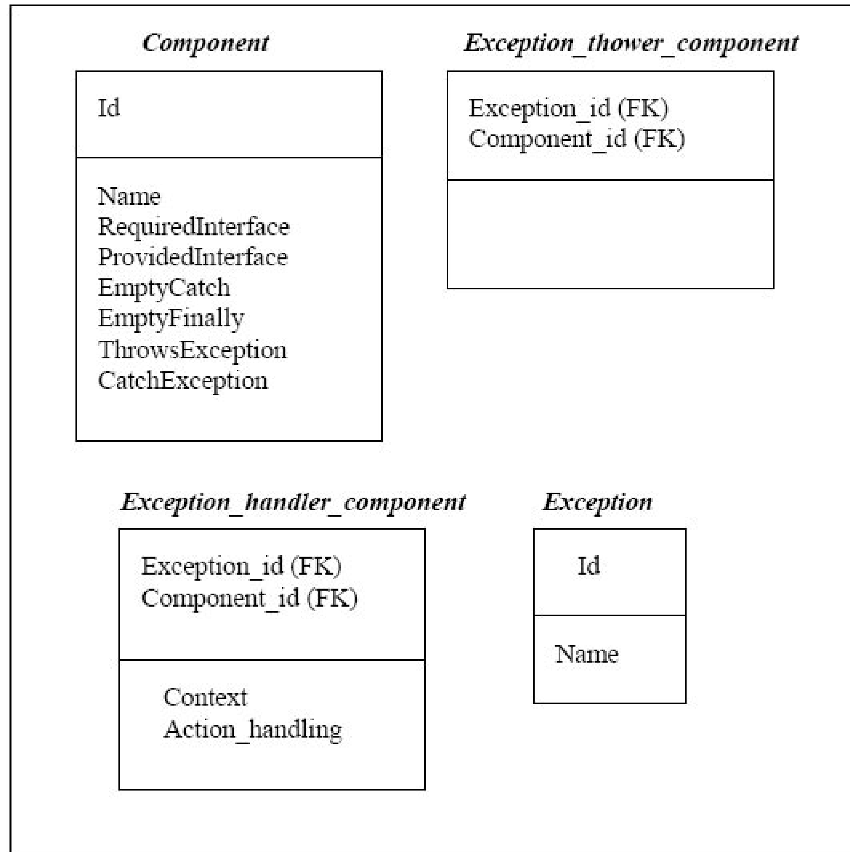

Figura 5.10: Tabelas do Reuse++ para os dados de exceções.

a partir do preenchimento de um conjunto de campos que representam as facetas do modelo original mais as duas facetas de exceções do modelo proposto.

A inclusão dessas duas facetas permitirá procurar componentes por exceção lançada e tratada. Com isso, o Reuse++ passa a contemplar situações em que o usuário queira somente componentes que tratem ou lancem determinadas exceções.

Após o preenchimento dos campos, as palavras menos importantes são removidas com a ajuda do *framework* Wordnet e as palavras-chaves resultantes forma uma expressão de consulta que é utilizada no mecanismo de busca. A Figura 5.11 mostra os campos a serem preenchidos pelo

usuário durante a busca.

Describe your query filling the facets
bellow and filtering it by attributes:

Domain:
Business Rules:
Roles:
Function:
Element:
Action:
Thrown Exceptions:
Caught Exceptions:
User:
Retrieval only SQA Components
Enable Automatic Query Expansion
Submit Query

Facetas do Comportamento Normal
Facetas do Comportamento Excepcional

Figura 5.11: Campos para preenchimento no mecanismo de busca.

Com a ajuda do *framework* Lucene a busca é então efetuada e a lista dos componentes que satisfazem a expressão de consulta é exibida pela ferramenta. Durante a busca, o Lucene realiza também o cálculo da relevância do componente selecionado em relação à expressão de consulta.

5.7.1 Exemplo de Busca

Na Figura 5.12 é mostrado um exemplo de busca onde o usuário procura por todos os componentes que fazem o tratamento da exceção *NamingException*. No lado direito da consulta é apresentada a lista de componentes que satisfazem a consulta. Cada componente da lista apresenta o seu grau de relevância em relação à consulta, uma breve descrição e um *link* para

a exibição dos detalhes.

