UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
PROGRAMA DE PÓS-GRADUAÇÃO EM COMPUTAÇÃO

SILVIA DE CASTRO BERTAGNOLLI

FRIDA: um método para elicitação e modelagem de RNFs

Tese apresentada como requisito parcial
para a obtenção do grau de
Doutor em Ciência da Computação

Profa. Dra. Ana Maria de Alencar Price
Orientadora

Profa. Dra. Maria Lúcia Blanck Lisbôa
Co-Orientadora

Porto Alegre, março de 2004.

CIP – CATALOGAÇÃO NA PUBLICAÇÃO

Bertagnolli, Silvia de Castro

FRIDA: Um Método para Elicitação e Modelagem de RNFs / Silvia de Castro Bertagnolli. – Porto Alegre: Programa de Pós-Graduação em Computação, 2004.

163 f.: il.

Tese (Doutorado) – Universidade Federal do Rio Grande do Sul, Programa de Pós-Graduação em Computação, Porto Alegre, BR-RS, 2004. Orientadora: Ana Maria de Alencar Price; Co-orientadora: Maria Lúcia Blanck Lisbôa.

1. Requisitos não funcionais para segurança, confiabilidade e desempenho. 2. Elicitação e modelagem de requisitos não funcionais. 3. Desenvolvimento de software orientado a aspectos. 4. Qualidade de software. I. Price, Ana Maria de Alencar. II. Lisbôa, Maria Lúcia Blanck. III. Título.

SUMÁRIO

LISTA DE ABREVIATURAS

ACLs	Access Control Lists
AO	Aspect-Oriented
AOCRE	Aspect-Oriented Component Requirements Engineering
AOD	Aspect-Oriented Design
AOSD	Aspect-Oriented Software Development
AOP	Aspect-Oriented Programming
AORE	Aspect-Oriented Requirements Engineering
AP	Adaptive Programming
API	Application Programming Interface
APIs	Application Programming Interfaces
AUD	Application Under Development
BNF	Backus Naur Form
BOs	Business Objects
CA	Combinação de Aspectos
CASE	Computer Aided Software Engineering
CBSE	Component-Based Software Engineering
CF	Composition Filters
COM	Component Object Model
CORBA	Common Object Request Broker Architecture
DA	Decomposição de Aspectos
EBTs	Enduring Business Themes
EJBs	Enterprise Java Beans
FRIDA	From Requirements to Design using Aspects
HTTP	HyperText Transfer Protocol
IDE	Integrated Development Environment
IDECoRe	Integrated Development Environment Core Reuse
IDEs	Integrated Development Environments
IEEE	Institute of Electrical and Electronics Engineers
ISO	International Organization for Standardization
J2SDK	Java 2 Standard Development Kit
JVM	Java Virtual Machine

JDBC	Java Database Connectivity
LAL	Léxico Ampliado da Linguagem
LRNFs	Léxico de Requisitos Não Funcionais
MDSOC	Multi- Dimensional Separation Of Concerns
MOP	Meta-Object Protocol
MOPs	Meta-Object Protocols
MOTF	Meta-Objetos Tolerantes a Falhas
NFR	Non-Functional Requirement
NFRs	Non-Functional Requirements
OCL	Object Constraint Language
OMG	Object Management Group
OO	Orientação a Objetos
PSEs	Programming Support Environments
PSEEs	Process-centered Software Engineering Environments
POP	Post-Object Programming
PREview	Process REquirements view
QARCC	Quality Attribute Risk and Conflict Consultant
QoS	Quality of Service
RFs	Requisitos Funcionais
RMI	Remote Method Invocation
RNF	Requisitos Não Funcional
RNFs	Requisitos Não Funcionais
RUP	Rational Unified Process
SEEs	Software Engineering Environments
SOP	Subject-Oriented Programming
SSC	Sistema de Segurança Corporativo
UdI	Universo de Informações
UofD	Universe of Discourse
UML	Unified Modeling Language
UXF/a	UML eXchange Format, aspect extension
VORD	Viewpoint-Oriented Requirements Definition
W3C	World Wide Web Consortium
XMI	XML Metadata Interchange
XML	eXtended Markup Language

LISTA DE FIGURAS

LISTA DE TABELAS

RESUMO

Os requisitos direcionam o desenvolvimento de software porque são cruciais para a sua qualidade. Como conseqüência tanto requisitos funcionais quanto não funcionais devem ser identificados o mais cedo possível e sua elicitação deve ser precisa e completa. Os requisitos funcionais exercem um papel importante uma vez que expressam os serviços esperados pela aplicação. Por outro lado, os requisitos não funcionais estão relacionados com as restrições e propriedades aplicadas ao software. Este trabalho descreve como identificar requisitos não funcionais e seu mapeamento para aspectos. O desenvolvimento de software orientado a aspectos é apontado como a solução para os problemas envolvidos na elicitação e modelagem dos requisitos não funcionais. No modelo orientado a aspectos, o aspecto é considerado o elemento de primeira ordem, onde o software pode ser modelado com classes e aspectos. As classes são comumente usadas para modelar e implementar os requisitos funcionais, já os aspectos são adotados para a modelagem e implementação dos requisitos não funcionais. Desse modo, é proposta a modelagem dos requisitos não funcionais através das fases do ciclo de vida do software, desde as primeiras etapas do processo de desenvolvimento. Este trabalho apresenta o método chamado FRIDA – *From RequIrements to Design using Aspects*, cujo objetivo é determinar uma forma sistemática para elicitar e modelar tanto os requisitos funcionais quanto os não funcionais, desde as fases iniciais do ciclo de desenvolvimento. Em FRIDA, a elicitação dos requisitos não funcionais é realizada usando-se *checklists* e léxicos, os quais auxiliam o desenvolvedor a descobrir os aspectos globais – utilizados por toda a aplicação – bem como, os aspectos parciais que podem ser empregados somente a algumas partes da aplicação. O próximo passo consiste na identificação dos possíveis conflitos gerados entre aspectos e como resolvê-los. No método FRIDA, a identificação e resolução de conflitos é tão importante quanto a elicitação de requisitos não funcionais, nas primeiras fases do ciclo de vida do software. Além disso, é descrito como usar a matriz de conflitos para automatizar esse processo sempre que possível. A extração dos aspectos e sua modelagem visual são características muito importantes, suportadas pelo método, porque elas possibilitam a criação de modelos que podem ser reutilizados no futuro. Em FRIDA, é demonstrado como transformar os requisitos em elementos da fase de projeto (classes e aspectos) e como traduzir esses elementos em código. Outra característica do método FRIDA é que a conexão entre diagramas, que pertencem a diferentes fases do processo de desenvolvimento do software, permite um alto nível de rastreabilidade. Em resumo, FRIDA requer que o desenvolvedor migre de uma visão puramente funcional para outra que contemple também os requisitos não funcionais.

Palavras-chave: requisitos não funcionais para confiabilidade, segurança e desempenho, elicitação e modelagem de requisitos não funcionais, desenvolvimento de software orientado a aspectos, qualidade de software.

FRIDA: a Method for Eliciting and Modeling NFRs

ABSTRACT

Since requirements drive the software development, they are crucial for quality. As a consequence, both functional and non-functional requirements shall be identified as soon as possible and their elicitation must be accurate and complete. The functional requirements play an important role, since they express the application expected services. On the other hand, non-functional requirements are immediately involved in the software constraints and properties. This work shows how to identify non-functional requirements and how to map them into aspects. The aspect-oriented software development is appointed as the solution to the problems involved in non-functional requirements elicitation and modeling. In the aspect-oriented model, the aspect is a first-class element, and software can be modeled with classes and aspects. The classes are commonly used to model and implement functional requirements, and the aspects are adopted to model and implement non-functional requirements.We propose to model non-functional requirements during the whole software lifecycle, from the initial phases of the process development. This work presents a method called FRIDA – From RequIrements to Design using Aspects. Its goal is to determine a systematic way to elicit and model both functional and non-functional requirements since the early stage of the development lifecycle. In FRIDA we propose the elicitation of non-functional requirements using checklists and lexicons. Both checklists and lexicons help the developer to discover the global aspects - applied to the whole application- as well as the aspects to be applied to some parts of the application. The next step is the identification of the possible conflicts generated among the aspects and how to solve them. In our method the identification and resolution of conflicts is as important as the elicitation of non-functional requirements in the early stages of the software development lifecycle. We show the use of a matrix of conflicts to automate this process whenever possible. The aspect extraction and visual modeling of aspects are very important features supported by our method, because they allow the creation of models that can be reused in the future. FRIDA demonstrates how to transform the requirements into design elements (classes and aspects), and how to translate these elements into code. Moreover, the connection between diagrams that belongs to distinct phases of the software development cycle allows a high level of traceability. In short, FRIDA requires that developers shift from the purely functional point of view to another that encompasses non-functional requirements.

Keywords: non-functional requirements to dependability, security and performance, non-functional requirements eliciting and modeling, aspect-oriented software development, software quality.

1 APRESENTAÇÃO

Para possibilitar uma visão geral do trabalho desenvolvido, bem como de seus objetivos e de suas principais contribuições este capítulo foi organizado nas seguintes seções: seção 1.1 introduz o problema abordado pelo trabalho e algumas das soluções adotadas; na seção 1.2 pode-se encontrar a origem das idéias do trabalho; a seção 1.3 enumera os objetivos do trabalho e indica as principais técnicas e conceitos selecionados para compor a solução; na seção 1.4 é possível encontrar as contribuições do presente trabalho e finalmente; a seção 1.5 mostra a organização do texto.

1.1 Introdução

Em busca de qualidade de software os desenvolvedores, constantemente, procuram por técnicas e metodologias que minimizem a complexidade do processo de desenvolvimento, aumentem a sua compreensão e promovam a reutilização, não apenas do processo mas também do produto. A decomposição funcional é uma conhecida técnica de redução de complexidade na etapa de projeto. Esta técnica possibilita subdividir um sistema em componentes (tais como procedimentos, funções, métodos, objetos, classes, módulos e APIs (***A**pplication **P**rogramming **I**nterfaces*), os quais podem atender uma ou mais funcionalidades [CZA 2000, KIC 97, OSS 2001a].

Embora a funcionalidade seja essencial, muitos sistemas têm propriedades conhecidas como propriedades não funcionais[1], as quais não estão necessariamente relacionadas com a funcionalidade dos componentes.

Com a inclusão dessas propriedades não funcionais no desenvolvimento do software as preocupações do desenvolvedor alteram-se. Ele deixa de concentrar-se, exclusivamente, nas características intrínsecas da aplicação para preocupar-se com as propriedades não funcionais que o sistema deve satisfazer. Alguns exemplos de características não funcionais encontrados freqüentemente na literatura compreendem [BOE 96, KIC 97, LAD 2002a, LAD 2003, OSS 2001b]: tratamento de falhas, persistência, comunicação, replicação, coordenação, gerenciamento de memória, restrições de tempo-real e muitos outros aspectos de comportamento do sistema [KIC 97].

A adoção de propriedades não funcionais em aplicações dificulta o processo de desenvolvimento, visto que pode gerar diversos problemas, tais como falta de transparência, código não funcional disperso e/ou entrelaçado em vários componentes e, em decorrência prejudicando a reutilização e a manutenção de componentes (ver

[1] Também denominadas requisitos não funcionais [ROM 85], ou atributos de qualidade [CHU 99, BRI 02], ou requisitos não comportamentais [DAV 93], ou "software ilities" (*reliability, availability, security, dependability, integrity, confidentiality, availability*, etc.).

detalhes no capítulo 3, seção 3.3). Percebe-se que esses problemas são originados da relação intrínseca que existe entre a funcionalidade e as características não comportamentais.

Várias técnicas foram desenvolvidas com o intuito de propiciar uma real separação de responsabilidades entre os elementos funcionais e os não funcionais e sua conexão transparente tentando eliminar ou reduzir a dispersão e o entrelaçamento de código. As principais delas compreendem [ELR 2001a, KIC 97]: (i) filtros de composição, (ii) técnicas adaptativas, (iii) programação orientada a assuntos, (iv) separação de responsabilidades de forma multi-dimensional, (v) protocolos de meta-objetos, e (vi) programação orientada a aspectos.

Cada uma dessas técnicas possui peculiaridades e elementos comuns (ver descrição detalhada na seção 3.4). A adoção de uma ou de outra técnica depende exclusivamente do desenvolvedor do sistema e do problema a ser solucionado. Além disso, para que uma nova técnica possa ser empregada como solução e para que seu uso seja incentivado deve ser oferecido algum suporte, principalmente através de alguma ferramenta associada.

As ferramentas auxiliam o desenvolvedor nas diferentes fases do ciclo de vida do software, bem como na sua gerência e documentação. Elas têm se mostrado cada vez mais importantes, porque proporcionam a construção de software confiável, de alta qualidade, e a um custo e tempo minimizados [GRU 99, GRU 2001b].

Desse modo, analisando-se o contexto acima descrito, neste trabalho é proposto um método sistemático de desenvolvimento, capaz de abranger grande parte do ciclo de desenvolvimento de um sistema computacional baseado em RNFs – ***R**equisitos **N**ão **F**uncionais*. Dentre as diversas técnicas propostas para solucionar os problemas envolvidos com a modelagem, elicitação e uso de RNFs optou-se por selecionar a orientação a aspectos ou AO – ***A**spect-**O**riented*. O método proposto busca aplicar o referencial teórico de orientação a aspectos a todo o processo de desenvolvimento, unificando as suas diferentes visões, tais como AORE – ***A**spect-**O**riented **R**equirements **E**ngineering* [BRI 2002, ARA 2002, RAS 2003], AOD – ***A**spect-**O**riented **D**esign* [CLA 2001, CON 2001, HER 2000] e AOP – ***A**spect-**O**riented **P**rogramming* [ELR 2001a, ELR 2001b, HIG 99].

Com base no método proposto pode-se construir uma ferramenta cujo objetivo é propiciar a automatização das tarefas pertinentes ao processo de desenvolvimento, desde a especificação de requisitos até a geração de código.

1.2 Histórico do Trabalho

As definições do método e da ferramenta que estão sendo apresentadas neste trabalho passaram por uma evolução. Essa evolução encontra-se dividida em fases, as quais são descritas uma a uma nas próximas seções.

1.2.1 Fase 1 – MOTF

A idéia de criar um método para apoiar o desenvolvimento de aplicações que utilizam requisitos não funcionais surgiu a partir do trabalho de Lisbôa [LIS 95, LIS 98]. Esse trabalho intitulado MOTF – ***M**eta **O**bjetos **T**olerantes a **F**alhas*, é um *framework* baseado em reflexão computacional, cujo objetivo é resolver problemas do domínio de tolerância a falhas.

Esse *framework* usa a reflexão computacional como forma de estabelecer uma clara separação entre os requisitos funcionais e os não funcionais (requisitos de tolerância a falhas) [LIS 98]. Isso deve-se principalmente ao fato de que a especificação do sistema pode ser particionada em níveis, onde o nível base concentra-se em informações do domínio do problema e os demais níveis manipulam as informações não funcionais do sistema.

A maioria dos trabalhos encontrados na literatura que abordam requisitos não funcionais do domínio de tolerância a falhas, também apontam a reflexão computacional como tendência na solução de vários problemas: (i) problemas de comunicação [ANC 95]; (ii) técnicas de replicação [FAB 95, GAR 98] ; (iii) problemas de *checkpoint* em sistemas concorrentes [KAS 98]; (iv) tratamento de atomicidade [STR 95] e (v) construção de sistemas com arquitetura de software distribuída [LEE 98, STR 92].

A partir do trabalho proposto por Lisbôa [LIS 95] ficou claro que existia a necessidade de ferramentas que auxiliassem o desenvolvimento de software reflexivo. Percebeu-se ainda que essas ferramentas deveriam propiciar um desenvolvimento rápido e se possível transparente. Com base nessas idéias surgiu o ambiente integrado de desenvolvimento JReflex.

1.2.2 Fase 2 – JReflex

Ferramentas e ambientes de programação que oferecem suporte à implementação de sistemas são, geralmente, denominados IDEs - ***I**ntegrated **D**evelopment **E**nvironments*, ou ambientes de desenvolvimento integrado. Eles costumam agregar ferramentas diversas, tais como montadores, compiladores, editores de programas e depuradores.

JReflex [BER 2000a] é um IDE - ***I**ntegrated **D**evelopment **E**nvironment,* com o objetivo de facilitar o desenvolvimento de software reflexivo na linguagem Java. Seu objetivo é oferecer um conjunto de ferramentas para automatizar o desenvolvimento deste tipo de software, procurando minimizar o esforço da fase de implementação, propondo-se a [BER 2000b]:

- disponibilizar uma biblioteca de componentes, os quais simplificam a utilização da API – ***A**pplication **P**rogramming **I**nterface* – de reflexão da linguagem Java;
- incorporar ao programa componentes que forneçam, de forma customizada, o serviço de introspecção;
- oferecer uma interface interativa permitindo, assim, ao programador identificar e selecionar os componentes alvo dos serviços.

JReflex, além do recurso de inspeção de objetos, oferece a geração automática de código possibilitando o desenvolvimento de programas reflexivos, sem que para isso o programador tenha que dominar a API de reflexão da linguagem Java. Outra característica importante é que o programador utiliza um código previamente testado, validado e documentado, diminuindo assim a taxa de erros, de falhas e o tempo para a finalização de sua aplicação.

Após o desenvolvimento desse ambiente iniciou-se um estudo sobre a viabilidade de utilizar a API de reflexão da linguagem Java para o desenvolvimento de aplicações tolerantes a falhas e distribuídas.

1.2.3 Fase 3 – Extensão de JReflex

Essa extensão que foi proposta em [BER 2000c] foi uma tentativa de integrar ao ambiente JReflex características pertinentes aos domínios de tolerância a falhas e distribuição. O objetivo dessa extensão consiste em simplificar o desenvolvimento de aplicações, nas quais a técnica de programação de meta-nível exerça um papel fundamental, tais como replicação transparente de componentes, para fins de tolerância a falhas de sistemas, e computação distribuída. A idéia principal era prover extensibilidade aos programas do desenvolvedor facilitando a incorporação de componentes para adaptação de programas às exigências dos RNFs. No entanto, essa idéia não obteve o sucesso esperado devido a uma deficiência encontrada na API reflexiva padrão da linguagem Java, a qual não oferecia mecanismos para interceptação[2] transparente de mensagens.

Mesmo assim, embora os resultados esperados não tenham sido atingidos, percebeu-se que existia um conjunto de componentes que poderiam ser extraídos e formar um núcleo de reutilização.

1.2.4 Fase 4 – IDECoRe

A ferramenta IDECoRe – ***I**ntegrated **D**evelopment **E**nvironment **C**ore **R**euse*, foi fundamentada, projetada e implementada com base na idéia de reutilizar, não apenas componentes, mas uma ferramenta como um todo. O objetivo dessa forma de reutilização é possibilitar a criação de ferramentas mais flexíveis e reutilizáveis, que foram denominadas derivações. IDECoRe é uma ferramenta que possui funcionalidades básicas, comum à maioria dos IDEs, e permite que algumas características sejam customizadas de acordo com a linguagem que pretende-se utilizar. Por exemplo, compilador e/ou interpretador, APIs ou bibliotecas disponíveis na linguagem.

A motivação para construir essa ferramenta surgiu durante o desenvolvimento da extensão do ambiente JReflex [BER 2000c, JRX 2001], pois nesse momento foi detectado que para construir um IDE com funcionalidades básicas não existiam ferramentas e/ou componentes de simples compreensão e fácil adaptação. Com isso, toda uma estrutura foi desenvolvida vislumbrando obter a reutilização futura do núcleo da ferramenta. Na verdade, foi exatamente o que aconteceu quando o ambiente JEduc [JED 2002, PER 2002] começou a ser desenvolvido. Esta derivação do JReflex foi desenvolvida como um IDE para a linguagem Java, especificamente para o ensino introdutório. Simplificar o desenvolvimento de programas Java, usando reflexão para esconder complexidades no tratamento de dados.

Em resumo, durante o desenvolvimento dessas ferramentas constatou-se a precariedade de métodos e ferramentas disponíveis para auxiliar no desenvolvimento de aplicações cujo foco são os RNFs. Assim, decidiu-se concentrar esforços na automatização dos processos de elicitação e modelagem de RNFs. Basicamente, a tecnologia adotada para a solução fundamentou-se em AOSD – ***A**spect-**O**riented **S**oftware **D**evelopment*. Além disso, tentando estabelecer uma continuidade dos trabalhos previamente desenvolvidos resolveu-se construir uma ferramenta que motivasse o desenvolvedor no uso do método definido.

---

[2] Esta deficiência foi parcialmente corrigida a partir da versão 1.3 do J2SDK - Java 2 Standard Development Kit - de 2001, com o oferecimento de *proxies* reflexivos.

1.3 Objetivos

Com base na literatura verifica-se que ainda não existe um consenso em relação ao processo de desenvolvimento de sistemas baseados em RNFs. O que existe é uma tendência dominante com relação ao uso de UML – ***U**nified **M**odeling **L**anguage* e suas extensões durante os diversos estágios do ciclo de vida de um software.

Examinando-se algumas metodologias, métodos e ferramentas para análise e projeto de sistemas disponíveis atualmente, verifica-se que as mesmas oferecem suporte a diversos recursos (Tabela 1.1).

Tabela 1.1: Comparação de Metodologias, Métodos e Ferramentas

Trabalho	M[3]	F[4]	UML	G[5]	A[6]
Chung [CHU 2000]	✓				
Cysneiros [CYS 2001]	✓	✓	✓		
Brito [BRI 2002]	✓		✓		✓
Rashid [RAS 2003]	✓		✓		✓
Grundy [GRU 2001a]	✓				✓
Sapir [SAP 2002]	✓		✓		✓
Zakaria [ZAK 2002]	✓		✓		✓
JMangler [JMA 2002]		✓		✓	✓
AspectBrowser [ASB 2002]		✓		✓	✓
AJDT [CLE 2003]		✓		✓	✓

Alguns exemplos de recursos oferecidos pelas ferramentas compreendem: armazenamento e uso compartilhado de modelos, suporte a alguma notação (geralmente, UML), engenharia reversa e geração automática de código. Neste último caso, percebe-se claramente que o código gerado (i) utiliza bibliotecas proprietárias (o que torna o código complexo e pouco intuitivo); (ii) gera código altamente entrelaçado, onde o código não consiste em uma reprodução fiel do projeto; ou ainda (iii) apenas esqueletos de código, que não estabelecem nenhuma distinção entre a parte funcional e a não funcional. Assim, verifica-se a carência de ferramentas de desenvolvimento que possibilitem a modelagem dos aspectos funcionais e não funcionais durante todas as fases do processo de desenvolvimento até o código em uma determinada linguagem de programação.

Ao efetuar a análise de alguns trabalhos identificou-se que apenas utilizar uma ferramenta não é o suficiente para se obter sucesso na separação de requisitos funcionais e não funcionais. A adoção de referencial teórico, bem como de passos bem definidos são essenciais para que a ferramenta alcance seus objetivos.

Segundo Tran [TRA 99] uma tática que deve ser considerada para contornar as dificuldades encontradas com a implementação de RNFs é considerá-los durante todo o ciclo de vida de desenvolvimento. Quando um requisito não funcional é identificado nos primeiros estágios do desenvolvimento os objetivos e intenções do cliente se refletem por todo o projeto da solução [TRA 99].

---

[3] M = Metodologia ou método
[4] F = Ferramenta
[5] G = Geração de código
[6] A = Modelo de aspectos

Assim, este trabalho tem como objetivo propor um método que propicie a elicitação e a modelagem de RNFs, desde as fases iniciais do ciclo de vida de uma aplicação até a codificação. Para tanto, o escopo deste trabalho parte de algumas premissas:

(i) adotar um processo de desenvolvimento iterativo permitindo o sistema seja projetado e construído de forma incremental, pois desse modo o cliente possui um retorno muito mais rápido sobre o sistema solicitado, e ainda a inclusão e/ou alteração dos requisitos ocorre de forma mais natural;

(ii) criar um método guiado por casos de uso uma vez que, com os casos de uso é possível compreender como o sistema será utilizado, além de oferecer ao cliente uma visão mais rápida e clara das funcionalidades do sistema;

(iii) cada fase do método é composta por artefatos que ora estão vinculados com os RFs – ***R****equisitos* ***F****uncionais*, ora com os RNFs;

(iv) utilizar o desenvolvimento orientado a aspectos para resolver problemas vinculados a código disperso e entrelaçado;

(v) requisitos funcionais são abstraídos na fase de projeto por alguma classe;

(vi) requisitos não funcionais são encapsulados por aspecto (fase de projeto);

(vii) estereótipos da UML adotados para realizar a modelagem dos aspectos.

Em resumo, pode-se afirmar que o objetivo geral deste trabalho é a concepção de um método fundamentado em AOSD para o desenvolvimento de sistemas que utilizam RNFs, englobando todo o seu ciclo de vida, e que permita a automatização via uma ferramenta de desenvolvimento. Para isto, foram estabelecidos mecanismos que propiciam a transição entre as diferentes fases de desenvolvimento. Ao longo deste trabalho, o termo "método FRIDA" é usado para designar o método e a ferramenta associada, onde FRIDA é o acrônimo de *"**F**rom **R**equ**I**rements to **D**esign using **A**spects"*.

1.4 Contribuições

A elicitação dos RNFs e sua propagação durante as demais fases do ciclo de vida compreendem itens fundamentais do método apresentado. Além disso, o uso de AOSD e OO (***O****rientação a* ***O****bjetos*) tem como objetivo oferecer uma separação de responsabilidades, ou seja, utilizando tanto classes quanto aspectos na solução. As principais contribuições podem ser enumeradas abaixo:

- componentes de programas: oferecer um modelo que estabeleça um vínculo entre os artefatos funcionais e os não funcionais;
- especificação e modelagem: estabelecer uma proposta de mapeamento e especificação para RNFs usando aspectos; e determinar formas de vínculo entre modelos de fases distintas;
- AOSD: descrever passo-a-passo um método que aborde aspectos, como elementos de primeira ordem, desde as fases iniciais do ciclo de vida do software; mapeamento da terminologia de AO para estereótipos; elaboração de uma BNF para aspectos, representação visual da separação de responsabilidades;
- tolerância a falhas: utilizando AOSD possibilitar a especificação e avaliação de requisitos de confiabilidade de um sistema; definição de um léxico de palavras comumente usadas na construção de aplicações tolerantes a falhas; e construção de uma *checklist* para uma melhor identificação dos atributos de tolerância a falhas na modelagem de uma aplicação.

Algumas contribuições adicionais do método também são esperadas:

- rastreabilidade: compreende a facilidade de encontrar os requisitos [COA 2003]. Isso significa que deve ser possível a partir do código ou do projeto, determinar os requisitos e vice-versa [FIR 2003]. No método descrito tentou-se determinar elementos que favoreçam a rastreabilidade. Isso é possível uma vez que as unidades de decomposição reproduzidas no código possuem um relacionamento direto com as unidades de projeto, e estas por sua vez encontram-se vinculadas aos requisitos que as originaram;
- reutilização: pretende-se atingir um maior nível de reutilização das especificações e do código decompondo o software em dois tipos de unidades de abstração: as classes e os aspectos;
- facilidade de entendimento e uso: as técnicas adotadas no desenvolvimento devem ser fáceis de entender e usar, pois isto garante um uso difundido. Mediante a adoção da UML pretende-se que o método facilite o uso de AOSD, uma vez que os elementos de AO encontram-se pouco disseminados;
- propagação de mudanças: compreende a facilidade de alterar o projeto ou o código quando qualquer mudança é introduzida em alguma parte da especificação. Com o método proposto pretende-se uma aproximação maior do caso ideal, o qual seria uma completa separação de responsabilidades possibilitando, assim, que modificações sejam realizadas em uma unidade sem que outras unidades sejam afetadas.

1.5 Estrutura do Texto

Para se entender o escopo e os pontos contemplados por um trabalho é necessário compreender os diversos critérios fundamentais nele envolvidos. Assim, para facilitar o entendimento, além de maximizar a compreensão desta tese foi criada a presente seção, que descreve resumidamente os dez capítulos integrantes desta tese:

- **Capítulo 2 – Fundamentos Teóricos**: aborda o referencial teórico usado para desenvolver o presente trabalho;
- **Capítulo 3 – Requisitos Não Funcionais**: apresenta as principais classificações para RNFs, bem como os principais problemas e soluções encontrados na literatura;
- **Capítulo 4 – AOSD Terminologia**: enumera alguns conceitos fundamentais usados no modelo de orientação a aspectos;
- **Capítulo 5 – Trabalhos Relacionados:** descreve e compara diversos trabalhos relacionados com o método FRIDA (nas diversas fases de desenvolvimento);
- **Capítulo 6 – O Método FRIDA:** aborda os passos do método e seus artefatos essenciais. Além, de apresentar os principais conceitos e características englobadas pela ferramenta;
- **Capítulo 7 – Estudos de Caso**: descreve o estudo de caso construído para validar o método desenvolvido. Neste capítulo, é descrito um estudo de caso, e sua respectiva solução aplicando-se o método FRIDA;
- **Capítulo 8 – Resultados**: apresenta e tabula os resultados obtidos a partir da avaliação do método FRIDA por alguns desenvolvedores;
- **Capítulo 9 – Conclusões:** descreve as impressões finais a respeito do trabalho.
- **Capítulo 10 – Trabalhos Futuros**: enumera as perspectivas futuras com relação ao trabalho.

2 FUNDAMENTOS TEÓRICOS

Este capítulo dedica-se a apresentar alguns conceitos fundamentais, relacionados com a área de engenharia de software, que orientam o desenvolvedor na construção de sistemas confiáveis e reutilizáveis. Para esta finalidade ele encontra-se dividido em seções: seção 2.1 apresenta a definição de processo de desenvolvimento de software, mecanismo fundamental para se obter sucesso na construção de qualquer sistema computacional; seção 2.2 descreve os fundamentos básicos do desenvolvimento de sistemas baseado em componentes; seção 2.3 aborda o conceito de arquitetura de software e como suas propriedades podem influenciar o sistema resultante; seção 2.4 introduz os conceitos, vantagens e exemplos de ferramentas de apoio ao desenvolvimento e finalmente; na seção 2.5 é introduzida a notação UML, visto que consiste na notação mais usada para a modelagem de sistemas.

2.1 Processo de Desenvolvimento de Software

Um processo de desenvolvimento de software é um conjunto de etapas, métodos e técnicas que utilizam pessoas para o desenvolvimento e manutenção de um software e seus artefatos associados (planos, requisitos, documentos, modelos, código, casos de teste, manuais, entre outros) [COA 2003]. Em resumo, um processo de desenvolvimento é um conjunto de passos bem definidos e ordenados que devem ser seguidos para se atingir um objetivo. No caso da engenharia de software o objetivo consiste na construção de um software [JAC 99].

A noção de processo de software é construída sobre o conceito de ciclo de vida do software. Ele define os princípios e as regras básicas de acordo com os estágios do ciclo de vida (por exemplo, o modelo cascata sugere que uma fase específica deve iniciar somente quando for atingido o término da fase anterior) [COA 2003, FUG 2000].

Nas últimas décadas, devido à crescente preocupação com a qualidade de software, os processos de desenvolvimento de software têm recebido considerável atenção [FUG 2000]. É possível encontrar na literatura diversos exemplos de processos de desenvolvimento, mas os que mais se destacam compreendem: Catalysis [DES 98], UML Components [CHE 2001] e o mais conhecido RUP – ***R**ational **U**nified **P**rocess* [JAC 99].

Atualmente, a categoria de processo de desenvolvimento de software mais adotada é "iterativo e incremental", ou seja, a análise/projeto/construção do sistema é organizada em uma série de pequenos mini-projetos denominados iterações.

O resultado gerado por cada iteração é um sistema executável testado e integrado, onde cada iteração representa um ciclo de desenvolvimento completo (Figura 2.1) [BOO 99, COA 2003, FUG 2000, PRS 2001, SOM 2001]: (i) análise especificação de requisitos; (ii) análise e projeto, (iii) implementação e (iv) testes.

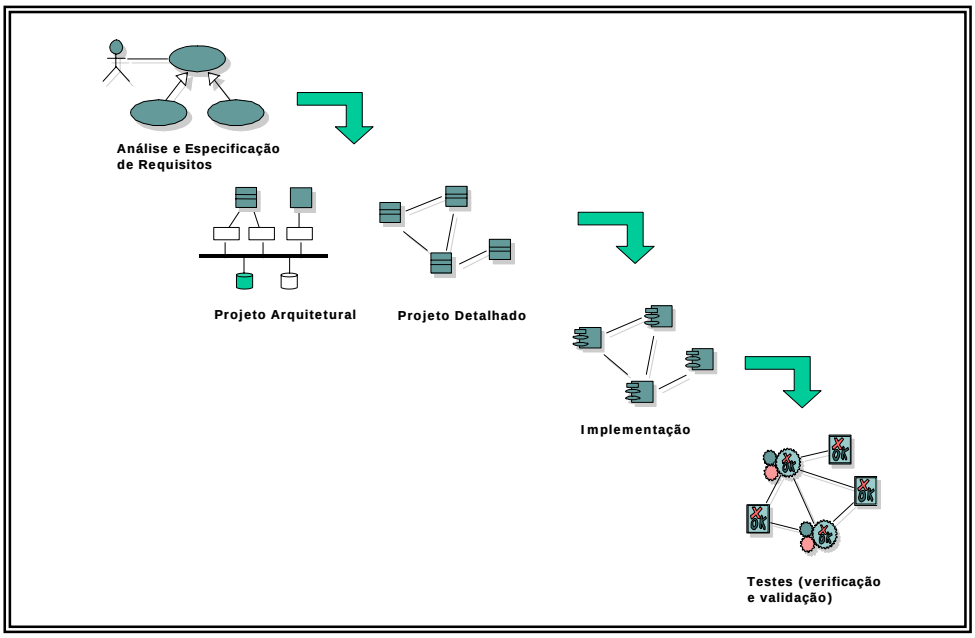

Figura 2.1: Fases do Ciclo de Vida

A primeira fase, análise e especificação de requisitos, concentra-se na identificação das funcionalidades que devem ser automatizadas e quais informações devem ser processadas. Essa fase concentra-se nos objetos de negócio, sua descrição funcional, podendo oferecer enfoque também para os aspectos não funcionais (tais como desempenho, confiabilidade, disponibilidade e segurança). Nessa etapa, o modelo de casos de uso é uma das técnicas mais empregadas [BOE 74, BRI 2002, BOO 99, JAC 99].

Durante a etapa de projeto os desenvolvedores concentram-se na definição da solução, ou seja, com a seleção da linguagem, dos componentes, da infra-estrutura necessária, da base de dados apropriada, com estruturação da arquitetura de software e com a caracterização das interfaces gráficas. A definição da arquitetura é muito importante para a construção de um sistema, pois as características arquiteturais influenciam, direcionam e restringem todas as fases do ciclo de vida do software.

O foco da fase de implementação é a tradução de todas as decisões de projeto para uma linguagem de implementação na forma de código fonte. Nesta fase é possível realizar o teste de unidade. A etapa de testes existe para que o sistema seja verificado e validado. Na verdade, essa última etapa foi criada para que o desenvolvedor certifique-se de que os requisitos especificados estão corretamente implementados.

2.2 Desenvolvimento Baseado em Componentes

O desenvolvimento de software baseado em componentes (CBSE - ***C**omponent-**B**ased **S**oftware **E**ngineering*) consiste em identificar blocos de construção reutilizáveis denominados componentes de software [GRU 2001a].

Existem inúmeras definições para componentes de software, mas de forma bem genérica pode-se afirmar que um componente é uma unidade executável desenvolvida e testada de forma isolada. Além disso, ele é distribuído como uma unidade que pode ser formada ou combinada com outros componentes, visando a construção de um sistema computacional [PRE 97].

O CBSE permite a configuração dos componentes para diferentes situações de reutilização e, algumas vezes, possibilita que os componentes sejam conectados à aplicação em tempo de execução [GRU 2001a]. Isso deve-se ao fato de que em uma aplicação baseada em componentes deve ser possível substituir um componente por outro. Deve ficar claro que o componente que substituirá necessita apresentar uma especificação equivalente à do substituído, pois essa ação não pode afetar a conexão e a funcionalidade disponível [PRE 97]. Evidentemente, isto só é possível porque a conexão entre os componentes ocorre exclusivamente pelas suas interfaces públicas, não existindo qualquer espécie de dependência da implementação.

O desenvolvimento baseado em componentes não é uma atividade trivial, pois é necessário: (i) identificar abstrações apropriadas, (ii) atribuir responsabilidades para os componentes, visando que o mínimo de duplicação e inconsistência ocorra, e (iii) ter prudência na combinação e configuração dos componentes.

Geralmente, costuma-se associar a reutilização de componentes com a idéia de que basta integrá-los a uma aplicação e reutilizá-los da forma como foram implementados. Porém, normalmente o componente deve ser adaptado a fim de adequá-lo ao contexto da aplicação.

Muitas tecnologias e métodos de desenvolvimento voltados para o uso de componentes têm sido definidas para auxiliar o desenvolvedor, entre elas destacam-se EJBs - ***E**nterprise **J**ava **B**eans*, COM+ (***C**omponent **O**bject **M**odel*) e CORBA – ***C**ommon **O**bject **R**equest **B**roker **A**rchitecture*. Um dos trabalhos mais recentes nessa área é o proposto por Grundy [GRU 2001a], o qual descreve como usar UML e aspectos para o desenvolvimento de componentes.

Na verdade, o modelo baseado em aspectos está se tornando um dos mais promissores para auxiliar o desenvolvimento de componentes, porque ele apresenta inúmeras vantagens [HIG 99, KIC 97, OSS 2001b]: (i) maior capacidade de adaptação, extensão e integração, (ii) redução do código disperso e (iii) minimização do entrelaçamento das propriedades.

A forma como o componente é especificado é mais importante do que o modo como a especificação será implementada. Dessa maneira, é indispensável o uso de um processo de desenvolvimento que assegure uma especificação completa e flexível dos componentes.

2.3 Arquitetura de Software

Um problema crítico no projeto e construção de qualquer sistema de software compreende a sua arquitetura, pois ela determina como estão organizados os elementos e como eles interagem entre si. A arquitetura de software possui um papel chave, uma verdadeira ponte de ligação entre os requisitos e a implementação, conforme ilustra a Figura 2.2 [SHA 96].

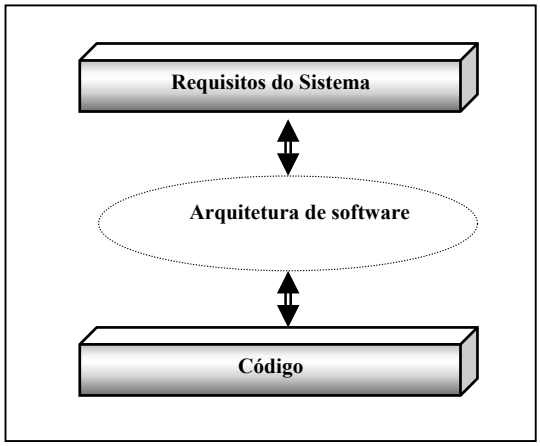

Figura 2.2: Arquitetura de Software – Ponte de Ligação

Na verdade, uma boa arquitetura pode ajudar a garantir que um sistema satisfaça requisitos chaves como, por exemplo, desempenho, confiabilidade, portabilidade, e interoperabilidade [SHA 96]. Dessa forma, pode-se afirmar que uma propriedade arquitetural costuma representar uma decisão de projeto relacionada com algum requisito não funcional [SOM 2001]. Alguns exemplos são:

- **modificabilidade:** define a capacidade do sistema de se adaptar a alterações/inclusão de requisitos. Depende de como o sistema foi modularizado, porque isto será refletido nas estratégias de encapsulamento;
- **reusabilidade:** define o grau de reutilização de um componente, isto é, o quanto este componente é independente do contexto. Além disso, é dependente do nível de acoplamento dos componentes do sistema;
- **desempenho:** consiste no tempo de resposta, ou de processamento, de uma requisição o qual deve ser compatível com as necessidades do cliente. Essa propriedade está relacionada à complexidade de comunicação entre os componentes, visto que eles costumam encontrar-se distribuídos fisicamente;
- **tolerância a falhas:** é capacidade do sistema reagir ou recuperar-se diante de situações excepcionais. Essa propriedade pode ser definida somente através da aplicação de técnicas de redundância de software e/ou hardware e do tratamento de exceções.

Os estilos arquiteturais costumam ser utilizados para garantir a preservação de uma determinada propriedade arquitetural durante o desenvolvimento do sistema.

Conforme pode ser observado até o presente momento, vários fatores devem ser considerados na definição e construção dos sistemas computacionais modernos. Nenhum processo de desenvolvimento pode ser eficientemente aplicado sem a utilização de ferramentas de apoio informatizado à construção e manutenção dos artefatos envolvidos.

2.4 Ferramentas CASE

O uso de ferramentas para auxiliar o desenvolvimento de software existe desde os primeiros montadores. Existem os mais diversos tipos de ferramentas, desde editores, compiladores até ambientes de engenharia centrados no processo que cobrem todo o ciclo de vida do software, ou pelo menos parte dele [HAR 2000].

Os primeiros esforços na produção de ferramentas foi na área de programação (PSEs - ***P**rogramming **S**upport **E**nvironments*), que correspondem a uma coleção de ferramentas que suportam apenas a atividade de codificação. Os PSEs foram e continuam sendo poderosas ferramentas que ainda são usadas pela maioria dos desenvolvedores. A maior limitação, entretanto, é que elas suportam somente uma atividade da engenharia de software – a implementação [HAR 2000].

O fato dos PSEs contemplarem apenas uma fase do ciclo de vida do software é prejudicial, porque todas as atividades estão interconectadas e muitas vezes os artefatos produzidos por uma etapa são utilizados como entrada na próxima etapa [HAR 2000]. Diferente dos PSEs os SEEs - ***S**oftware **E**ngineering **E**nvironments*, fornecem suporte integrado às atividades de engenharia de software através do ciclo de vida do software [HAR 2000].

Outros ambientes são os de suporte semi-automatizado para o processo de desenvolvimento. Esses apresentam uma representação explícita dos processos, seus produtos e suas interações. Nesse tipo de ambiente o próprio processo de engenharia de software deve ser tratado como um pedaço de software. Ele dá origem aos PSEEs (***P**rocess-centered **S**oftware **E**ngineering **E**nvironments*), os quais integram ferramentas para desenvolvimento de artefatos. Além disso, oferecem suporte para a modelagem e execução do processo de engenharia de software que produziu os artefatos.

As ferramentas podem ser agrupadas conforme a sua funcionalidade. Assim, as ferramentas usadas para a fase de requisitos auxiliam a elicitação, codificação, validação e evolução dos requisitos do sistema. Ferramentas para arquitetura de software e projeto visual são usadas para modelar e refinar decisões de implementação, além de possibilitar a geração automática de esqueletos de código e engenharia reversa. As ferramentas denominadas "Interface com o Usuário" incluem sistemas de gerenciamento de interfaces gráficas, fornecendo um ambiente para projeto gráfico.

PSEs e IDEs estão focalizados em oferecer soluções para a etapa de implementação. Costumam ser limitados a uma linguagem e algumas vezes incorporar documentação, visualizadores de código e depuradores. Os exemplos mais conhecidos desse tipo de ambiente compreendem: Visual Age[7], Visual Basic[8], Visual C^{++}[8], JBuilder[9] e Delphi[9].

Ferramentas para depuração, teste e monitoramento costumam ser usadas para avaliar a qualidade do software, ou para auxiliar os desenvolvedores na identificação e correção de erros no software. As ferramentas de depuração concentram-se na análise do código fonte, geralmente oferecendo facilidade de execução passo a passo, monitoramento de variáveis, etc.

Uma variedade de ferramentas de testes tem sido desenvolvidas para controlar e validar os artefatos envolvidos no software. Outras ferramentas encontradas são as de controle de versões e gerenciamento de configurações. As primeiras são usadas para

---

[7] Visual Age™ IBM
[8] Visual Basic™ e Visual C^{++}™ Microsoft
[9] JBuilder™ e Delphi™ Borland

gerenciar a criação, a organização e o uso de múltiplas versões dos artefatos de software.

A Figura 2.3 ilustra a distribuição das ferramentas através do ciclo de vida de um software. Essa figura foi extraída e adaptada do trabalho de Grundy [GRU 2001b]. Foram omitidas algumas categorias de ferramentas, pois não fazem parte do escopo deste trabalho.

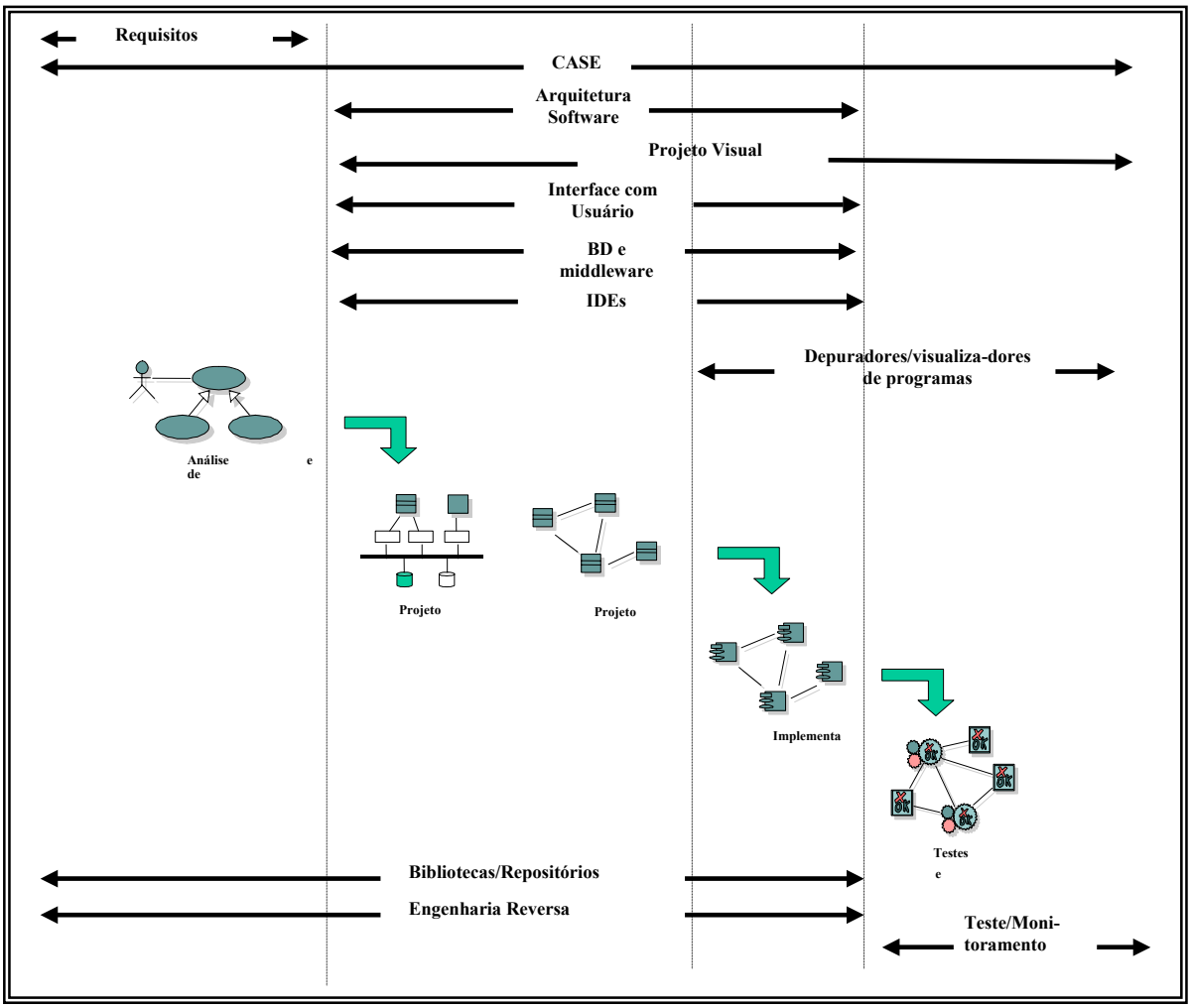

Figura 2.3: Ferramentas e o Ciclo de Vida do Software – adaptada de [GRU 2001b]

Conforme pode-se observar anteriormente, as ferramentas CASE – ***C****omputer* ***A****ided* ***S****oftware* ***E****ngineering* - representam um papel fundamental na solução para os problemas de desenvolvimento e manutenção de sistemas. Se usadas adequadamente, estas ferramentas proporcionam os seguintes benefícios [SOM 2001]:

- aumento da qualidade do software: diminuem a probabilidade de erros através de verificação automatizada da consistência dos modelos, integridade de dados e testes;
- manutenção de programas simplificada: através da automatização da documentação, atualização de modelos através de recursos como engenharia reversa e atualização de código com base em alterações no modelo;
- aumento da produtividade de equipes de desenvolvimento através de recursos como controle de versão e compartilhamento de informações.

Em resumo, pode-se determinar que as ferramentas são usadas para descrever os processos de software, planejar e gerenciar as equipes de trabalho, capturar informações e aperfeiçoar os processos.

As ferramentas CASE estão voltadas para as fases de análise e projeto, pois costumam fornecer diagramas e modelos de análise. Além disso, essas ferramentas freqüentemente fornecem outras funcionalidades como geração de código, documentação e engenharia reversa. Uma característica chave dessas ferramentas tem sido o uso parcial ou total da linguagem UML. Isso porque atualmente a UML é a linguagem padrão para modelagem de sistemas computacionais.

2.5 UML e o Desenvolvimento de Software

Um processo é sustentado por vários conceitos e a notação é responsável por expressar estes conceitos. A notação é apenas uma linguagem que deve ser utilizada em conjunto com um processo. O uso de uma notação provê uma linguagem padronizada e organizada que facilita o entendimento entre usuários, gerentes e desenvolvedores. Atualmente, a maioria das organizações utilizam a UML como linguagem de comunicação padrão [OMG 2001].

A UML, conforme definido pela OMG – ***O****bject* ***M****anagement* ***G****roup*, pode ser vista como uma maneira significativa para "capturar, comunicar e influenciar o conhecimento em uma organização" [OMG 2001].

Na verdade, a UML é uma notação gráfica definida segundo uma semântica de meta-modelos, a qual define uma forma de capturar e comunicar a estrutura e comportamento de um objeto [ALD 2001].

A UML oferece como forma de separação de responsabilidades a decomposição do sistema em quatro dimensões [LIO 2002]:

1. dimensão funcional - diagramas de caso de uso:
   - são usados para expressar os requisitos do sistema que esta sendo modelado;
   - para mapear aspectos nesses diagramas são necessárias descrições adicionais.
2. dimensão estática - diagrama de classes e pacotes
   - fornece um núcleo para a modelagem de conceitos OO;
   - para descrever aspectos utiliza estereótipos e *tagged-values*.
3. dimensão dinâmica - diagramas de seqüência, colaboração, estado e atividades
   - esta dimensão possibilita ao desenvolvedor descrever vários aspectos comportamentais do sistema;
   - o impacto dessa dimensão em AOSD é a habilidade para modelar os efeitos da composição de aspectos visualmente, além dos *advices* relacionados com cada aspecto.
4. dimensão física - diagramas de componentes e implantação
   - representa as características de implementação é especificada independentemente das outras três dimensões, proporcionando uma completa separação de responsabilidades do projeto e características de implementação.

A UML é usada freqüentemente para a modelagem de requisitos funcionais. Para modelar requisitos não funcionais é necessário estender ou customizar os elementos dos modelos da UML [ALD 2001, SAP 2002, ZAK 2002].

Um dos pontos fundamentais da UML compreende os mecanismos de extensão [OMG 2001], os quais habilitam o uso da linguagem nos mais diferentes tipos de sistemas, domínios, métodos e processos [OMG 2001]. A UML fornece três mecanismos de extensão [ALD 2001, FON 2000, OMG 2001]: estereótipos, *tagged values* e restrições (*constraints)*.

Os estereótipos são usados para introduzir um novo tipo de elemento a um dado modelo. Esse novo tipo de elemento deve possuir a mesma estrutura que o bloco a partir do qual é derivado. Por exemplo, estereótipos para o diagrama de classes devem possuir atributos, associações, operações. A estrutura sintática é a mesma, porém eles podem possuir diferentes semânticas e podem especificar regras adicionais usando para tanto *tagged values* e restrições [ALD 2001].

Os *tagged values* são usados para estender as propriedades de um elemento, também conhecidas como meta-propriedades. Eles são usados para atribuir informações arbitrárias aos elementos do modelo (por exemplo informação de gerenciamento (autor, data, estado) ou informação relacionada com a geração de código (nível de otimização, classe *contêiner*, entre outros)). Uma *tagged value* é basicamente um par <*nome-valor*>, onde ambos são descritos utilizando-se um conjunto de caracteres descritivo [FON 2000]. A sua representação no modelo é dada pelo nome (*tag*) e por um valor de atribuição entre chaves (*{tag=value}*), conforme esquematiza a Figura 2.4.

Convém observar que tanto um estereótipo quanto um *tagged value* podem ser usados para o mesmo propósito, mas cada elemento pode ter um único estereótipo, enquanto o número de *tagged values* pode ser ilimitado [ZAK 2002].

Com os *constraints* novas semânticas podem ser especificadas para qualquer elemento do modelo usando uma determinada linguagem de restrição (OCL – ***O****bject* ***C****onstraint* ***L****anguage*). Para representar uma restrição em um modelo deve-se colocar entre chaves o nome da restrições e o seu significado (*{constraint expression}*).

Por exemplo, na linguagem Java existe uma estrutura denominada *interface* que possibilita a criação de elementos semelhantes às classes, mas que caracterizam-se por possuir apenas constantes e métodos abstratos. Para representar esse elemento na UML podem ser utilizados dois tipos de estereótipos: o gráfico ou o textual [BOO 99]. Na Figura 2.4 pode-se observar o uso do estereótipo <<*interface*>> criado para diferenciar esse elemento estrutural das classes.

Nesse contexto, essa estrutura (*interface*), por restrições da linguagem, não pode ser instanciada, ou seja, não é permitida a criação de objetos a partir de uma interface. Baseado no que foi descrito anteriormente, o uso de restrições é o modo mais adequado para representar o comportamento restritivo de um elemento em um diagrama de classes da UML.

Além disso, a linguagem Java possui modificadores de acesso (*static, final, synchronized, volatile, transient*, entre outros) que não possuem representação na UML padrão. Para contornar isso pode-se associar uma *tagged value*.

No exemplo ilustrado pela Figura 2.4 pode-se observar que o atributo *numObj* da classe *Pessoa* é considerado uma constante e uma variável de classe. Isso significa que significa que ele deve ser declarado como um atributo *static final*. Na UML as variáveis de classe são representadas por um sublinhado, já constantes não possuem representação, logo é atribuído para a *tagged value final* o valor verdadeiro.

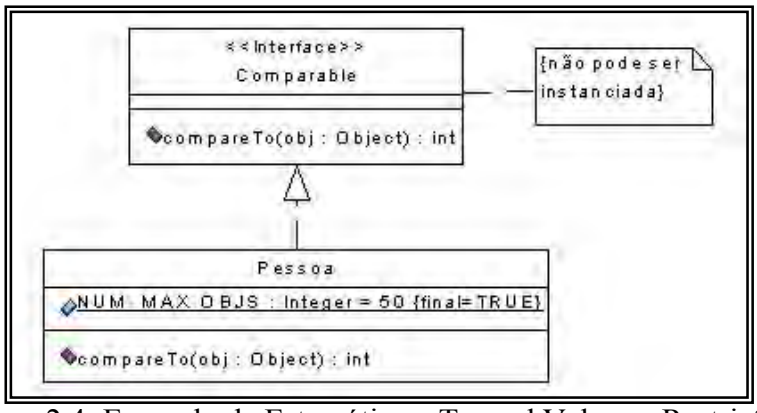

Figura 2.4: Exemplo de Estereótipos, Tagged Values e Restrições

2.6 Conclusões

Conforme pode-se perceber, nesta seção, vários conceitos e técnicas exercem influência no desenvolvimento de um software. Além disso, é possível observar que grande parte desses mecanismos encontra-se voltado para o contexto funcional da aplicação, ou seja, tudo que é considerado não funcional exerce, praticamente, nenhuma influência.

3 REQUISITOS NÃO FUNCIONAIS

A tradução dos RNFs pelas diversas fases do processo de desenvolvimento é um item fundamental para a construção de sistemas computacionais. Assim, este capítulo foi criado visando realizar uma introdução aos RNFs e algumas de suas principais classificações. Além disso, são enumerados os principais problemas que ocorrem com a adoção de RNFs, bem como a descrição de algumas soluções adotadas na resolução dos problemas relacionados com os RNFs.

3.1 Introdução

Os sistemas modernos demandam que o foco do desenvolvimento migre de uma visão puramente funcional, para uma que lide também com propriedades não funcionais. Essas propriedades, também denominadas requisitos não funcionais, compreendem as tarefas que servem de suporte para que a funcionalidade seja contemplada. Ou ainda, os RNFs são propriedades ortogonais as classes.

Segundo Cysneiros [CYS 97] para que um software possua qualidade deve ser construído considerando-se os RNFs. Assim, pode-se concluir que os RNFs encontram-se diretamente relacionados com a qualidade do software e ainda, a sua elicitação influencia positivamente um software. Em contrapartida, existe a possibilidade da elicitação incompleta, imprecisa ou inexistente desses requisitos levando a: inconsistência, insatisfação de clientes e desenvolvedores; dimensionamento incorreto do tempo e do custo de desenvolvimento [CYS 2001].

Incorporar RNFs em qualquer tipo de software é uma das atividades mais complexas, devido principalmente à natureza inerente dos RNFs [SUB 2001]. Além disso, esses requisitos são freqüentemente vagos, porque eles podem ser vistos e avaliados de maneiras diferentes por diferentes grupos de pessoas [CHU 95, CHU 2000].

Os RNFs podem ser tratados de forma qualitativa e/ou quantitativa [CHU 93]. No primeiro caso enquadram-se as abordagens orientadas a processo [BAS 91, BOE 78], ou seja, os RNFs afetam o processo de desenvolvimento (como por exemplo usar a plataforma X, manter o custo menor que Y, integrar com o sistema Z). Por outro lado, as abordagens quantitativas são orientadas a produto [KEL 90], isto é, concentram-se em requisitos que afetam o software (ciclo de desenvolvimento do software), como por exemplo, o sistema deve ser seguro e confiável, o sistema deve oferecer suporte à características de tempo real, etc.

Grande parte dos trabalhos encontrados na literatura [BAS 91, FIN 96, KEL 90] preocupam-se exclusivamente com a relação dos RNFs e o seu grau de conformidade, ou seja, se o sistema desenvolvido atende aos RNFs que deveria satisfazer. Poucos trabalhos preocupam-se em propor um tratamento explícito para RNFs durante o

processo de desenvolvimento de software [CHU 2000]. Porém, os erros ocasionados pela omissão ou avaliação parcial de um RNF – ***R**equisito **N**ão **F**uncional* - são apontados como erros caros e difíceis de corrigir [LIN 93]. Para evitar esses problema pensou-se em fundamentar este trabalho na abordagem orientada a produto, onde cada RNF será considerado um elemento essencial para o processo de desenvolvimento.

3.2 Classificações dos RNFs

Os RNFs foram primeiramente classificados como atributos de qualidade por McCall e Boehm, respectivamente em [BOE 78, McC77]. Yeh [YEH 84] foi o primeiro pesquisador a introduzir o conceito de requisitos não funcionais. A partir desse trabalho vários outros pesquisadores perceberam a necessidade de considerar os RNFs nas diversas fases do processo de desenvolvimento do software. Roman [ROM 85] incluiu em sua taxonomia de engenharia de requisitos os RNFs, apoiando as idéias de Yeh sobre a necessidade da elicitação e modelagem dos RNFs.

Com o tempo percebeu-se que o processo de elicitação e modelagem de requisitos necessita de um certo grau de sistematização. Isso pode ser obtido através de metodologias de projeto, que fornecem uma estrutura dentro da qual o processo de projeto ocorre e através de notações ou linguagens para representar os requisitos e suas características.

Toda metodologia e notação pressupõe uma classificação de requisitos. A literatura propõe tradicionalmente a caracterização de requisitos funcionais e não funcionais de forma que [BOE 76, LEI 95, MAC 99, MAM 99, SOM 2001]:

- requisitos funcionais: definem as funções que o sistema ou seus componentes devem executar;
- requisitos não funcionais: também referenciados como requisitos de qualidade. Incluem limitações no produto (desempenho, interface do usuário, confiabilidade, segurança, interoperabilidade) e no processo de desenvolvimento (custos e atrasos, metodologias a serem adotadas no desenvolvimento, componentes a serem reutilizados, etc.).

Na literatura é possível encontrar diversas classificações dos RNFs. A Figura 3.1 ilustra uma taxonomia mais detalhada de RNFs apresentada por Mamani [MAN 99] e Macedo [MAC 99].

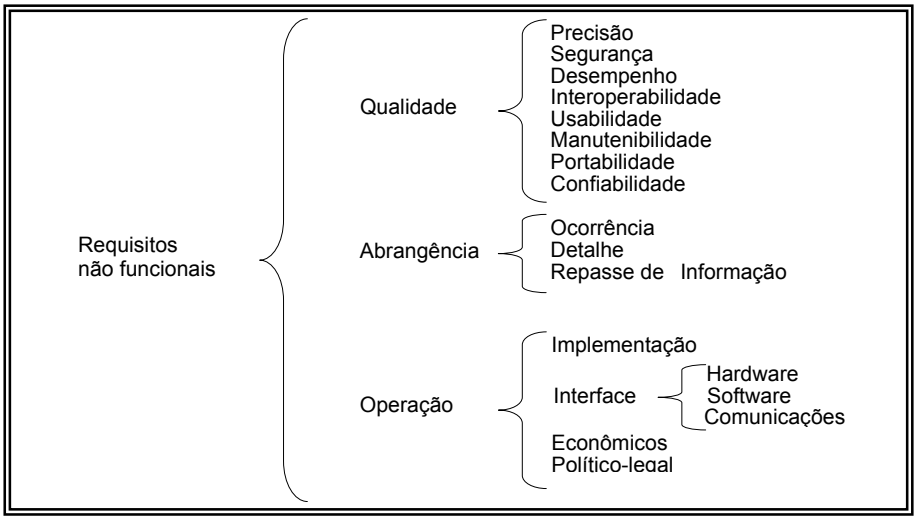

Figura 3.1: Classificação de RNFs Mamani e Macedo [MAN 99, MAC 99]

Os RNFs podem ser classificados em termos de qualidades que o software pode possuir [BOE 78]. Assim, Sommerville [SOM 2001] apresenta uma classificação mais geral que distingue os requisitos de produto, processo e externos, conforme esquematiza a Figura 3.2.

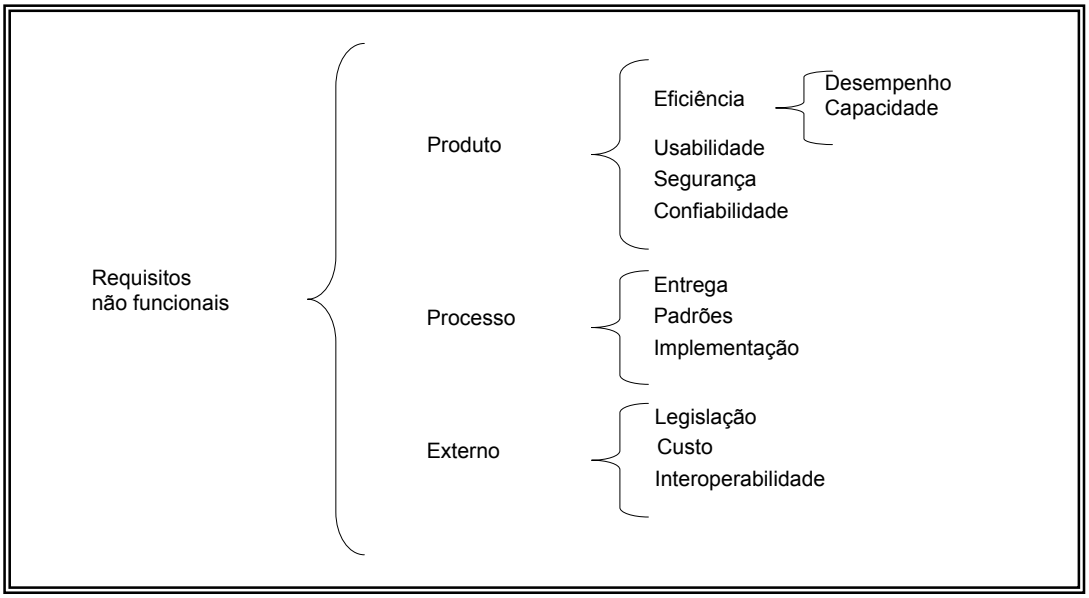

Figura 3.2: Classificação de RNFs Sommerville [SOM 2001]

Outra taxonomia considerada neste trabalho é a definida por Pimenta [PIM 97], conforme ilustra a Figura 3.3. Essa taxonomia é uma forma de evitar que os requisitos de interação sejam tratados dispersamente junto com outros requisitos não-funcionais. Dessa maneira, procura-se minimizar a situação onde os requisitos de interação são insuficientemente determinados e pobremente expressos.

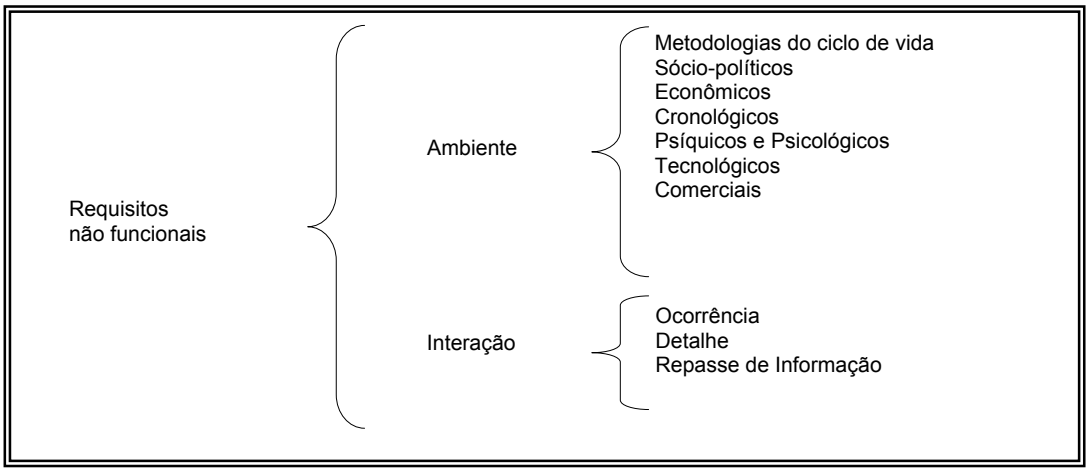

Figura 3.3: Classificação de RNFs Pimenta [PIM 97]

O padrão internacional, também denominado modelo de qualidade, ISO 9126-1 (***I**nternational **O**rganization for **S**tandardization*) pode ser considerado outra classificação encontrada na literatura. Basicamente, é adotado como referência para avaliação da qualidade de um software [ROC 2001]. Ele indica que um sistema de software de qualidade deve possuir as seguintes características [COR 2001, ISO 92]: funcionalidade, confiabilidade, usabilidade, eficiência, manutenabilidade e portabilidade.

Nessa classificação cada atributo de qualidade podem ser decomposto em subcaracterísticas. Além disso, percebe-se claramente que o item funcionalidade não pode ser enquadrado como RNF. A Figura 3.4 esquematiza o relacionamento dos RNFs segundo esse padrão.

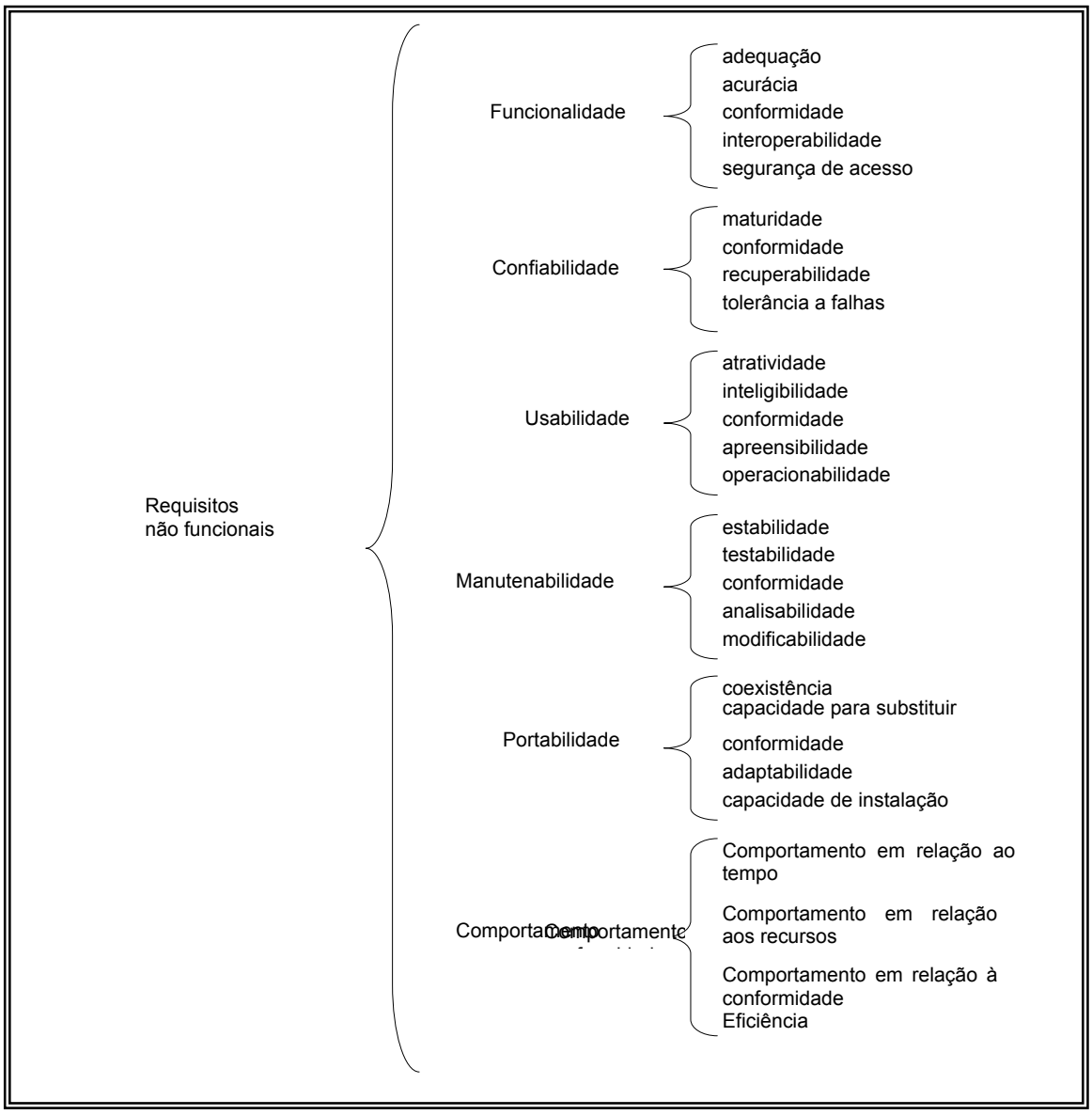

Figura 3.4: Classificação de RNFs ISO 9126-1 [ISO 92]

Para este trabalho é proposta uma taxonomia baseada nos demais trabalhos encontrados na literatura, mas cujo foco principal seja sistemas tolerantes a falhas. Para tanto, os RNFs são divididos em três contextos genéricos. Cada RNF genérico é decomposto em RNFs específicos, os quais descrevem ou detalham os genéricos. Um RNF específico pode sofrer sucessivos refinamentos até obter-se um alto nível de abstração.

A Figura 3.5 ilustra a classificação proposta para este trabalho, bem como apresenta os RNFs que serão considerados no decorrer de todo o trabalho.

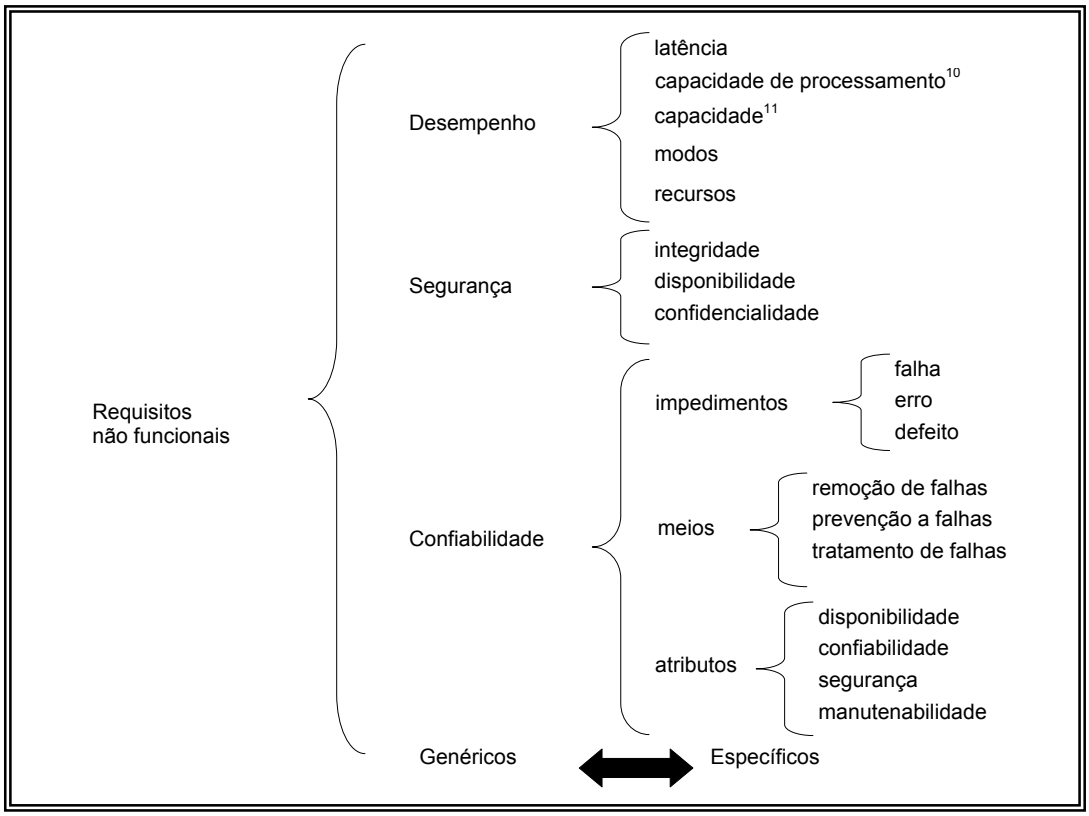

Figura 3.5: Classificação de RNFs Proposta

A elicitação de um RNF é fortemente influenciada pelos problemas que o desenvolvedor irá enfrentar ao usar um dado RNF. A seguir são identificados diversos problemas que podem ser enfrentados pelo desenvolvedor, assim como possibilidades de solução.

3.3 Problemas Identificados

A adoção dos requisitos não funcionais no desenvolvimento de sistemas ocasiona vários problemas para o desenvolvedor. Dentre os principais, alguns estão enumerados a seguir:

(i) modelagem incompleta: os requisitos funcionais de um sistema são representados ou modelados usando conceitos, tais como classes, objetos, atributos, operações e associações entre classes. Por outro lado, os RNFs raramente são modelados, porque eles não estão relacionados com as entidades do mundo real, mas compreendem propriedades dos objetos de software, que representam essas entidades [CAZ 99];

(ii) representação dos RNFs: notações, meta-modelos, métodos, metodologias usados para modelagem orientada a objetos convencional, geralmente não são aplicados diretamente na modelagem de requisitos não funcionais [CAZ 2000b];

[10] do inglês = throughput
[11] do inglês = capacity

(iii) transição da análise para o projeto: requisitos funcionais são relativamente fáceis de ser traduzidos para elementos da fase de projeto, sendo que o mesmo não ocorre com RNFs [CAZ 2000a];

(iv) transparência: as abordagens tradicionais que tratam com requisitos não funcionais não costumam ser transparentes para o desenvolvedor da aplicação, aumentando dessa forma a complexidade do software [STR 96];

(v) código altamente entrelaçado: o que costuma ocorrer em sistemas que utilizam RNFs é que o código relacionado com a funcionalidade do sistema fica muito misturado ao código não funcional, o que reduz as possibilidades de modificação e reutilização [KIC 97];

(vi) código disperso: é o código responsável por atender um requisito, mas que se encontra distribuído por mais de uma unidade funcional [LAD 2002a]. Assim, as probabilidades de reutilização diminuem e a produtividade é afetada. Além disso, o código fonte fica difícil de entender, desenvolver e evoluir [CAZ 2000b, CLA 2001, CON 2000a, KIC 97, MEN 97].

Os itens acima descrevem, de forma resumida, os principais problemas que os RNFs introduziram no desenvolvimento de software evidenciando sua abrangência a todas as fases do processo de desenvolvimento.

Várias abordagens tem sido usadas para solucionar ou amenizar os problemas citados. Estas incluem: (i) projetar para a mudança (usando técnicas como padrões de projeto), (ii) realizar refatoração de partes do código periodicamente, (iii) usar múltiplas versões de componentes para representar diferentes conjuntos de características. Essas soluções fornecem alguma forma de flexibilidade para que o código passe por uma evolução, porém elas fracassam em fornecer um método de modularização de todos os tipos de responsabilidades.

Com a intenção de reverter essa situação vários pesquisadores definiram diversas soluções, a maioria das quais desenvolvidas visando apenas algumas fases do processo de desenvolvimento. Atualmente, as soluções mais adotadas são baseadas em técnicas de decomposição, ou seja, aquelas que realizam uma verdadeira separação de responsabilidades.

3.4 Soluções Empregadas

Kiczales [KIC 97] observa que os atuais métodos e notações concentram-se em encontrar e compor as unidades funcionais, as quais são representadas por procedimentos, funções, módulos, objetos, etc. Porém, dessa forma, aspectos não funcionais que são importantes para preservar a funcionalidade da aplicação, não são tratados. Esse fato é decorrente, principalmente, de algumas limitações das linguagens de programação.

Vários estudos e experiências práticas determinaram que existem inúmeras limitações no modelo de desenvolvimento atual – o orientado a objetos. Isso ocorre, porque, nesse modelo, o objeto é uma unidade indivisível que executa um conjunto de ações [ELR 2001a]. O código do objeto especifica o seu comportamento como uma totalidade, isto é, qualquer ação deve ser implementada no mesmo lugar, mesmo se seus objetivos são diferentes (funcional x não funcional) [HER 2000].

Conforme pode-se observar anteriormente, a maioria dos problemas está relacionada principalmente com reutilização, adaptabilidade, gerenciamento da complexidade, qualidade, rastreabilidade e desempenho [CZA 98, LAD 2002a]. Além de metodologias e técnicas de separação de responsabilidades inadequadas [HÜR 95].

Visando reduzir a complexidade do projeto de programas e seu código de implementação, vários pesquisadores iniciaram a definição de novos estilos de programação, que implicam o aumento da expressividade do paradigma orientado a objetos. Elrad, em [ELR 2001a], classifica esse novos estilos de programação como POP[12] – ***P**ost-**O**bject **P**rogramming* - destacando-se [ELR 2001a, VRA 2001]: *intentional programming* [MIC 2002], *feature-oriented development* [TUR 98], *views/view-points* [LEI 91], *generic programming*, g*enerative programming* [CZA 98], entre outros descritos em detalhes neste capítulo.

Esses estilos estendem o modelo orientado a objetos, permitindo o encapsulamento de aspectos, os quais, em programas orientados a objetos convencionais, são usualmente implementados por pedaços de múltiplos objetos.

3.4.1 CF – *Composition Filters*

A abordagem de filtros de composição (***C**omposition **F**ilters*) definida por Aksit e outros [AKS 88, AKS 89, AKS 96, BRG 94, CF 2002] foi motivada pelas dificuldades de expressar qualquer tipo de coordenação de mensagens no modelo de objetos convencional. Isso porque para realizar a sincronização de mensagens nesse modelo seria necessário misturar, nos métodos, código de sincronização com código funcional [CZA 2000].

O modelo CF estende o modelo de objetos convencional através da adição de filtros (*wrappers*), onde os objetos podem estar associados com um número diferente de filtros. Esse filtros determinam quando uma mensagem deve ser aceita/rejeitada, e qual ação deve ser realizada em cada caso [AKS 94]. Um filtro consiste de três elementos:

(i) condição: consiste em um critério que deve ser satisfeito para se poder avaliar o filtro;

(ii) parte de combinação – *matching part*: corresponde a uma mensagem que é avaliada e combinada com um padrão definido;

(iii) parte de substituição - *substituting part*: determina onde e quando as partes da mensagem podem ser substituídas.

Um exemplo de um CF simples usado em um protocolo de sincronização pode ser observado na Figura 3.6. O primeiro elemento do filtro *queue* é um filtro *Wait*. Se a variável *locked* for verdadeira, então somente a mensagem *unlock* poderá prosseguir para o próximo elemento de filtro. Se a variável *unlocked* é verdadeira, então todos os métodos poderão prosseguir.

```
queue:    Wait       = {locked => unlock, unlocked => * };
execute:  Dispatch   = {true => {inner.*, additional.*}};
```

Figura 3.6: CF um Exemplo – Extraído de [AKS 98]

[12] Em [CZA 00] pode-se encontrar uma descrição de outros tipos de POPs.

Na técnica de CF, um objeto é composto por duas partes (Figura 3.7): uma camada de interface e um objeto interno (também denominado *kernel object*). Este último pode ser considerado como um objeto regular definido em uma linguagem de programação orientada a objetos convencional, como Java ou C^{++}. A camada de interface contém um número arbitrário de filtros de mensagens de entrada e saída. As mensagens passam pelos filtros de entrada e saem através dos filtros de saída. Os filtros podem modificar as mensagens, redirecioná-las para outros objetos (internos ou externos) [CZA 2000].

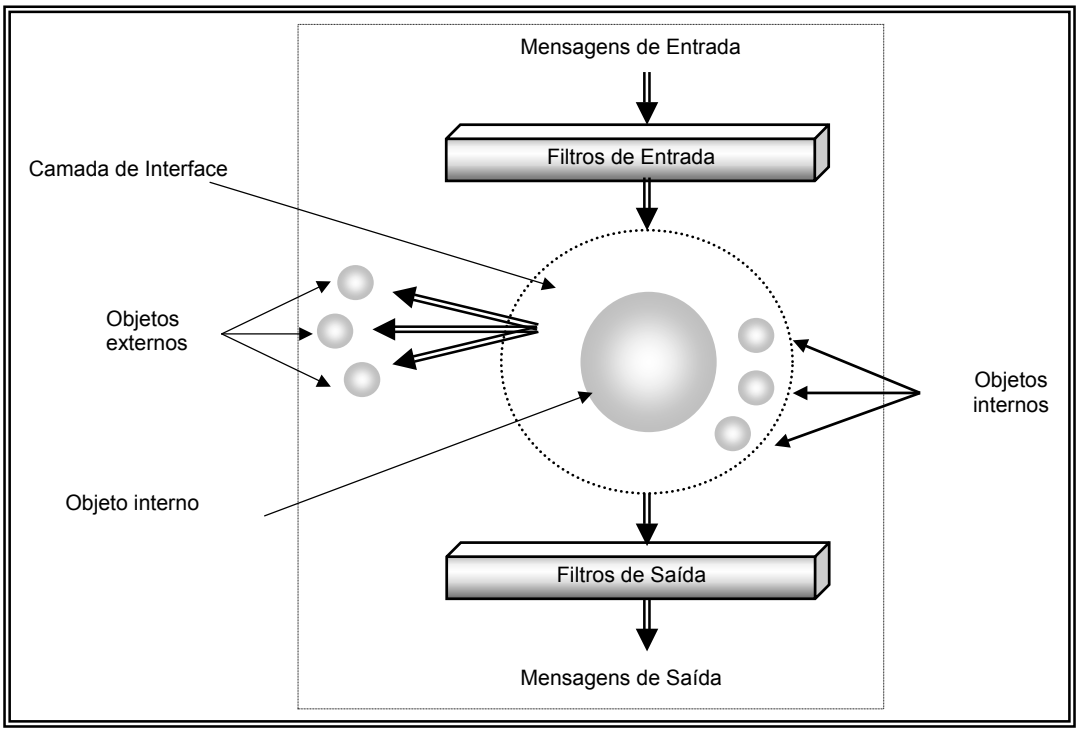

Figura 3.7: Elementos de um Objeto em CF – figura adaptada de [CZA 2000]

A ação que será realizada depende do tipo do filtro. Existem alguns filtros pré-definidos: filtros de delegação para delegar mensagens (*delegation filters*), filtros de espera para buferizar mensagens (*wait filters*), filtros de erros para exceções causadas (*error filters*), e novos tipos de filtros podem se adicionados [PRY 2000].