Figura 5.12: Exemplo de Busca por exceção no Reuse++

5.7.2 Ontologia

Como visto na seção 2.6 um sistema de recuperação de informação deve fornecer uma rápida seleção dos itens de interesse do usuário. Sendo assim, a ontologia de exceção foi utilizada com o objetivo de melhorar o mecanismo de busca da ferramenta original. A estrutura semântica da ontologia permitirá recuperar componentes que se aproximam da expressão de consulta, no caso de insucesso da busca exata.

A partir do modelo de ontologia apresentado na Seção 4.5.1 foi criado o modelo relacional. O conjunto de tabelas do modelo é mostrado na Figura 5.10. Dessa forma, toda vez em que a

busca exata não recupera nenhum componente, a busca alternativa que usa ontologia é efetuada utilizando as palavras-chaves das facetas de exceções. Essa busca alternativa consiste em uma consulta às tabelas, recuperando uma lista de componentes alternativos.

A Figura 5.13 mostra uma consulta que se fosse feita somente com o Lucene não traria resultados. No exemplo da figura, o usuário queria todos os componentes que tratam a exceção *ProcessingErrorException*. Porém, como o Reuse++ não possui nenhum componente com essas características a busca utilizou a ontologia. O resultado foi uma lista de componentes de lançam a exceção ProcessingErrorException.

Figura 5.13: Resultado de Busca utilizando ontologia.

5.8 Visualização das informações do Componente

O Reuse++ permite ao usuário obter os detalhes do componente. Embora o foco desse trabalho sejam as informações de exceções, durante o processo de análise de código-fonte outras informações do componente são também extraídas: o pacote e as interfaces provida e requerida. Essas informações foram incluídas na tela de detalhes do componente juntamente com um *link* para as informações de exceções como mostrado na Figura 5.14.

Figura 5.14: Informações do Componente no Reuse++

A tela com os detalhes das exceções exibirá as seguintes informações:

- lista das exceçõs lançadas pelo componente cada uma contendo o *link* para o respectivo javadoc
- lista das exceções tratadas pelo componente cada uma contendo o *link* para o respectivo javadoc
- o trecho de código de lançamento de cada exceção
- o trecho de código de tratamento de cada exceção
- o total de *catch Exception* encontrado
- o total de *throws Exception* encontrado
- o total de estrutura *finally* vazia
- o total de estrutura *catch* vazia

A Figura 5.15 mostra um exemplo com os detalhes de exceções do componente BankBean.

5.9 Avaliação do Modelo e Método

As avaliações experimentais do Reuse++ fizeram uso de um conjunto controlado de documentos. As seguintes tarefas foram executadas:

1. Obtenção de uma coleção de testes
2. Conhecimento prévio das informações de exceções da coleção
3. Classificação e armazenamento dessa coleção de testes
4. Criação de 20 consultas
5. Execução automática das consultas no Reuse++
6. Coleta e medição dos resultados

5.9.1 A Coleção de Testes e as Consultas

A coleção de testes é um conjunto de 40 componentes EJB versão 2.1. [Microsystems, 2008]. Cada componente foi primeiramente examinado para a obtenção das informações de exceções. Os componentes possuem domínios variados e em sua maior parte foram obtidos no pacote de documentações dos seguintes sites:

Figura 5.15: Informações das Exceções do Componente no Reuse++

- *http://java.sun.com/j2ee/tutorial/1-3-fcs/doc/Examples.html*
- *http://java.sun.com/j2ee/1.4/docs/tutorial/examples/*

Foram criadas 20 consultas com o conhecimento prévio das exceções existentes nos componentes. Para facilitar as consultas, a recuperação foi feita baseando-se apenas nas exceções lançadas e tratadas, excluindo as informações de negócio do componente. Sendo assim, cada consulta corresponde ao par $<$*consulta*, *componente relevante* $>$ e foi subdividida em 3 grupos:

- **grupo 1**: Consulta por componentes que lançam um conjunto de exceções
- **grupo 2**: Consulta por componentes que tratam um conjunto de exceções
- **grupo 3**: Consulta por componentes que lançam e tratam um conjunto de exceções

5.9.2 Medidas de Avaliação

Para avaliar o sistema Reuse++ foram utilizadas as seguintes medidas:

- **Cobertura** : mede o número de componentes relevantes que são recuperados no processo de seleção versus o número total de candidatos que obedecem aos requisitos. Uma recuperação baixa significa que muitos componentes interessantes estão sendo ignorados.
- **Precisão** : mede o número de componentes realmente utilizáveis frente ao número de componentes obtidos no processo de seleção. Sendo assim, se um repositório recupera 100 documentos e 50 deles são relevantes, a precisão do repositório é de 50 por cento. Uma baixa precisão significa que o método de seleção devolve muitos componentes que não obedecem aos requisitos informados.