Essa técnica de decomposição é extensível e adaptável, porque através da adição e/ou modificação de filtros é possível encapsular os entrelaçamentos, preservando assim a modularidade do sistema.

A composição de responsabilidades ocorre em tempo de execução evitando código entrelaçado e permitindo em tempo de execução a avaliação dos parâmetros para tomar decisões. Entretanto, os mecanismos de composição podem apresentar algumas desvantagens quando os filtros não são ortogonais entre si. A ordem dos filtros permite a manipulação de certos conflitos quando existe mais de um entrelaçamentos [PRY 2002].

As principais características[13] do modelo CF compreendem (i) ser declarativo, (ii) possuir semântica de alto-nível, (iii) estar propício a extensões (novos filtros e operadores podem ser definidos), (iv) apresentar forte encapsulamento, (vi) ser modular e (vii) ser componível.

[13] Em [BRG 01] é possível encontrar uma descrição mais detalhada dessas características.

A técnica de filtrar mensagens é muito poderosa, uma vez que permite implementar restrições de sincronização [BRG 94], restrições de tempo real [AKS 94], transações atômicas [AKS 92], verificação de erros [BRG 94], e outros aspectos. De fato, qualquer aspecto que adote interceptação de mensagens enviadas ou capture ações executadas antes ou depois dos métodos pode ser adequadamente representado no modelo CF [CZA 2000]. A capacidade de redirecionamento também pode ser usada para implementar delegação e herança dinâmica.

Existem poucas implementações do modelo CF, por exemplo (i) Glandrup [GLA 95] definiu extensões em C^{++}, (ii) Mordhorst [MOR 95] para Smalltalk e (iii) Wichman [WIC 99] fez uma para Java - ComposeJ (ainda está em estágio de protótipo).

3.4.2 AP – *Adaptive Programming*

Programação adaptativa (*Adaptive Object-Oriented Programming* ou ***A**daptive **P**rogramming – AP*) é um subconjunto da orientação a objetos, que pretende desacoplar os algoritmos das hierarquias das classes. Isso possibilita que um sistema seja decomposto em duas dimensões: classes e algoritmos transversais a essas classes.

O objetivo desse modelo é fornecer a melhor separação de responsabilidades entre o comportamento e a estrutura dos objetos nos programas orientados a objetos [DEM 2002, LIE 92, LIE 96]. A principal motivação para a definição desse modelo é a de que os programas orientados a objetos possuem a tendência de conter um conjunto de métodos pequenos que não realizam nenhuma ou muito poucas ações e chamam outros métodos passando informações [CZA 2000]. Esse tipo de computação costuma degradar a compreensão do sistema e mudanças simples nos algoritmos podem requerer a revisão de um grande número de métodos [PRY 2002].

Para resolver esse problema a AP define a *Lei de Demeter* [LIE 88], a qual determina que um método deve acessar somente variáveis de instância e realizar o envio de mensagens somente para si mesmo. Com essa lei pretende-se evitar o problema de *structure-hardwiring-in-large-methods*, ou seja, um grande número de métodos pequenos que ficam trocando mensagens entre si.

Para se ter uma idéia do código usado por esse modelo foi criado o exemplo esquematizado na Figura 3.8. Esse exemplo especifica um padrão de propagação para calcular o total dos salários dos empregados de uma determinada companhia.

```
*operation * void calculaSalario(int& salarioTotal)
*traverse*
*from* Companhia
*to* Salario
*wrapper* Salario
*prefix*
(@ salarioTotal = salarioTotal + *(this->getValue()); @)
```

Figura 3.8: AP um exemplo – Extraído e Adaptado de [LIE 96]

O processo envolvido para gerar um sistema adaptativo é esquematizado na Figura 3.9. Para desenvolver uma aplicação usando AP e Demeter o programador deve seguir os seguintes passos: (i) escrever as classes que compreendem a solução da aplicação, incluindo seus atributos e operações; (ii) escrever o programa adaptativo (na linguagem *propagation patterns*), que especifica a intenção de comportamento da aplicação; (iii) combinar o *class dictionary graph* com o programa adaptativo, e finalmente (iv) gerar o código da aplicação usando o compilador Demeter.

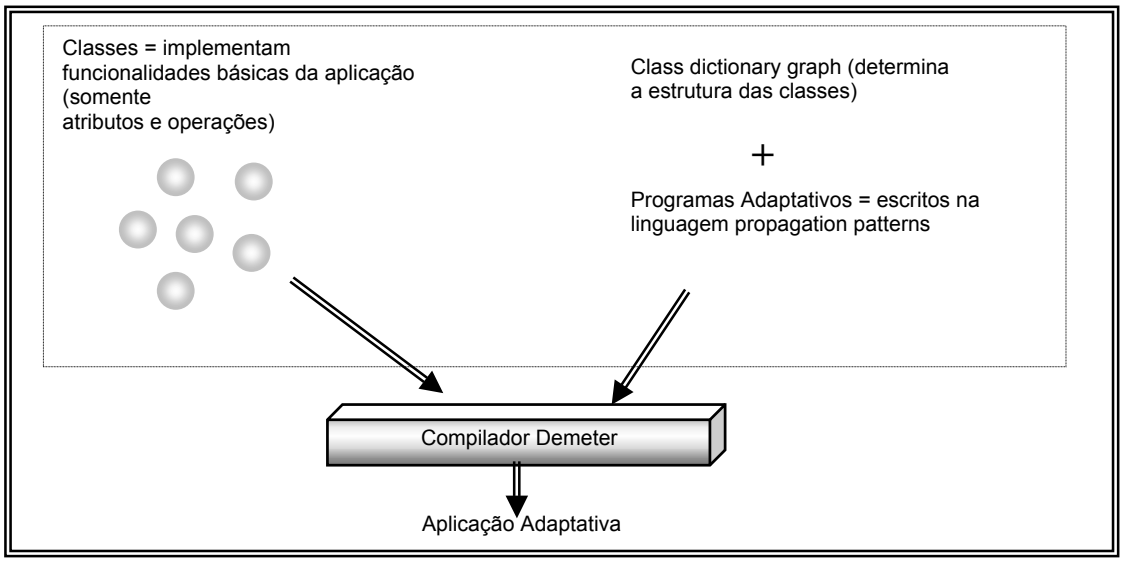

Figura 3.9: Gerando Programas Adaptativos

As idéias de Demeter foram integradas em linguagens orientadas a objetos como C^{++} (Demeter/C^{++}) [LIE 96, SIL 94] e Java (Demeter/Java ou DemeterJ) [LOP 97, ORL 2001]. DemeterJ é uma ferramenta que processa especificações de comportamento e diagramas de classe produzindo programas Java. As vantagens dessa ferramenta são programas mais flexíveis, que se adaptam automaticamente a uma variedade de mudanças estruturais. DemeterJ usa reflexão computacional para interpretar e executar sua navegação.

3.4.3 SOP – *Subject-Oriented Programming*

Programação orientada a assuntos *(**S**ubject-**O**riented **P**rogramming*) foi proposta por Harrison e Ossher [HAR 93, OSS 96, OSS 95, SOP 2002] como uma extensão do paradigma orientado a objetos. Esse modelo é adotado para manipular as diferentes perspectivas subjetivas dos objetos modelados. Isto é, pessoas diferentes em papéis diferentes vêem um objeto sob perspectivas diferentes [CLA 2001].

Nesse modelo, um assunto (*subject*) é uma coleção de classes e/ou fragmentos de classes relacionados por herança e outros relacionamentos suportados por um assunto. Os assuntos são compostos para construir sistemas orientados a objetos [PRY 2000].

SOP determina como um sistema será subdividido em assuntos e especifica como escrever regras de composição[14] para realizar a composição desses assuntos. A composição orientada a assuntos não opera diretamente sobre o programa, mas sim em uma descrição abstrata do programa chamada de "*subject label*" [OSS 96].

[14] Regras de composição (inglês: *composition rules*) = podem ser encontradas em detalhes em [SOP 02, OSS 96, OSS 95].

As regras utilizadas no processo de composição podem ser classificadas como [CLA 99]: (i) regras de correspondência, (ii) regras de combinação e (iii) regras de correspondência e combinação.

Nesse modelo, a composição é realizada em tempo de compilação, o que acarreta a inexistência de composição dinâmica de assuntos. Além disso, requer uma linguagem especifica, e não possibilita a manipulação de RNFs [HÜR 95].

A principal característica do modelo é que ele é independente de linguagem [CZA 2000]. Existe uma versão em C^{++}, onde cada assunto é escrito como uma coleção de classes C^{++} em um espaço de nomes (*namespace*), e qualquer código C^{++} pode ser tratado como parte de um assunto, e existe também um protótipo para Smalltalk [MIL 96].

3.4.4 MDSOC – *Multidimensional Separation of Concerns*

A MDSOC - ***M**ulti**D**imensional **S**eparation of **C**oncerns*, foi introduzida pelo mesmo grupo de pesquisa do SOP [HAR 93], com o intuito de permitir o encapsulamento de diversos tipos de responsabilidades simultaneamente [OSS 2001a, TAR 99]. Para tanto, o software é decomposto e encapsulado em todos os tipos de responsabilidades relevantes (dimensões). Além disso, MDSOC, também possibilita uma poderosa capacidade de composição, possibilitando integrar/compor essas responsabilidades.

Esse modelo propõe a modelagem do sistema utilizando-se de vários conceitos [HYP 2002a]:

- *concern space:* é responsável por organizar as unidade no corpo do software com o objetivo de separar as diversas responsabilidades e estabelecer seus relacionamentos. Além disso, indica como os componentes podem ser construídos e integrados a partir de unidades dessas responsabilidades;
- *unit:* é uma construção sintática (instrução/classe/interface), ou seja, uma unidade de implementação que possa ser descrita através de um formalismo;
- *hyperspaces:* possibilitam estruturar o desenvolvimento de software em diferentes níveis, permitindo uma separação de responsabilidades explícita em cada nível. Eles são compostos por unidades (*units*) e por *hypermodules*, os quais especificam como os componentes podem ser construídos a partir das *units;*
- *hyperslices:* podem encapsular colaborações, padrões de projeto e outras unidades de reutilização, e relacionamentos que possibilitam a adaptação de componentes reutilizáveis [OSS 2001b].

Os aspectos teóricos do modelo MDSOC foram implementados na ferramenta Hyper/J [HYP 2002b]. O objetivo dela não é modificar ou estender a linguagem Java, mas manipular os arquivos *.class*. Analisando esses arquivos é possível realizar extensões, extrações, adaptações e integrações dos componentes Java.

As características que determinam MDSOC com Hyper/J são [OSS 2001b]: (i) habilidade de extrair e encapsular responsabilidades a partir de software preexistente de forma não invasiva; (ii) baixo acoplamento; (iii) tratamento dos responsabilidades de maneira simétrica; (iv) habilidade de integrar e especificar relacionamento entre responsabilidades; (v) a possibilidade de utilizar qualquer software Java, mesmo não possuindo o código fonte. Essas são características que tornam esse modelo propício à evolução, reutilização e integração.

3.4.5 MOPs – Meta-Object Protocols

Essa técnica de construção de sistemas tem sido utilizada nas mais diversas áreas: linguagens de programação, sistemas operacionais, bancos de dados, engenharia de software, computação distribuída, *middleware*, sistemas inteligentes e computação na Web. Isso deve-se principalmente, à implícita separação de responsabilidades e à flexibilidade que o ambiente fornece. Essas vantagens são obtidas devido as características de modificação em tempo de execução [CAZ 2000b].

A reflexão computacional [MAE 87, SMI 82] habilita o programa a acessar sua estrutura e comportamento, assim como manipular sua própria estrutura modificando seu comportamento. Essa técnica oferece uma arquitetura reflexiva, divida em dois níveis: o nível base e o meta nível. A interface entre esses níveis é estabelecida pelos MOPs – ***M****eta-****O****bject* ***P****rotocols* [KIC 91].

Entre as principiais desvantagens no uso de MOPs encontram-se: (i) a complexidade, pois os MOPs são difíceis de entender e propiciam ao programador muito poder, e (ii) a sobrecarga, porque assume-se que a presença de um MOP – ***M****eta-****O****bject* ***P****rotocol*, possui impacto negativo no desempenho do sistema [SUL 2001].

A reflexão computacional pode ser usada de diversas formas no desenvolvimento de software:

- monitoramento: é possível inspecionar as atividades do desenvolvedor e do usuário a respeito das informações sobre os recursos, prazos, etc. Possibilita também, o acompanhamento de versões liberadas para o cliente;
- depuração: durante o processo de desenvolvimento informações sobre o processo podem ser obtidas visando a melhoria do processo;
- adaptabilidade: com a inclusão de MOPs em um sistema pode-se estender as funcionalidades originais da aplicação introduzindo modificações via os meta-objetos;
- separação conceitual: o nível base implementa as funcionalidades relevantes da aplicação, delegando ao meta-nível a implementação de requisitos não funcionais – reduzindo a complexidade de implementação.

Com relação a esse último item, os MOPs oferecem uma clara separação entre as funcionalidades e as características não funcionais. Por outro lado, eles não possibilitam separar aspectos não funcionais uns dos outros [CLA 2001].

3.4.6 AOP – *Aspect Oriented Programming*

Programação orientada a aspectos é a abordagem proposta por Kiczales e Lopes em [KIC 97]. É um estilo de programação que possibilita que tanto código como projeto sejam estruturados de forma a refletir aquilo que os desenvolvedores imaginam sobre o sistema [ELR 2001a]. Essa técnica fornece mecanismos denominados aspectos, os quais estão além das sub-rotinas e da herança para localização dos entrelaçamentos. Os aspectos são usados para simplificar a compreensão dos entrelaçamentos [ELR 2001b].

Assim como todas as outras tecnologias voltadas para a separação de responsabilidades, a AOP foi desenvolvida para tornar o projeto e o código mais modular, significando que as responsabilidades estão de preferência desassociadas e possuem interfaces bem definidas com o resto do sistema [ELR 2001b].

Através dessas interfaces, a AOP oferece ao programador suporte para que ele possa separar claramente componentes e aspectos, uns dos outros (componente de componente, aspecto de aspecto e componente de aspecto), bem como mecanismos que tornam possível abstraí-los e compô-los para produzir o sistema. Isto contrasta com a programação atual que possibilita que o programador separe somente componentes uns dos outros [KIC 97].

Um componente e um aspecto diferem entre si em muitas características. O primeiro é usado quando uma funcionalidade do sistema pode ser claramente encapsulada em um procedimento generalizado (isto é, objeto/método/procedimento/API). Já os aspectos tendem a não ser unidades de decomposição funcional do sistema, mas unidades com propriedades que afetam a performance ou semântica dos componentes de forma sistêmica [HIG 99, KIC 97].

Existem algumas ferramentas para AOP, mas a principal delas é a ferramenta AspectJ [KIC 2001], que compreende uma extensão dos conceitos de orientação a aspectos da linguagem Java. A integração de AOP em AspectJ é realizada em tempo de compilação. O arquivo fonte *.java* e os arquivos executáveis (.class) compreendem a entrada para o compilador de aspectos, também denominado *compiler aspect weaving*. O código produzido por essa compilação é um código *.java* que contém o código relacionado com os aspectos e o código relacionado com as classes. Esse código deve ser compilado com um compilador Java padrão [CLA 2001].

Dentre todas as técnicas de decomposição a que está crescendo mais rapidamente é a AOP, pois essa técnica possibilita que tanto código quanto projeto sejam estruturados de forma a refletir aquilo que os desenvolvedores imaginam sobre o sistema [ELR 2001a]. Além disso, é possível vê-la sendo usada em aplicações das mais diversas áreas como *middleware*, segurança, tolerância a falhas, QoS (***Q****uality* ***o****f* ***S****ervice*) e sistemas operacionais. Também é possível encontrar pesquisas onde a AOP é aplicada nos mais diversos estágios do ciclo de vida do software [ELR 2001a, ELR 2001b, KIC 2001, OSS 2001b].

A Tabela 3.1 apresenta uma comparação entre as abordagens mais utilizadas para a decomposição de sistemas computacionais. Para construí-la foram utilizadas como base as seguintes referências [CLA 2001, ELR 2001b, PRY 2002].

Tabela 3.1: Comparação entre as Técnicas de Decomposição

	CF[15]	AP[16]	SOP[17]	MDSOC[18]	MOP[19]	AOP[20]
Unidade Básica de decomposição	Filtros	Classe e algoritmos	Sujeitos	Objetos e aspectos	Nível base e meta	Objetos e aspectos
Dimensões	n dimensões	2 dimensões	1 dimensão	n dimensões	2 dimensões	2 dimensões
Tipo de composição	Dinâmica	Estática	Estática	Estática	Dinâmica (e estática)	Estática
Join-points	Mensagens enviadas e recebidas pelos objetos	Definidos por uma especificação sucinta denominada estratégia transversal	Não encontrada	Classes, interfaces, métodos e atributos	Reificação, introspecção e interceptação	Chamadas de métodos, acesso a atributos e construção de objetos
Separação clara entre RNFs e funcionais	Sim	Não	Sim	Não	Sim	Sim
Paradigma Orientado a Objetos	Extensão	Subconjunto	Extensão	Extensão	Extensão	Extensão
Manipulação de Conflitos	Ordenação dos filtros	Não encontrada	Com regras de composição	Com regras de composição	Não encontrada	Precedência de aspectos
Mecanismo de coordenação	Qualquer	Grafo resumido de especificações	Identidade do objeto e regras de composição	Não encontrada	Qualquer	Qualquer
Exemplos	C++ [GLA 95] Smalltalk [MOR 95] ComposeJ [WIC 99]	Demeter/C++ [LIE 96] DemeterJ [ORL 2001]	VisualAge for C++ e Visual Age for Smalltalk	HyperJ [HYP 2002b, OSS 2001b]	DJ Aspects [DEM 2002, ORL 2001]	AspectJ [KIC 97, KIC 2001]

[15] CF = do inglês *Composition Filter*
[16] AP = do inglês *Adaptive Programming*
[17] SOP = do inglês *Subject-Oriented Programming*
[18] MDSOC = do inglês *Muli-Dimensional Separtion of Concerns*
[19] MOP = do inglês *Meta-Object Protocol*
[20] AOP = do inglês *Aspect-Oriented Programming*

4 TERMINOLOGIA PARA AOSD

Para entender a AO é necessário conhecer vários termos fundamentais que fazem parte da maioria das definições e implementações. Para melhor compreender os conceitos envolvidos com essa tecnologia este capítulo inicialmente introduz a terminologia referente ao desenvolvimento de software orientado a aspectos. Após é apresentada uma visão geral de um estudo de caso, bem como suas soluções orientadas a objetos e a aspectos.

4.1 Introdução

O desenvolvimento de software orientado a aspectos surgiu com o trabalho de Kiczales [KIC 97], onde o foco residia basicamente na fase de programação usando aspectos. Mens, em [MEN 97], já previa a necessidade dessa evolução, onde os conceitos de orientação a aspectos estariam presentes nas diversas fases do processo de desenvolvimento. Desse modo, similar ao desenvolvimento orientado a objetos, o desenvolvimento baseado em aspectos seguiu um ciclo natural de progressão, ou seja, partindo da programação para os outros estágios de desenvolvimento.

Da mesma forma que as demais técnicas de decomposição (ver seção 3.4), a AOP foi desenvolvida para tornar o projeto e o código mais modular. Isso estabelece que as responsabilidades estão de preferência desassociadas umas das outras [ELR 2001b].

A orientação a aspectos ajuda a superar problemas relacionados com a decomposição ineficiente, tais como dispersão, entrelaçamento, falta de transparência, entre outros [AOS 2002]. A conseqüência prática de utilizar AOP é um maior nível de reutilização e flexibilidade, pois é necessário escrever menos código. Além disso, todo código que estaria distribuído fica localizado em um único ponto, ou seja, minimizando as redundâncias [ELR 2001a]. Isso resulta em implementações altamente modularizadas, mais fáceis de compreender e manter [LAD 2002a].

4.2 Terminologia Elaborada

Na literatura, existem vários esforços [ALD 2001, ALD 2003, SAP 2002, ZAK 2002] para o estabelecimento de uma terminologia voltada ao desenvolvimento de software orientado a aspectos. Esta seção concentra-se em apresentar os principais conceitos encontrados na orientação a aspectos.

4.2.1 Responsabilidade (*Concern*)

A palavra inglesa *concern* possui vários significados, denotando tanto preocupação com algo como responsabilidade sobre algo. Este último termo é adotado neste trabalho, uma vez que no modelo de objetos, cada classe é responsável por um ou mais serviços a serem oferecidos a seus clientes; e a tais serviços podem ser adicionadas mais responsabilidades, como por exemplo, a responsabilidade de fornecer um serviço confiável.

Responsabilidade, em se tratando de Engenharia de Software, pode ser, basicamente, qualquer elemento (como por exemplo, um protocolo, uma característica, um requisito, uma classe, etc.) que possa ser adotado na solução do sistema, tanto a nível conceitual quanto prático (construção) [LAD 2002a].

A IEEE - ***I**nstitute of **E**lectrical and **E**lectronics **E**ngineers* [IEE 2000] - define responsabilidade de um sistema como "... o interesse relacionado ao desenvolvimento do sistema, suas operações e quaisquer outros aspectos que são críticos ou importantes para um ou mais envolvidos (*stakeholders*)". As responsabilidades surgem em todos os estágios do ciclo de vida, desde a análise e especificação de requisitos, passando pelo projeto e implementação até a manutenção e evolução [HIL 99].

Para poder compreender os efeitos que uma responsabilidade possui em um sistema, seria necessário olhar o interior de todas as classes que, de alguma forma, utilizam essa responsabilidade [HIG 99].

Se for possível descrever uma funcionalidade de um sistema usando uma linguagem natural, pode-se assegurar que uma responsabilidade foi identificada, pelo menos a nível conceitual [HÜR 95]. Quando é desejável identificar responsabilidades nos projetos ou aplicações, deve-se localizar unidades funcionais[21] que estão relacionadas com algum tipo de funcionalidade.

Geralmente existe uma certa confusão com relação aos termos artefatos e responsabilidades. Embora os artefatos possam ser considerados responsabilidades, elas em geral não constituem-se em artefatos. Na verdade, as responsabilidades são essencialmente conceituais, enquanto os artefatos contribuem para o desenvolvimento de sistemas de software, pois são instruções concretas. Os artefatos podem ser usados para identificar, representar, definir, descrever, modelar, implementar, afetar, refletir ou incorporar as responsabilidades [CON 2000, CLA 2001].

4.2.2 Entrelaçamentos (*Crosscutting Concerns*)

Entrelaçamentos ou propriedades ortogonais (ver Anexo 1 para outras traduções) compreendem elementos que não são facilmente encapsulados em uma estrutura modular.

Exemplos de entrelaçamentos não são difíceis de encontrar em sistemas computacionais, a exemplo de: desempenho, sincronização, comunicação, tratamento de interfaces gráficas, depuração, persistência, autorização, etc. Esses elementos podem associar-se de alguma forma com diferentes componentes do sistema e assim facilmente introduzem código altamente relacionado ou até mesmo duplicado em um ou mais módulos [ELR 2001a, KIC 97]. Segundo Laddad [LAD 2002a] os entrelaçamentos podem ser classificados como código entrelaçado e código disperso (ver seção 3.3).

---

[21] O termo unidade funcional é usada para descrever os elementos que compõe o programa, por exemplo classes, interfaces, características (*features*), assuntos (*subjects*), aspectos.

4.2.3 Aspecto (*Aspect*)

Outra dificuldade encontrada no desenvolvimento AO é distinguir o que é uma simples responsabilidade e o que é um aspecto. As diferenças concentram-se em dois pontos básicos: as responsabilidades representam funcionalidades – o que o sistema deve fazer, enquanto que os aspectos são responsabilidades adicionais do sistema, independentes do domínio da aplicação, ou seja, aspectos concentram-se em características não funcionais. Adicionalmente, cada funcionalidade é geralmente associada a um componente, enquanto que aspectos podem afetar vários componentes, ou seja, a forma como eles são projetados e implementados é diferente [LAD 2002a].

O objetivo da AOD é organizar os programas decompondo-os em aspectos e classes. Um aspecto é uma unidade de decomposição que implementa uma propriedade que é ortogonal a uma determinada aplicação. Na verdade, ele pode ser considerado como a representação de um entrelaçamento [HIG 99]. Kiczales, em [KIC 97], define que um aspecto é algo que permeia vários grupos de componentes funcionais ou que mescla-se a funcionalidade básica do sistema.

Assim, sincronização, restrições de tempo real e verificação de erros são exemplos de aspectos. Existem muitos outros exemplos de aspectos, tais como comunicação entre objetos, gerenciamento de memória, persistência, segurança, políticas de *caching*, perfil, monitoramento, chamada remota, balanceamento de carga, replicação, tolerância a falhas, QoS, transações distribuídas [HIG 99, KIC 2001, LAD 2002a, LAD 2003].

Em resumo, aspectos tornam possível criar módulos coesos que implementam responsabilidades adicionais. O código das classes- responsáveis pelas funcionalidades do domínio da aplicação - e dos aspectos é integrado através de um mecanismo denominado combinação de aspectos. Esse mecanismo insere o código de um aspecto em locais bem definidos, denominados pontos de combinação (*join-points*), na estrutura de uma unidade funcional. Esse mecanismo será apresentado em detalhes na próxima seção.

4.2.4 Combinador de Aspectos (*Aspect Weaver*)

A capacidade de combinação é um elemento importante em AO. O processo de combinação de aspectos, geralmente conduzido com o auxilio de uma linguagem orientada a aspectos e uma ferramenta associada, ocorre em três fases distintas [LAD 2002a] esquematizadas na Figura 4.1.

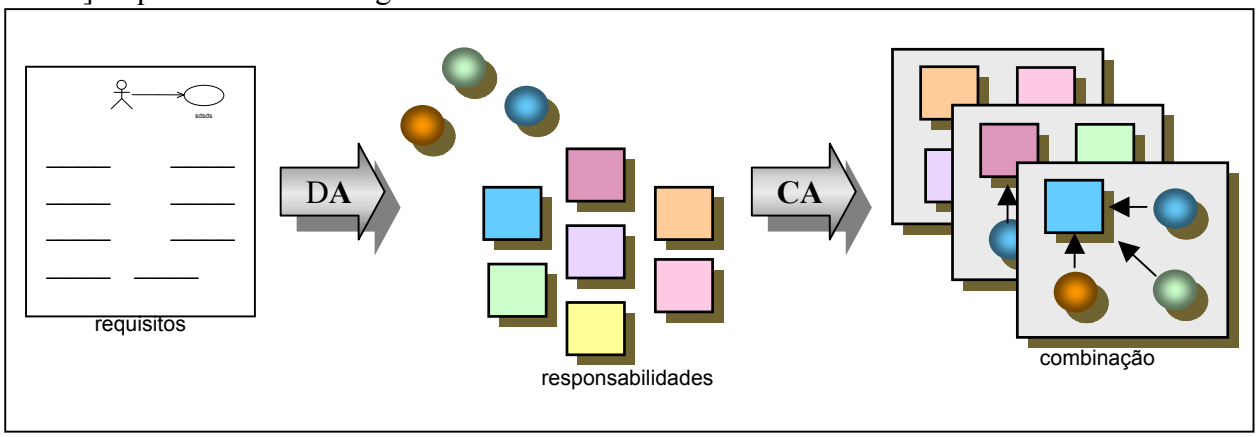

Figura 4.1: Combinação de Aspectos

Essas três fases podem ser resumidas e descritas do seguinte modo:

1. fase 1: decomposição de aspectos (DA) - (*aspectual decomposition*): decompor os requisitos para identificar entrelaçamento;
2. fase 2: implementação de responsabilidades (*concern implementation*): implementar cada responsabilidade de forma independente e isolada*;*
3. fase 3: combinação de aspectos (CA) - (*aspectual recomposition*): nessa etapa, um integrador de aspectos especifica regras de combinação para criar as unidades de modularização. Esse processo, também conhecido como combinação ou integração*,* usa informações para compor o sistema final.

O processo de combinação definido pelo combinador de aspectos pode ocorrer em dois momentos [CON 2001, KIC 97]:

(i) tempo de compilação ou combinação estática: é mais eficiente, pois não existe nenhum mecanismo dinâmico que possa sobrecarregar o sistema. Porém, nesse tipo de combinação características como adaptabilidade e customização são muito difíceis de serem alcançadas;

(ii) tempo de execução ou combinação dinâmica: esse tipo de combinação costuma gerar mais sobrecarga. Geralmente utiliza uma arquitetura reflexiva para que, em tempo de execução, seja possível adicionar, adaptar e remover aspectos.

Com o aumento da abstração e dos níveis de entrelaçamento das linguagens de programação é possível eliminar grande parte da codificação manual realizada pelos programadores com a ajuda de um combinador de aspectos, assim permitindo que o programador se concentre nas etapas mais criativas do desenvolvimento de software [HIG 99].

Esse mecanismo de combinar os aspectos aos componentes somente poderá ser efetivo se os pontos de combinação estiverem corretamente definidos.

4.2.5 Pontos de Combinação (*join-points*)

O objetivo do combinador de aspectos é juntar os aspectos e os componentes a fim de produzir um sistema coerente. Para que isso ocorra utiliza-se o conceito de pontos de combinação ou *join-points*, os quais se constituem nos elementos principais das linguagens de programação orientadas a aspectos [LAD 2002b], uma vez que eles determinam o ponto onde o combinador insere o código relacionado com os aspectos [OSS 98].

Segundo Bardou, em [BAR 98], o entrelaçamento - entre componentes e aspectos é uma característica chave da AOP. Ele é representado na Figura 4.2, onde *n* componentes estão distribuídos em barras verticais e *m* aspectos estão distribuídos em barras horizontais. Os pontos de combinação, representados por círculos, mostram os pontos em que os aspectos interagem com os componentes.

O mesmo aspecto pode interagir com todos os componentes (por exemplo A_m) ou somente com alguns deles (por exemplo A_1). Ou ainda, o mesmo componente pode interagir com todos os aspectos (por exemplo C_2) ou somente com alguns aspectos (por exemplo C_3).

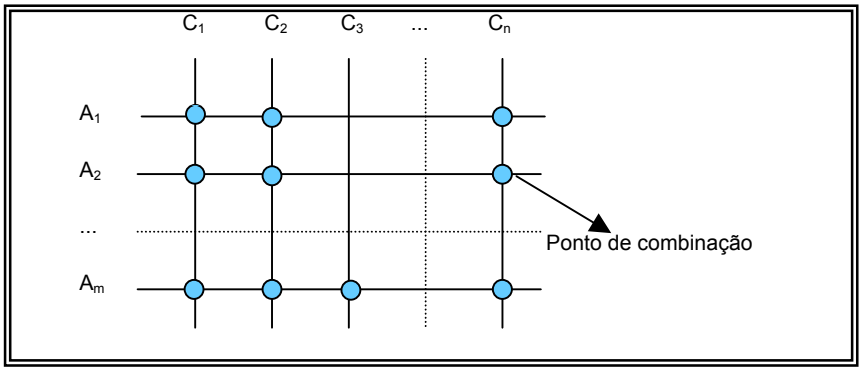

Figura 4.2: Exemplo de Pontos de Combinação

Um ponto de combinação é qualquer ponto de execução identificável em um sistema. Exemplos de pontos de combinação compreendem: (i) invocação e execução de um método, (ii) atribuição de um valor para uma variável, (iii) uma instrução *return*, (iv) a construção de um objeto, (v) a verificação de uma condição, (vi) o tratador de exceção, (vii) laços *for/while/do-while*, entre outros [LAD 2003].

Em Ossher [OSS 98] encontra-se que os pontos de combinação podem estar localizados a nível de instrução ou a nível de operação. No primeiro caso, implica que o conjunto dos possíveis pontos de combinação inclui qualquer instrução (linha de código) no sistema. Já o segundo especifica que o conjunto de pontos de combinação inclui qualquer operação (invocação de métodos) que o sistema realiza. As atuais implementações de AOP usam pontos de combinação a nível de operação [HIG 99].

Os pontos de combinação podem ser gerados em tempo de execução usando um ambiente reflexivo para a programação dos componentes. Nesse caso, a linguagem de aspectos é implementada através de meta-objetos [SUL 2001], os quais utilizam as informações dos pontos de combinação e dos programas de aspectos para combinação [KIC 97].

Os pontos de combinação somente serão introduzidos no código orientado a aspectos por meio de pontos de corte, a seguir descritos.

4.2.6 Ponto de Corte (*Pointcut*)

Ponto de corte ou *pointcut* é uma construção de linguagem que une um conjunto de pontos de combinação baseando-se em um critério pré-definido. Os pontos de corte capturam ou identificam pontos de combinação no fluxo de um programa. Após determinar os pontos de combinação é possível especificar as regras de combinação envolvendo os pontos de combinação.

Os pontos de corte definem um conjunto de pontos na execução de um programa, mas eles usam somente um número finito de tipos de pontos. Esses tipos de pontos compreendem basicamente: invocação e execução de método, tratamento de exceções, instanciação e execução de construtores [ASJ 2002].

A maioria das implementações atuais de linguagens orientadas a aspectos especifica ainda que um ponto de corte pode ser executado em alguns momentos chave, definidos por *advices* (ver a próxima seção).

4.2.7 Advice

Um *advice* determina o momento em que um ponto de corte será ativado. A sua definição, ou seja o código nele contido, é similar ao corpo de um método.

Um advice pode pertencer a uma das seguintes categorias [ASJ 2002, KIC 97, POP 2001]: antes (*before*), depois (*after*) e simultaneamente (*around*) a execução de um ponto de combinação. O *advice before* determina que será imediatamente antes da execução do corpo do método/construtor, o *after* especifica que será executado após o corpo do método/construtor executar, e o *around* que executa antes e após o término da execução do método/construtor [LAD 2002a, SAP 2002].

A seguir é apresentado um estudo de caso bastante simplificado, que objetiva aumentar a compreensão destes conceitos e ilustrar as suas diferenças em relação ao modelo de objetos através de duas soluções: uma solução OO e uma solução AO.

4.3 Estudo de Caso: OO x AO

4.3.1 Descrição do Estudo de Caso

Com a intenção de complementar os conceitos apresentados nesse capítulo criou-se o estudo de caso baseado em uma aplicação bancária simples, o qual será construído com um *framework* de classes.

Esse *framework* será composto das classes fundamentais para a resolução do problema. As principais considerações realizadas sobre a aplicação compreendem:

- usuários: podem ser de dois tipos – funcionário do banco e cliente;
- ações permitidas para o usuário cliente: creditar e debitar valores em sua conta, além de consultar o seu saldo;
- o usuário cliente é caracterizado por conter um único tipo de conta e por possuir uma senha de acesso aos terminais;
- ações permitidas para o usuário funcionário: verificar histórico do cliente e realizar auditoria (consultar últimas operações realizadas por um cliente).

No banco é permitido que todo e qualquer cliente possua um dos seguintes tipos de conta: conta corrente ou conta especial. Toda conta é caracterizada por um saldo, mas uma conta especial é descrita também por um limite. Além disso, toda conta deve pertencer a uma agência (caracterizada por número e nome).

Com o intuito de manter o histórico do cliente as operações de crédito e débito devem ser auditadas. Essa auditoria deve ser realizada em dois momentos: antes da execução da ação e após a execução da ação.

4.4 Solução Orientada a Objetos

Para criar uma solução orientada a objetos é necessário construir o código orientado a objetos, ou seja as classes. Para identificar as classes, que contemplam as regras de negócio definidas para o problema, é necessário realizar uma especificação detalhada dos requisitos.

A partir dessa especificação é possível derivar o diagrama de classes (usando técnicas como extração de substantivos, adjetivos e verbos). Após, o diagrama de classes é traduzido para código em uma linguagem de programação orientada a objetos, no caso deste trabalho a linguagem de programação Java.

O conjunto de classes resultante a partir da análise do problema pode variar de acordo com a experiência e o ponto de vista do analista que está modelando a aplicação. Assim, a solução apresentada neste capítulo pode variar se ela for construída por outros analistas, mas a essência deve ser a mesma.

O ponto de partida para definir a solução do problema é a análise e especificação dos requisitos. Para atingir esse objetivo foi construído o diagrama de casos de uso, conforme ilustra a Figura 4.3.

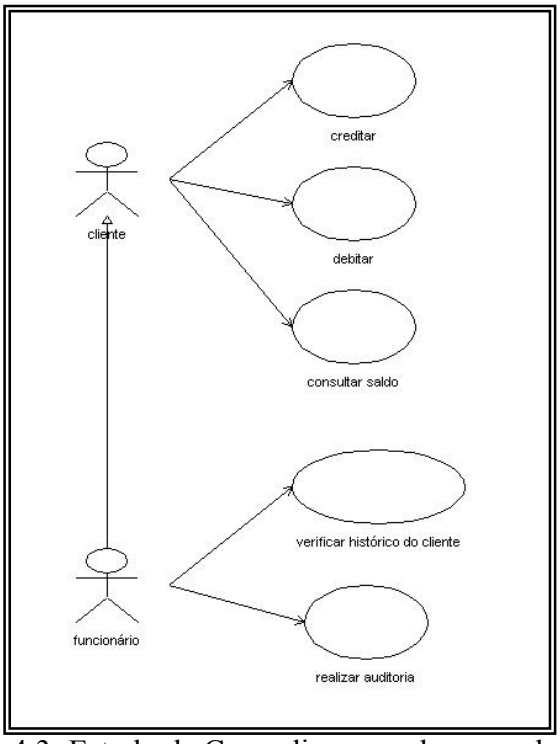

Figura 4.3: Estudo de Caso: diagrama de casos de uso

Para cada caso de uso foram associadas descrições textuais (formato conforme exposto pela Figura 4.4), cujo objetivo é descrever em detalhes os casos de uso (para posteriormente extrair as classes).

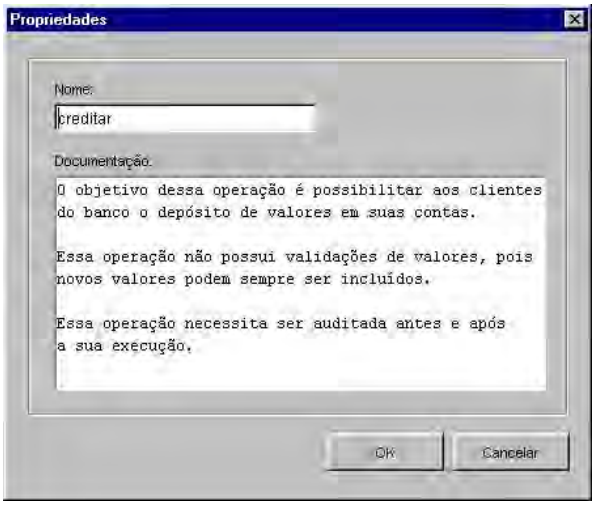

Figura 4.4: Estudo de Caso: descrições textuais (um exemplo)

Após realizar a especificação dos requisitos o próximo passo consiste no projeto das classes através do diagrama de classes. A Figura 4.5 apresenta as classes que compõe a solução do sistema bancário descrito previamente.

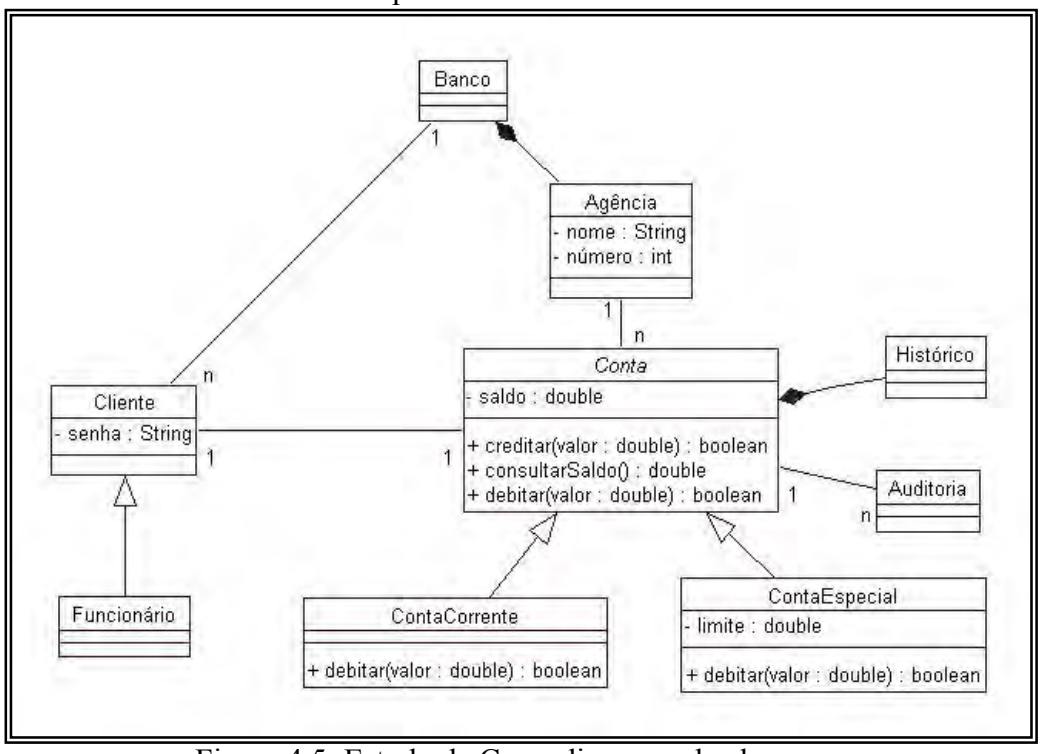

Figura 4.5: Estudo de Caso: diagrama de classes

Apenas as classes/interfaces consideradas essenciais para exemplificar o uso de aspectos durante o desenvolvimento de um software serão apresentadas nesse capítulo.