5.9.3 Resultados

A coleção de componentes obtida foi cadastrada, classificada e as consultas foram submetidas no Reuse++. As medidas de avaliação: cobertura e precisão foram coletadas e armazenadas.

Os principais resultados obtidos com o Reuse++ foram:

- Considerando todas as consultas, o valor médio de cobertura foi de 0,75.
- Considerando todas as consultas, valor médio de precisão foi de 0,47.
- Considerando as consultas que lançam um conjunto de exceções (grupo 1), o valor médio de cobertura foi de 0,71.

- Considerando as consultas que lançam um conjunto de exceções (grupo 1), valor médio de precisão foi de 0,5.
- Considerando as consultas que tratam um conjunto de exceções (grupo 2), o valor médio de cobertura foi de 0,69.
- Considerando as consultas que tratam um conjunto de exceções (grupo 2), valor médio de precisão foi de 0,53.
- Considerando as consultas que lançam e tratam um conjunto de exceções (grupo 3), o valor médio de cobertura foi de 0,9.
- Considerando as consultas que lançam e tratam um conjunto de exceções (grupo 3), valor médio de precisão foi de 0,3.

As medidas mostraram que a cobertura da ferramenta, em média, é boa para os três grupos de consultas, porém, a precisão para as consultas do grupo 1 e 2 foi razoável e para o grupo 3 foi baixa.

A Figura 5.16 mostra as medidas de cobertura e precisão das consultas.

5.10 Conclusão

Neste Capítulo foi apresentado o funcionamento, as principais telas e a avaliação da ferramenta Reuse++. Foi mostrada a viabilidade da solução reforçando a idéia que a classificação e busca de componentes com tratamento de exceções pode facilitar a escolha do componente.

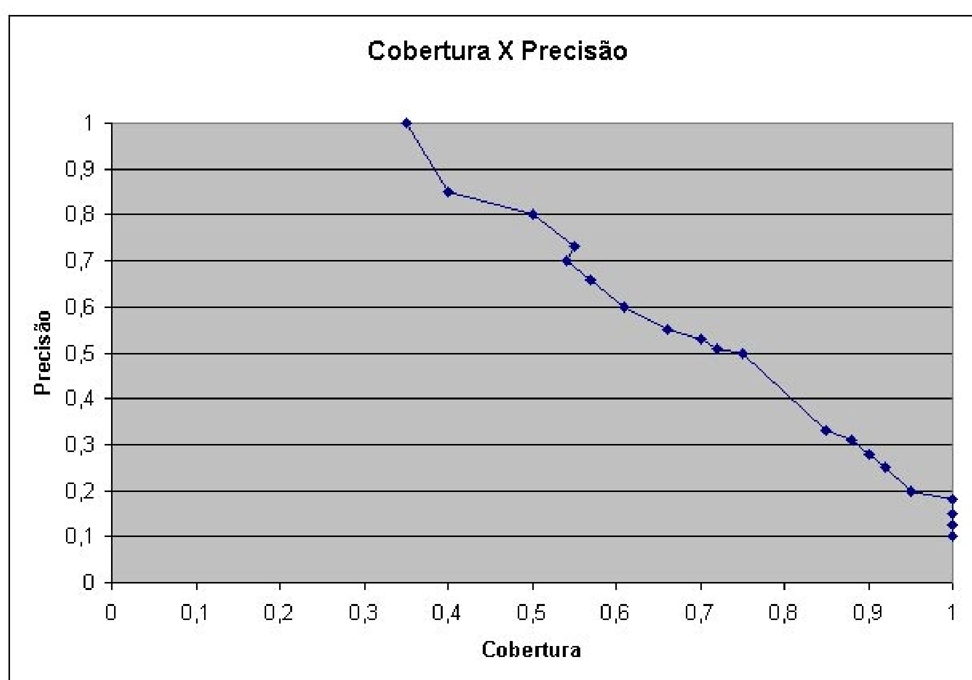

Figura 5.16: Curva com a cobertura e precisão das consultas submetidas no Reuse++

Capítulo 6

Conclusões

Desde 1968, quando McIlroy propôs a idéia de uma indústria de componentes de software, pesquisas têm sido feitas para explorar melhor o tema. Descobriu-se que o emprego de componentes na estruturação dos sistemas promove maior qualidade e flexibilidade ao produto e agiliza o processo de desenvolvimento.

Contudo, para que estes benefícios sejam realmente observados, é fundamental que se conheça os detalhes dos componentes selecionados. Isso implica em conhecer também o funcionamento dos mesmos em situações de falhas.

Nos sistemas de missão crítica tolerantes a falhas, grande parte da implementação é destinada à detecção e recuperação de erros [Ran, 1995]. Entretanto, a introdução de mecanismos para tolerância a falhas eleva consideravelmente a complexidade do sistema, aumentando também a probabilidade de ocorrência de falhas de projeto [T. Anderson, 1990].

Nesse contexto, os mecanismos para tratamento de exceções possuem uma importância fundamental ao permitir a separação clara do código responsável pela recuperação do erro do código normal do sistema, diminuindo a complexidade e, conseqüentemente, reduzindo as eventuais falhas de projeto [Guerra, 2004].

Portanto, o mecanismo para tratamento de exceções, quando inseridos corretamente no código, provê uma estrutura bastante adequada para a implementação das técnicas de tolerância a falhas nos sistemas.

Porém, a literatura não apresentou até então um trabalho que inserisse no processo de recuperação, mecanismos que considerasse também as informações sobre o comportamento anormal dos componentes.

Assim, diante da importância de se conhecer o comportamento excepcional e da necessidade de se ter mecanismos de classificação e busca que considerasse também as exceções dos componentes, uma solução foi apresentada neste trabalho.

A solução, baseada em um conjunto de facetas e atributos, foi mostrada e concretizada através do desenvolvimento da ferramenta Reuse++.

6.1 Contribuições

As principais contribuições deste trabalho são:

- Mostrar uma alternativa para o armazenamento de informações a respeito das exceções dos componentes nos repositórios.
- Mostrar a viabilidade de se efetuar buscas de componentes por suas exceções lançadas e tratadas.
- Apresentação de medidas associadas às exceções que possibilitem avaliar a qualidade do componente.
- Melhoramento do processo de busca e armazenamento através do uso de uma ontologia de exceção.
- Apresentação da classificação de componentes considerando também as suas interfaces.

6.2 Trabalhos Futuros

Uma das limitações do trabalho é a manipulação apenas dos componentes EJB da versão 2.1. Uma vez que a versão EJB 3.0 é a versão mais atualizada e, sob ponto de vista de desenvolvimento, a criação de componentes nesta versão é mais fácil, seria interessante a extensão da ferramenta para aceitar componentes EJB 3.0 também.

Alguns tópicos mencionados nesse trabalho podem dar origem a trabalhos relevantes, seguem abaixo algumas sugestões para estudos futuros:

- Métricas para avaliar a qualidade do componente e do repositório - neste trabalho foram criadas quatro medidas de qualidade de código relacionadas às exceções: duas que medem a quantidade de blocos vazios *catch* e *finally* e duas que medem a quantidade de declarações *throws Exception* e *catch Exception*. Em relação às métricas de repositório existem duas nesse trabalho: uma que representa o total de componentes do repositório e outra que

é a quantidade de reuso de cada componente. Porém, para avaliar melhor a qualidade dos componentes e os respectivos repositórios é necessário um estudo mais profundo e incrementar o Reuse++ com um conjunto maior de métricas.

– Ontologia - nesse trabalho foi apresentado uma ontologia de exceção. Porém seria interessante um estudo mais detalhado para avaliar a possibilidade de inserção de mais ontologias para o melhoramento do processo de busca.

– Mecanismo de Inferência - neste trabalho não foi utilizado nenhum mecanismo de inferência baseado em conhecimento obtido no processo de classificação dos componentes. A aplicação de inferência no modelo de classificação e busca de componentes poderia trazer melhorias no processo de busca.

Referências Bibliográficas

[Ran, 1995] (1995). *The evolution of the recovery block concept.* Lyu.

[Alan W. Brown, 1996] Alan W. Brown, K. C. W. (1996). Engeneering of component based systems. *IEEE Computer Society Press*, 15:7–15.