A classe Conta, apresentada no Código 1, define uma estrutura e um comportamento de reutilização comum a todos os tipos de conta existentes, logo ela foi definida como abstrata (serve apenas como um modelo para as subclasses - *ContaCorrente* e *ContaEspecial*).

(1) CÓDIGO

```
1.  public abstract class Conta{
2.      private double saldo;
3.
4.      public Conta(double saldo){
5.           this.saldo = saldo;
6.      }
7.
8.      public boolean creditar(double valor){
9.            Auditoria.auditar("creditar","antes", saldo);
10.           saldo+=valor;
11.           Auditoria.auditar("creditar","após", saldo);
12.     }
13.     public double consultarSaldo(){return saldo;}
14.     public void setSaldo(double nSaldo){
15.          saldo = nSaldo;
16.     }
17.     public double getSaldo(){
18.         return saldo;
19.     }
20.     public abstract boolean debitar(double valor);
21.   }}
22. class ContaCorrente extends Conta{
23.     public ContaCorrente(double saldo){
```

```
24.             super(saldo);
25.     }
26.
27.     public boolean debitar(double valor) {
28.          if(getSaldo() >= valor){
29.             Auditoria.auditar("debita","antes", saldo);
30.             setSaldo(getSaldo()-valor);
31.             Auditoria.auditar("debita","após", saldo);
32              return true;
33.          }
34.          return false;
35.     }
36. }
37. class ContaEspecial extends Conta{
38.     private double limite;
39.
40.     public ContaEspecial(double s, double l){
41.              super(s);
42.              this.limite = l;
43.     }
44.     public void setLimite(double nLimite){
45.            limite = nLimite;
46.     }
47.     public double getLimite(){
48.           return limite;
49.     }
50.     public boolean debitar(double valor){
51.          if(getSaldo()+limite >= valor){
52.             Auditoria.auditar("debitar","antes", saldo);
53.             Auditoria.auditar("debitar","antes", limite);
54.             setSaldo(getSaldo()-valor);
55.             Auditoria.auditar("debitar","após", saldo);
56.             Auditoria.auditar("debitar","após", limite);
57.             return true;
58.          }
59.          return false;
60.     }
61. }
```

Conforme pode-se perceber no Código 1, o código relacionado com a auditoria (requisito não funcional), linhas 9 e 11 ou linhas 52-53 e 55-56, encontra-se altamente vinculado com o código funcional.

O problema dessa abordagem é que essa adição de código nas classes ocorre de forma invasiva, além de que o código responsável pela auditoria estará replicado por todo o sistema.

Uma outra forma de introduzir a auditoria seria o uso de herança, onde cada classe que necessite desse serviço herde da classe *Auditoria*. O problema com esse tipo de solução é que cada classe deverá ser responsável pelas suas auditorias. Além desse problema, essa construção não seria permitida na linguagem Java, pois ela permite apenas uso de herança simples. Nesse caso, como as classes *ContaCorrente* e *ContaEspecial* já herdam da classe *Conta* não seria possível realizar a herança da classe *Auditoria*.

Percebe-se que embora a orientação a objetos ofereça diversos mecanismos para modularidade (herança, composição, agregação, polimorfismo, etc.) eles não são suficientes para a construção de módulos independentes. Assim, com o desenvolvimento baseado em aspectos pretende-se oferecer um grau de modularidade maior, bem como um menor entrelaçamento.

4.5 Solução Orientada a Aspectos

Para permitir o uso de auditoria de modo não invasivo, com alto nível de modularidade, e com pouco entrelaçamento criou-se o aspecto *AuditoriaNF* (Código 2).

Esse aspecto é responsável por determinar os pontos em que os serviços de auditoria deverão ser introduzidos. A classe de *Auditoria* é responsável pelo serviço de auditoria propriamente dito, enquanto que o aspecto determina onde esse serviço deve ser introduzido com relação ao código funcional.

(2)	CÓDIGO
1.	`Aspect AuditoriaNF{`
2.	`    pointcut audita(): call(boolean Conta.credita(double)) ||`
3.	`                       call(boolean ContaCorrente.debita(double)) ||`
4.	`                       call(boolean ContaEspecial.debita(double));`
5.	
6.	`   before() returning: auditar(){`
7.	`         Auditoria.auditar();`
8.	`    }`
9.	
10.	`   after() returning: auditar(){`
11.	`          Auditoria.auditar();`
12.	`    }`
13.	`}`

O aspecto *AuditoriaNF* (definido no Código 2, linha 1 a 13, do) determina que a auditoria deverá ser ativada tanto antes quanto após a invocação dos métodos *credita()* e *debita()*. a auditoria deverá ser ativada. Observa-se que o ponto de corte (linhas 2 a 4) define que o aspecto será ativado somente se o método *credita()* estiver definido na superclasse *Conta*, e o *debita()* em suas subclasses (*ContaCorrente* e *ContaEspecial*).

A Figura 4.6 esquematiza o processo de combinação desse aspecto com a classe `ContaCorrente`, conforme determina a classe de `Teste`.

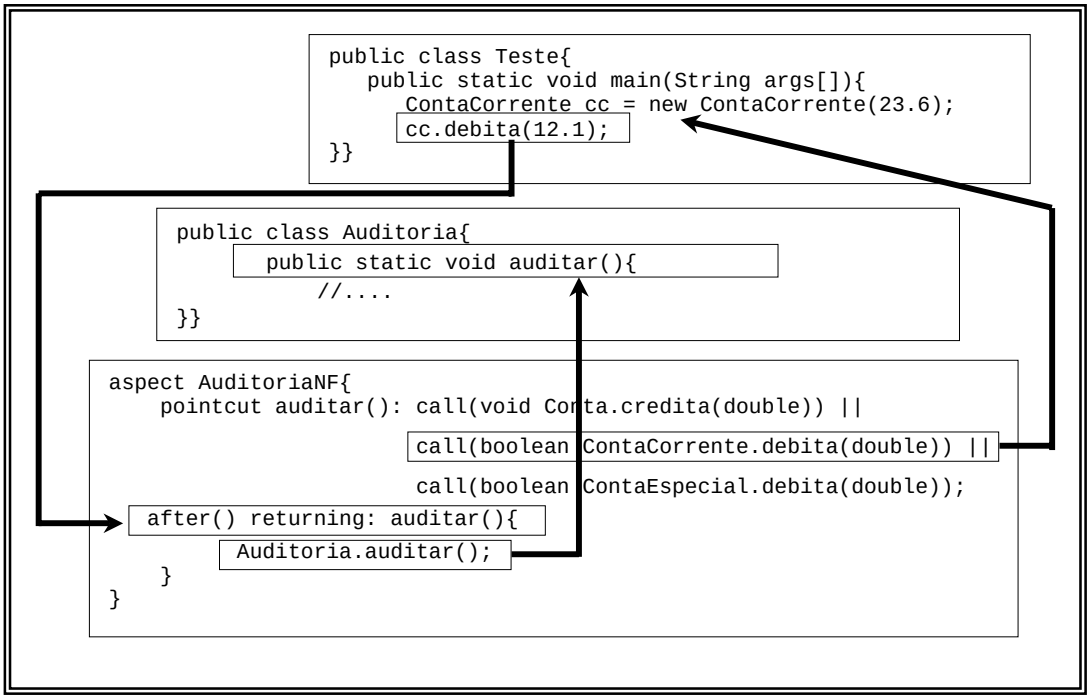

Figura 4.6: Combinando Aspectos e Componentes

A sintaxe da linguagem (AspectJ) apresentada nos exemplos descritos nesse capítulo, bem como exemplos e explicações detalhadas sobre essa linguagem podem ser encontrados em detalhes no Anexo 2.

4.6 Conclusões

Uma vez definidas as classes, responsáveis pela funcionalidade, o código fica mais fácil de entender, pois são eliminadas as porções de código não funcional, agora localizadas no aspecto.

Além disso, a implementação, a manutenção e os testes são facilitados, porque podem ser realizados em pontos específicos. Caso a aplicação sofra uma redefinição, por exemplo a auditoria deve ser aplicada somente antes da execução do método, no modelo orientado a objetos, é necessário percorrer todas as classes e alterar o código. Isso pode gerar algumas inconsistências nas classes, por exemplo a eliminação do código não é completa. Por outro lado, com o uso de aspectos, basta alterar o aspecto para ter certeza de que a alteração será refletida nas diversas classes com ele relacionadas.

5 TRABALHOS RELACIONADOS

Neste capítulo são descritos alguns trabalhos que influenciaram as soluções adotadas por FRIDA. O capítulo encontra-se organizado de modo que sejam apresentadas descrições resumidas dos trabalhos usados como fundamentação para cada fase do processo de desenvolvimento, assim como a indicação das principais idéias/artefatos que influenciaram o método. Apresenta também tabelas comparativas entre os trabalhos indicando as contribuições de FRIDA.

5.1 Introdução

Existem muitas propostas de técnicas e métodos para tratamento de RNFs, porém poucas abrangem todo o ciclo de desenvolvimento de software. Geralmente uma ou algumas poucas fases do ciclo merecem atenção no que diz respeito ao tratamento integrado de RNFs, e, mais raramente, existe vinculação de tais técnicas e métodos com aquelas adotadas nas demais fases. Um fato que costuma ocorrer, freqüentemente, é a elicitação dos RNFs utilizar algum método durante a fase de especificação de requisitos totalmente desvinculado das técnicas adotadas nas demais fases.

É possível encontrar diversos trabalhos relacionados com RNFs, porém ainda são poucos os trabalhos que (i) preocupam-se com a relação dos RNFs e o seu grau de conformidade com especificação; (ii) concentram-se em requisitos que permeiam todo o ciclo de desenvolvimento do software; e (iii) abordam o uso de aspectos como solução desde as fases iniciais do desenvolvimento.

Neste capítulo são resumidos os principais trabalhos que influenciaram as soluções adotadas no método, comparando-os com as contribuições de FRIDA, no sentido de conduzir o tratamento de RNFs com técnicas particularizadas em cada uma das fases aliadas à preocupação com a integração ao longo do processo de desenvolvimento.

Todas as comparações são realizadas através de tabelas. Onde cada coluna representa um trabalho analisado e as linhas expressam as idéias que cada trabalho aborda. Em alguns casos as idéias de FRIDA são aplicadas da mesma maneira que nos trabalhos relacionados, mas na maioria dos casos as idéias constituem apenas a fonte de inspiração para algum passo ou artefato do método FRIDA.

5.2 Estratégias para Elicitação/Modelagem de Requisitos

Nesta seção, são apresentadas abordagens que incluem: (i) o uso de *templates* textuais [ARA 2003, BRI 2002, COL 98, RAS 2003, REE 2002] para a descrição dos requisitos; (ii) diagramas e extensões da UML [BRI 2002, CYS 2001, RAS 2003, REE 2002]; (iii) construção e definição de léxico [CYS 2001, LEI 95]; e (iv) aspectos [ARA 2003, BRI 2002, GRU 2001a, RAS 2003]. Além dessas descrições é apresentada uma tabela (Tabela 5.1), cujo objetivo é esquematizar as contribuições de FRIDA com relação aos demais trabalhos, ou seja, apresentar os mecanismos/artefatos que FRIDA adota em comparação com cada um dos trabalhos descritos.

Trabalho 1 – Coleman [COL 98]

Descrição: O primeiro trabalho analisado foi o de Coleman [COL 98], que é baseado no modelo de casos de uso. Este trabalho propõe um *template* que busca integrar a especificação dos RFs e RNFs. O seu objetivo principal é estruturar e descrever os casos de uso pertencentes a um diagrama de casos de uso.

Idéia(s)/Artefato(s) adotado(s) por FRIDA: (i) diagrama de casos de uso e (ii) *template* para descrição de RFs**.**

Trabalho 2 – Reed [REE 2002]

Descrição: Reed, em [REE 2002], apresenta um processo de transformação gradativa de diagramas UML em código Java. Neste processo, o diagrama de casos de uso é usado como ponto de partida, sendo incrementalmente descrito através de um *template* classificado em seções. Cada seção possui um foco e é responsável por descrever atividades bem específicas. O objetivo principal deste trabalho é a análise funcional, ou seja, os RNFs não constituem a parte fundamental. Além disso, neste trabalho não fica claro como os RNFs serão traduzidos nas fases posteriores, bem como a sua manipulação.

Idéia(s)/Artefato(s) adotado(s) por FRIDA: (i) diagrama de casos de uso, (ii) *template* para descrição de RFs, e (iii) definição das fases de análise e especificação de requisitos, e também das fases de projeto e construção**.**

Trabalho 3 – Chung [CHU 2000]

Descrição: Mylopoulos [MYL 92] e Chung [CHU 95, CHU 99, CHU 2000] propõem, através do uso de um framework, denominado NFR Framework – ***N**on-**F**unctional **R**equirements*, a elicitação e modelagem de RNFs. Neste trabalho os RNFs são representados explicitamente como metas que devem ser satisfeitas. Um RNF geral, ou amplo, sofre sucessivos refinamentos até se obter um conjunto de requisitos específicos representados por meio de um grafo de RNFs. A proposta desse modelo é tratar todos os tipos de RNFs desde as primeiras etapas do processo de desenvolvimento de software.

Idéia(s)/Artefato(s) adotado(s) por FRIDA: (i) refinamento dos RNFs e (ii) o tratamento explícito dos RNFs desde as primeiras fases do processo de desenvolvimento**.**

Trabalho 4 – Leite [LEI 93, LEI 95]

Descrição: para Leite [LEI 93, LEI 95] é possível utilizar um LAL (***L**éxico **A**mpliado da **L**inguagem*) para registrar a linguagem utilizada pelos atores do UdI (***U**niverso de*

***I**nformações*) sem que seja necessário preocupar-se com a funcionalidade. Nesse léxico, através de regras de construção bem definidas, é definido um vocabulário (palavras e frases) pertinente à aplicação.

Idéia(s)/Artefato(s) adotado(s) por FRIDA: (i) construção de um léxico aplicado para um determinado UdI (ou contexto).

Trabalho 5 – Cysneiros [CYS 2001]

Descrição: Cysneiros [CYS 97, CYS 99, CYS 2000, CYS 2001] apresenta um processo para lidar com RNFs desde as etapas iniciais do processo de desenvolvimento de software até o projeto. Ele apresenta uma forma de integração dos RNFs, elicitados durante as fases iniciais do desenvolvimento do software, com os modelos conceituais gerados a partir dos RFs. Para atingir esses objetivos utiliza um LAL que servirá como base para a construção dos cenários e do diagrama de casos de uso, bem como para a estruturação do grafo de RNFs.

Idéia(s)/Artefato(s) adotado(s) por FRIDA: (i) estereótipos da UML (diagrama de classes); (ii) uso de *checklist* para guiar o processo de elicitação, e (iii) o tratamento explícito dos RNFs desde as primeiras fases do processo de desenvolvimento**.**

Trabalho 6 – Boehm [BOE 96]

Descrição: Boehm e In, em [BOE 96], focalizam seu trabalho na resolução de conflitos entre requisitos. Para tanto, criaram uma base de conhecimento denominada QARCC – ***Q**uality **A**ttribute **R**isk and **C**onflict **C**onsultant*. Nessa base os RNFs são armazenados e priorizados segundo a visão do cliente. A resolução de conflitos ocorre analisando-se os efeitos que cada RNF possui e então, é atribuído um peso positivo (+) ou negativo (-) ao RNF.

Idéia(s)/Artefato(s) adotado(s) por FRIDA: (i) criação da matriz de conflitos, e (ii) etapa do método para identificar e resolver conflitos.

Trabalho 7 – Brito [BRI 2002]

Descrição: Brito et al., em [ARA 2002, BRI 2002], definiu um modelo onde os RFs são especificados usando casos de uso e os RNFs são abstraídos como aspectos. O objetivo principal deste modelo é a manipulação dos RNFs, desde as etapas iniciais do processo de desenvolvimento de software, e o tratamento das propriedades ortogonais, sejam elas funcionais ou não funcionais. Além disso, propõe uma forma sistemática de integração dos RNFs com os RFs através de um conjunto de modelos e *templates*.

Idéia(s)/Artefato(s) adotado(s) por FRIDA: (i) usar aspectos para mapear RNFs, (ii) *template* para RFs, (iii) diagrama de casos de uso, (iv) *template* para aspectos, e (v) tratamento explícito dos RNFs desde as primeiras fases do processo de desenvolvimento.

Trabalho 8 – Grundy [GRU 2001a]

Descrição: Grundy [GRU 2001a] propõe um modelo denominado AOCRE (***A**spect-**O**riented **C**omponent **R**equirements **E**ngineering*), cujo ponto central é o desenvolvimento de sistemas baseados em componentes usando os modelos OO e AO. Essa técnica é caracterizada por diversos elementos relacionados com componentes de software e é altamente específica para o desenvolvimento baseado em componentes.

Idéia(s)/Artefato(s) adotado(s) por FRIDA: (i) componentes relacionam-se (associações, generalização, composição, agregação) com os aspectos.

Trabalho 9 – Rashid [RAS 2003]
Descrição: Rashid et. al. [RAS 2003]: utiliza-se de um modelo genérico para engenharia de requisitos orientada a aspectos. Este trabalho utiliza pontos de visão para elicitar os requisitos, e de forma complementar utiliza matrizes para relacionar os requisitos com os pontos de visão. Além disso, neste trabalho é possível encontrar uma nítida separação dos requisitos funcionais e não funcionais, desde as fases iniciais do ciclo de vida de um software, até as fases posteriores.
Idéia(s)/Artefato(s) adotado(s) por FRIDA: (i) usar aspectos para mapear RNFs, (ii) matriz de conflitos, e (iii) tratamento explícito dos RNFs desde as primeiras fases do processo de desenvolvimento.

Trabalho 10 – Araújo [ARA 2003]
Descrição: Araújo [ARA 2003] mostra neste trabalho que é possível desenvolver uma extensão baseada em aspectos e pontos de visão (*viewpoints)* para o método Vision. Este método é baseado em outros modelos VORD (***V**iewpoint-**O**riented **R**equirements **D**efinition*) [KON 92] e PREview (***P**rocess **RE**quirements **view***) [SAW 2002], sendo que as principais diferenças existentes entre este trabalho e os demais é o uso dos diagramas UML com a definição de associação e agregação entre pontos de vista, assim como a utilização de AO para o tratamento das propriedades ortogonais modeladas para o sistema.
Idéia(s)/Artefato(s) adotado(s) por FRIDA: usar aspectos para mapear RNFs.

Trabalho 11 – Baniassad [BAN 2003]
Descrição: Baniassad e Clarke [BAN 2003], mostram através de Theme/Doc que é possível especificar os requisitos usando descrições textuais. O trabalho consiste na identificação de ações e entidades, na classificação das ações, na associação os requisitos com as ações ortogonais, para posteriormente oferecer uma descrição visual dos requisitos.
Idéia(s)/Artefato(s) adotado(s) por FRIDA: extração das características funcionais e sua associação com as não funcionais.

Tabela 5.1: Comparação Estratégias para Elicitação/Modelagem de Requisitos

Item avaliado	FRIDA	1	2	3	4	5	6	7	8	9	10	11
Fases do ciclo de desenvolvimento do software cobertas												
Requisitos	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔
Análise e Projeto	✔		✔	✔		✔		✔	✔			✔
Implementação	✔		✔	✔					✔			
Orientado a Objetos	✔	✔	✔			✔		✔	✔	✔		✔
Orientado a Aspectos	✔							✔	✔	✔	✔	✔
Fase de Especificação de Requisitos – Técnicas e Artefatos adotados												
Casos de Uso	✔	✔	✔			✔		✔				
Objetivos (*Goals*)												
Pontos de Vista (*viewpoints*)					✔					✔	✔	
Cenários	✔		✔			✔		✔				
Diagrama de Seqüência do Sistema								✔			✔	
Template Com RNFs		✔	✔								✔	
Template Sem RNFs	✔							✔				
Template para RNFs	✔							✔			✔	
Grafo				✔		✔						
Léxico Geral					✔	✔						
Léxico Específico para um contexto	✔											
Base de Conhecimento	✔						✔					
Conflitos	✔			✔		✔	✔	✔		✔	✔	
Checklist para vários Contextos						✔						

Checklist específica para o UdI	✔											

5.3 Estratégias para Projeto com RNFs

Nesta seção, são apresentadas abordagens para o projeto de RNFs, destacando-se: (i) o uso de estereótipos para o diagrama de classes, [ALD 2001, ALD 2003, CYS 2001, SAP 2002, SUZ 99a, SUZ 99b, ZAK 2002], e (ii) a inclusão de novos modelos/elementos [BAN 2003]. A Tabela 5.2 apresenta um quadro comparativo entre as descrições apresentadas a seguir e o método FRIDA.

Trabalho 1 – Cysneiros [CYS 2001]

Descrição: Cysneiros [CYS 97, CYS 99, CYS 2000, CYS 2001], conforme mencionado anteriormente, apresenta um processo para lidar com RNFs desde as etapas iniciais do processo de desenvolvimento de software até o projeto. Nessa última fase, o projeto, utiliza-se de extensões da UML (estereótipos) para modelar os RNFs. Cada RNF é introduzido no diagrama de classes como um estereótipo.

Idéia(s)/Artefato(s) adotado(s) por FRIDA: (i) estereótipos UML (diagrama de classes); (ii) tratamento explícito de RNFs desde as primeiras fases do desenvolvimento.

Trabalho 2 – Suzuki [SUZ 99a, SUZ 99b]

Descrição: Suzuki e Yamamoto [SUZ 99a, SUZ 99b] foram os primeiros autores a definir uma extensão que incorpora na UML o aspecto como uma unidade estrutural. Para tanto, criaram uma extensão denominada UXF/a (***U**ML e**X**change **F**ormat, **a**spect extension*), a qual oferece suporte à atividade de projeto. Eles propõem uma linguagem de descrição de aspectos baseada em XML (*e**X**tended **M**arkup **L**anguage*), que é similar, em alguns aspectos, ao formato XMI (***X**ML **M**etadata **I**nterchange*), padronizado pela OMG.

Idéia(s)/Artefato(s) adotado(s) por FRIDA: (i) estereótipos da UML para modelar aspectos, e (ii) uso de XML para estabelecer vínculos.

Trabalho 3 – Sapir [SAP 2002]

Descrição: Sapir [SAP 2002] concentra-se na definição de uma metodologia para o desenvolvimento de software com aspectos. Esta metodologia usa o ponto de vista "dividir para conquistar". A aplicação é dividida em duas partes: (i) aspectos e (ii) a aplicação que está em desenvolvimento (AUD – ***A**pplication **U**nder **D**evelopment*). Os aspectos são vistos como unidades separadas que encontram-se mescladas com a aplicação.

Idéia(s)/Artefato(s) adotado(s) por FRIDA: (i) divisão da modelagem da aplicação em duas visões: funcional e não funcional; e (ii) tratar aspectos como unidades independentes.

Trabalho 4 – Aldawud [ALD 2001, ALD 2003]

Descrição: Aldawud et al. [ALD 2001, ALD 2003] definem um *profile* UML para AOSD. Um *profile* consiste em um conjunto de estereótipos, *tagged values,* restrições e ícones (imagens gráficas) que possibilitam a modelagem de um sistema que pertence a um domínio específico. Este *profile* fornece convenções e regras aplicadas ao padrão UML visando a construção de aplicações orientadas a aspectos. Com esse *profile* é possível capturar toda a terminologia relacionada com AOSD.

Idéia(s)/Artefato(s) adotado(s) por FRIDA: (i) terminologia, e (ii) identificação dos elementos essenciais de um aspecto.

Trabalho 5 – Zakaria [ZAK 2002]

Descrição: Zakaria et al. [ZAK 2002] apresentam uma extensão da UML para a modelagem de sistemas AO. A extensão é realizada através de estereótipos no diagrama de classes. Neste trabalho são apresentados estereótipos textuais e visuais, o que facilita a identificação dos elementos da AO.

Idéia(s)/Artefato(s) adotado(s) por FRIDA: (i) estereótipos para os elementos que compõe um aspecto: atributos, métodos, pontos de corte, *advices*.

Trabalho 6 – Baniassad [BAN 2003]

Descrição: Baniassad e Clarke [BAN 2003], através de Theme/UML realizam a modelagem de classes e aspectos, bem como a sua combinação. A UML padrão é usada para os elementos da fase de projeto, mas foram criadas outras extensões que complementam a UML.

Idéia(s)/Artefato(s) adotado(s) por FRIDA: a combinação de classes e aspectos.

Tabela 5.2: Comparação Estratégias para Projeto com RNFs

Item avaliado	FRIDA	1	2	3	4	5	6
Paradigma de Desenvolvimento							
Orientado a Objetos	✔	✔	✔	✔	✔	✔	✔
Orientado a Aspectos	✔		✔	✔	✔	✔	✔
Fase de Análise e Projeto – Técnicas e Artefatos adotados							
Modelo Conceitual	✔	✔					
Diagrama de Classes	✔	✔	✔	✔	✔	✔	✔
Diagrama de Seqüência		✔					
Diagrama de Colaboração		✔					
Estereótipos aplicação prática Classes	✔	✔	✔	✔		✔	
Estereótipos aplicação prática Relacionamentos						✔	
Estereótipos aplicação prática Seqüência		✔					
Estereótipos aplicação prática Colaboração		✔					
Estereótipos aplicação teórica Classes					✔		
Estereótipos aplicação teórica Relacionamentos					✔		
Componentes/Classes	✔					✔	✔
XML/XMI	✔		✔				
Separar Aspectos de Classes	✔			✔		✔	✔

5.4 Estratégias de Implementação

O objetivo desta seção é apresentar as principais ferramentas encontradas na literatura para a fase de implementação de aspectos, mais especificamente para a geração de código relacionado com classes e aspectos. A Tabela 5.3 esquematiza as principais funcionalidades oferecidas por essas ferramentas, juntamente com aquelas propostas para a ferramenta FRIDA.

Trabalho 1 –JMangler

Descrição: JMangler [JMA 2002] é um framework implementado em Java que permite transformações de classes Java em tempo de carga, ou seja, antes que as classes sejam ligadas pela JVM – ***Java Virtual Machine***. JMangler habilita a transformação de programas Java sem a necessidade de possuir acesso aos arquivos fonte, pois ele usa as informações contidas nos arquivos *.class* [COS 2001]. Usa a técnica de *bytecode-*

rewriting, ou seja, rescreve algumas partes dos arquivos em *bytecodes* para possibilitar a inclusão de aspectos no código java.

Trabalho 2 –AspectBrowser

Descrição: AspectBrowser [ASB 2002] é uma ferramenta que auxilia os programadores na identificação e no gerenciamento de aspectos [ASB 2002]. Essa ferramenta usa a metáfora de mapas e atlas para mostrar e gerenciar as propriedades ortogonais. Todos os arquivos no programa são mostrados em pequenas janelas, nas quais cada linha de código no arquivo corresponde a uma linha de *pixels* na janela. Cada entrelaçamento identificado é destacado nas janelas com uma cor especifica [ASB 2002]. Essas linhas destacadas são como símbolos em um mapa, elas mostram ao programador uma forma rápida de entender como o código encontra-se disperso através dos vários arquivos fonte [GRI 99].

Trabalho 3 –Hyper/J

Descrição: Hyper/J [HYP 2002b, OSS 2001b] é uma ferramenta baseada na linguagem Java que oferece suporte à separação de responsabilidades de forma multidimensional. Ela propicia a identificação de responsabilidades e a especificação de módulos em função dessas responsabilidades. Ela opera sobre os arquivos .class que serão usados para execução da aplicação. Além disso, usa um arquivo de controle que determina como mapear construções Java para responsabilidades, quais responsabilidades devem ser compostas, e seus relacionamentos de composição, inclusive dando detalhes de como a composição deve ser estabelecida. O objetivo dessa ferramenta não é modificar ou estender a linguagem Java, mas manipular os arquivos *.class*. Analisando esses arquivos é possível realizar extensões, extrações, adaptações e integrações dos componentes Java.

Trabalho 4 – AJDT

Descrição: AJDT [CLE 2003] é uma ferramenta para desenvolvimento de sistemas orientados a aspectos, ela permite: editar, construir e depurar programas escritos na linguagem de programação AspectJ. É possível observar de forma visual todos os elementos do modelo orientado a aspectos, bem como seu relacionamento com o código e os demais elementos.

Trabalho 5 – AspectJ

Descrição: AspectJ [KIC 2001] é uma extensão da linguagem Java para a inclusão de conceitos de orientação a aspectos. A integração de AOP em AspectJ é realizada pela ferramenta associada em tempo de compilação. O arquivo fonte (.java) e os arquivos executáveis (.class) constituem a entrada para o compilador de aspectos, também denominado *compiler aspect weaving*. O resultado desta compilação é um código fonte (.java) que contém o código relacionado com os aspectos e com as classes. Esse código deve ser compilado com um compilador Java padrão [CLA 2001].

Tabela 5.3: Comparação Estratégias de Implementação

Item avaliado	FRIDA	1	2	3	4	5
Paradigma de Desenvolvimento						
Orientado a Objetos	✔	✔	✔	✔		✔
Orientado a Aspectos	✔	✔	✔	✔	✔	✔
Ferramentas de Apoio						
Geração de Aspectos	✔					
Modelagem de Aspectos	✔					
Gerenciamento de Aspectos	✔		✔			
Wizards para criação de aspectos	✔					
Wizards para criação de pontos de corte	✔					
Wizards para criação de advices	✔					
Wizards para criação de Métodos/Atributos						
Classes	✔					
Aspectos	✔					
Manipulação de bytecodes		✔		✔		
Pré-processador	✔[22]	✔				✔
Linguagem Alvo						
Java	✔	✔	✔	✔	✔	✔
Independente						
Rastreabilidade						
Classes	✔					
Aspectos	✔					

5.5 Modelagem dos RNFs: desempenho, confiabilidade e segurança

Trabalhos 1 – Desempenho [FAY 2003]

Descrição: em [FAY 2003] os padrões de projeto são utilizados para mapear os diversos aspectos envolvidos com o desempenho. Essa técnica foi selecionada pois permite variações na modelagem, conforme a arquitetura da aplicação (wireless, cliente-servidor, etc.). Além disso, utiliza conceitos de EBTs (***E**nduring **B**usiness **T**hemes*) e BOs (***B**usiness **O**bjects*) que são estruturas similares aos aspectos. Essas estruturas caracterizam-se por serem: dinâmicas, reutilizáveis e independentes de domínio.

Trabalhos 2 – Desempenho [COO 2003]

Descrição: Cooper em [COO 2003] propõe um framework, onde o desempenho é modelado como um aspecto. Na verdade, esse aspecto é definido como uma coleção de subaspectos que incluem tempo de resposta, medida da capacidade de processamento, utilização de recursos, tempo entre os erros, duração de um evento, tempo entre eventos, entre outros. Ainda nesse trabalho é proposta uma extensão ao diagrama de classes para a representação do aspecto desempenho e de seus subaspectos.

Trabalhos 3 e 4 – Confiabilidade [BON 99a, BON 99b]

Descrição: os trabalhos de Bondavalli propõem uma análise automática de confiabilidade usando UML. Na verdade, os diagramas estruturais da UML compreendem o ponto de partida da identificação, refinamento e captura somente das informações pertinentes ao contexto da confiabilidade. Para essa finalidade utiliza como mecanismos auxiliares estereótipos da UML e redes de Petri.

[22] Utiliza AspectJ

Trabalhos 5 e 6 – Segurança [WIN 2003, WIN 2001]
Descrição: nos trabalhos propostos por Win et al. [WIN 2003, WIN 2001] a segurança de aplicações compreende o foco. Para atingir esse objetivo eles definiram tanto segurança[23], quanto seus atributos (identificação, autenticação e autorização) como aspectos. Além disso, nesses trabalhos são destacadas as vantagens de se utilizar aspectos para a modelagem e construção de aplicações seguras.

5.6 Conclusões

A ferramenta FRIDA associada ao método aqui apresentado neste trabalho é o resultado de uma integração e de um refinamento de idéias e artefatos relacionados com o tratamento de requisitos não funcionais ao longo de processo de desenvolvimento de software. Do universo de trabalhos analisados e resumidos neste capítulo, emergiram muitas das idéias fundamentais: a adoção de casos de uso e seus *templates* associados, o auxílio ao desenvolvedor por meio de *checklists* baseadas em léxicos específicos e de *wizards* para a identificação de aspectos.

[23] do inglês = *security* (falhas intencionais).

6 O MÉTODO FRIDA

Este capítulo dedica-se a apresentar o método FRIDA em detalhes, para este fim cada passo é descrito em uma seção correspondente. Além disso, cada seção cobre três itens fundamentais: (i) conceitos básicos, onde são enumerados os principais conceitos relacionados com o método; (ii) a fundamentação teórica, a qual examina os aspectos teóricos que guiam o método FRIDA; e finalmente (iii) a ferramenta, que descreve os principais critérios adotados pelo protótipo com a intenção de contemplar a fundamentação teórica.

6.1 Introdução

A palavra método origina-se do latim *methodu*, que significa: "(i) caminho para chegar a um fim; (ii) processo racional que se segue para chegar a um fim; (iii) modo ordenado de proceder; (iv) conjunto de procedimentos técnicos e científicos." [DIC 2004]. De certo modo, esse é o objetivo deste trabalho, apresentar um processo (composto por passos bem definidos) que guie o desenvolvedor através dos estágios básicos do ciclo de vida de um software, de forma que tanto requisitos funcionais quanto não funcionais sejam devidamente elicitados.

Os requisitos funcionais geralmente originam hierarquias de classes, formadas por uma ou mais classes interrelacionadas. Por outro lado, como os requisitos não funcionais são propriedades ortogonais das classes o seu encapsulamento em unidades de reutilização é difícil. Isto ocorre porque não existe nenhuma forma de representação, definida no modelo orientado a objetos, para esse tipo de requisito. Ou seja, as metodologias tradicionais (baseadas em classes e relacionamentos) são inadequadas para expressar os atributos de qualidade [WAL 2002].

AOSD é um modelo de separação que possibilita que os requisitos não funcionais possam ser facilmente encapsulados em unidades denominadas aspectos. Como o objetivo desse modelo é fornecer mecanismos para identificar, separar, representar e compor propriedades ortogonais [ARA 2003, BRI 2002], a orientação a aspectos guia todos os passos do método desenvolvido.

Em linhas gerais, o método FRIDA concentra-se na modelagem dos RNFs, bem como na análise e na decomposição dos RNFs através das fases do ciclo de desenvolvimento do software. Relacionamentos entre os artefatos envolvidos/gerados em cada fase do método permitem não apenas a estruturação do método, mas também a automatização de todo o processo através da ferramenta FRIDA (ver Anexo 8).

Cada passo do método utiliza um ou mais artefatos, os quais neste trabalho foram denominados unidades de decomposição (Tabela 6.1).

Tabela 6.1: Unidades de Decomposição Relevantes em FRIDA

Fase do Modelo	Unidade de Decomposição
Identificação e Modelagem de Requisitos Funcionais	Diagrama de casos de uso templates
Identificação e Modelagem de RNFs	Checklist
	Léxico – palavras e expressões
	Matriz de conflitos
Associação dos Requisitos Funcionais com o projeto	Classes e XML
Modelagem Visual de Aspectos	Template para aspectos
	Aspectos (estereótipos UML)

Os passos fundamentais do método encontram-se esquematizados na Figura 6.1.

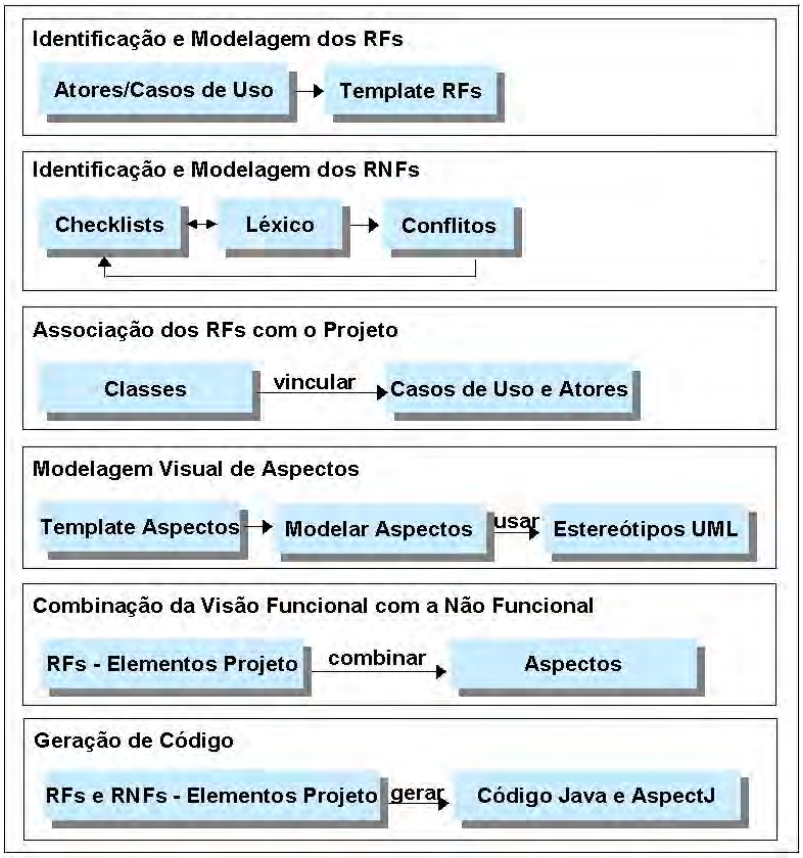

Figura 6.1: FRIDA - passos e artefatos

As próximas seções deste capítulo irão apresentar o método proposto, os conceitos envolvidos e a proposta realizada para cada uma das fases que compõem o método.

6.2 Identificação e Modelagem de Requisitos Funcionais

6.2.1 Conceitos Básicos

Segundo Conallen [COA 2003], a especificação de requisitos compreende "uma coleção de artefatos – documentos, registros de bancos de dados, modelos - que tentam descrever sem ambigüidade um sistema de software a ser criado". Com base nesses artefatos é possível iniciar a especificação dos requisitos, assim como a identificação dos fragmentos de funcionalidade que o sistema deverá apresentar.

Essa fase do ciclo de vida é responsável por estabelecer os objetivos do projeto, investigar as necessidades do cliente e do domínio da aplicação, além de extrair, analisar, documentar e validar os requisitos [HOF 93, SOM 2001]. O objetivo desta fase, após a análise do problema, é definir uma completa especificação do comportamento externo esperado do software que está sendo construído [DAS 88].

Cada processo de desenvolvimento de software costuma utilizar uma determinada técnica para elicitação de requisitos. Alguns trabalhos baseiam-se em *features*, *viewpoints*, e/ou cenários [CLA 2001]. Porém, muitos trabalhos encontrados na literatura [BOO 99, HSI 94, JAC 99, POT 94, REG 95, RUM 94] descrevem os requisitos utilizando o modelo de casos de uso.

O modelo de casos de uso, proposto por Jacobson [JAC 92], tem sido adotado para a especificação de requisitos, pois com ele é possível capturar e expressar o comportamento detalhado do sistema [JAC 2003].

Existem diversos fatores que influenciam a adoção desse modelo para a especificação dos requisitos. Os principais são:

(i) orientado a objetivos - os casos de uso representam o que o sistema deverá fazer (objetivo) e não como ele deve atender os requisitos [REE 2002];

(ii) orientado ao usuário - além de fornecer uma descrição abrangente do que o sistema deve satisfazer, ele ainda oferece um maior entendimento para o usuário, uma vez que é uma forma simples e visual de elicitação de requisitos [JAC 99, BOO 99].

Um caso de uso é inicializado por um ator que possui um determinado objetivo em mente, e ele somente irá finalizar com sucesso se seu objetivo for alcançado. Na verdade, um caso de uso delimita o sistema, porque resume quem ou o que irá interagir com o sistema, e qual funcionalidade é esperada do sistema [COA 2003, REE 2002].

Outro fator determinante para o uso desse modelo é que ele direciona as demais fases do processo (*use-case-driven process*), conforme ilustra a Figura 6.2 (extraída e adaptada de [JAC 99, REE 2002, RUB 98]).

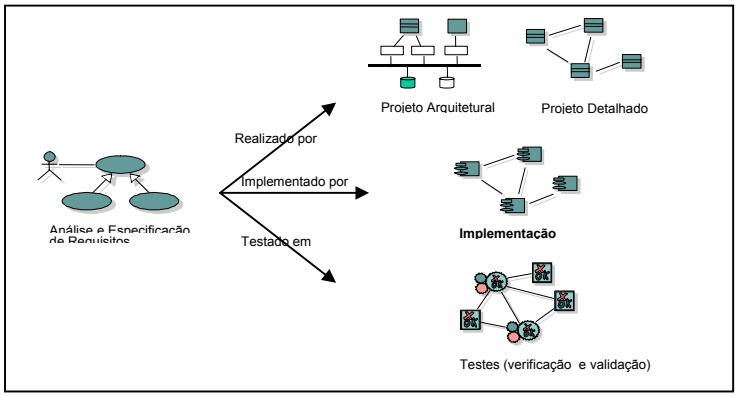

Figura 6.2: Modelo de Casos de Uso X Outros Modelos

O primeiro passo para a construção do modelo de casos de uso consiste em identificar os atores e os casos de uso; o próximo passo concentra-se em estabelecer relacionamentos entre esses elementos e, finalmente, chegar ao diagrama de casos de uso que descreve o problema. Após a estruturação do diagrama é necessário documentar o modelo, pois o diagrama por si só não é suficiente.

Desse modo, para reproduzir todas as funcionalidades e restrições em detalhes de um diagrama de casos de uso desenvolve-se a documentação, que pode ser representada usando (i) documentos gráficos (por exemplo, cenários através de diagramas de seqüência); ou (ii) documentos textuais (por exemplo, *templates*).

Geralmente, cada caso de uso tem associado um documento de especificação, também denominado *template*, para descrever em detalhes o caso de uso. Os *templates*, segundo Coleman [COL 98], não estão relacionados com a implementação, mas com os objetivos do caso de uso.

6.2.2 Método FRIDA: fundamentação teórica

Um requisito pode ser expresso de diversas formas, desde linguagem natural (declarações em alto nível), até o uso de formalismos matemáticos [SOM 2001]. Neste trabalho, a descrição dos requisitos funcionais será realizada de duas formas: (i) visualmente, através do diagrama de casos de uso, e (ii) textualmente utilizando-se templates.

Os *templates* servem como um estilo para guiar e promover a consistência entre as diversas pessoas e os diferentes casos de uso. Para FRIDA, o *template* possui um papel fundamental, pois ele será o ponto de partida para as demais etapas do método.

Conforme pode-se observar na Figura 6.3 o *template* encontra-se dividido em três seções principais:

(i) **geral**: esta seção é indispensável para o entendimento do caso de uso, pois é responsável por agrupar informações de propósito geral. Ela está focalizada nas informações de alto nível que possuem um caráter informativo e encontram-se diretamente vinculadas ao objetivo do elemento que está sendo descrito;

(ii) **caminho**: o objetivo dessa seção é apresentar uma breve descrição dos caminhos (normal, alternativo e excepcional) suportados pelo caso de uso;

(iii) **cenário**: como o modelo de casos de uso é pouco formal para a especificação de cenários [COA 2003] desenvolveu-se esta seção. Nela são detalhadas as ações que devem ser realizadas para garantir que a funcionalidade do caso de uso seja preservada. Além disso, deve-se considerar também que a validação dos cenários deve ocorrer em conjunto com os clientes, porque, assim, é possível verificar a existência de erros e a omissão de informações.

A Figura 6.3 esquematiza o modelo de *template* definido, que consiste em uma combinação de características dos *templates* propostos por [COA 2003, COL 98, JAC 99, MAL 2001, REE 2002, RUB 98, WIE 99].

Geral	Identificador	Compreende uma identificação única que possibilitará a um caso de uso manter a rastreabilidade ao longo de todo o ciclo de vida.
	Nome	Compreende uma pequena frase que irá denominar esse caso de uso. Este nome deve expressar a funcionalidade do caso de uso ou o que ele realiza.
	Objetivo	Uma declaração sucinta que apresenta o objetivo do caso de uso a partir da perspectiva do usuário.
	Autor	Pessoa responsável pela definição.
	Pré-condição	O estado em que o sistema deve encontrar-se antes que esse caso de uso possa ser executado.
	Pós-condição	O estado em que o sistema deve encontrar-se após o cenário primário ter sido finalizado.
	Ator primário	Consiste no ator chave do caso de uso, ou seja é o ator considerado a fonte dos eventos que estimulam a execução do cenário primário.
	Ator secundário	Compreende um ator que pode interagir com o caso de uso. Neste caso, pode-se afirmar que este ator manipula alguma informação relacionada com o sistema, mas não executa nenhuma ação direta sobre o mesmo.
	Prioridade	Usada para decidir a ordem em que os casos de uso serão desenvolvidos, isto é, analisados, projetados e implementados. Prioridade pode estar entre 0 e 3. Onde: 0 – relevante para o domínio da aplicação 1 – prioridade alta 2 – prioridade média ou intermediária 3 – prioridade baixa
	Situação	Um requisito pode estar em uma das seguintes situações: 0 – identificado 1 – analisado 2 – especificado 3 – aprovado 4 – cancelado 5 - finalizado
Caminho	Primário (Normal)	Consiste em uma breve descrição do fluxo principal do caso de uso. Esse caminho não apresenta condições de erro, apenas o caminho que apresenta resultados positivos.
	Alternativo	Compreende uma descrição do fluxo alternativo envolvido na descrição do caso de uso.
	Excepcional	Apresenta uma descrição resumida do fluxo excepcional esperado para o caso de uso.
Cenário	Principal	Nesta parte do *template* as tarefas, ou passos, principais do cenário devem ser descritos.
	Variações	Os passos descritos nesta área do *template* são aqueles que modificam um ou mais passos do cenário principal.

Figura 6.3: Template para Descrição dos Requisitos

Nas primeiras versões do *template* apresentado na Figura 6.3, existia uma região destinada à indicação dos RNFs genéricos que faziam parte do caso de uso. Porém, após uma análise detalhada de alguns estudos de caso realizados chegou-se a conclusão de que nesse momento apenas os RFs devem ser elicitados. Essa decisão leva em consideração que a seção denominada "RNFs" era usada apenas para indicar a presença dos requisitos não funcionais. Porém essa indicação dos RNFs ocorria de forma vaga e incompleta e acabava não contribuindo para a verdadeira elicitação dos RNFs.

Além disso, outros fatores que influenciaram essa decisão compreendem: (i) Chung [CHU 2000] afirma que o detalhamento dos RNFs exige um processo de decomposição, não recomendável neste momento; (ii) Cysneiros [CYS 97] argumenta que para que o desenvolvedor fique concentrado na solução de sua aplicação, deve-se elicitar os RNFs separadamente com relação aos requisitos funcionais. Com isto é possível oferecer um certo grau de transparência a nível de especificação de requisitos.

6.2.3 Método FRIDA: a ferramenta

Todo o método inicia quando o analista constrói o diagrama de casos de uso. Para realizar esta tarefa a ferramenta possui embutido um desenhador de diagramas de casos de uso, o qual possibilita a criação de atores, casos de uso e de seus principais relacionamentos (ver detalhes de ativação no Anexo 8 – seção A8.3).

Entre as funcionalidades proporcionadas por essa etapa de FRIDA, a mais importante é a automatização do *template*. Para tanto, uma janela (Figura 6.4) composta de três abas é ativada, onde cada aba descreve respectivamente: (i) informações gerais do caso de uso; (ii) os caminhos suportados pelo caso de uso; (iii) a representação dos cenários e dos seus respectivos fluxos.

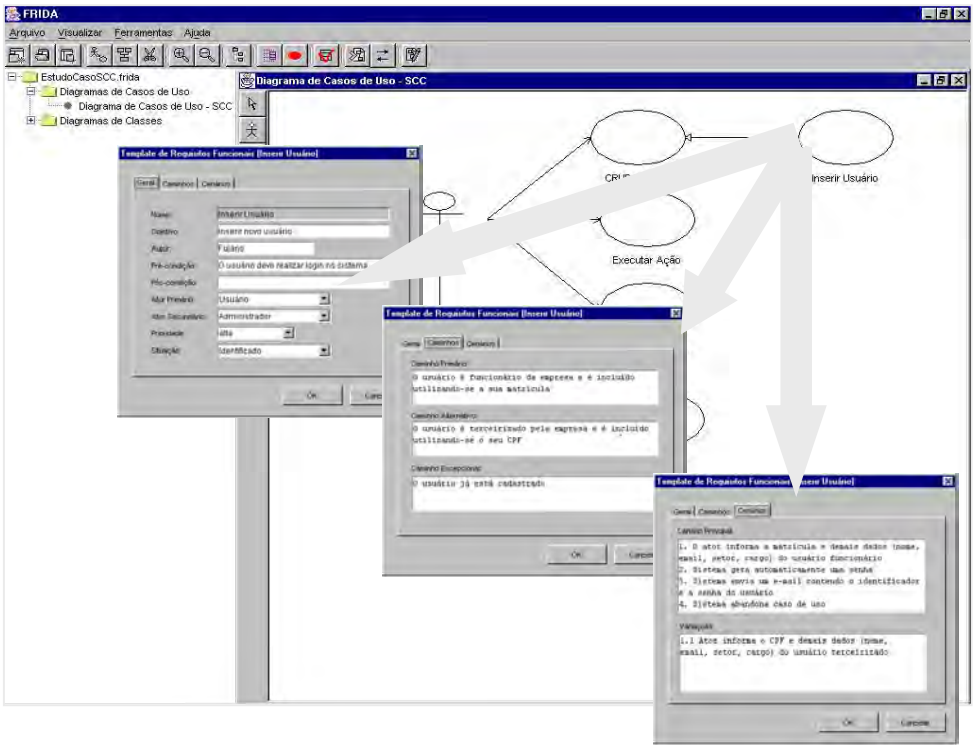

Figura 6.4: Associação do Template a um Caso de Uso

Em resumo, através da automatização de tarefas vinculadas com a especificação de requisitos funcionais busca-se a possibilidade de realizar os demais passos do método, uma vez que o passo atual é o ponto de partida para as demais etapas.

Conforme mencionado anteriormente, o objetivo dessa fase não é o tratamento explícito dos RNFs. Isto é feito a seguir, através de artefatos como *checklists* e léxicos, que auxiliam na elicitação dos RNFs de um modo efetivo.

6.3 Identificação e Modelagem de RNFs: artefato Checklists

6.3.1 Conceitos Básicos

RNFs não possuem um mecanismo padrão para especificação e descrição. Algumas das soluções propostas para a inclusão e o tratamento desses atributos incluem: "*shadow use cases*" [REE 2002], "*Supplementary Specification*" [JAC 99], descrições textuais [COA 2003], *checklists* [BOE 84, CYS 97, McC 77, SOM 2001] entre outros.

Os "*shadow use cases*" compreendem casos de uso, em um diagrama de casos de uso, que ao invés de representarem funcionalidades, são utilizados para especificar requisitos não-funcionais, tais como segurança, auditoria, infraestrutura de arquitetura (RMI – ***R**emote **M**ethod **I**nvocation*, CORBA - ***C**ommon **O**bject **R**equest **B**roker **A**rchitecture*, JDBC – ***J**ava **D**atab**a**se **C**onnectivity*), etc. [REE 2002].

Uma "*Supplementary Specification*" captura os requisitos não funcionais que estão vinculados com cada caso de uso, ou ao sistema como um todo. Ela encontra-se dividida em várias seções (usabilidade, confiabilidade, desempenho, restrições de implementação, etc.) que possibilitam a descrição de RNFs associados a determinados contextos.

As descrições textuais usadas por Connalen [COA 2003] são divididas em (i) seções as quais são usadas para apresentar o RNF e; (ii) subseções adotadas para descrever em detalhe o RNF em cada situação em que for aplicado.

As *checklists* são listas especializadas, baseadas na experiência, de características significantes para garantir o sucesso do desenvolvimento de um software [BOE 84]. Elas são utilizadas na maioria das vezes para a elicitação de requisitos funcionais. O seu uso para RNFs foi apresentado pela primeira vez por [McC 77], onde elas eram usadas somente para realizar a especificação dos RNFs. Uma outra abordagem foi proposta por Cysneiros [CYS 97], na qual são indicados alguns RNFs que podem influenciar uma aplicação e que não podem ser esquecidos.

6.3.2 Método FRIDA: fundamentação teórica

Segundo o que foi mencionado nos primeiros capítulos desse trabalho, a elicitação e a modelagem de RNFs compreendem atividades complexas, uma vez que esses requisitos podem ser observados e avaliados de diversas formas por diferentes desenvolvedores. Além disso, esses requisitos, não são fáceis de interpretar e não costumam ser facilmente percebidos pelos envolvidos. Desse modo, resolveu-se utilizar o conceito de *checklists* [BOE 84], as quais podem ser utilizadas para realizar a verificação de requisitos, pois são excelentes para capturar omissões, além de oferecer completude para a especificação.

Assim, em conjunto com as *checklists*, é utilizada uma taxonomia que classifica os RNFs em genéricos e específicos. Os genéricos compreendem os mais abstratos (por exemplo, confiabilidade, segurança, desempenho, etc.). Já os específicos consistem em decomposições dos RNFs genéricos. Pretende-se assim obter uma maior granularidade, ou seja, um tratamento detalhado da informação. Com esta classificação pode-se afirmar que os RNFs encontram-se organizados em uma hierarquia de requisitos, tornando, assim, o gerenciamento de requisitos mais fácil [COA 2003].

Na classificação apresentada foi aplicada a idéia de usar refinamentos sucessivos sobre os RNFs, pois com uma pequena granularidade o impacto do uso de RNFs é amenizado nas fases subseqüentes do processo de desenvolvimento. Em FRIDA, a

forma adotada para realizar o refinamento foi a aplicação de uma *checklist* – lista de verificação ou conferência. O uso desse artefato é um instrumento fundamental na identificação dos RNFs, porque serve como um guia para o desenvolvedor. Na verdade, a *checklist* pode ser usada no questionamento direto do cliente sobre cada RNFs que ele deseja, ou para lembrar ao engenheiro de software alguns pontos que devem ser observados ou indagados.

Para fins de delimitação de escopo, o método FRIDA trata apenas dos RNFs pertinentes ao contexto de tolerância a falhas e concentra-se em três áreas específicas [LAP 92, LEM 2000, RAN 98]:

1. desempenho: aspectos que descrevem o desempenho de execução do sistema computacional, que em geral encontram-se relacionados com o tempo;
2. confiabilidade/robustez: torna prioritárias as características diretamente relacionadas com a disponibilidade e a confiabilidade no uso da aplicação;
3. segurança: propriedades que especificam perfis de usuários e níveis de acesso ao sistema.

Para cada uma dessas áreas foi criada uma *checklist*, que descreve algumas das principais questões relacionadas com desempenho, segurança e confiabilidade (ver Anexo 3).

A estrutura fundamental da *checklist* pode ser observada na Figura 6.5, a qual também ilustra o cabeçalho.

RNFs	R[24]	P[25]	Restrições
RNF_Específico			
RNF_Genérico_Questão ?			

Figura 6.5: Estrutura da Checklist

Independentemente da área a qual pertence o RNF, algumas informações devem ser indicadas pelo desenvolvedor:

(i) determinar, para cada questionamento, se o RNF específico em análise é relevante para a solução do problema (R);

(ii) atribuir uma prioridade (P) para cada RNF, de modo que, posteriormente, seja possível resolver os conflitos identificados e;

(iii) especificar as restrições vinculadas com o RNF para o sistema que está sendo analisado.

Em resumo, a *checklist* é o primeiro artefato adotado no refinamento dos RNFs. O seu objetivo é incrementar a qualidade de um produto final de software, simplesmente tornando os desenvolvedores conscientes de que alguns critérios de qualidade devem ser cumpridos [BOE 96].

[24] R = Relevante
[25] P = Prioridade

6.3.3 Método FRIDA: a ferramenta

Após construir o diagrama de casos de uso (representação da funcionalidade), o analista inicia com a especificação dos RNFs. Para isso, ele deve determinar a qual nível um RNF pertence. Cada RNF encontra-se classificado conforme seu nível de influência sobre a aplicação:

(i) nível global (ou do sistema): o RNF é pertinente ao sistema como um todo Por exemplo, o requisito confidencialidade pode ser considerado global ao sistema, porque ele é usado por todas as funcionalidades do sistema;

(ii) nível parcial (ou de aplicação): esse tipo de entrelaçamento ocorre quando um RNF encontra-se relacionado com apenas alguns dos RFs, ou seja, é aplicado somente a partes da aplicação. Por exemplo, é possível analisar o item confiabilidade, onde apenas algumas transações necessitam oferecer suporte à replicação e as demais não.

Assim, após essa classificação, cada questão apresentada pela *checklist* deve ser examinada com cuidado. A *checklist* será sempre a mesma, o que muda é a fonte de ativação (ver detalhes de ativação no Anexo 8 – seção A8.4) e o seu escopo com relação à aplicação.

No método FRIDA, a *checklist* encontra-se organizada conforme esquematiza a Figura 6.6. À esquerda da Figura 6.6 é possível encontrar os RNFs organizados de forma hierárquica, onde a raiz compreende um RNF genérico e suas folhas RNFs específicos. Pode-se observar ainda que para cada RNF específico existem associadas algumas questões relevantes visando uma completa elicitação do RNF atual. Na parte direita da Figura 6.6 são esquematizadas algumas informações: (❶) a pergunta para verificar a existência do RNF, (❷) as restrições associadas com o RNF, e (❸) outras informações.

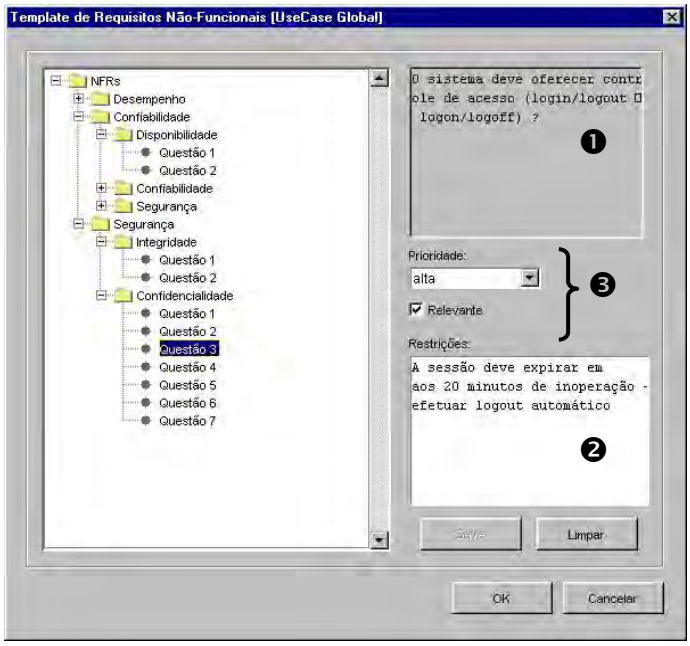

Figura 6.6: FRIDA e a Checklist

Embora o apoio oferecido pelas *checklists* seja satisfatório, ele não é completo, pois a inabilidade de representar explicitamente os RNFs ainda é muito grande [TRA 99]. Assim, a forma encontrada para complementar o uso de *checklists* na obtenção dos RNFs foi a adoção de um léxico.

6.4 Identificação e Modelagem de RNFs: artefato Léxico de RNFs

6.4.1 Conceitos Básicos

Segundo Leite [LEI 2001] todo o processo para a elicitação de requisitos ocorre em um contexto denominado UdI – ***U**niverso de **I**nformações*. Esse contexto compreende o domínio a partir do qual o software deverá ser desenvolvido.

A partir do UdI é possível identificar símbolos (palavras ou frases) que possuem sentido no contexto em que estão sendo aplicados. Assim, é possível criar um léxico, denominado por Leite [LEI 93] como LAL (***L**éxico **A**mpliado da **L**inguagem*), que registra a linguagem utilizada pelos envolvidos preocupando-se apenas com a linguagem do problema, sem a necessidade de compreender a funcionalidade. Basicamente, o foco do LAL reside na identificação dos símbolos e de seus sinônimos.

Uma abordagem aproximada à definida por Leite [LEI 93] é a descrita por Cysneiros [CYS 2001], onde o LAL é usado como um meio de se conhecer a linguagem utilizada no domínio dos envolvidos, bem como para estabelecer uma conexão entre as visões funcional e não funcional (descritas em detalhes naquele trabalho).

Em resumo, nestes trabalhos o formato do léxico é baseado em símbolos e é aplicado sobre os requisitos funcionais e não funcionais. Para tanto, é necessário registrar o vocabulário de um universo do discurso (UofD - ***U**niverse **of** **D**iscourse*) especificado, o qual preocupa-se apenas com a linguagem do problema e não com o problema.

Para usar e descrever um léxico é necessário utilizar uma representação formal. Assim, sabendo-se que o léxico assemelha-se à definição de uma linguagem, a forma selecionada para defini-lo foi a adoção da BNF (***B**ackus-**N**aur **F**orm*) [NAU 69]. A BNF é uma meta-linguagem formal usada para descrever a sintaxe de linguagens. Ela encontra-se dividida em dois tipos de elementos fundamentais:

(i) meta-variáveis ou variáveis meta-lingüísticas: seqüência de caracteres envolvidos pelos sinais de menor e maior, "<>";

(ii) meta-símbolos ou conectores meta-lingüísticos: símbolos da linguagem que denotam definição (::=) ou alternativa (|).

6.4.2 Método FRIDA: fundamentação teórica

A maioria dos requisitos é elicitada, analisada e especificada por desenvolvedores que possuem pouco, ou nenhum treinamento em engenharia de requisitos [CYS 97]. Desse modo, o objetivo do léxico proposto é tentar identificar, nas descrições dos casos de uso, RNFs que foram esquecidos ou negligenciados pelo desenvolvedor.

De forma similar aos trabalhos de Leite [LEI 93] e Cysneiros [CYS 2001], neste trabalho, foi aplicada a idéia de construir um léxico, porém, o formato e o objetivo do léxico não são os mesmos. Em FRIDA, o léxico é similar a um glossário para RNFs, onde cada RNF é descrito e associado a uma entrada específica no léxico.

O léxico apresentado neste trabalho, assim como a definição das *checklists*, engloba a definição de apenas três contextos: desempenho, confiabilidade e segurança (detalhes sobre os termos, palavras e expressões do léxico podem ser encontrados no Anexo 4).

Para entender o mecanismo adotado para o léxico, deve-se observar os passos descritos a seguir:

1. a detecção de um RNF inicia com o processamento e análise da descrição do caso de uso – o *template*. Neste *template* tenta-se identificar palavras, ou palavras derivadas ou expressões que fazem sentido para o léxico;
2. assim que uma palavra é identificada ela torna-se um RNF candidato;
3. verifica-se na *checklist*, se esse RNF candidato já consta como um RNF real. Nesta situação aquele é descartado em favor deste, caso contrário deve-se prosseguir com o processo (passo 4);
4. todos os envolvidos analisam os RNFs candidatos e realizam a confirmação, ou não, da existência do RNF. Caso ocorra a confirmação, uma *checklist* deverá ser associada com o caso de uso.

O léxico contém apenas palavras-chave relacionadas ao domínio do problema. No caso deste trabalho, o léxico descrito no Anexo 4, em formato de BNF, possui um vocabulário pertinente a três RNFs genéricos: desempenho, segurança e confiabilidade. Esse léxico, além de servir como um glossário, pode ser utilizado como uma gramática, que especifica algumas regras para a escrita de RNFs em linguagem natural.

6.4.3 Método FRIDA: a ferramenta

No protótipo desenvolvido, essa fase foi dividida em três etapas (Figura 6.7): construção do léxico, processamento do léxico e verificação dos resultados encontrados (ver detalhes de uso e ativação no Anexo 8 – seção A8.5).

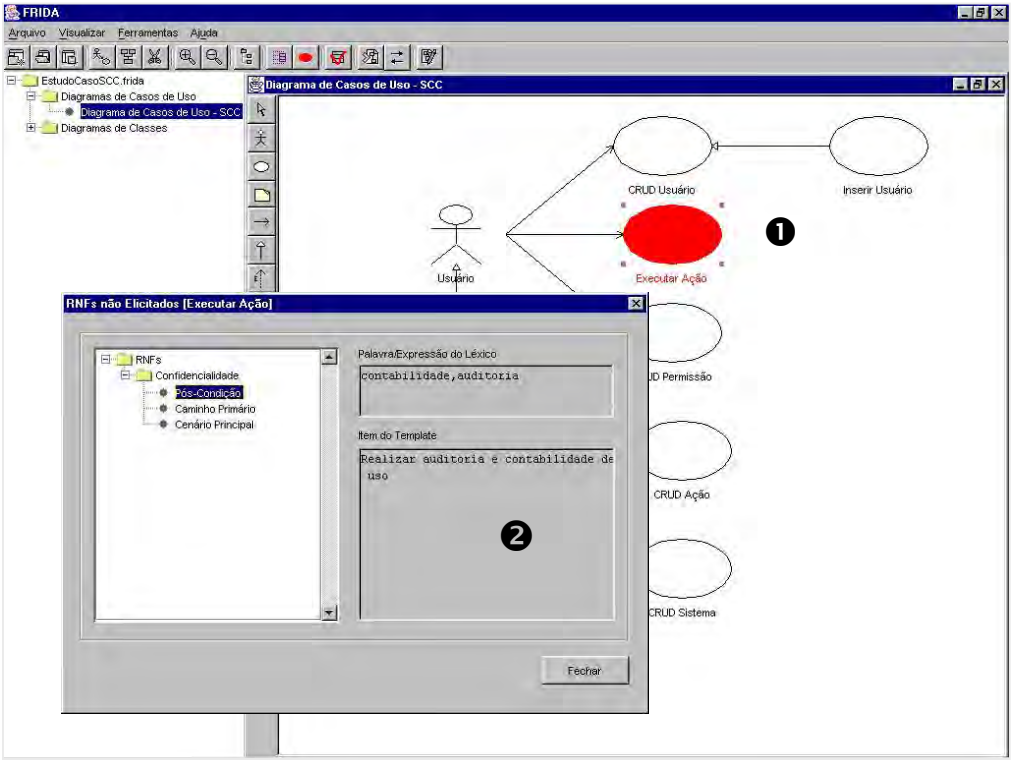

Figura 6.7: Processando e Usando o Léxico

A primeira etapa, construção do léxico, é a mais simples, porque nela ocorre apenas a montagem e a análise das palavras e expressões que compõe o léxico.

O processamento do léxico, segunda etapa, compreende a análise da descrição textual de cada caso de uso. Percebeu-se a necessidade de utilizar um mecanismo para destacar de alguma forma os casos de uso, nos quais foi identificada alguma forma de combinação com as entradas do léxico. Portanto, foi definido um estereótipo para o diagrama de casos de uso baseado em cores (ver Anexo 6). Assim, todo e qualquer caso de uso desenhado na cor vermelha encontra-se estereotipado (Figura 6.7 - ❶), indicando que ele possui algum RNF associado ainda não foi elicitado.

A terceira etapa resume-se à verificação dos resultados encontrados onde, para cada caso de uso estereotipado, é possível reconhecer quais palavras e/ou expressões foram identificadas através do léxico e em quais campos do *template* elas foram reconhecidas (Figura 6.7- ❷).

Deve-se observar que o processamento do léxico apenas indica a existência de RNFs ainda não elicitados. A ativação ou não da *checklist* é responsabilidade do analista, ou seja, o método FRIDA não se encarrega de ativar automaticamente as *checklists*.

O objetivo desses dois artefatos, checklist e léxico, é a elicitação de RNFs e seu detalhamento na construção de sistemas computacionais.

6.5 Identificação e Modelagem de RNFs: Resolução de Conflitos

6.5.1 Conceitos Básicos

Durante o desenvolvimento de um sistema computacional é possível que alguns requisitos entrem em conflito. Basicamente, podem existir três categorias de conflito: (i) requisitos funcionais x requisitos funcionais, (ii) requisitos funcionais x RNFs, e (iii) RNFs x RNFs.

Esses conflitos podem ser tratados em diferentes estágios do ciclo de vida de um produto de software. A escolha da fase em que um conflito será resolvido é definida pelo analista em conjunto com o projetista. Geralmente, a identificação e a resolução de conflitos ocorre nas fases posteriores à da elicitação de requisitos do processo de desenvolvimento. Porém isso costuma ocasionar algumas desvantagens [CLA 2001]: (i) conflitos gerados entre requisitos terão de ser resolvidos por pessoas que não possuem domínio sobre o problema, (ii) as soluções propostas para resolver o impacto dos conflitos, no modelo de projeto, não é propagada para o modelo de casos de uso ocasionando inconsistência no modelo de requisitos.

Caso os conflitos não sejam resolvidos previamente, pode-se chegar na fase de implementação de um produto de software e identificar alguns requisitos conflitantes. Nesse caso, toda a análise deverá ser repensada considerando o impacto que estes conflitos poderão gerar na solução construída [CYS 97].

Conforme afirmações de Clarke [CLA 2001], os conflitos devem ser resolvidos antes da fase de especificação de requisitos ser finalizada, pois quanto antes os elementos conflitantes forem identificados mais fácil e rápida será a sua integração com as fases posteriores.

Com relação a conflitos entre requisitos, além do trabalho de Clarke [CLA 2001] é possível encontrar os trabalhos de Boehm [BOE 96]. Onde, o foco compreende a resolução de conflitos entre requisitos, através de bases de conhecimento, nas quais os RNFs são priorizados pela visão do cliente.

6.5.2 Método FRIDA: fundamentação teórica

Conhecendo-se a natureza conflitante dos RNFs, é importante estabelecer alguns mecanismos para a sua resolução. Conforme citado anteriormente, existem três categorias de conflitos, neste trabalho apenas os conflitos entre RNFs serão abordados.

O primeiro passo para resolver conflitos compreende a análise de uma base de conhecimento, onde os conflitos existentes entre RNFs, previamente determinados, são armazenados. Considerando a posição de Finkelstein e Kramer [FIN 2000] foram selecionadas regras semânticas como mecanismo principal para definir a base de conhecimento. Essas regras permitem identificar quando dois requisitos encontram-se em conflito e também podem ser usadas para determinar como um conflito é integrado à base.

Optou-se por descrever as regras na forma de textos matemáticos porque, ao utilizar um modelo formal para definição das regras, obtém-se mais precisão, além de propiciar a evolução do sistema com relação às modificações que possam ser necessárias [MOU 2001].

Além desse instrumento os conflitos são armazenados em uma matriz de conflitos que indica relacionamentos conflitantes entre RNFs já conhecidos na literatura [BOE 94, CHU 2000].

Após todos os conflitos serem identificados, parte-se para o próximo passo desta fase, que consiste na resolução dos conflitos. Usando a base de conhecimento como ponto de partida é possível estabelecer uma heurística para a resolução dos conflitos identificados.

A heurística determinada é baseada em uma estratégia que avalia as prioridades. Basicamente, ela classifica os conflitos em duas categorias:

(i) ordemPrioridade: quando a resolução do conflito fundamenta-se na prioridade especificada para cada RNF. Se a prioridade atribuída a cada RNF é diferente, então é possível resolver o conflito, pois o RNF com maior prioridade será o dominante. Por exemplo, se o atributo confidencialidade (controle de acesso) possuir prioridade máxima em relação à latência (tempo de resposta), então a heurística determinará que o primeiro atributo é predominante sobre o segundo;

(ii) dependenteContexto: essa categoria é determinada quando um conflito ocorre entre RNFs que possuem a mesma prioridade. Neste caso, o engenheiro deve procurar determinar qual dos RNFs deve predominar, ou ainda, se um requisito deverá ser renegociado com o cliente.

Sabe-se que embora a solução baseada em prioridades possa ser útil ela não é expressiva para capturar todos os relacionamentos de aspectos que um sistema pode requerer [PAC 2002]. Logo, sempre será necessária a intervenção do engenheiro para decidir o que deve ser feito com relação ao conflito.

6.5.3 Método FRIDA: a ferramenta

No protótipo desenvolvido, a resolução de conflitos utiliza somente a heurística "ordemPrioridade". Na verdade, os conflitos são resolvidos de forma transparente, o que significa que o usuário não necessita intervir de forma alguma para resolver os conflitos identificados.

6.6 Extração do Modelo de Análise e Projeto

6.6.1 Conceitos Básicos

A fase de análise, também denominada análise do domínio do problema, possui como objetivo principal refinar e estruturar os requisitos. O refinamento concentra-se em resolver ambigüidades e inconsistências relacionadas com a especificação dos requisitos, além de contemplar a estruturação dos requisitos através da construção de um modelo conceitual [CLA 2001, KAI 99].

Um modelo conceitual é um conjunto de conceitos [CAZ 99], cujo foco está nos elementos/entidades pertinentes ao domínio do problema. Este modelo é utilizado para estruturar os requisitos em três elementos essenciais: conceitos, atributos e relacionamentos. A construção do modelo conceitual é realizada de forma incremental, por tanto deve seguir três passos bem definidos [KAI 99].

O primeiro passo compreende a análise de todo o diagrama de casos de uso para a seleção do conjunto de casos de uso que apresentam a maior prioridade [BOO 99, JAC 99, LAR 2000]. No segundo passo, para cada caso de uso analisa-se a documentação (neste trabalho denominada *template*) de cada caso de uso para obter-se os conceitos e atributos [BOO 99, COL 98, LAR 2000, REE 2002]. Para identificar os elementos do modelo conceitual deve-se realizar uma análise gramatical [RUB 98, SOM 2001, LAR 2000] dos artefatos relacionados com os casos de uso. Essa análise consiste na extração de substantivos e adjetivos de forma que:

(i) substantivos – são selecionados como conceitos e/ou atributos candidatos;

(ii) adjetivos – são eleitos como atributos candidatos.

Após essa seleção deve-se indicar entre os conceitos e atributos candidatos quais serão efetivamente incluídos no modelo conceitual. Finalmente, o terceiro passo é usado para identificar quais os relacionamentos existentes entre os conceitos.

Assim que o modelo conceitual é construído torna-se possível transitar para a fase subsequente: o projeto [CLA 2001]. O ponto de partida da fase de projeto é o modelo conceitual, pois a partir dele será construído o diagrama de classes que dará origem aos demais elementos/artefatos dessa fase. Esta é uma das vantagens essenciais da OO, as classes encontradas na análise são preservadas na etapa do projeto [CAZ 99].

Existe uma tradução quase que automática entre as fases de análise e de projeto, pois os conceitos, atributos e associações são traduzidos respectivamente para classes, atributos e relacionamentos. Deve-se observar que, após a transformação do conceitos para classe, o diagrama de classes pode ser enriquecido progressivamente, através da inclusão de comportamentos, de novos estados, e ainda a complementação por classes auxiliares. O foco do desenvolvedor deve ser basicamente encontrar o comportamento e refinar os relacionamentos entre as classes.

Ao concluir essa fase, o desenvolvedor deve ter em mãos uma hierarquia de classes cujo foco é a representação das soluções de software.

6.6.2 Método FRIDA: fundamentação teórica

O modelo conceitual é construído na fase de análise com o objetivo de detalhar a especificação dos requisitos, além de ser o modelo inicial usado pela fase de projeto [JAC 99]. Para o projeto o modelo mais comumente usado é o diagrama de classes que expressa a solução com relação aos elementos do produto de software.

Nesta etapa do método FRIDA, o desenvolvedor constrói o modelo conceitual e após deriva este para o diagrama de classes. Tendo o diagrama de classes final, deve-se proceder com uma análise detalhada do mesmo tentando identificar se algum dos RNFs detalhados nos Anexos 3 e 4 foi mapeado como classe. Caso algum RNF (pertencente às áreas de: segurança, desempenho ou confiabilidade) tenha sido mapeado no diagrama de classes, ele deverá ser reavaliado, pois em FRIDA esses requisitos são encapsulados em aspectos. A idéia desse mapeamento (RFs para classes x RNFs para aspectos) consiste em evitar o entrelaçamento e a dispersão de informações através de elementos fundamentalmente funcionais [CAZ 99].

Conclui-se que essa fase do método FRIDA é basicamente utilizada para criar o diagrama de classes, onde cada classe representará somente requisitos funcionais.

6.6.3 Método FRIDA: a ferramenta

De forma similar ao primeiro passo do método, nesta fase a ferramenta FRIDA oferece suporte através de um desenhador de diagramas de classes, o qual permitirá o projeto das classes identificadas nos casos de uso, bem como o seu estado e comportamento (Figura 6.8).

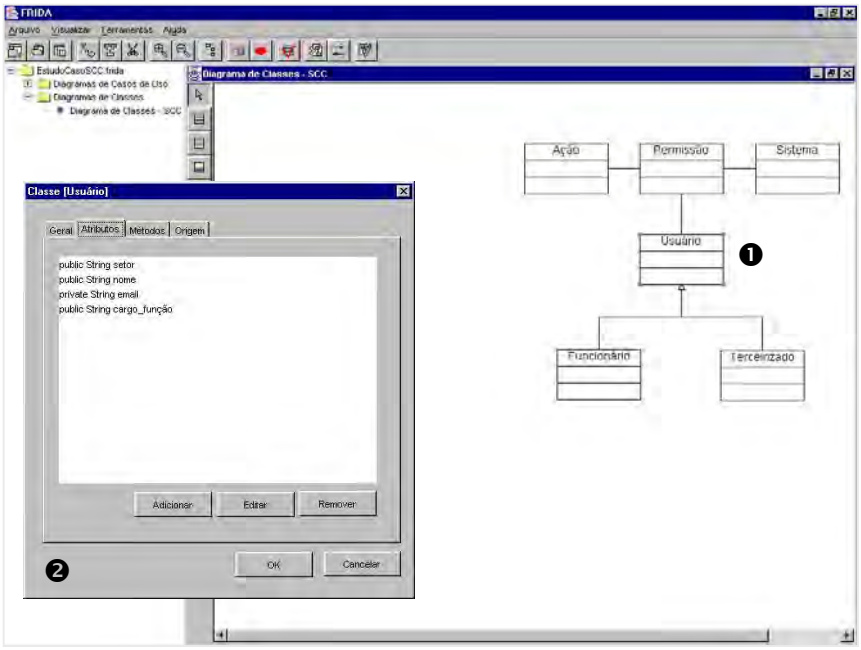

Figura 6.8: Diagrama de Classes

Conforme pode ser observado na Figura 6.8 (parte ❶), detalhes da classe, atributos e métodos não encontram-se visíveis. Eles poderão ser observados somente através de uma consulta às propriedades da cada classe, como ilustra a Figura 6.8 (parte ❷). Outros detalhes sobre a especificação de uma classe podem ser encontrados no Anexo 6 – seção A8.6

6.7 Associação dos Requisitos Funcionais com o Projeto

6.7.1 Conceitos Básicos

À medida que os sistemas são desenvolvidos, tanto os requisitos, como as equipes de desenvolvimento, podem sofrer alterações. Assim, os modelos devem oferecer uma forma rápida e fácil de compreensão do sistema, para possibilitar a inclusão de novos requisitos no sistema ou de novos membros na equipe.

Antes de tomar qualquer decisão que altere alguma parte do sistema é necessário identificar e examinar as conseqüências, para que a decisão seja tomada de forma consciente. Neste contexto, é necessário "rastrear" de alguma maneira o impacto de qualquer ação sobre o sistema. Para este fim um dos conceitos mais empregados é da rastreabilidade, a qual possibilita identificar como um requisito é propagado através dos diversos artefatos usados na construção do sistema.

Em resumo, pode-se afirmar que a rastreabilidade compreende a facilidade de encontrar, tanto de forma *top-down* quanto *bottom-up* os requisitos [PIN 96, COA 2003]. Isso significa que deve ser possível a partir do código, ou do projeto determinar os requisitos e vice-versa [FIR 2003].

A necessidade de se rastrear informações durante o desenvolvimento de um sistema computacional é muito grande. Ela pode ocorrer nas diversas etapas do ciclo de vida, porém, existem alguns pontos nos quais a necessidade de rastreamento é fundamental:

- requisitos x projeto: ocorre quando o engenheiro do sistema necessita conhecer quais elementos de um diagrama de classes se originaram a partir de um determinado caso de uso;
- projeto x requisitos: ocorre quando o projetista do sistema deseja identificar qual(is) caso(s) de uso deu(ram) origem a um determinado elemento no diagrama de classes;
- projeto x código: ocorre quando o projetista necessita analisar as classes que foram geradas a partir da especificação de um diagrama de classes;
- código x projeto: ocorre quando o programador deseja determinar quais elementos, em um diagrama de classes, originaram o código que ele está desenvolvendo.

Alguns dos pontos enumerados anteriormente são facilmente rastreáveis, porém outros necessitam de mecanismos auxiliares. Por exemplo, os itens 3 e 4 são facilmente identificáveis se a organização que está desenvolvendo o software adotar como padrão que toda classe implementada em uma linguagem orientada a objetos qualquer deve, obrigatoriamente, possuir o mesmo nome presente no diagrama de classes, ou se a organização adotar algum gerador de código que siga o padrão especificado no diagrama de classes.

Existem diversas formas de rastrear os artefatos em um projeto: (i) usando convenções de nomes, (ii) utilizar ferramentas que possibilitem o uso de padrões, (iii) utilizar relacionamentos de dependências (UML), (iv) adotar o uso de *tags* para capturar os vínculos entre os artefatos, e (v) a definição de estereótipos [COA 2003].

6.7.2 Método FRIDA: fundamentação teórica

No método FRIDA o elemento fundamental de rastreabilidade o "*link*". A sintaxe do *link* é dada por um documento XML (Figura 6.9) que especifica a partir de qual(is) caso(s) de uso e, consequentemente, em qual(is) *template*(*s*) uma determinada classe.

```
<concept_name = "value">
    <use_cases>
      <template_id>value</template_id>[<template_id>value</template_id> ...]
    </use_cases>
    <actors>  <actor>name</actor>[<actor>name</actor> ...] </actors>
</ concept_name>
```

Figura 6.9: Sintaxe do Link para Rastreabilidade

Através dos casos de uso e de suas descrições textuais é possível identificar as classes que compõem a solução do problema [JAC 99]. Assim, o *link* definido representa uma associação dos requisitos (casos de uso e/ou atores) com as classes de um diagrama de classes. Onde a associação será uma referência entre os diagramas da fase de especificação de requisitos com o da fase de projeto [COA 2003].

6.7.3 Método FRIDA: a ferramenta

Nesse contexto, o método FRIDA estabelece um vínculo entre elementos de diagramas pertencentes a diferentes fases do processo de desenvolvimento. O primeiro passo para acompanhar a propagação de alguma mudança consiste em determinar a origem de uma classe, ou seja, a partir de uma especificação de requisitos qual(is) elemento(s) do diagrama de casos de uso deu(ram) origem a uma determinada classe (Figura 6.10).