[Alan W. Brown, 1998] Alan W. Brown, K. C. W. (1998). The current state of cbse. *IEEE Computer Society*, 15:37–46.

[Ancelmo Maciel Nunes, 2006] Ancelmo Maciel Nunes, R. F. (2006). Uma arquitetura para recuperação de informação baseada em semântica e sua aplicação no apoio a jurisprudência.

[And, 2004] And, K. S. (2004). Container-managed exception handling framework.

[Bachmann et al., 2000] Bachmann, F., Bass, L., Buhman, C., Santiago, D. C., Long, F., Robert, J., Seacord, and Wallnau, K. (2000). Technical concepts of component-based software engeneering. Technical report, Technical Report CMU/SEI.

[Brito, 2005] Brito, P. (2005). Um método para modelagem de exceções em desenvolvimento baseado em componentes. Master's thesis, Universidade Estadual de Campinas - UNICAMP.

[Buchanan, 1979] Buchanan, B. (1979). *Theory of Library Classification.* New York: K.G.

[Chris Leer, 2001] Chris Leer, David Rosenblum, D. S. W. (2001). Wren - an evironment for component-based development. In *In ESEC/FSE-9: Proceedings of the 8th European Software Engineering conference held jointly with 9th ACM SIGSOFT International Symposium On Foundations Of Software Engineering*, pages 207–217.

[Cristian, 1995] Cristian, F. (1995). Exception handling and software fault tolerance. In *FTCS-25:Highlights from Twenty-Five Years*, page 120.

[D. Garlan, 2000] D. Garlan, Robert T. Monroe, D. W. (2000). Ame: Architectural description of component-based systems. *Cambridge University Press, Cambridge, UK*, pages 47–67.

[da Silva Xavier, 2008] da Silva Xavier, K. (2008). Ambiente de testes utilizando verificação de componentes java com tratamento de exceções. Master's thesis, Universidade de São Paulo - USP.

[Donna Malayeri, 2005] Donna Malayeri, J. A. (2005). Practical exception specifications.

[Ferreira, 2001] Ferreira, G. R. M. (2001). Tratamento de exceções no desenvolvimento de sistemas confiáveis baseado em componentes. Master's thesis, Universidade Estadual de Campinas-UNICAMP.

[Foundation, 2006] Foundation, A. S. (2006). Apache lucene.

[Gómez-Pérez and Benjamins, 1999] Gómez-Pérez, A. and Benjamins, V. (1999). Overview of knowledge sharing and reuse components: Ontologies and problem-solving methods. In *International Joint Conference on Artificial Intelligence(IJCAI-99)*.

[Gruber, 1995] Gruber, T. R. (1995). Toward principles for the design of ontologies used for knowledge sharing. *International Journal of Human-Computer Studies*, pages 907–928.

[Guarino, 1997] Guarino, N. (1997). Understanding, building, and using ontologies: A commentary to "using explicit ontologies in kbs development", by van heijst and schreiber and wielinga. *International Journal of Human and Computer Studies*, pages 293–310.

[Guerra, 2004] Guerra, P. A. C. (2004). *Uma abordagem Arquitetural para Tolerância a Falhas em Sistemas de Software Baseados em Componentes*. PhD thesis, Universidade Estadual de Campinas - UNICAMP.

[Guo, 2000] Guo, J. L. (2000). A survey of software reuse repositories. In *International Conference And Workshop On The Engeneering Of Computer Based Systems*, pages 92–100.

[Henninger, 1997] Henninger, S. (1997). An evolutionary approach to constructing effective software reuse repositories. *In: ACM Transactionson Software Engineering and Methodology*, 6.

[Johannes, 1997] Johannes, S. (1997). *Software Engineering with Reusable Components*. Springer.

[Lucredio et al., 2004] Lucredio, D., d. Almeida, E. S., and d. Prado, A. F. (2004). A survey on software components search and retrieval. In *30th IEEE EUROMICRO Conference, Component-Based Software Engineering Track*, page 152159.

[Mary Shaw, 1996] Mary Shaw, Robert DeLine, G. Z. (1996). Abstractions and implementations for architectural connections. In *Third International Conference on Configurable Distributed Systems*.

[McIlroy, 1968] McIlroy, M. D. (1968). Mass-produced software components. In *Proceedings of the NATO Software Engineering Conference*.

[Melo, 2006] Melo, C. (2006). Classificação semi-automática de componentes java. Master's thesis, Universidadade de São Paulo-USP.