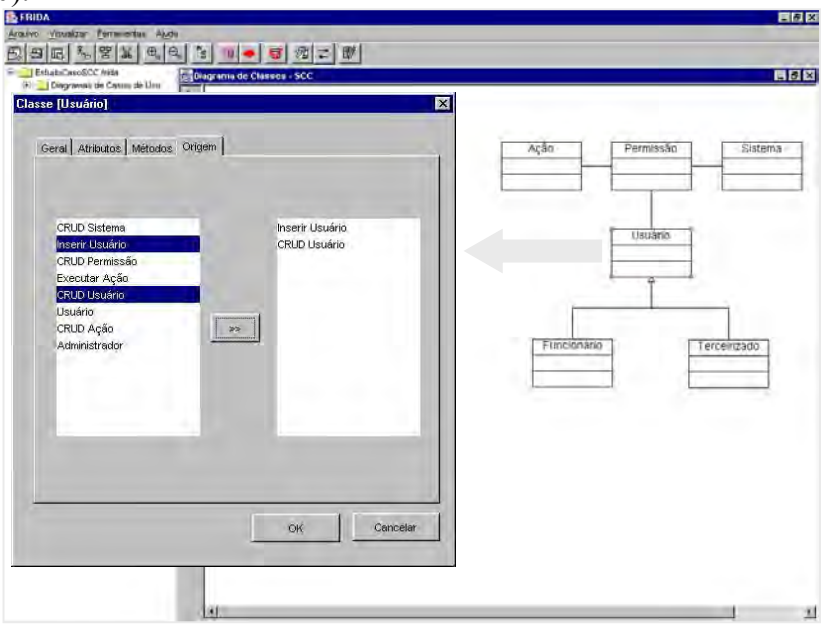

Figura 6.10: Associando Classes com Requisitos

A automatização ilustrada pela Figura 6.11 é o primeiro passo para realizar um gerenciamento efetivo das mudanças. Quando uma alteração é efetuada na especificação de requisitos ela é propagada aos elementos da fase de projeto, que de alguma forma encontram-se vinculados com essa especificação de requisitos. Como esquematiza a Figura 6.11, os elementos que sofreram o efeito de alguma alteração são diferenciados dos demais. Para tanto, resolveu-se colorir (cor vermelha) os elementos afetados pela mudança.

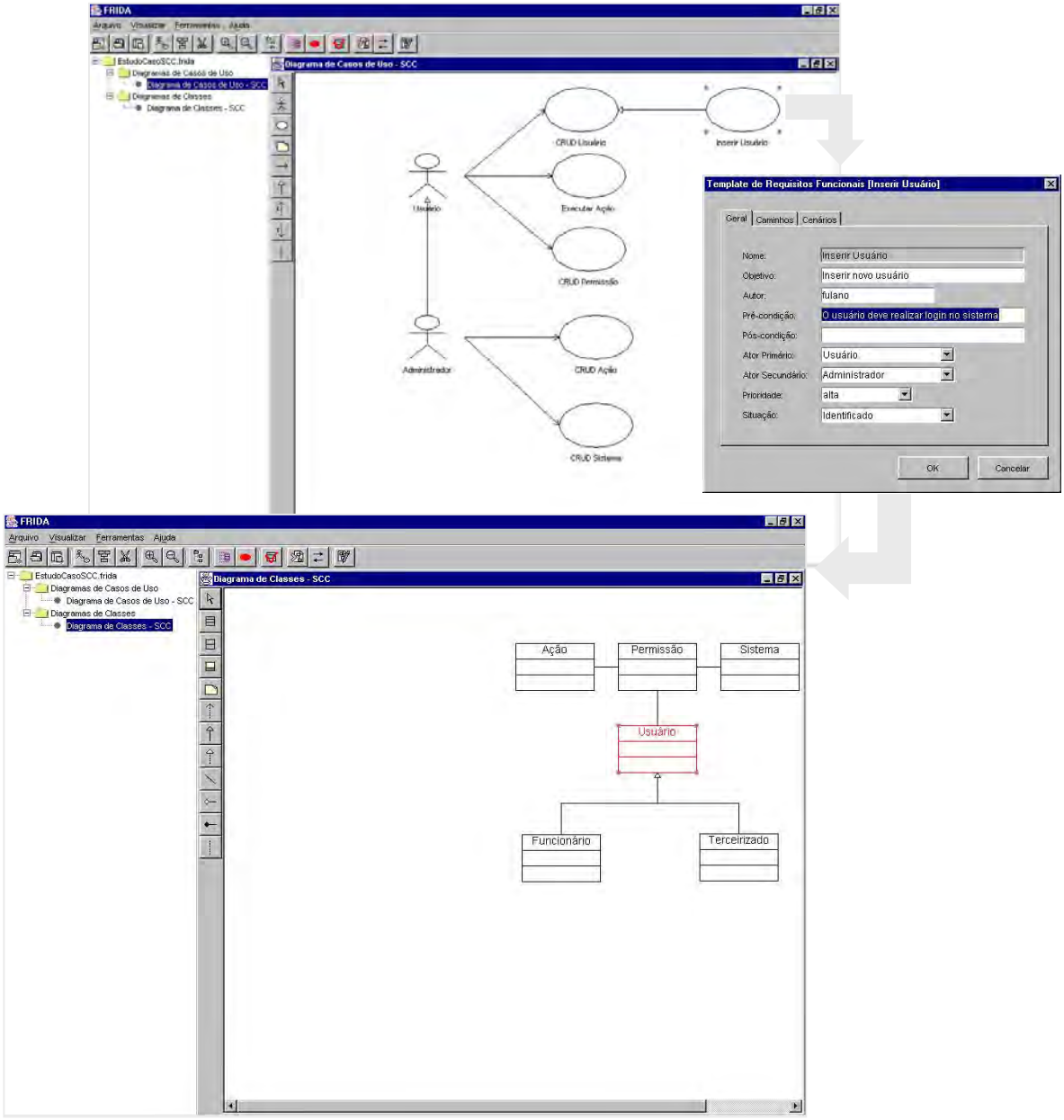

Figura 6.11: Propagação de Mudanças

O impacto relacionado a qualquer alteração na especificação dos requisitos deve ser considerado no contexto do projeto e da implementação. Assim, o objetivo do mecanismo desenvolvido é apresentar ao analista as conseqüências que algumas decisões podem ter no projeto de uma aplicação específica.

No Anexo 8 – seção A8.7 – encontra-se uma descrição sobre como desmarcar uma classe sinalizada pelo protótipo.

6.8 Modelagem Visual de Aspectos: Extração de Aspectos

6.8.1 Conceitos Básicos

Até o presente momento, os RNFs estavam sendo tratados como elementos complementares aos requisitos funcionais. Porém, devido à natureza entrelaçada [BRI 2002, KIC 97] dos mesmos, a sua extração possibilita que sejam tratados como aspectos, com implementação separada do código funcional. Na verdade, cada RNF corresponderá a um aspecto, o qual irá, posteriormente, encapsular toda a implementação relacionada ao RNF.

Segundo Connalen [COA 2003] é importante capturar os RNFs como elementos distintos para permitir o rastreamento. Assim, a partir deste momento, todo RNF será tratado como um aspecto. Consequentemente, será necessário enumerar algumas informações que antes não seriam consideradas essenciais, tais como o nome do aspecto, seu nível (local ou global) e sua prioridade, entre outras.

6.8.2 Método FRIDA: fundamentação teórica

A extração de aspectos consiste em realizar uma busca dos RNFs definidos na primeira fase do modelo. Essa pesquisa identifica os pontos em que existem propriedades ortogonais, ou seja, os pontos de entrelaçamento de um mesmo RNF através dos casos de uso.

Conforme foi observado na seção 6.2, os RNFs não possuem um mecanismo padrão para especificação e descrição. Neste trabalho, os artefatos adotados para este fim compreendem as *checklists* e o léxico. Porém, estes artefatos não são suficientes para capturar precisamente todos os elementos envolvidos com os RNFs.

Uma maneira adicional de especificar requisitos é definir atributos que possam armazenar informações extras sobre um dado requisito. O artefato adotado para descrição dos RNFs é um *template*, que estabelece uma forma de interação única entre as visões (funcional e não funcional). Única porque a descrição da funcionalidade também é realizada via *template*.

Cada aspecto possuirá um *template* (Figura 6.12) vinculado, o qual armazena diversas meta-informações fundamentais, onde cada uma serve para um determinado propósito: identificação, aplicabilidade, prioridade de desenvolvimento, origem, etc.

Nome	Compreende o nome do aspecto que será descrito por este template. Está relacionado com os requisitos genéricos ou específicos, por exemplo segurança, confidencialidade, etc.
Nível	Indica a qual nível (global ou parcial) o aspecto pertence. O nível do aspecto é definido como global (afeta todo o sistema) ou parcial (afeta apenas parte da aplicação).
Descrição	Apresenta uma descrição resumida dos objetivos do aspecto.
Prioridade	Determina o grau de importância do aspecto para o sistema em desenvolvimento, segundo a visão dos envolvidos. Este elemento é extraído das *checklists* após a resolução de conflitos.
Lista de Modelos	Enumera todos os casos de uso em que esse aspecto foi encontrado. A rasteabilidade entre casos de uso e outros elementos do modelo torna mais fácil a manutenção da integridade do sistema. Estabelecer esse *link* ajuda na identificação dos requisito que deram origem ao aspecto.
Lista de Requisitos	Compreendem os requisitos que fazem parte deste aspecto. Este elemento é obtido junto as descrições informadas durante a construção das *checklists*.

Figura 6.12: Template para Aspectos

6.8.3 Método FRIDA: a ferramenta

No protótipo construído a automatização dessa fase ocorre em conjunto com a próxima fase. A Figura 6.13 ilustra apenas a geração automática do *template*. O *template* para o projetista torna-se apenas uma aba que contém os detalhes do aspecto em questão. Com esses detalhes é possível identificar a origem do aspecto (parte ❷), assim como o seu escopo (parte ❶) e suas restrições (parte ❸) com relação à aplicação em desenvolvimento.

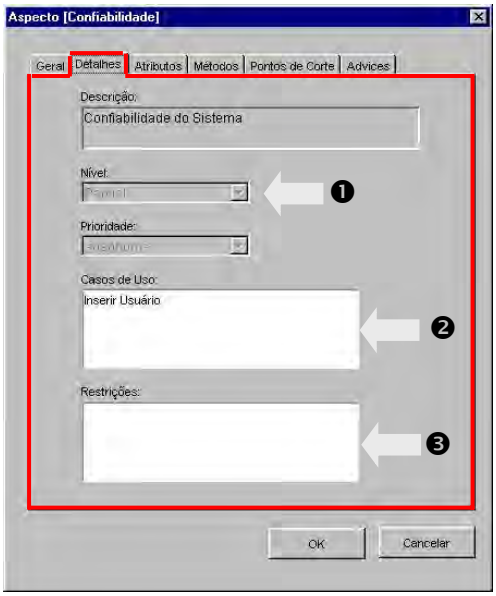

Figura 6.13: Automatizando o Template para Aspectos

6.9 Modelagem Visual de Aspectos: Estereótipos UML

6.9.1 Conceitos Básicos

A análise e o projeto orientados a objetos estão divididos em dois tipos de unidades [CLA 2001]: estrutural (classes, objetos e atributos), e comportamental (operação, método e interface). Nota-se que os aspectos não estão incluídos em nenhuma dessas unidades.

Segundo diversos autores [ALD 2001, ALD 2003, SUZ 99a, SUZ 99b, SAP 2002, ZAK 2002] uma das melhores formas de representar a modelagem de um aspecto é definindo extensões para a UML, pois assim como a UML é a linguagem padrão para modelagem de OO, nada mais natural do que usá-la para a AOD.

Na maioria dos casos, as extensões são realizadas sobre o diagrama de classes e seus elementos. Um diagrama de classes apresenta uma descrição dos objetos de um sistema e, também, dos relacionamentos entre as classes, tais como associação, generalização, agregação, composição e dependência.

As classes são representadas utilizando-se um retângulo compartimentado em três áreas, onde são mostrados respectivamente, o nome da classe, seus atributos e seus métodos. Os relacionamentos entre as classes são representados como linhas estilizadas conectando classes.

A UML fornece alguns mecanismos para realizar uma extensão formal conforme descreve a seção 2.6.

6.9.2 Método FRIDA: fundamentação teórica

Neste trabalho, os aspectos serão representados utilizando uma extensão ao diagrama de classes, conforme ilustra a Figura 6.14. Nele os aspectos serão identificados por um rótulo (<<*aspect*>>) e um nome (indicando o RNF que este aspecto abstrai).

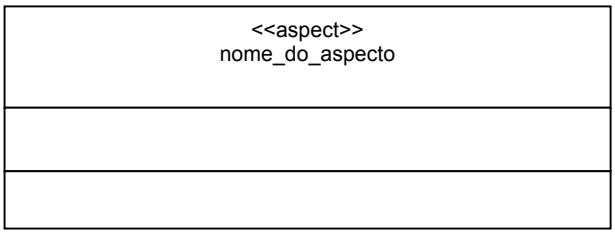

Figura 6.14: Representando um Aspecto na UML

Aspectos, assim como classes, são divididos em estado (atributos) e comportamento (métodos) [SUZ 99b, ZAK 2002]. Além disso, eles também podem encontrar-se organizados sob uma hierarquia de herança [LAD 2003].

Outros elementos relacionados com aspectos ainda devem ser modelados nesta fase. Um ponto de combinação (por exemplo chamada de método, chamada de construtor, acesso a um atributo, entre outros) não será representado em FRIDA, porque ele somente será executado quando associado a um ponto de corte. Um outro ponto de vista que pode ser levado em consideração para justificar essa decisão é o de Stein [STE 2003], o qual afirma que não existem classificadores legíveis para representar os pontos de combinação na UML.

Os pontos de corte são estabelecidos entre uma ou mais classes e um aspecto. Em FRIDA esses elementos são representados como um "*UML Operation meta-model element*". O que significa dizer que todo ponto de corte será representado como um método abstrato. Para formar um ponto de corte vários pontos de combinação devem ser conectados usando-se operadores lógicos, tais como *and*, *or* e *not*. Aqui a conexão desses pontos utiliza sempre o operador *or*; caso seja necessário é possível alterá-lo.

O *advice* (Figura 6.15) é outro elemento relacionado com AOSD. Ele é usado para especificar o momento em que um ponto de corte deve executar. Em FRIDA, o elemento *advice* também é representado como um "*UML Operation meta-model element*". Algumas considerações devem ser realizadas: (i) o comportamento de um *advice* não é opcional, e portanto, ele deve ser associado ao ponto de corte utilizando-se uma descrição textual; e (ii) somente três tipos de *advice* são permitidos: *before, after e around.*

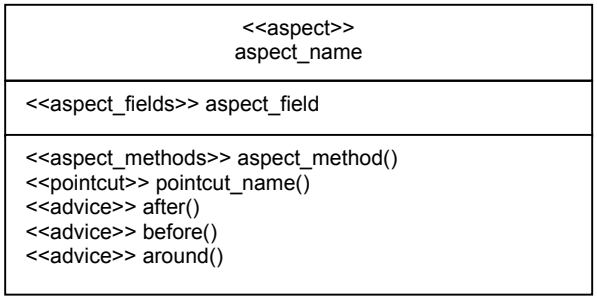

Figura 6.15: Representando todos os Elementos de um Aspecto

6.9.3 Método FRIDA: a ferramenta

Para esta etapa a ferramenta irá gerar automaticamente todos os aspectos identificados através das *checklists* e do reconhecimento automático do léxico (Figura 6.16).

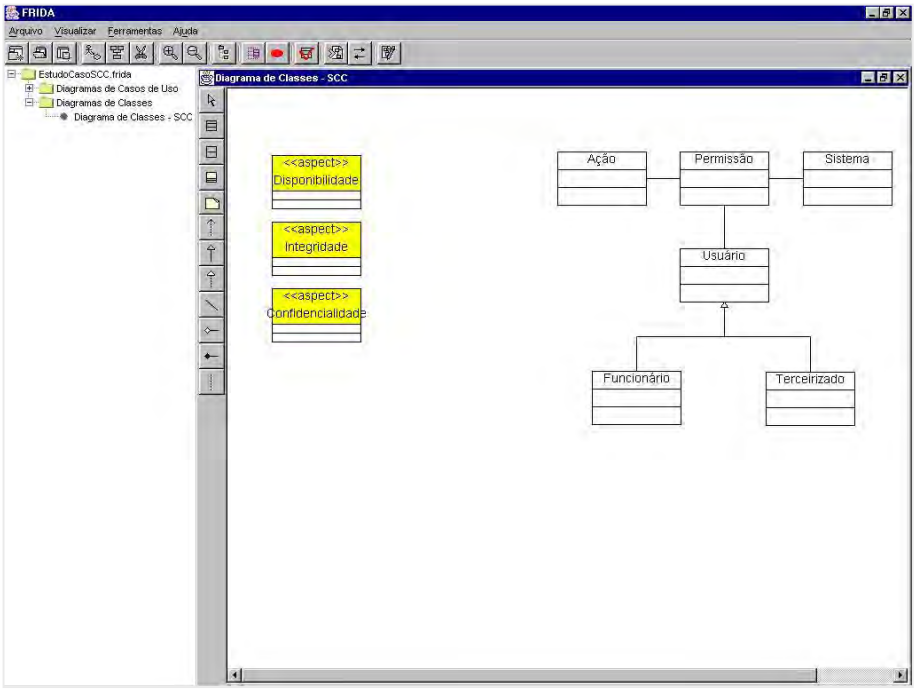

Figura 6.16: Geração Automática de Aspectos

Conforme pode ser observado na Figura 6.16, além do estereótipo <<*aspect*>>, foi definido um estereótipo baseado em cores, oferecendo um maior destaque para o elemento estrutural – o aspecto (definições e detalhes podem ser encontrados no Anexo 6).

A automatização dessa fase possibilita ao desenvolvedor associar os atributos e métodos de cada aspecto, bem como seus *advices* e pontos de corte. Com relação aos métodos, para cada requisito enumerado na lista de requisitos do *template* apresentado na seção 6.8, será gerada de forma automática a assinatura de um método. No Anexo 8 – seção A8.8 pode-se encontrar as demais janelas que são utilizadas para descrever um aspecto.

6.10 Combinação da Visão Funcional com a Não Funcional

6.10.1 Conceitos Básicos

O desenvolvimento de um sistema deve levar em consideração tanto os requisitos funcionais quanto os não funcionais, pois a funcionalidade é a base do sistema, e os RNFS compreendem as restrições impostas a essa funcionalidade. Logo, não basta identificar elementos de projeto, que representem a funcionalidade e suas restrições, se esses elementos (componentes e aspectos) não estão de alguma forma combinados, ou relacionados.

Componentes são unidades de produção, aquisição e instalação independentes, que interagem para determinar a funcionalidade do sistema [LOR 2001]. Sob o ponto de

vista de implementação de programas, componentes podem ser vistos como unidades de compilação ou mesmo outras entidades passivas usadas em programas, como arquivos e documentos.

O uso de componentes é motivado por inúmeros fatores, entre os quais: reutilização de projeto e implementação, encapsulamento, possibilidade de adicionar novas funcionalidades pela adição de novos componentes à aplicação, bem como sua atualização através da troca de componentes por outros de versões mais recentes [BEL 99, PIF 99].

No contexto deste trabalho, componentes são definidos como unidades ativas, encapsuladas na forma de classes, que são consideradas tipos de componentes; ou seja, um componente de um programa é obtido por instanciação de um tipo de componente. Os tipos de componentes se relacionam de forma estática (estrutural), através de relacionamentos da OO, ou de forma dinâmica (funcional), emitindo e recebendo mensagens [LIS 95].

6.10.2 Método FRIDA: fundamentação teórica

Até este momento existem dois elementos principais relacionados com a fase de projeto: classes e aspectos. Porém esses elementos encontram-se desconectados uns dos outros.

Para realizar a ligação entre a visão funcional (classes) e a não funcional (aspectos) deve-se seguir uma seqüência de passos pré-determinados:

1. usando o *link* definido na seção 6.7 é possível determinar quais requisitos funcionais originaram cada classe;
2. analisando o *template* definido na seção 6.8 pode-se identificar a partir de quais RNFs foram gerados os aspectos e com quais casos de uso o aspecto encontra-se de alguma forma vinculado;
3. realizando o cruzamento dessas informações é possível identificar com quais aspectos cada classe está associada.

Ao término desses passos cada classe deve estar associada a um ou vários aspectos. Deve-se observar que existirá um único aspecto para cada RNF. Para determinar o relacionamento das classes com os aspectos deve-se observar a qual nível o aspecto pertence: global ou parcial. Em um primeiro momento, o relacionamento entre as classes e os aspectos são realizados através de "relacionamentos estereotipados", ou seja, através de extensões do "*UML Association meta-model element*". Em FRIDA, o estereótipo utilizado foi <<*crosscutting*>>, justamente para enfatizar que o aspecto é uma propriedade ortogonal à classe que está ligada a ele.

A representação dos relacionamentos parciais e globais pode gerar cruzamento de linhas tornando o entendimento do modelo muito difícil. Assim, quando o nível do aspecto for global, ele não estará associado com nenhuma classe em particular, uma vez que o seu nível determina que ele afetará todas as classes.

6.10.3 Método FRIDA: a ferramenta

Com o objetivo de facilitar o trabalho do projetista, o protótipo investiga os relacionamentos estabelecidos entre os elementos da visão funcional com os da visão não funcional (através dos *templates*, *links*, etc.) e gera, automaticamente, os relacionamentos entre componentes e aspectos, conforme esquematiza a Figura 6.17.

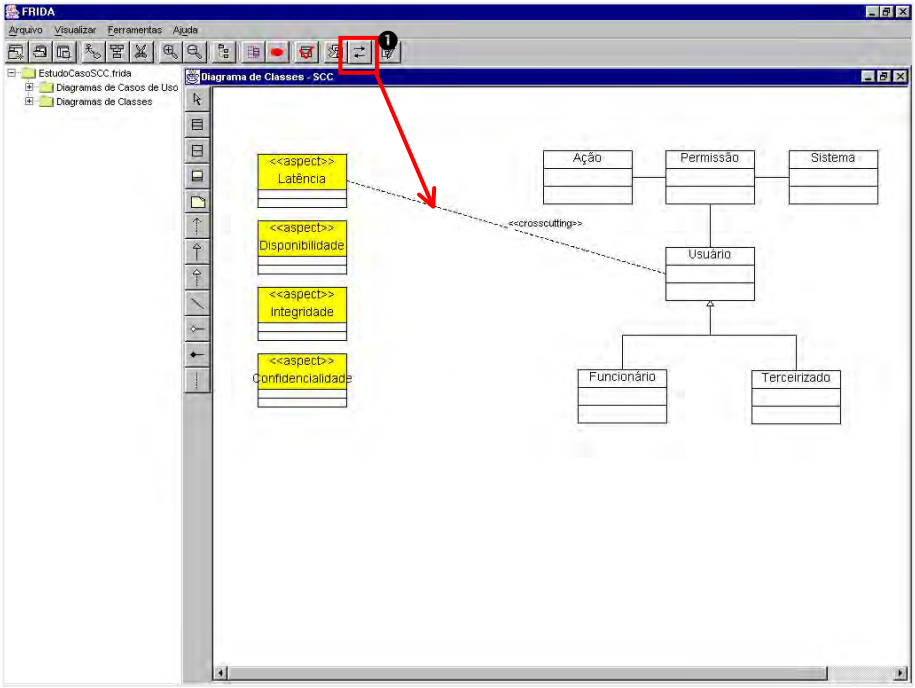

Figura 6.17: Combinando Componentes e Aspectos

Nota-se na Figura 6.17 que apenas o aspecto Latência encontra-se vinculado a classe usuário. Isso deve-se ao fato de que esse RNF possui um escopo parcial, enquanto os demais aspectos são globais à aplicação.

6.11 Geração de Código

6.11.1 Conceitos Básicos

Os casos de uso foram finalizados, a análise encontra-se estabilizada, o projeto foi considerado adequado e liberado para a construção da aplicação. Nesse momento, começam muitos dos problemas do processo de desenvolvimento. O principal deles consiste no mapeamento dos diagramas para código, uma vez que o programador deve possuir um conhecimento dos casos de uso para garantir que seus significados e objetivos sejam materializados na forma de código executável.

Além disso, a geração de código pode encontrar restrições de implementação, as quais compreendem condições que limitam a implementação do software, tais como o software foi desenvolvido para uma única plataforma, ou o software deve ser implementado em uma determinada linguagem visto que alguns conceitos ainda não existem nas demais linguagens de programação, entre outras.

6.11.2 Método FRIDA: fundamentação teórica

Como o método FRIDA encontra-se fundamentado no uso de orientação a aspectos, é conveniente usar um gerador de código específico.

O objetivo do gerador é aumentar a produtividade, gerando de forma automática trechos de código. Esses trechos são construídos com base nas informações apresentadas nos *templates*, no diagrama de classes e aspectos construído, bem como na definição de alguns estereótipos. A geração automática de código traz como benefícios a redução do tempo de desenvolvimento, um menor esforço de testes e maior qualidade das classes e aspectos gerados, visto que pode-se pressupor que o código gerado esteja correto.

Convém observar que, em FRIDA algumas restrições de implementação foram adotadas para viabilizar a geração de código: a linguagem selecionada foi Java, e a linguagem de aspectos escolhida foi AspectJ.

6.11.3 Método FRIDA: a ferramenta

Atualmente, o protótipo FRIDA gera apenas os esqueletos de código relacionados com: interfaces, classes e aspectos (ver Anexo 6).

6.12 Conclusões

Pode-se observar pelos diversos trabalhos encontrados na literatura que a correta elicitação de requisitos é vital para se obter um software com qualidade. Neste processo de elicitação geralmente os desenvolvedores focalizam na funcionalidade, mas é fundamental que os RNFs também sejam elicitados.

Nesse contexto, o método FRIDA foi definido visando principalmente a elicitação dos RNFs, mas não esquecendo os RFs pois eles são dependentes. A idéia aplicada ao método é estabelecer passos bem definidos, compostos por diversos artefatos, que possibilitem demonstrar como os requisitos elicitados podem guiar o resto do processo de desenvolvimento de software.

Alguns passos concentram-se apenas na visão funcional (*templates*, diagramas de classes e vínculo com os casos de uso), enquanto outros foca na visão não funcional (*checklists*, léxico, conflitos, *template* para aspectos e modelagem visual dos aspectos). Em FRIDA, é prevista a combinação desses artefatos, bem como a geração do código resultante de todos os passos do método.

Em resumo, o método FRIDA consiste em uma estratégia que estabelece como elicitar e tratar RNFs desde o início do processo de desenvolvimento de software. Através de seus passos é proposto um processo sistematizado de integração dos RNFs ao projeto do sistema, e posteriormente ao código.

7 ESTUDO DE CASO

Este capítulo apresenta um estudo de caso realizado para validar a estratégia apresentada nesta tese. Para tal, ele realiza uma breve introdução a algumas características do desenvolvimento de sistemas seguros, assim como introduz algumas definições desta área, objeto deste estudo de caso. Após descreve o estudo de caso e propõe a sua resolução utilizando os passos e artefatos do método FRIDA.

7.1 Introdução

O termo segurança, quando aplicado a software, pode corresponder a expectativas diversas, de acordo com a comunidade que o interpreta. Na modelagem de sistemas seguros, é conveniente estabelecer quais propriedades envolvidas que realmente constituem segurança, de forma a tornar consistente seu projeto e implementação.

Neste texto, o sentido adotado para o termo segurança[26] é o mesmo apresentado por Avizienis et. al. [AVI 2001], segundo o qual segurança é uma propriedade do sistema, sendo ela própria constituída por três outras:

(i) disponibilidade: propriedade do sistema que indica se ele está pronto para uso somente por usuários autorizados, assumindo a entrega do serviço corretamente;
(ii) confidencialidade: é a não-divulgação de informações sigilosas e;
(iii) integridade: compreende a inexistência de alterações realizadas por usuários não autorizados, levando o sistema a estados errôneos e inconsistentes.

Randell, em [RAN 98], apresenta segurança e outras propriedades do domínio de tolerância a falhas como pertencentes a um contexto mais amplo, e congregadas sob o conceito de confiabilidade[27], do inglês *dependability*, que também é considerada uma propriedade importante dos sistemas computacionais atuais.

Além das propriedades acima mencionadas, segurança pode ter atributos (ou propriedades) secundárias, tais como:

- contabilidade[28]: registra as atividades dos usuários com vistas a cobrar pelo uso de recursos, possivelmente em um nível mais abstrato do que a auditoria. Tipicamente, registra-se o tempo de uso e o custo de determinadas operações. Convém observar que esta técnica por si só não representa um mecanismo de

[26] do inglês *security*, composta pelos atributos confidencialidade (*confidentiality*) e integridade (*integrity*).
[27] do inglês *dependability*, composta pelos atributos confiabilidade (*reliability*) e disponibilidade (*availability*)
28 do inglês *accounting*

segurança, somente quando aplicada em conjunto com outras técnicas de segurança;
- autenticidade: relativa a integridade do conteúdo e da origem das mensagens trocadas.

A fim de atender às propriedades de segurança, um sistema dispõe de vários mecanismos, dentre os quais destacam-se:

- autenticação: que identifica e permite o acesso aos recursos somente por parte de usuários autorizados (por exemplo, *login*);
- autorização: que contém regras sobre a utilização de recursos por parte de usuários, a exemplo das listas de controle de acessos - ACLs (***A**ccess **C**ontrol **L**ists*). Essas listas contém uma lista dos usuários, recursos e operações, combinando-os garantindo ou negando o acesso;
- auditoria: que registra as atividades dos usuários com vistas a seguir/registrar suas ações, tal como nos sistemas de *logs*.

Todos esses mecanismos podem apresentar vantagens e desvantagens, porém, eles devem ser levados em consideração no projeto de sistemas seguros.

7.2 Descrição do Estudo de Caso

O estudo de caso baseia-se no desenvolvimento de um sistema de segurança corporativo (SSC), cujo foco é implementar políticas de segurança baseadas em autenticação, autorização, auditoria e contabilidade em um sistema de banco de dados. O SSC pode ser visto como uma fachada (padrão de projeto façade), que consiste em uma interface única e simplificada para o acesso as funcionalidades que o sistema tem para oferecer [GAM 2000]. Na verdade, com esse padrão pretende-se expor a maioria dos serviços de segurança do sistema de modo que possam ser usados por outros sistemas.

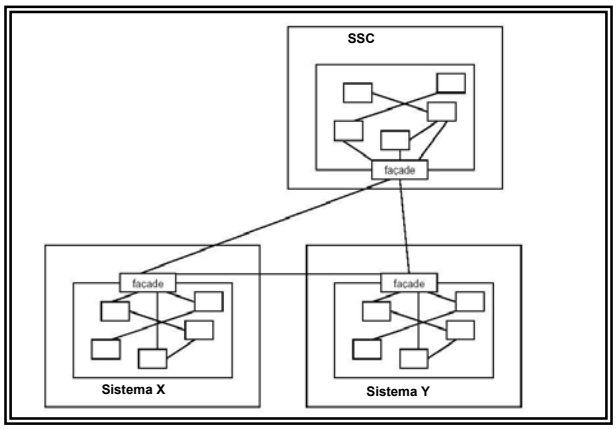

Figura 7.1: Arquitetura do Sistema Proposto para o Estudo de Caso

Os clientes deste sistema utilizam plataformas de acesso heterogêneas, e estão dispersos em uma área geográfica considerável. As propriedades de segurança a serem consideradas neste sistema envolvem disponibilidade, confidencialidade e integridade. Dada a relevância do sistema, também são consideradas como propriedades principais a autenticidade e a contabilidade.

Para a autenticação, um mecanismo de *login*, utilizando protocolo HTTP[29] seguro (https) permite acesso dos clientes aos dados pertinentes à sua aplicação. Uma vez que um usuário tenha obtido acesso, as ações requeridas pelo cliente (inserções, remoções e consultas) passam por mecanismos de autorização, que efetivam ou negam de acordo com as permissões (descritas através de ACLs) que o cliente tenha contratado com os provedores de serviço.

Não é permitido aos clientes o acesso a bases que não tenham sido contratadas, assim como é realizada uma distinção entre administradores e usuários das mesmas.

Como forma de auxílio na detecção, prevenção e remoção de falhas, um serviço de auditoria monitora as ações executadas pelo cliente. Para o controle de auditoria são registrados parâmetros e estados do sistema, e estes registros são mantidos por prolongados períodos de tempo, eventualmente anos.

Para que os provedores de serviço possam efetuar a cobrança, as ações do cliente passam por um sistema de contabilidade, que atribui pesos às ações. Este sistema é distinto do de auditoria, pois registra as ações em um nível mais abstrato, e não mantém os registros após a cobrança dos serviços.

7.3 Método FRIDA Aplicado ao Estudo de Caso

O objetivo deste estudo de caso é avaliar se a estratégia proposta nesta tese efetivamente traz benefícios de qualidade ao software.

7.3.1 Identificação e Modelagem dos Requisitos Funcionais

O primeiro passo da análise e especificação de requisito consiste na identificação dos limites e da descrição das características primárias do sistema a ser desenvolvido. Após o analista capturar as expectativas dos envolvidos no sistema e compreender as necessidades do sistema, o diagrama de casos de uso é construído. Ele esboça a funcionalidade básica do sistema e através de seus elementos é possível definir quem irá interagir com o sistema e quais funcionalidades serão contempladas. Desse modo, após uma análise detalhada do sistema proposto como estudo de caso o diagrama de casos de uso definido encontra-se esquematizado na Figura 7.2.

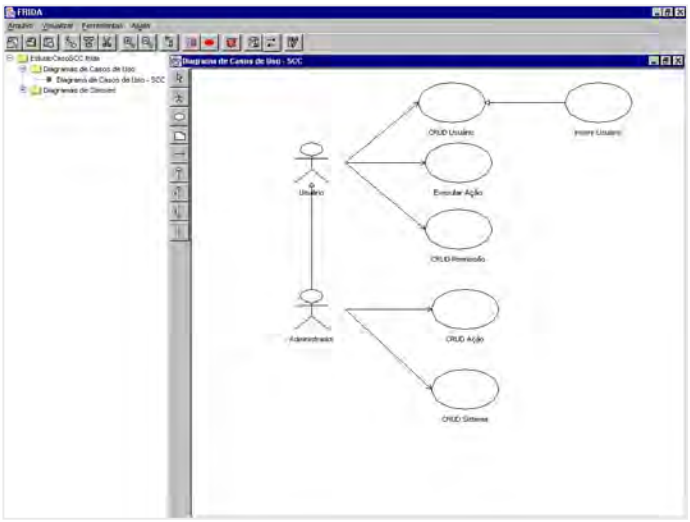

Figura 7.2: Estudo de Caso: diagrama de casos de uso

[29] HTTP = ***H**yper**T**ext **T**ransfer **P**rotocol*

O diagrama de casos de uso por si só não é suficiente para expressar a funcionalidade do sistema que está sendo modelado. Assim, é necessário agregar a esses diagramas descrições adicionais, de modo que a identificação dos requisitos seja o mais completa possível. No método FRIDA, essas descrições encontram-se textualmente diagramadas através de *templates* (Figura 7.3 - *template* relacionado ao caso de uso *Incluir Usuário*).

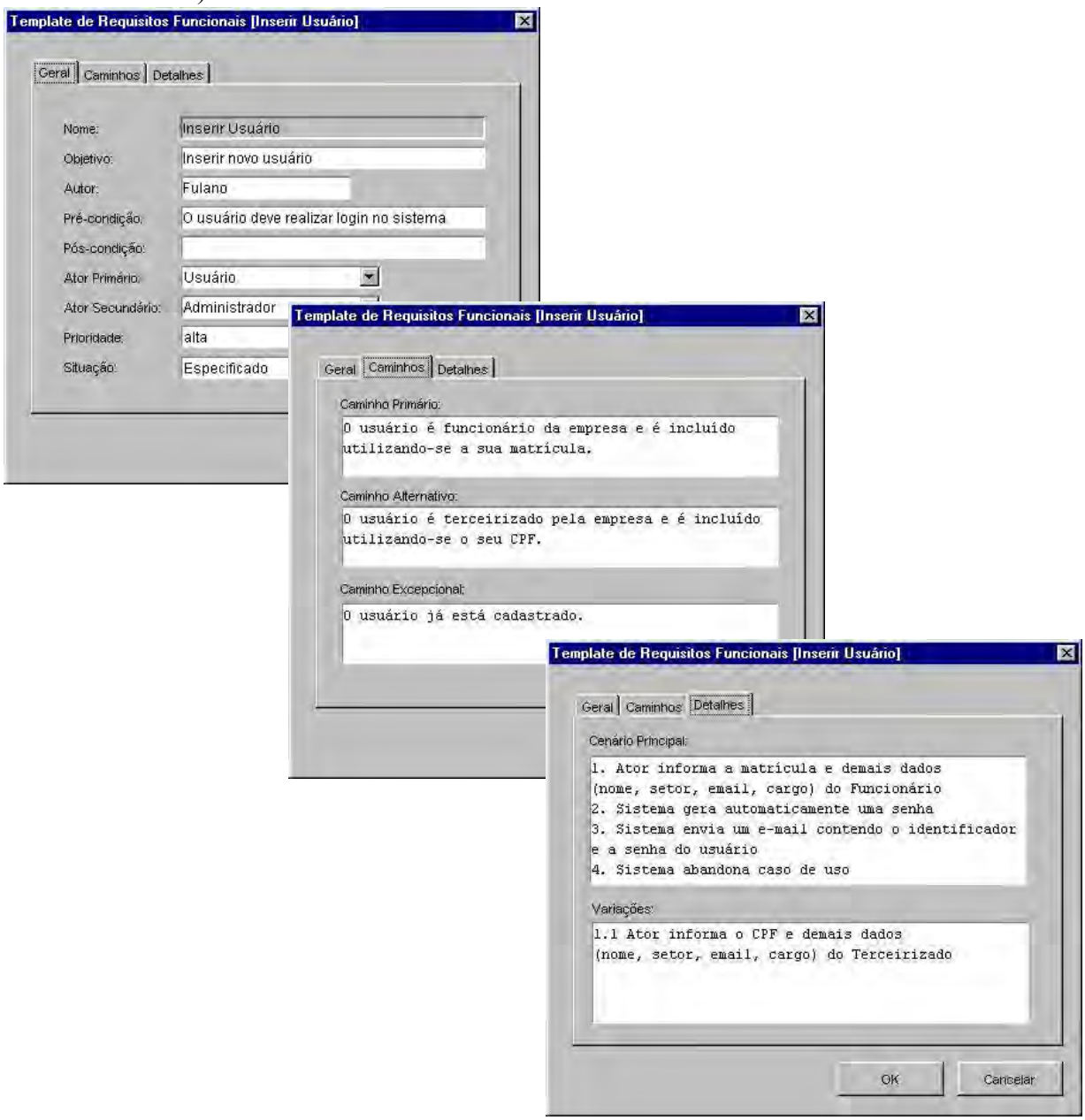

Figura 7.3: Estudo de Caso: template caso de uso incluir usuário

Outro exemplo é o ilustrado pela Figura 7.4 que exibe o *template* para o caso de uso *Executar Ação*.

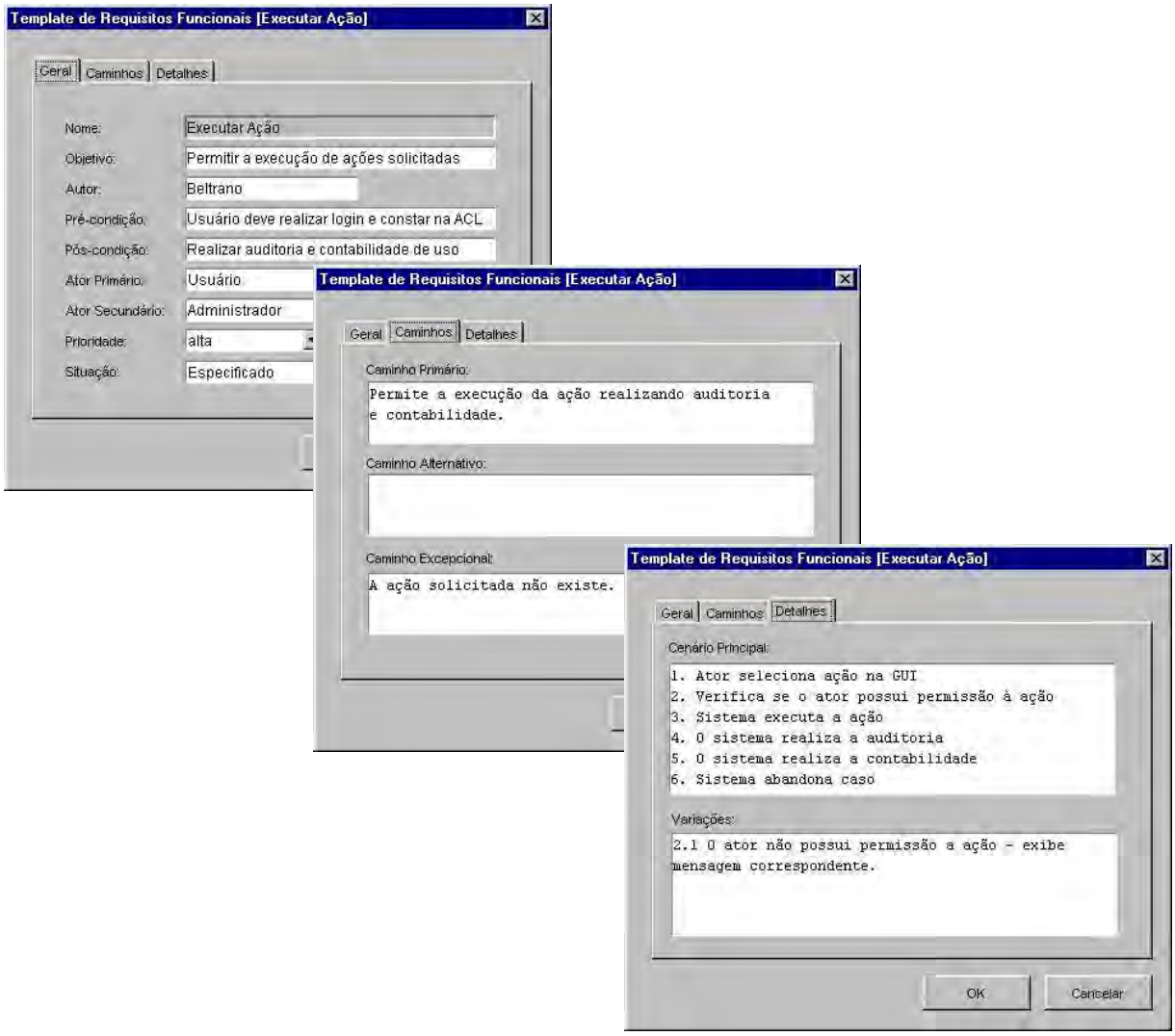

Figura 7.4: Estudo de Caso: template caso de uso executar ação

Ao término desse passo o analista deve: (i) chegar a um acordo com o cliente e o usuário sobre o que o sistema deve fazer; (ii) oferecer ao desenvolvedor um melhor entendimento dos requisitos do sistema; (iii) delimitar o escopo sistema através do diagrama de casos de uso e (iv) refinar o diagrama de casos de uso através da associação dos *templates*. Da mesma maneira foram elaborados os demais *templates*. Note-se que os RNFs correspondentes ainda não estão explicitados.

7.3.2 Identificação e Especificação dos RNFs

A identificação e especificação dos RNFs ocorre através de dois artefatos básicos: as *checklists* e o léxico.

No caso das *checklists*, é possível extrair RNFs compartilhados, que podem ser usados por mais de um caso de uso, os quais foram denominados nesse trabalho como RNFs globais. Desse modo, o primeiro passo desta etapa consiste em verificar a existência de RNFs através das *checklists*, conforme ilustra a Figura 7.5.

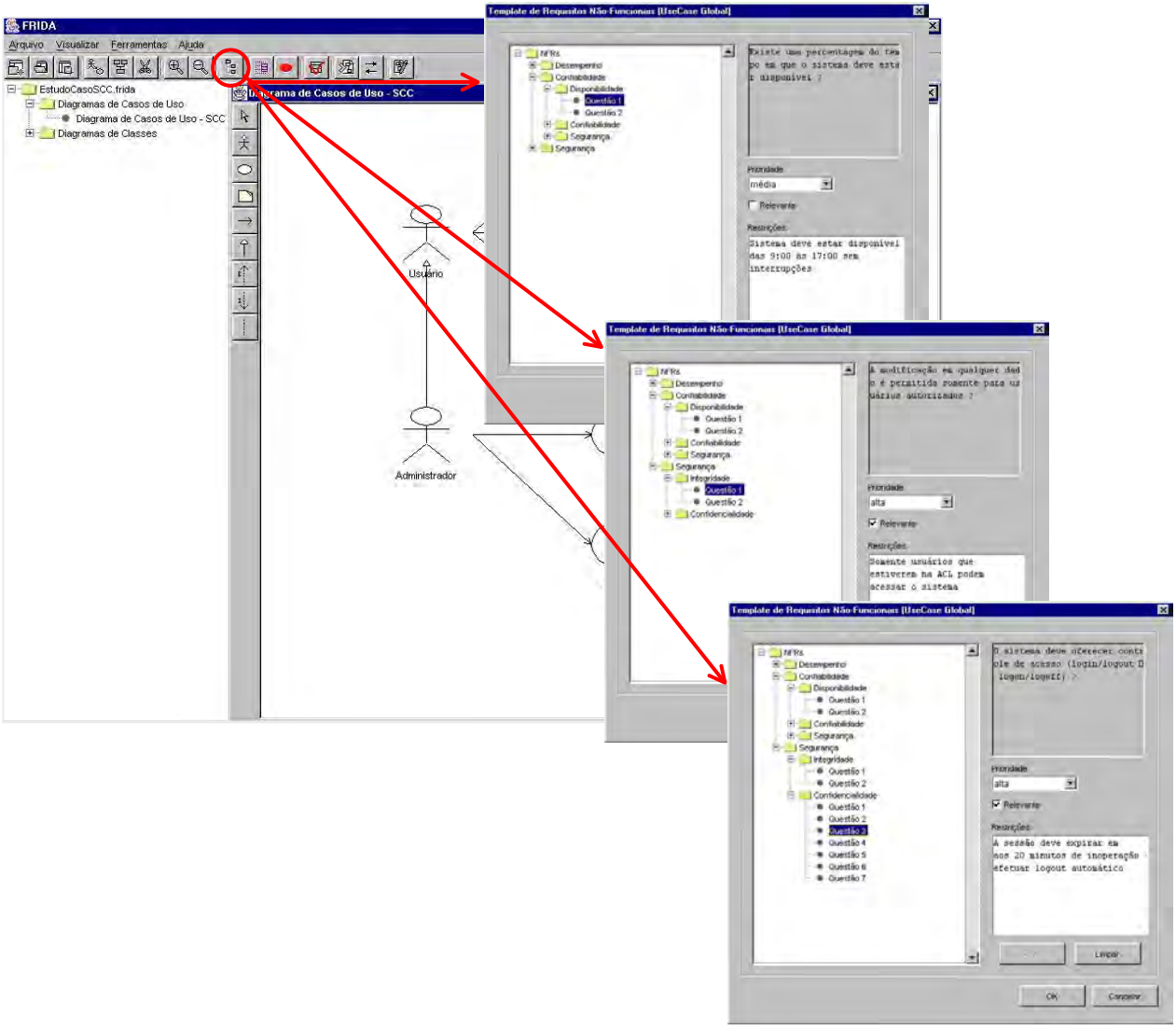

Figura 7.5: Estudo de Caso: checklist global

Sabe-se que os RNFs são por natureza mais abstratos que os RFs, e que geralmente são especificados de forma breve e vaga. Através do uso das *checklists*, o analista é mais enfático em suas especificações não funcionais, pois para um dado RNF ele deve especificar algumas propriedades que serão devidamente utilizadas em alguma fase posterior do processo de desenvolvimento.

A adoção das *checklists* pode ocorrer em dois momentos, o primeiro quando o RNF é global e o segundo quando seu escopo é parcial. Com base na avaliação do método FRIDA (ver Capítulo 8) verificou-se que a maioria dos participantes concentrou-se na elicitação apenas dos RNFs globais. Isso ocorre devido à ligação forte que existe entre

os RNFs parciais e os casos de uso, e também porque o nível de conhecimento e de detalhamento da modelagem do sistema deve ser muito alto.

Assim, percebe-se claramente a necessidade de um mecanismo que sinalize ao analista que algum requisito ficou "esquecido". Para cada *template* associado ao diagrama de casos de uso o LNFR – ***Léxico de Requisitos Não Funcionais*** – foi percorrido. Durante essa atividade cada palavra ou expressão contida no léxico, é combinada em uma tentativa de se encontrar algum RNF ainda não elicitado. Logo, ao realizar o processamento do léxico percebeu-se que alguns RNFs que encontravam-se vinculados a poucos ou a apenas um caso de uso não foram corretamente elicitados.

Conforme pode-se observar na Figura 7.6 o caso de uso *Executar Ação* encontra-se em destaque, pois após o processamento do léxico os seguintes RNFs foram identificados nas descrições: (i) auditoria: combinada no item *Cenário Principal* do *template*; (ii) contabilidade: identificada no item *Pós-Condição* do *template*.

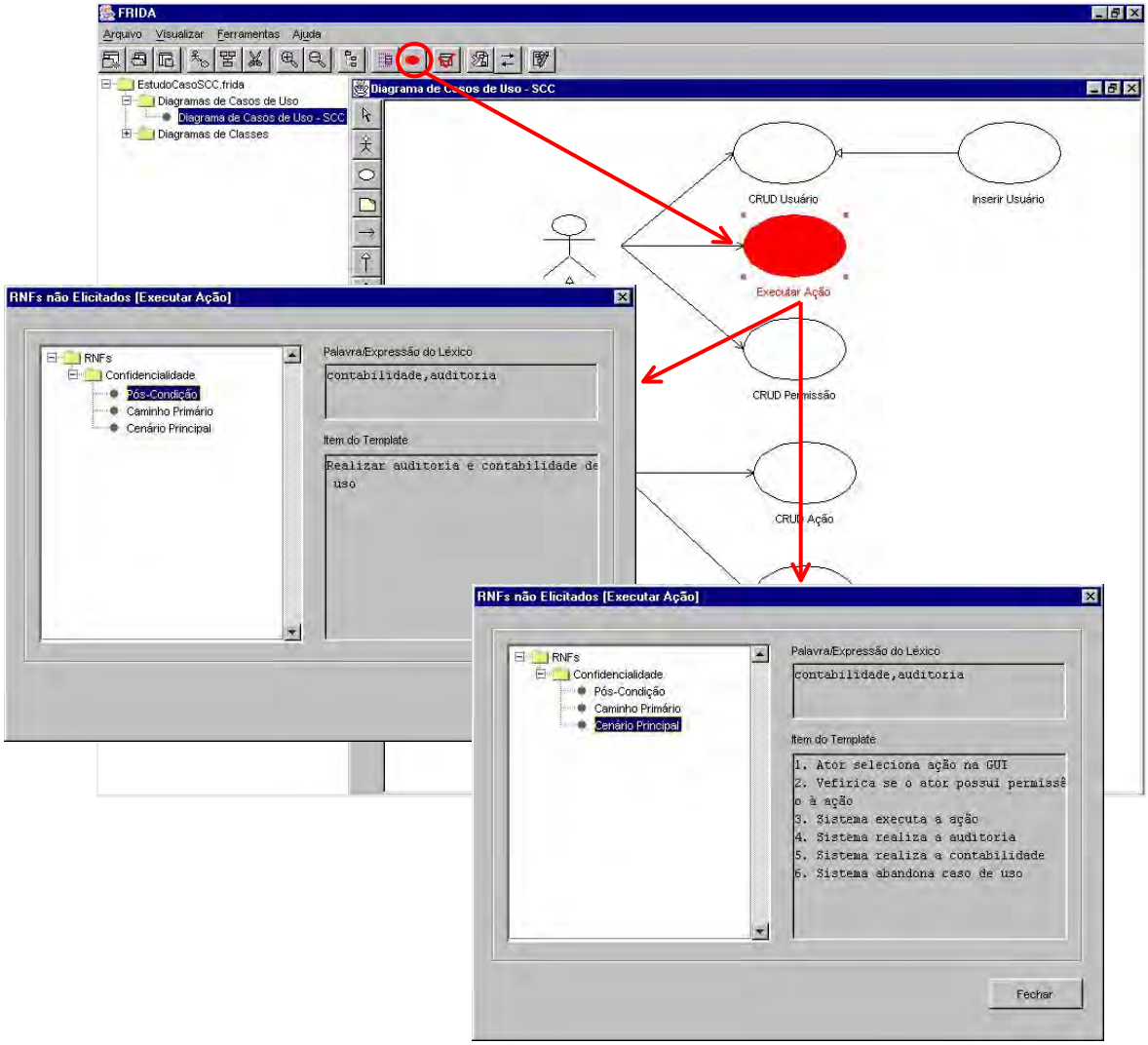

Figura 7.6: Estudo de Caso: ativação checklist global

Pode-se observar que esses RNFs demandam o uso de técnicas de segurança no acesso e uso de alguns módulos do software. A adoção dessas técnicas torna mais complexo o processo de elicitação dos RNFs. Porém, a não elicitação desses requisitos certamente implicaria em retrabalho (analisar e projetar novamente) de inúmeras partes do sistema em desenvolvimento.

Sabe-se que é possível elicitar e identificar os RNFs de auditoria e contabilidade para o caso de uso em destaque sem o auxílio do léxico. Neste caso, a contribuição que se deseja destacar é o fato de que o léxico reduz o tempo de especificação dos RNFs, além disso se o seu processamento for realizado no momento correto o projeto não corre o risco de chegar na fase de implementação sem que alguns requisitos não tenham sido adequadamente especificados.

Conforme já mencionado anteriormente, o processamento do léxico não é responsável pela ativação automática das *checklists*. Essa é uma tarefa que cabe ao analista, pois pode ocorrer que uma palavra identificada compreenda apenas um RNF candidato.

Outro item que merece destaque nessa fase é o fato de existirem interdependências entre os RNFs. O tratamento delas ocorre de forma simples, usando o critério de priorização, ou seja, quanto mais alta for a prioridade de um RNF maior será a sua influência no sistema resultante. Desse modo, o RNF com maior prioridade irá predominar sobre os demais. Essa resolução de conflitos entre RNFs é muito importante, porque essas interdependências podem ser vistas como a origem de importantes decisões de projeto. Além disso, a resolução prévia de conflitos pode influenciar positivamente o sistema em desenvolvimento.

7.3.3 Definindo a Visão Funcional

A associação dos RFs com o projeto ocorre em dois momentos distintos: o primeiro consiste na construção e definição do diagrama de classes, e o segundo compreende a vinculação de cada classe com a sua origem.

Para construir o diagrama de classes foi definido em primeiro lugar o modelo conceitual. Esse modelo concentra-se no foco no problema, ou seja, a funcionalidade (RFs), onde o comportamento pode ser observado como uma caixa preta, sem detalhes de implementação.

Para determinar o modelo conceitual realiza-se uma extração dos substantivos e adjetivos obtendo-se um conjunto de conceito e atributos candidatos (Tabela 7.1). Após, uma análise detalhada é realizada sobre esses elementos, confirmando a sua existência, ou excluindo-o da lista de conceitos/atributos candidatos.

Tabela 7.1: Estudo de Caso: conceitos e atributos do modelo conceitual

Palavra identificada	Categoria Candidata	Categoria Final
Usuário	Conceito	Conceito
Nome do usuário	Conceito/Atributo	Atributo
Sigla_Setor	Conceito/Atributo	Atributo
Email	Conceito/Atributo	Atributo
Cargo	Conceito/Atributo	Atributo
Administrador	Conceito	Conceito
Ação	Conceito	Conceito
Sistema	Conceito	Conceito
Sigla_Sistema	Conceito/Atributo	Atributo
Nome_Sistema	Conceito/Atributo	Atributo
Permissão	Conceito	Conceito
Funcionário	Conceito	Conceito
Matrícula	Conceito/Atributo	Atributo
Terceirizado	Conceito	Conceito
CPF	Conceito/Atributo	Atributo

Uma vez tendo sido confirmada a existência dos conceitos e dos atributos, bem como estabelecido as principais associações entre os conceitos, o modelo conceitual encontra-se finalizado. O próximo passo compreende a definição do diagrama de classes e de seus principais elementos (Figura 7.7). O foco desse diagrama está na representação da solução (sistema executável), pois é a representação mais próxima do código que um diagrama da UML oferece.

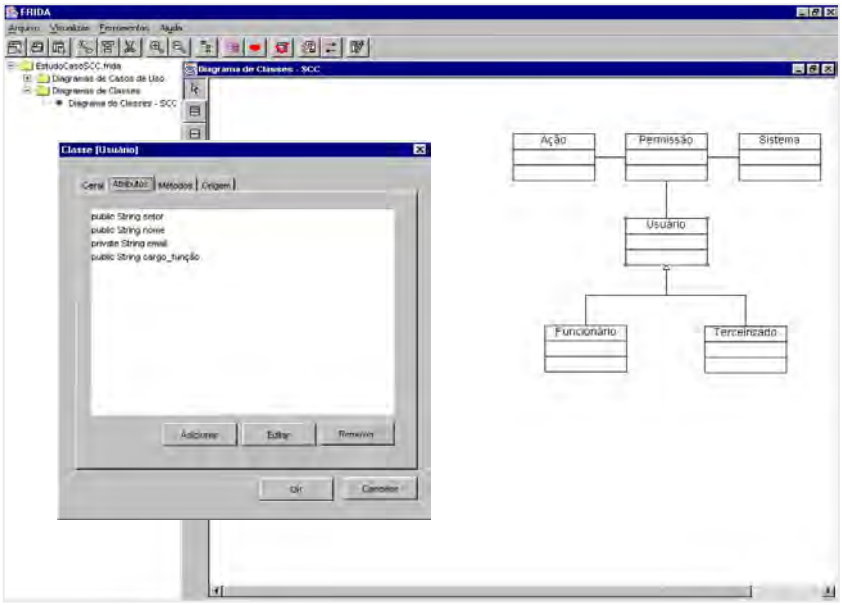

Figura 7.7: Estudo de Caso: diagrama de classes e atributos da classe

Em abordagens tradicionais, após a definição de cada classe e dos seus elementos, pode-se deduzir que o diagrama de classes está pronto. Porém, com o método FRIDA, para que uma classe esteja completamente descrita ela deve encontrar-se associada com suas origens, isto é, os requisitos que foram o ponto de partida para a sua definição. No estudo de caso (Figura 7.8), a partir dos casos de uso *CRUD*[30] *Ação* e *Executar Ação* foram extraídas as definições da classe *Ação*.

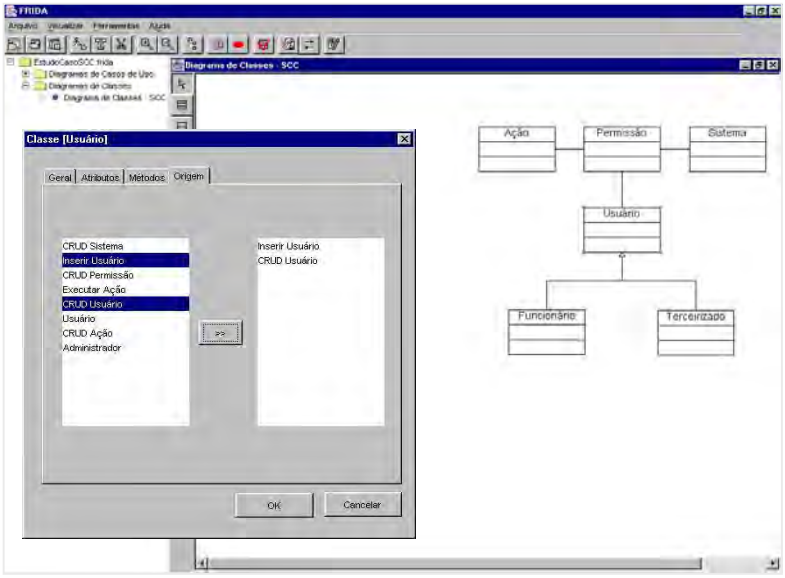

Figura 7.8: Estudo de Caso: diagrama de classes e origem da classe

[30] CRUD = Create, Retrieve, Update e Delete [REE 02].

Essa adaptação proposta pelo método FRIDA, tem por objetivo propiciar algum nível de rastreabilidade dos RFs de forma a auxiliar na evolução dos mesmos. Essa rastreabilidade preocupa-se com a fonte de conhecimento que gerou a necessidade do RF. Um segundo objetivo dessa adaptação é facilitar a integração dos RNFs ao diagrama de classes.

No método FRIDA, o diagrama de classes compreende o artefato usado para representar os requisitos funcionais na fase de projeto. Posteriormente, esse artefato contemplará todos os requisitos do sistema, pois nele serão agregados os aspectos, os quais modelam os RNFs.

7.3.4 Definindo a Visão Não Funcional

Nessa fase, os RNFs descritos através das *checklists* são traduzidos para aspectos. Para cada ocorrência encontrada na *checklist*, isto é, para cada RNF elicitado um aspecto correspondente é gerado.

A tradução de RNFs para aspectos ocorre de forma estereotipada. Para esta finalidade, o diagrama de classes sofreu dois tipos de extensão: (i) identificação textual do aspecto por meio do estereótipo <<*aspect*>>, e (ii) uso da cor amarela para destacar o aspectos dos demais elementos do diagrama (ver Anexo 6).

A Figura 7.9 apresenta os aspectos *Disponibilidade*, *Integridade* e *Confidencialidade*, relativos ao estudo de caso em questão. Esse aspectos foram elicitados previamente, conforme descreve a seção 7.3.2.

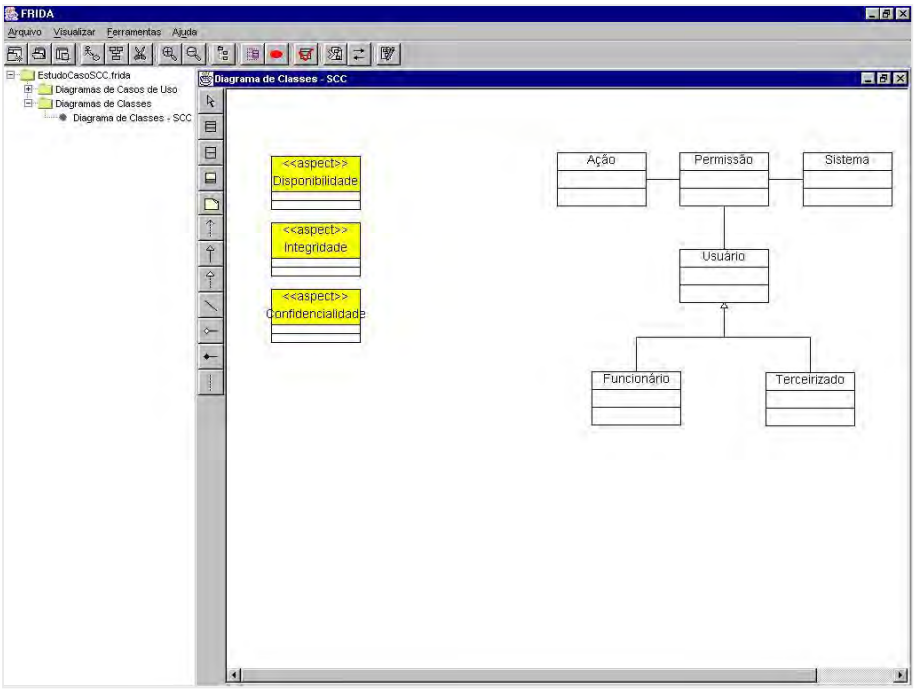

Figura 7.9: Estudo de Caso: representando os aspectos visualmente

Cada aspecto possui associado um conjunto de propriedades que é usado para descrevê-lo. Na Figura 7.10, podem ser visualizadas algumas informações pertinentes a cada aspecto que foram extraídas dos demais artefatos. Por exemplo, é possível identificar os casos de uso com os quais um aspecto encontra-se relacionado, além das restrições a ele associadas.

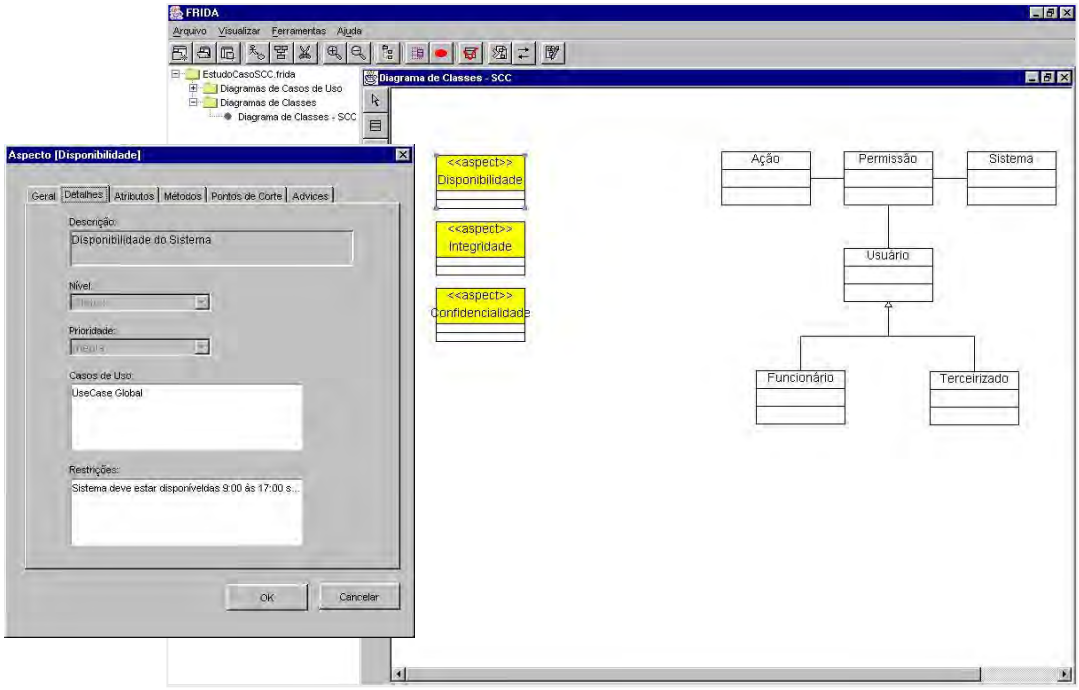

Figura 7.10: Estudo de Caso: detalhando os aspectos

Com as informações descritas na aba Detalhes é possível propiciar um certo nível de rastreabilidade, tanto com relação aos artefatos não funcionais (*checklists*) quanto com os artefatos funcionais (casos de uso). Essa rastreabilidade é definida em termos de algumas informações introduzidas no método FRIDA (*template* de aspectos), que possibilita integrar a visão não funcional com a funcional.

É importante ressaltar que existe a obrigatoriedade de se criar os aspectos para satisfazer os RNFs elicitados. Essa decisão é de responsabilidade do método FRIDA, visto que ele analisa os artefatos associados com a modelagem do projeto e decide pela geração ou não dos aspectos.

Após finalizar a construção de ambas as visões (funcional e não funcional) realiza-se a sua integração.

7.3.5 Combinando as Visões Funcional e Não Funcional

O processo de combinação baseia-se no uso dos vínculos estabelecidos entre os elementos pertencentes à visão funcional (diagramas de casos de uso e de classes) com os da visão não funcional (*checklists* e aspectos).

Através do uso destes vínculos, o processo de integração das visões é facilitado, ficando centrado na busca da convergência de um caso de uso que encontra-se relacionado tanto com elementos do projeto (visão funcional), quanto com as *checklists*.

A integração das visões será realizada através da incorporação de relacionamentos estereotipados e explícitos entre os aspectos e as classes. Observa-se que esses relacionamentos serão gerados apenas para aqueles aspectos cujo escopo seja parcial, o escopo global não é descrito para aumentar a compreensão do diagrama.

No estudo de caso deste capítulo alguns aspectos globais foram identificados: disponibilidade, integridade e confidencialidade. Como esses aspectos foram considerados globais nenhum relacionamento será estabelecido.

Com relação ao aspecto confidencialidade pode-se afirmar que ele pode sofrer um refinamento e gerar outros dois aspectos contabilidade e auditoria. Uma vez que esses aspectos foram vinculados apenas ao caso de uso *Executar Ação* eles serão associados com as classes originadas a partir desse caso de uso, conforme esquematiza a Figura 7.11.

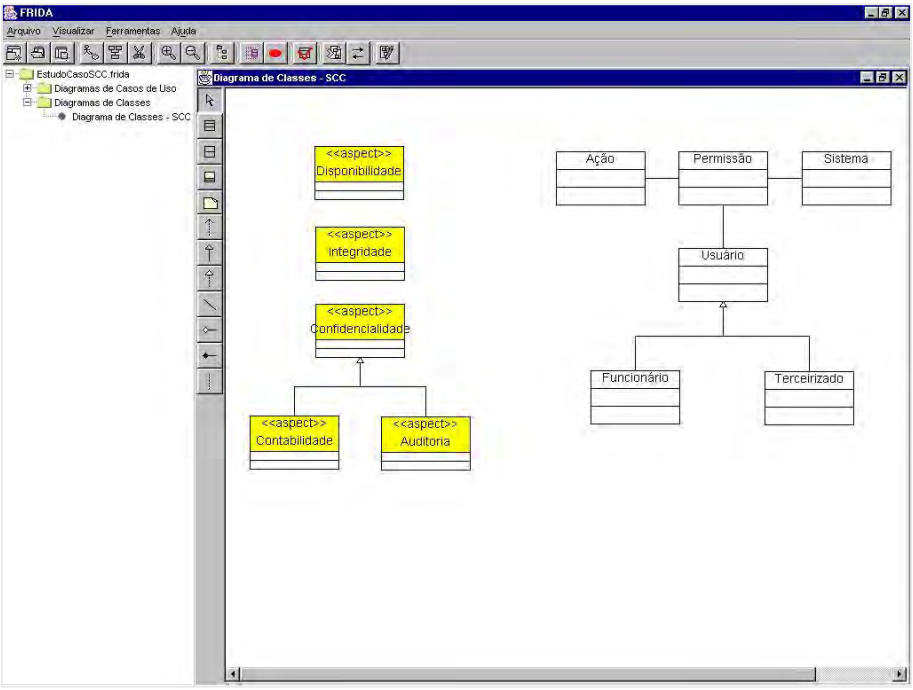

Figura 7.11: Estudo de Caso: combinando aspectos e classes

7.3.6 Código Gerado

A geração do código é realizada percorrendo-se todos os aspectos e classes, bem como examinando-se suas descrições e propriedades associadas.

Com relação aos aspectos, um peculiaridade pode ser destacada: cada RNF genérico é considerado uma raiz na hierarquia de herança entre os aspectos. Tendo gerado a raiz da árvore de herança dos RNFs, cada RNF sofre sucessivas decomposições até chegar nas folhas (RNFs específicos)

Assim, para o estudo de caso temos como raiz três RNFs: Disponibilidade, Integridade e Confidencialidade. E também uma sub-árvore composta pelas folhas contabilidade e auditoria (aspectos parciais).

Prosseguindo com a análise anterior, o código gerado para os RNFs, descritos e implementados através de aspectos, é esquematizado na Figura 7.12.

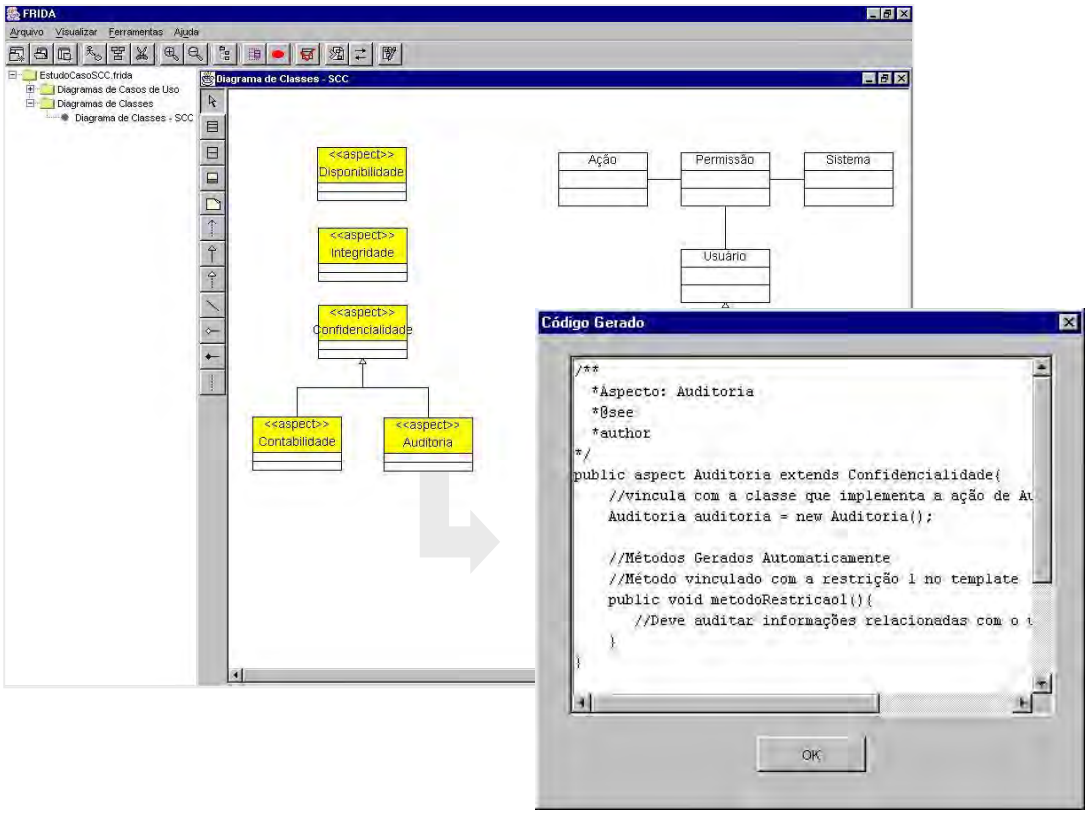

Figura 7.12: Estudo de Caso: hierarquia dos aspectos gerados

7.4 Conclusões

Os RNFs abordam importantes características relacionadas à qualidade do software. A não elicitação de RNFs pode resultar em: (i) software inconsistentes e de baixa qualidade; (ii) insatisfação de clientes e desenvolvedores; (iii) tempo e custo de desenvolvimento dimensionados imprecisamente. Com o método FRIDA pretende-se propiciar o desenvolvimento de sistemas computacionais de forma que os problemas destacados previamente possam ser contornados.

Pode-se perceber no desenvolver desse estudo de caso que a aplicação do método FRIDA resultou em uma melhoria da qualidade final do produto, assim como as técnicas e artefatos adotados possibilitaram o gerenciamento de algumas atividades e a realização da modelagem em um tempo menor, uma vez que a ferramenta associada ao método (descrita no Anexo 8), além de guiar o desenvolvedor na descoberta de RNFs através de *checklists*, também automatiza parte do processo. Além disso, com o mapeamento dos RNFs em aspectos pode-se perceber que a implementação torna-se mais clara, pois cada RNF é modelado como uma unidade individual, e também porque seu código não encontra-se disperso e/ou entrelaçado.

8 RESULTADOS

8.1 Introdução

De modo a validar todas as etapas e artefatos definidos, bem como o protótipo construído foi realizada uma avaliação com profissionais da área, representados por um grupo de 8 funcionários voluntários da Cia. De Processamento de Dados do Rio Grande do Sul – PROCERGS, situada em Porto Alegre.

O processo de avaliação ocorreu em dois momentos: no primeiro foram respondidas questões relacionadas com a área de Engenharia de Software e após, foram conduzidos experimentos com o protótipo da ferramenta FRIDA, objetivando uma validação dos principais artefatos adotados pelo método.

A avaliação ocorreu na empresa, durante quatro horas, sendo que em média cada participante levou trinta minutos para responder seu respectivo questionário. As atividades foram realizadas na presença da autora deste trabalho, visto que o tempo determinado para os experimentos era de certa forma pequeno.

A escolha dos profissionais que realizaram a pesquisa ocorreu aleatoriamente para que a avaliação não fosse invalidada. Conforme pode-se observar na Figura 8.1 optou-se por selecionar pessoas de diversas idades, mas que possuíssem alguma experiência na área.

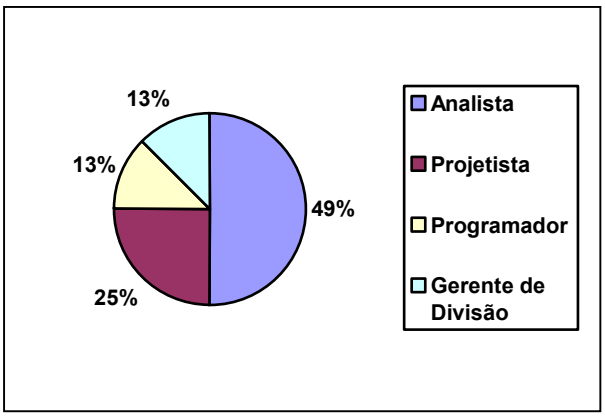

Figura 8.1: Função dos Participantes da Avaliação

As faixas etárias podem ser indicativas do grau de experiência profissional, bem como de diferenças de formação acadêmica.

A função que cada avaliador exerce também é importante, pois representa exatamente o perfil das pessoas que trabalham para as empresas de tecnologia. A Figura 8.2 ilustra a distribuição das funções de cada participante do processo de validação do método FRIDA.

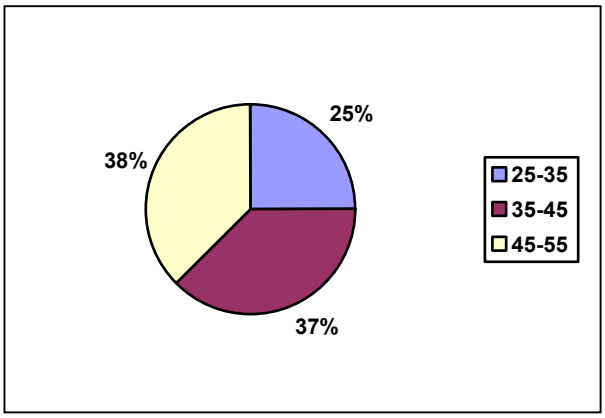

Figura 8.2: Idade dos Participantes da Avaliação

O processo de avaliação ocorreu em dois momentos, no primeiro foram respondidas questões relacionadas com a área de Engenharia de Software e após, iniciou-se uma validação dos principais artefatos adotados pelo método.

8.2 Avaliação Inicial

O objetivo dessa avaliação inicial consiste em demonstrar qual o nível de conhecimento dos profissionais participantes com relação a conceitos e técnicas da área de Engenharia de Software, os quais foram empregados no método FRIDA.

A primeira questão levantada foi o nível de conhecimento do participante na área de análise/projeto de sistemas. As respostas obtidas podem ser observadas na Figura 8.3.

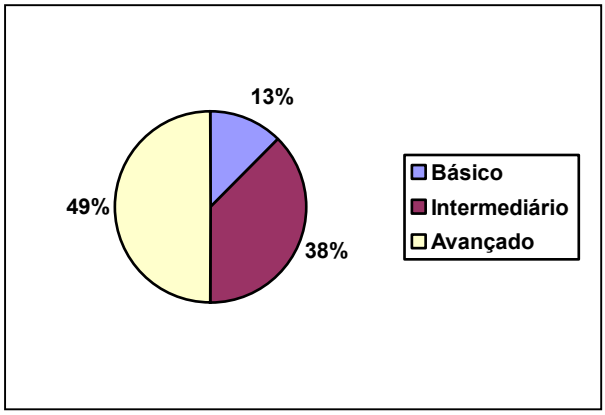

Figura 8.3: Nível de Conhecimento dos Participantes

Como o foco deste trabalho são os RNFs resolveu-se perguntar aos participantes se eles conheciam requisito não funcionais e quais eram os requisitos, dessa categoria, que eram mais comuns em seu ambiente de trabalho.

A maioria dos avaliadores não conhecia esta classificação para os requisitos, logo foi necessário esclarecer quais eram esses requisitos para que a avaliação pudesse prosseguir. A Figura 8.4 apresenta a distribuição dos RNFs mais lembrados pelos participantes durante a avaliação.

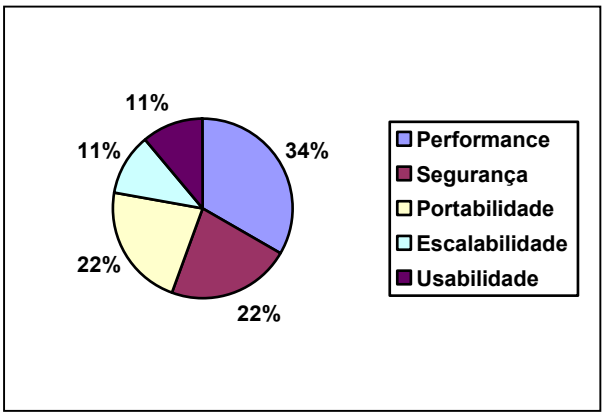

Figura 8.4: RNFs Identificados pelos Participantes

Em virtude do método proposto utilizar-se da UML para a definição de alguns artefatos, os participantes foram indagados a respeito de seu nível de conhecimento sobre esta linguagem de modelagem. As respostas informadas podem ser examinadas na Figura 8.5.

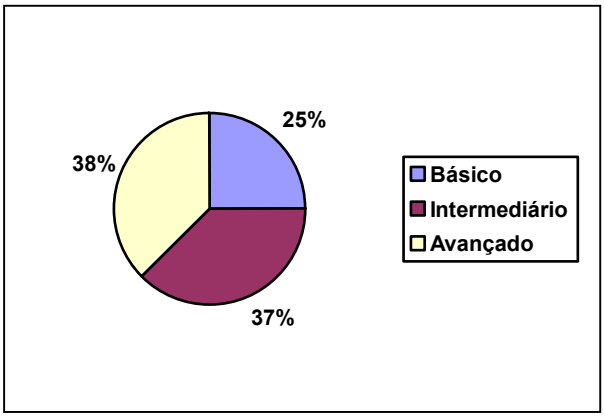

Figura 8.5: Nível de Conhecimento da UML pelos Participantes

Ainda com relação à UML, foi questionado se os participantes já haviam utilizado estereótipos. Muitos não sabiam o que eram os estereótipos, após uma breve explicação eles perceberam que usavam no seu dia a dia, mas que não conheciam com esta terminologia.

Para proporcionar um maior entendimento do método FRIDA foi perguntado o quanto os participantes sabiam sobre AOSD. Como era de se esperar o conhecimento sobre este item era muito pouco (Figura 8.6). Assim, resolveu-se expor brevemente o modelo de orientação a aspectos e suas diferenças com relação ao modelo OO.

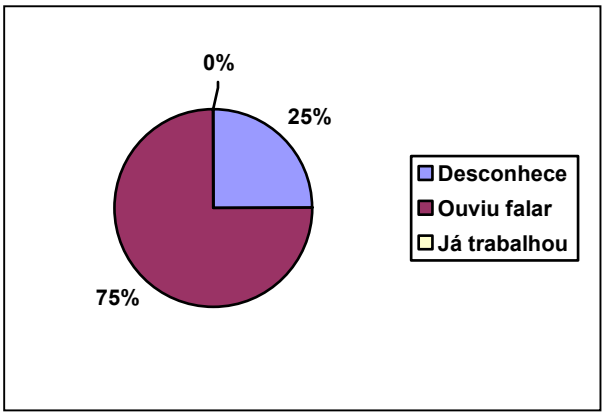

Figura 8.6: Nível de Conhecimento dos Participantes sobre AOSD

Após o término dessa etapa o processo de avaliação iniciou com uma análise do protótipo e de seus artefatos.