[Microsystems, 2002] Microsystems, S. (2002). *Java Programming Language SL-275*. Sun Service.

[Microsystems, 2008] Microsystems, S. (2008). Enterprise javabeans specification version 2.1.

[Miller, 1998] Miller, G. A. (1998). Wordnet an electronic lexical database.

[Mittermeir et al., 1998] Mittermeir, R., Pozewaunig, H., Mili, A., and Mili, R. (1998). Uncertainty aspects in component retrieval.

[Novello, 2003] Novello, T. C. (2003). Ontologias, sistemas baseados em conhecimento e modelo de banco de dados.

[Padmal Vitharana, 2003] Padmal Vitharana, Fatemeh M. Zahedi, H. J. (2003). Knowledge-based repository scheme for storing and retrieving business components: A theoretical design and an empirical analysis. *IEEE Transactions on Software Engineering*, 29(7):649–664.

[Panos Constantoupolos, 1993] Panos Constantoupolos, Martin Doerr, Y. V. (1993). Repositories for software reuse: The software information base. In *Working Conference On Information System Development Process*, pages 285–307.

[Pasi, 2002] Pasi, G. (2002). Flexible information retrieval: some research trends. *Mathware and Soft Computing*, pages 107–121.

[Paul Clements, 1996] Paul Clements, L. M. N. (1996). Software architecture: An executive overview. Technical report, Relatório Técnico-MU-SEI-96-TR-003.

[Pereira, 2007] Pereira, D. P. (2007). Um framework para coordenação do tratamento de exceções em sistemas tolerantes a falhas. Master's thesis, Universidade de São Paulo - USP.

[Prieto-Diaz, 1991] Prieto-Diaz, R. (1991). Implementing faceted classification for software reuse. *Especial Issue on Software Engineering*, 34.

[Ranganathan, 1960] Ranganathan, S. R. (1960). *Colon Classification: basic classification*. Asia Publishing House.

[Redolph et al., 2004] Redolph, G., de Araujo Spagnoli, L., Bastos, R. M., Cristal, M., and Espindola, A. P. (2004). Especificando informações para componentes reutilizáveis. Technical report, Pontifícia Universidade Católica do Rio Grande do Sul - PUCRS.

[Regina Braga, 2001] Regina Braga, Marta Mattoso, C. W. (2001). The use of mediation and ontology technologies for software component information retrieval. *Symposium on Software Reusability*, pages 19–28.

[Romanovsky et al., 2005] Romanovsky, A., Dony, C., Knudsen, J. L., and Tripathi, A., editors (2005). *Developing Systems that Handle Exceptions LIRMM (Laboratoire d'Informatique, de Robotique et Micro-Electronique de Montpellier)*.

[Sanchez, 2005] Sanchez, M. G. (2005). Um estudo sobre os riscos inerentes à implantação de reuso de componentes no processo de desenvolvimento de software. Master's thesis, Universidade Estadual de Campinas-UNICAMP.

[Souchon et al., 2003] Souchon, F., Urtado, C., Vauttier, S., and Dony, C. (2003). Exception handling in component based-systems: a first study. In *Of the Exception Handling in Object Oriented Systems: towards Emerging Application Areas and New Programming Paradigms Workshop (at ECOOP´03 international conference)*.

[Szyperski, 2002] Szyperski, C. (2002). *Component Software: Beyond Object-Oriented Programming*. Harlow: Addison Wesley Longman Publishing Co., Inc., Boston, MA, USA.

[T. Anderson, 1990] T. Anderson, P. A. L. (1990). *Fault Tolerance: Principles and Practice*. Springer-Verlag.

[Van Harmelen, 2003] Van Harmelen, J. Davies, D. F. (2003). *Towards the Semantic Web: Ontology-Driven Knowledge Management*. John Wiley and Sons Ltd.

[Vickery, 1960] Vickery, B. C. (1960). *Facet Classification: A Guide to the Construction and Use of Special Schemes*. London: Aslib.

[Vijayan Sugumaran, 2003] Vijayan Sugumaran, V. C. S. (2003). A semantic-based approach to component retrieval. *SIGMIS Database*, 34(3):8–24.

[Ye and Fischer, 2002] Ye, Y. and Fischer, G. (2002). Supporting reuse by delivering task-relevant and personalized information. In *In Proceedings of the 24th ICSE*, pages 513–523.