8.3 Avaliação do Método FRIDA

Para observar o comportamento de profissionais da área no uso do protótipo definido, bem como para sistematizar a avaliação do método, foi aplicado um questionário organizado em seções, contendo perguntas relativas a cada passo do método FRIDA, conforme apresentam as próximas subseções.

8.3.1 PASSO 1 – IDENTIFICAÇÃO DOS REQUISITOS

O objetivo da primeira pergunta formulada para esta fase era identificar a técnica mais utilizada para a elicitação/modelagem de requisitos. As respostas obtidas mostram claramente que o diagrama de casos de uso é a técnica mais empregada (gráfico à esquerda da Figura 8.7), e a adoção de documentações adicionais é um elemento essencial (75% de índice de adoção), conforme esquematiza o gráfico à direita na Figura 8.7. Por documentações adicionais entende-se arquivos em formato texto, imagens, telas prototipadas, etc.

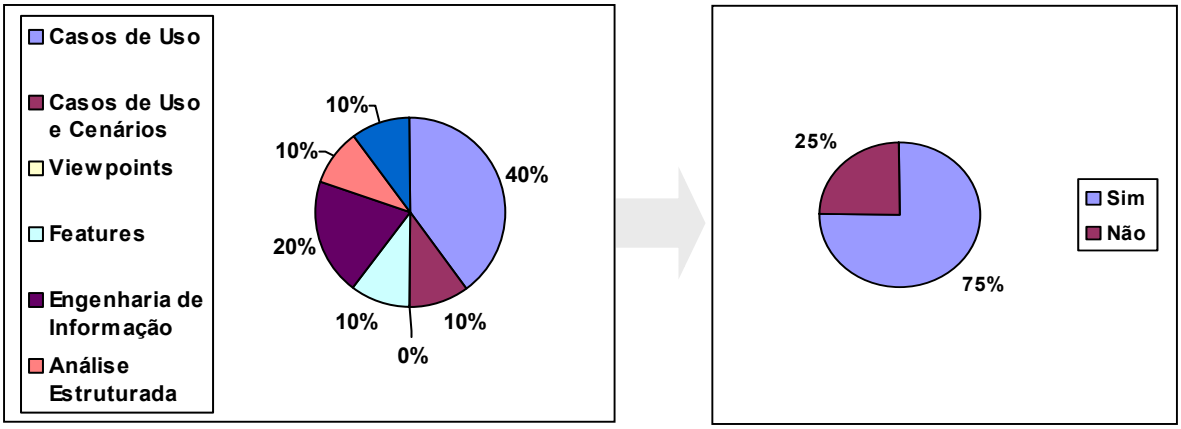

Figura 8.7: Técnicas Adotada para Elicitar/Modelar Requisitos

Com base nas respostas do questionário e nos resultados ilustrados na Figura 8.7 é possível confirmar a necessidade de uma documentação extra para os elementos de um diagrama de casos de uso. No caso do método FRIDA , essa documentação é feita através do uso de *templates*. Ainda nesse sentido, uma outra questão levantada para os participantes seria a forma proposta de organização do *template*. Neste caso, todos julgaram adequada a divisão das informações em seções. Porém alguns sugeriram a inclusão de outros itens, com o intuito de que o *template* possa ser utilizados para diversos fins, como demonstra a Figura 8.8.

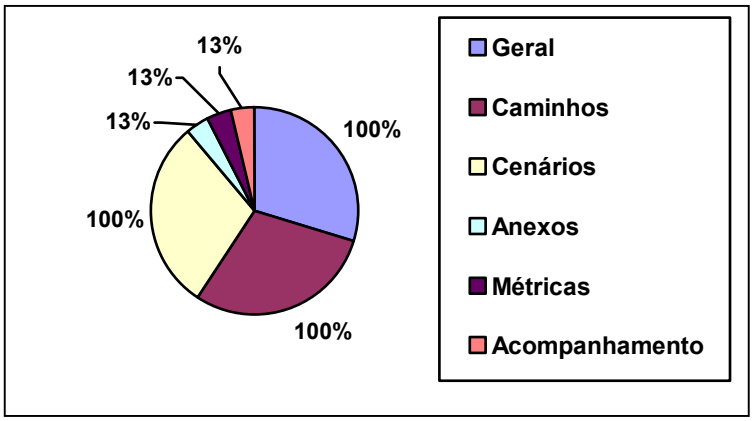

Figura 8.8: Divisão das Informações no *Template* do método FRIDA

Após finalizar a avaliação dos artefatos envolvidos com o primeiro passo do método FRIDA a avaliação prosseguiu com os passos relacionados com a elicitação dos RNFs.

8.3.2 PASSO 2 e 3 REFINANDO RNFs (CHECKLISTS e LÉXICO)

Para introduzir os artefatos utilizados por FRIDA no processo de refinamento dos RNFs foi solicitado aos participantes que eles indicassem quais artefatos costumam usar para elicitar e descrever RNFs. As respostas podem ser conferidas na Figura 8.9.

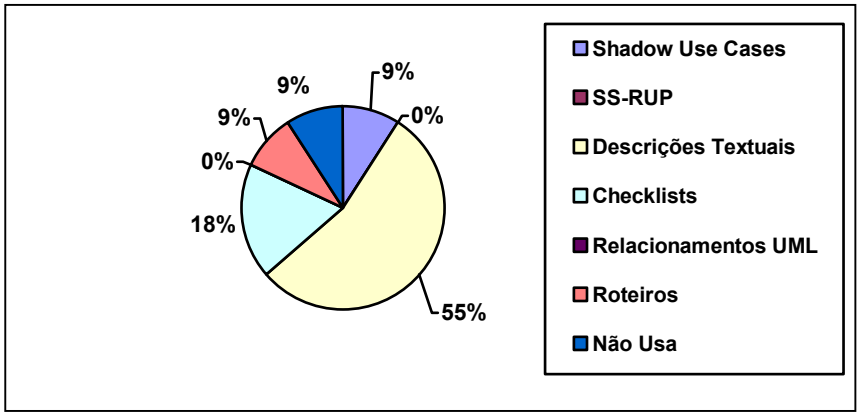

Figura 8.9: Artefatos usados na Elicitação e Descrição de RNFs

Embora o mecanismo mais usado para resolver os problemas de elicitação relativos a RNFs tenha sido o uso de descrições textuais, todos os participantes da avaliação concordaram que o uso de *checklists* pode lembrar alguns pontos que devem ser observados ou indagados ao cliente.

Nesse contexto, a *checklist* proposta pelo método FRIDA foi aplicada ao estudo de caso e os participantes identificaram que, mesmo possuindo experiência na área, alguns RNFs passariam desapercebidos se a *checklist* não existisse. Com essa avaliação pode-se constatar explicitamente que o artefato (estrutura e propriedades) proposto no método FRIDA é válido e pode auxiliar na identificação de RNFs.

Além dos RNFs de segurança, desempenho e confiabilidade foi indagado aos participantes da avaliação quais outros RNFs requisitos não funcionais também deveriam fazer parte da *checklist*. As respostas encontram-se ilustradas na Figura 8.10.

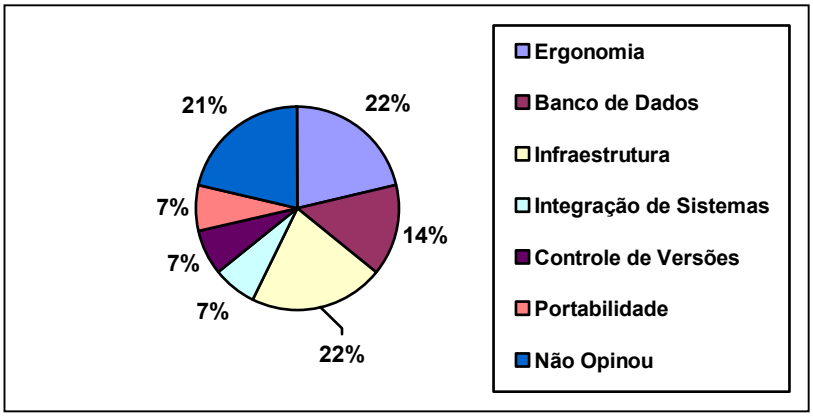

Figura 8.10: RNFs Sugeridos pelos Participantes

Assim como as *checklists*, o léxico também obteve uma boa avaliação por parte dos participantes. Todos concordaram que a definição de um universo de informações relacionado a um determinado domínio pode auxiliar na elicitação dos RNFs. Informalmente, alguns avaliadores declararam que esta técnica poderia se estender para a elicitação de requisitos funcionais.

O processamento do léxico sobre as descrições textuais (*template*) associadas ao diagrama de casos de uso foi outra característica do método FRIDA aprovada pelos avaliadores. Na verdade, todos acreditam que esse mecanismo pode incentivar uma melhor especificação dos requisitos por parte dos envolvidos, visto que a partir dessa documentação a extração de requisitos pode ser semi-automatizada.

8.3.3 PASSO 5 e 6 – DEFINIÇÃO DO MODELO DE ANÁLISE E PROJETO

O mecanismo de associação das classes com o(s) caso(s) de uso e/ou atores que deram origem à classe obteve aprovação imediata, pois os participantes acreditam que isso pode interligar a fase de especificação de requisitos com a fase de projeto. Além disso, segundo um dos avaliadores o que costuma ocorrer é que: “o analista vê seu trabalho de certa forma perdido, porque não reconhece claramente onde as suas especificações se refletem nos elementos da fase de projeto”.

Outro item bem recebido pelos participantes da avaliação foi o esquema de propagação das mudanças especificado para o protótipo FRIDA. Os participantes acreditam que é importante perceber nitidamente onde serão refletidas as alterações realizadas nas especificações funcionais, uma vez que elas podem influenciar na tomada de decisão.

Acredita-se que técnicas desse tipo deveriam constar em toda e qualquer fase do processo de desenvolvimento, pois demonstram claramente a ligação entre os elementos de cada fase, bem como possibilitam a rastreabilidade das informações.

Com relação a rastreabilidade foi questionado aos participantes qual o tipo de rastreamento de informações a ferramenta por eles utilizada oferecia. O objetivo dessa indagação consiste na obtenção de respaldo para afirmar que a idéia apresentada pelo método FRIDA oferece uma real contribuição nessa área. A avaliação demonstrou que as ferramentas utilizadas, pelos avaliadores, não oferecem nenhum tipo de suporte para a ratreabilidade dos elementos entre as fases de especificação de requisitos e projeto.

8.3.4 PASSOS 7, 8, 9 e 10 – EXTRAÇÃO DE ASPECTOS, VINCULAÇÃO e GERAÇÃO DE CÓDIGO

Todos os avaliadores acreditam que é importante demonstrar visualmente os aspectos no diagrama de classes, visto que são elementos essenciais para o desenvolvimento da aplicação. Alguns participantes justificaram que a visualização dos aspectos no diagrama de classes pode ser uma forma de se acostumar com a nomenclatura e terminologia usada na área de AOSD.

O uso de estereótipos para a representação de aspectos gerou uma certa controvérsia, pois muitos avaliadores nunca haviam usado estereótipos. De modo geral, a adoção de estereótipos para modelar aspectos foi bem aceita.

Da mesma forma que a avaliação do uso de estereótipos textuais, foi realizada uma consulta informal do estereótipo baseado em cores, apresentado por este trabalho. Basicamente, a maioria dos envolvidos aprovou a idéia pois assim é possível "identificar claramente quais são os aspectos em um grande diagrama de classes ".

Quanto a geração de código o questionário concentrou-se na criação de *wizards* para geração de código relacionado com aspectos. Todos os avaliadores confirmam que o uso de *wizards* e que a geração de código são essenciais, principalmente quando o assunto é AOSD – uma tecnologia pouco disseminada.

8.4 Conclusões

Após terem usado os diversos artefatos do método FRIDA, os participantes da avaliação julgaram a aplicabilidade das idéias embutidas em FRIDA. Todos indicaram que o método FRIDA, apresentado de forma prática através do protótipo, contribui para o desenvolvimento de aplicações, inclusive comerciais.

Com a avaliação realizada foi possível ter uma idéia real de como o método poderia ser aplicado por profissionais da área de tecnologia. Pode-se perceber que a maioria das idéias foram bem aceitas pelos avaliadores. Já com relação ao protótipo pode-se constatar que os participantes da avaliação indicaram várias melhorias, todas elas foram devidamente documentadas e encontram-se transcritas nesse trabalho de duas formas:

(i) atualizações/alterações: foram realizadas algumas atualizações/alterações no protótipo para que o mesmo ficasse mais adequado às exigências dos avaliadores;

(ii) trabalhos futuros: como o objetivo deste trabalho não é a construção do protótipo, alguns itens sugeridos foram incluídos na seção de trabalhos futuros.

Pode-se observar que alguns conceitos não encontram-se disseminados no mercado de trabalho, e que as diferenças de terminologias podem afetar a comunicação dos envolvidos. Em resumo, a experiência realizada mostrou que, embora fundamentado em idéias não muito difundidas, o trabalho pode ser aplicado no desenvolvimento de aplicações comerciais.

9 CONCLUSÕES

A qualidade dos artefatos gerados nas fases iniciais do processo de desenvolvimento de software, é essencial para o êxito de um sistema. Quando o foco da elicitação e da modelagem leva em consideração, além da funcionalidade, atributos de qualidade (RNFs) as chances de sucesso do sistema que está sendo construído aumentam ainda mais.

Conrow, em [COW 97], apresenta os resultados de uma pesquisa, onde foram levantados os principais problemas no desenvolvimento de 8.380 projetos de software comerciais: 61% dos projetos apresentavam os requisitos inicialmente esperados; (ii) a maior parte dos erros de um sistema (64%) encontrava-se associada às fases de análise de requisitos e projeto; (iii) grande parte dos erros era descoberto somente durante as etapas mais avançadas do processo (por exemplo, codificação e testes).

Desse modo, percebe-se claramente que o uso de um método sistemático, que permita a integração dos RNFs com os requisitos funcionais, deve ser adotado no desenvolvimento de sistemas computacionais visando reduzir as taxas de erros apresentadas anteriormente.

Diversos problemas surgem durante a fase de especificação dos requisitos, pois existem diversas dificuldades encontradas pelo desenvolvedor: (i) incertezas e erros podem ocorrer em função da má comunicação ou falta de especialistas nas informações do **domínio do problema** que está sendo analisado; (ii) desenvolvedores pertencentes a uma mesma equipe podem apresentar vocabulário conflitante e/ou ambíguo; (iii) má interpretação dos conceitos/ funcionalidades e/ou atributos de qualidade, em função do uso de linguagem natural. Nesse contexto, são propostas no método FRIDA, para a especificação funcional, o uso agregado de diagramas de casos de uso e *templates*. Já para minimizar os problemas relativos aos RNFs apresenta-se o uso combinado de checklists, léxico e conflitos. Estes três mecanismos, no contexto deste trabalho, foram focalizados na determinação de um vocabulário comum ao domínio de aplicações tolerantes a falhas, embora sua aplicabilidade não seja restrita a esta área.

A técnica de rastreabilidade proposta auxilia na verificação das características de um sistema com relação ao seu projeto, ajuda na identificação das unidades afetadas por mudanças na especificação de requisitos, e assim propicia um certo nível no gerenciamento das mudanças.

A geração de aspectos e a definição de alguns de seus elementos fundamentais auxiliam a identificação, separação, representação e composição das propriedades ortogonais (RNFs).

O uso de estereótipos baseados em descrições textuais, e principalmente em cores, permite representar relações simbólicas, chamar e direcionar a atenção do desenvolvedor e também enfatizar a importância de alguns elementos estruturais.

As principais contribuições deste trabalho envolvem:

(i) a possibilidade da melhoria de qualidade do software, visto que os domínios estabelecidos para neste estudo (desempenho, segurança e confiabilidade) são tratados de forma breve e na maioria das vezes apenas durante a fase de implementação. Com o modelo pretende-se que esses RNFS sejam examinados desde as fases iniciais do desenvolvimento diminuindo as taxas de inconsistência, erro e ambigüidade através dos artefatos que compõe o projeto;

(ii) a visão integrada de técnicas de análise orientada a aspectos e programação orientada a aspectos, de modo a possibilitar a automatização de processos de desenvolvimento de software, nos quais requisitos funcionais sejam tão importantes quanto os não funcionais;

(iii) o desenvolvimento e a avaliação de uma ferramenta, mostrando ser exeqüível a proposta de um modelo integrado.

10 TRABALHOS FUTUROS

Considerando que a ferramenta construída para automatizar o método de desenvolvimento AOSD se atém às questões relacionadas com os passos de FRIDA, pode-se concluir que outras técnicas podem vir a ser adotadas em algumas das diversas fases do método e que futuras implementações da ferramenta possam vir a permitir a melhoria de algumas funcionalidades do protótipo, bem como a inclusão de funcionalidades ainda não tratadas.

Com relação ao método FRIDA algumas das melhorias sugeridas para a complementação das funcionalidades já existentes, incluem:

(i) a resolução de conflitos utilizando-se ambas as heurísticas propostas – ordemPrioridade e dependenteContexto, visto que hoje apenas a primeira encontra-se implementada;

(ii) criar um processador de textos mais completo, que possibilite processar as descrições textuais em função de termos e expressões mais complexas.

No que diz respeito à inclusão de funcionalidades ainda não tratadas sugere-se:

(i) criação de um mecanismo para a definição e a inclusão de novos RNFs, como por exemplo, para os contextos de sistemas distribuídos e computação móvel. Observe-se que neste item deve-se criar mecanismos para a inclusão de RNFs nas *checklists*, no léxico, e na matriz de conflitos;

(ii) a possibilidade de customização do léxico, ou seja, permitir, de forma visual, a inclusão de novas palavras ou expressões, tanto para novos contextos quanto para os contextos já existentes;

(iii) determinar quais módulos devem ser construídos em primeiro lugar, através da análise da prioridade dos casos de uso e das *checklists* indicando, respectivamente, a ordem de implementação das classes e dos aspectos;

(iv) permitir um acompanhamento visual dos pontos em que os aspectos atuam - usando definições estereotipadas no diagrama de seqüências;

(v) criar um gerador que possua recursos mais elaborados, possibilitando uma geração mais completa e consistente, de acordo com a especificação dos requisitos.

Além das sugestões anteriores, durante a aplicação do estudo de caso, outras sugestões foram levantadas. Entre elas destacam-se:

(i) incluir questões de consistência a partir de regras da UML;

(ii) possibilitar a inclusão de regras para a verificação da sintaxe e semântica dos aspectos;

(iii) criar um mecanismo de críticas que examine as definições dos aspectos e verifique se são de natureza conflitante com os RNFs indicados, como por exemplo, solicitar a ativação de um ponto de corte para a chamada e execução de todo e qualquer método conflita com o RNF desempenho.

REFERÊNCIAS

[ABR 2003] AOSD-BR: Lista de discussão dos grupos de AOSD no Brasil. Disponível em: <http://groups.yahoo.com/group/aosd-br/>. Acesso em: out. 2003.

[AKS 98] AKSIT, M.; TEKINERDOGAN, B. Solving the modeling problems of object-oriented languages by composing multiple aspects using composition filters. In: ASPECT-ORIENTED PROGRAMMING WORKSHOP, 1998, Berlin. **Proceedings...** Berlin: Springer-Verlag, 1998. p. 12-23.

[AKS 96] AKSIT, M. Composition and separation of concerns in the object-oriented model. **ACM Computing Surveys**, New York, v. 28, n. 4, 1996.

[AKS 94] AKSIT, M. et al. Specification Inheritance Anomalies and Real-Time Filters. In: EUROPEAN CONFERENCE FOR OBJECT-ORIENTED PROGRAMMING, ECOOP, 1994, Berlin. **Proceedings...** Berlin: Springer-Verlag, 1994. p. 386-407.

[AKS 92] AKSIT, M.; BERGMANS, L.; VURAL, S. An Object-Oriented Language-Database Integration Model: The Composition-Filters Approach. In: EUROPEAN CONFERENCE FOR OBJECT-ORIENTED PROGRAMMING, ECOOP, 1992, Berlin. **Proceedings...** Berlin: Springer-Verlag, 1992. p. 372-395.

[AKS 89] AKSIT, M. **On the Design of the Object-Oriented Language Sina**. 1989. Tese de Doutorado (Doutorado em Ciência da Computação) – Department of Computer Science, University of Twente, Netherlands. Disponível em: <http://trese.cs.utwente.nl/publications/paperinfo/aksit.phd.pi.top.htm>. Acesso em: out. 2001.

[AKS 88] AKSIT, M.; TRIPATHI, A. Data Abstraction Mechanisms in Sina/ST. **ACM SIGPLAN Noticies**, New York, v. 23, n. 11, p. 265-275, 1988.

[ALB 2002] ALBUQUERQUE, R.; RIBEIRO, B. **Segurança no Desenvolvimento de Software**. Rio de Janeiro: Campus, 2002.

[ALD 2003] ALDAWUD, O.; ELRAD, T.; BADER, A. A UML Profile for Aspect-Oriented Software Development. In: WORKSHOP ON ASPECT-ORIENTED MODELING WITH UML, AOM, 2003, Boston. **Proceedings...** Boston: [s.n.], 2003. Disponível em: < http://lglwww.epfl.ch/workshops/aosd2003/papers/AldawudAOSD_UML_Profile.pdf>. Acesso em: maio 2003.

[ALD 2001] ALDAWUD, O.; ELRAD, T.; BADER, A. A UML Profile for Aspect Oriented Modeling. In: WORKSHOP ON ADVANCED SEPARATION OF CONCERNS IN OBJECT-ORIENTED SYSTEMS, OOPSLA, 2001, Tampa. **Proceedings...** Tampa: [s.n.], 2001. p. 1-6. Disponível em: <http://www.cs.ubc.ca/~kdvolder/Workshops/OOPSLA2001/submissions/26-aldawud.pdf>. Acesso em: abr. 2003.

[ANC 95] ANCONA, M. et al. Channel Reification: a Reflective Approach to Fault-Tolerant Software Development. **ACM SIGPLAN Notices**, New York, v. 30, n. 10, p. 137, 1995.

[AOS 2002] ASPECT-Oriented Software Development – Home Page. Disponível em: <http://aosd.net/>. Acesso em: jan. 2002.

[ARA 2003] ARAÚJO, J. et al. Identifying aspectual use cases using a viewpoint-oriented requirements method. In: ASPECT-ORIENTED REQUIREMENTS ENGINEERING AND ARCHITECTURE DESIGN, Early Aspects, 2003, Boston. **Proceedings...** Boston: [s.n.], 2003. Disponível em: <http://www.cs.bilkent.edu.tr/AOSD-EarlyAspects/Papers/AraujoCoutinho.pdf>. Acesso em: maio 2003.

[ARA 2002] ARAÚJO, J. et al. Aspect-Oriented Requirements with UML. In: INTERNATIONAL WORKSHOP ON ASPECT-ORIENTED MODELING WITH UML, 2002, Dresden. **Proceedings...** Dresden: [s.n.], 2002. Disponível em: <http://lglwww.epfl.ch/workshops/uml2002/papers/Rashid.pdf>. Acesso em: jan. 2003.

[ASB 2002] ASPECTBROWSER – Home Page. Disponível em: <http://www-cse.ucsd.edu/users/wgg/Software/AB/>. Acesso em: maio 2002.

[ASJ 2002] THE ASPECTJ™ Programming Guide. Disponível em: <http://www.mcs.vuw.ac.nz/~rkhaled/aj/progguide/>. Acesso em: maio 2002.

[AVI 2001] AVIZIENIS, A. LAPRIE; J.-C.; RANDELL, B. **Fundamental Concepts of Dependability**. [S.l.]: LAAS-CNRS, 2001. (Research Report N01145). Disponível em: <http://citeseer.nj.nec.com/avizienis01fundamental.html>. Acesso em: jan. 2003.

[BAN 2003] BANIASSAD, E.; CLARKE, S. **Theme**: An Approach for Aspect-Oriented Analysis and Design. Trinity College: [s.n.], 2003. (Technical Report TCD-CS-2003-40) . Disponível em: <http://www.dsg.cs.tcd.ie/uploads/category360/116.pdf>. Acesso em: nov. 2003.

[BAR 98] BARDOU, D. Roles, Subjects and Aspects: How do they relate ? In: ASPECT ORIENTED PROGRAMMING WORKSHOP, ECOOP, 1998, Brussels. **Proceedings...** Belgium: Springer-Verlag, 1998. p. 418-419. Disponível em: <http://trese.cs.utwente.nl/aop-ecoop98/papers/Bardou.pdf>. Acesso em: dez. 2001.

[BAS 91] BASILI, V. R.; MUSA, J. The Future Engineering of Software a Management Perspective. **IEEE Computer**, Los Alamitos, v. 24, n. 9, p. 90-96, 1991.

[BEL 99] BELLUR, U. **The role of components & standards in software reuse**. Disponível em: <http:// www.obs.com/workshops/ws9801/paper012.pdf>. Acesso em: mar. 1999.

[BER 2000a] BERTAGNOLLI, S. C. **Ambiente Visual para o Desenvolvimento de Aplicações Java Reflexivas**. 2000. Dissertação (Mestrado em Computação) – Instituto de Informática, UFRGS, Porto Alegre.

[BER 2000b] BERTAGNOLLI, S. C.; LISBÔA, M. L. B. JReflex: um Ambiente Visual para o Desenvolvimento de Aplicações Java Reflexivas. In: SIMPÓSIO BRASILEIRO DE LINGUAGENS DE PROGRAMAÇÃO, SBLP, 4., 2000. **Anais...** Recife: SBC, 2000. p. 139-143.

[BER 2000c] BERTAGNOLLI, S. C. **Integrando Aspectos de Distribuição e de Tolerância a Falhas no Ambiente Java Reflexivo (Jreflex).** 2000. Trabalho Individual (Mestrado em Ciência da Computação) – Instituto de Informática, UFRGS, Proto Alegre.

[BER 2003] BERTAGNOLLI, S. C.; LISBÔA, M. L. B. FRIDA: a model to eliciting NFRs. In: ANALYSIS OF ASPECT-ORIENTED SOFTWARE, AAOS, Darmstadt, 2003. **Proceedings...** Darmstadt: [s.n.], 2003. Disponível em: <http://www.comp.lancs.ac.uk/computing/users/chitchya/AAOS2003/ Assets/bertagnolli_llisboa.pdf>. Acesso em: nov. 2003.

[BOE 96] BOEHM, B.; IN, H. Identifying Quality-Requirement Conflicts. **IEEE Software**, Los Alamitos, v. 13, n. 2, p. 25-35, 1996.

[BOE 84] BOEHM, B. Verifying and Validating Software Requirements and Design Specifications. **IEEE Software**, Los Alamitos, v.1, n. 1, p. 75-88, 1984.

[BOE 78] BOEHM, B. **Characteristics of Software Quality**. New York: North Holland Press, 1978.

[BOE 76] BOEHM, B.; BROWN, J. R.; LIPOW, M. Quantitative Evaluation of Software Quality. In: INTERNATIONAL CONFERENCE ON SOFTWARE ENGINEERING, ICSE, 2., 1976, San Francisco. **Proceedings...** Los Alamitos: IEEE Computer Society Press, 1976. p. 592-605.

[BOE 74] BOEHM, B. Some Steps Toward Formal and Automated Aids to Software Requirements Analysis and Design. In: IFIP CONGRESS, 1974, Stockholm. **Proceedings...** Stockholm: North-Holland, 1974. p. 192-197.

[BON 99a] BONDAVALIIi, A.; MAJZIK, I.; MURA, I. Automatic Dependability Analysis for Supporting Design Decisions in UML. In: IEEE INTERNATIONAL SYMPOSIUM ON HIGH-ASSURANCE SYSTEMS ENGINEERING, 4., Washington, 1999. **Proceedings...** Los Alamitos: IEEE Computer Society Press, 1999. p. 64-74.

[BON 99b] BONDAVALIIi, A.; MAJZIK, I.; MURA, I. Automated Dependability Analysis of UML Designs. In: IEEE INTERNATIONAL SYMPOSIUM ON OBJECT-ORIENTED REAL-TIME DISTRIBUTED COMPUTING, 2., Saint-Malo, 1999. **Proceedings...** Los Alamitos: IEEE Computer Society Press, 1999. p. 139-146.

[BOO 99] BOOCH, G. et al. **UML**: guia do usuário. Rio de Janeiro: Campus, 1999.

[BRG 2001] BERGMANS, L.; AKSIT, M. Composing Crosscutting Concerns Using Composition Filters. **Communications of the ACM**, New York, v. 44, n. 10, p. 51-57, 2001.

[BRG 94] BERGMANS, L. M. J. **Composing Concurrent Objects**: Applying Composition Filters for the Development and Reuse of Concurrent Object-Oriented Programs. 1994. Tese de Doutorado. University of Twente, Department of Computer Science, Netherlands. Disponível em: <ftp://ftp.cs.utwente.nl/pub/doc/TRESE/bergmans.phd.tar>. Acesso em: out. 2001.

[BRI 2002] BRITO, I.; MOREIRA, A.; ARAÚJO, J. A requirements model for quality attributes. In: ASPECT-ORIENTED REQUIREMENTS ENGINEERING AND ARCHITECTURE DESIGN, 2002, Enschede. **Proceedings...** Enschede: University of Twente, 2002. p. 1-6. Disponível em: <http://trese.cs.utwente.nl/AOSD-EarlyAspectsWS/Papers/Brito.pdf>. Acesso em: set. 2002.

[CAZ 2000a] CAZZOLA, W.; SOSIO, A.; TISATO, F. Shifting Up Reflection from the Implementation to the Analysis Level. In: WORKSHOP ON REFLECTION AND SOFTWARE ENGINEERING, 2000, Denver. **Proceedings...** Denver: Springer, 2000. p.1-20

[CAZ 2000b] CAZZOLA, W.; CHIBA, S.; LEDOUX, T. Reflection and Meta-Level Architectures: State of the Art, and Future Trends. In: EUROPEAN CONFERENCE FOR OBJECT-ORIENTED PROGRAMMING, ECOOP, 2000, Berlin. **Proceedings...** Berlin: Springer-Verlag, 2000. p. 1-15.

[CAZ 99] CAZZOLA, W.; SOSIO, A.; TISATO, F. Reflection and Object-Oriented Analysis. In: WORKSHOP ON REFLECTION AND SOFTWARE ENGINEERING, OORaSE, 1999, Denver. **Proceedings...** [S.l.: s.n], 1999. p. 95-106. Disponível em: <ftp://ftp.disi.unige.it/pub/person/CazzolaW/OORaSE99/095-106%20Cazzola.ps.gz>. Acesso em dez. 2001.

[CF 2002] COMPOSITION Filters. Disponível em: <http://trese.cs.utwente.nl/composition_filters/index.htm>. Acesso em: jan. 2002.

[CHE 2001] CHEESMAN, J.; DANIELS, J. **UML Components**: A Simple Process for Specifying Component-Based Software. New York: Addison-Wesley, 2001.

[CHU 2000] CHUNG, L. et al. **Non-Functional Requirements in Software Engineering**. [S.l.]:Kluwer Publishing, 2000.

[CHU 99] MYLOPOULOS, J.; CHUNG, L.; YU, E. From Object-Oriented to Goal-Oriented Requirements Analysis. **Communications of the ACM**, New York, v. 42, n.1, p. 31-37, 1999.

[CHU 95] CHUNG, L.; NIXON, B. A. Dealing with Non-Functional Requirements: Three Experimental Studies of a Process-Oriented Approach. In: INTERNATIONAL CONFERENCE ON SOFTWARE ENGINEERING, 17., 1995, Seattle. **Proceedings...** New York: ACM Press, 1995. p. 25-37.

[CHU 93] CHUNG, L. **Representing and Using Non-Functional Requirements**: A Process Oriented Approach. 1993. Tese de Doutorado (Doutorado em Ciência da Computação) - Department of Computer Science , University of Toronto, Toronto.

[CLA 2001] CLARKE, S.; WALKER, R. J. Composition Patterns: An Approach to Designing Reusable Aspects. In: INTERNATIONAL CONFERENCE ON SOFTWARE ENGINEERING, ICSE, 23., 2001, Toronto. **Proceedings...** Los Alamitos: IEEE Computer Society Press, 2001. p. 5-14.

[CLA 99b] CLARKE, S. et al. Subject-Oriented Design: Towards Improved Alignment of Requirements, Design and Code. In: OBJECT-ORIENTED PROGRAMMING SYSTEMS, LANGUAGES AND APPLICATIONS, OOPSLA, 14., 1999, Denver. **Proceedings...** New York: ACM Press, 1999. p. 325–339.

[CLE 2003] CLEMENT, A.; COLYER, A.; KERSTEN, M. Aspect-Oriented Programming with AJDT. In: ANALYSIS OF ASPECT-ORIENTED SOFTWARE, AAOS, 2003, Darmstadt. **Proceedings...** Darmstadt: [s.n.], 2003. p. 1-6. Disponível em: <www.comp.lancs.ac.uk/computing/users/chitchya/AAOS2003/Assets/clemas_colyer_kersten.pdf>. Acesso em: nov. 2003.

[COA 2003] CONALLEN, J. **Desenvolvendo Aplicações Web com UML**. Rio de Janeiro: Campus, 2003.

[COL 98] COLEMAN, D. A Use Case Template: Draft for discussion. **Fusion Newsletter**, Apr. 1998. Disponível em: <http://www.hpl.hp.com/fusion/md_newsletters.html>. Acesso em: nov. 2002.7

[CON 2001] CONSTANTINIDES, C. A. et al. Designing an Aspect-Oriented Framework in an Object-Oriented Environment. **ACM Computing Surveys**, New York, v. 32, n. 1, 2001. Disponível em: <http://www.cse.unl.edu/%7Efayad/column/CACM/ACMPub/a41-constantinides.pdf>. Acesso em: abr. 2001.

[CON 2000a] CONSTANTINIDES, C. A.; ELRAD, T.; FAYAD, M. E. Extending the object model to provide explicit support for crosscutting concerns. **Software Practice and Experience**, New York, v. 1, n. 7, p. 1-36, 2000.

[COO 2003] COOPER, K.; DAI, L.; DENG, Y. Modeling Performance as an Aspect: a UML Based Approach. In: AOSD MODELING WITH UML WORKSHOP, 4. 2003, San Francisco. **Proceedings...** [S.l. : s.n.], 2003.

[COR 2001] CORTES, M. L.; CHIOSSI, T. C. S. **Modelos de Qualidade de Software**. São Paulo: Ed. da Unicamp, 2001.

[COS 2001] COSTANZA, P.; KNIESEL, G.; AUSTERMANN, M. Independent Extensibility for Aspect-Oriented Systems. In: WORKSHOP ON ADVANCED SEPARATION OF CONCERNS, ASC, 2001, Budapest. **Proceedings...** [S.l.:s.n.], 2001. p. 1-6. Disponível em:<http:zSzzSzwww.informatik.uni-bonn.dezSz%7EcostanzazSz.zSz.zSzindependentextensibility.pdf/independent-extensibility-for-aspect.pdf>. Acesso em: out. 2002.

[COW 97] CONROW E.H.; SHISHIDO P.S. Implementing Risk Management on Software Intensive Projects. **IEEE Software**, Los Alamitos, v. 14, n. 3, p. 83-89, May/June 1997.

[CYS 2001] CYSNEIROS,L. M. **Requisitos não funcionais**: da elicitação ao modelo conceitual. 2001. Tese de Doutorado. Pontifícia Universidade Católica do Rio de Janeiro, Rio de Janeiro. Disponível em: <http://www.cs.toronto.edu/~cysneiro/articles/Tese.zip>. Acesso em: out. 2002.

[CYS 2000] CYSNEIROS, L.M.; LEITE, J.C.S.P.; NETO, J.S.M. A Framework for Integrating Non-Functional Requirements into Conceptual Models. **Requirements Engineering Journal**, London, v. 6, n. 2, p. 97-115, 2001.

[CYS 99] CYSNEIROS, L.M.; LEITE, J.C.S.P. Integrating Non-Functional Requirements into data model. In: INTERNATIONAL SYMPOSIUM ON REQUIREMENTS ENGINEERING, 4., 1999, Ireland. **Proceedings...** Los Alamitos: IEEE Computer Society Press, 1999. p. 162-171.

[CYS 97] CYSNEIROS, L.M.; LEITE, J.C.S.P. Definindo Requisitos Não Funcionais. In: SIMPÓSIO BRASILERIO DE ENGENHARIA DE SOFTWARE, SBES, 11., 1997, Ceará. **Anais...** Ceará: SBC, 1997. p. 49-64.

[CZA 2000] CZANERCKI, K.; ULRICH, W. E. **Generative Programming**: Methods, Tools, and Applications. Massachusetts: Addison-Wesley, 2000.

[CZA 98] CZANERCKI, K. **Generative Programming**: principles and Techniques of Software Engineering Based on Automated Configuration and Fragment-Based Component Models. 1998. Tese de Doutorado, Technische Universitat, Ilmenau. Disponível em: <http://www.prakinf.tu.ilmenau.de/czarn>. Acesso em: jan. 2001.

[DAD 2002] DAVID, P. C.; LEDOUX, T. Dynamic Adaptation of Non-Functional Concerns. In: INTERNATIONAL WORKSHOP ON UNANTICIPATED SOFTWARE EVOLUTION, USE, 2002, Málaga. **Proceedings...** [S.l.: s.n.], 2002. p. 1-8. Disponível em: <http://joint.org/use2002/use2002.pdf>. Acesso em: abr. 2002.

[DAS 88] DAVIS, A. M. A comparison of techniques for the specification of external system behavior. **Communications of the ACM**, New York, v.31, n.9, p.1098-114, 1988.

[DAV 93] DAVIS, A. **Software Requirements – Objects, Functions and States**. London: Prentice-Hall, 1993.

[DEM 2002] DEMETER – Aspect-Oriented Software Development. Disponível em: <http://www.ccs.neu.edu/research/demeter/>. Acesso em: jan 2002.

[DES 98] DESMOND, S.; WILL A. **Objects, Components and Frameworks with UML**: The Catalysis Approach. New York: Addison-Wesley, 1998.

[DIC 2004] DICIONÁRIO de Português**.** Disponível em: <http://www.priberam.pt/dlpo/>. Acesso em: jan. 2004.

[ELR 2001a] ELRAD, T.; FILMAN, R. E.; BADER, A. Aspect-Oriented Programming. **Communications of the ACM**, New York, v. 44, n. 10, p. 29-32, Oct. 2001.

[ELR 2001b] ELRAD, T.; et al. Discussing Aspects of AOP. **Communications of the ACM**, New York, v. 44, n. 10, p. 33-38, Oct. 2001.

[FAB 95] FABRE, J. C. et al. Implementing Fault-Tolerant Applications using Reflective Object-Oriented Programming. In: INTERNATIONAL SYMPOSIUM ON FAULT-TOLERANT COMPUTING, 25., 1995, Pasadena. **Proceedings...** Los Alamitos: IEEE Computer Society Press, 1995. p. 489-498.

[FAY 2003] FAYAD, M. E.; PRADEEP, R. S.; SEDDIQUI, F. Aspects in Communications: Performance. In: AOSD MODELING WITH UML WORKSHOP, 4. 2003, San Francisco. **Proceedings...** [S.l. : s.n.], 2003.

[FER 99] FERREIRA, S. B. L. et al. Requisitos Não Funcionais para Interfaces com o Usuário – O Uso de Cores. In: WORKSHOP IBEROAMERICANO DE ENGENHARIA DE REQUISITOS E AMBIENTES DE SOFTWARE, IDEAS, 1999, Costa Rica. **Proceedings...** Costa Rica: CYTED & Instituto Tecnológico de Costa Rita (TEC), 1999. p. 279-291.

[FIN 2000] FINKELSTEIN, A.; KRAMER, J. Software Engineering: A Roadmap, the Future of Software Engineering. In: INTERNATIONAL CONFERENCE ON SOFTWARE ENGINEERING, ICSE, 2000, Limerick. **Proceedings...** New York: ACM Press, 2000. p. 3-22.

[FIN 96] FINKELSTEIN, A.; DOWELL, J. A comedy of Errors: The London Ambulance Service Case Study. In: INTERNATIONAL WORKSHOP ON SOFTWARE SPECIFICATION AND DESIGN, 8., 1996, Washington. **Proceedings...** Los Alamitos: IEEE Computer Society Press, 1996. p. 2-5.

[FIR 2003] FIRESMITH, D. Specifying Good Requirements. **Journal of Object Technology**, [S.l.], v. 2, n. 4, p. 77-87, 2003.

[FON 2000] FONTOURA, M.; PREE, W.; RUMPE, B. Modeling Language for Object-Oriented Frameworks. [Berlin: Springer Verlag], p. 63-68, 2000. (Lecture Notes in Computer Science, v. 1850). Disponível em: <citeseer.nj.nec.com/314906.html>. Acesso em: dez. 2001.

[FOW 2000] FOWLER, M.; KENDALL S. **UML Essencial**. Porto Alegre: Bookman, 2000.

[FUG 2000] FUGGETTA, A. Software Process: A Roadmap. In: INTERNATIONAL CONFERENCE ON SOFTWARE ENGINEERING, ICSE, 2000, Limerick. **Proceedings...** New York: ACM Press, 2000. p. 25-34.

[GAM 2000] GAMMA, E. et al. **Padrões de Projeto**. Porto Alegre: Bookman, 2000.

[GAR 98] GARCIA, J. C. R. et al. A Metaobject Protocol for Fault-Tolerant CORBA Applications. In: SYMPOSIUM ON RELIABLE DISTRIBUTED SYSTEMS, SRDS, 17., 1998, Indiana. **Proceedings...** Indiana: IEEE Computer Society Press, 1998. p. 127-134.

[GLA 95] GLANDRUP, M. **Extending C++ Using the Concepts of Composition Filters**. 1995. Dissertação de Mestrado. University of Twente, Tewnte. Disponível em: <ftp.cs.utwente.nl/pub/doc/TRESE/glandrup.thesis.ps.Z> Acesso em: out. 2001.

[GRI 99] GRISWOLD, W.; KATO, Y.; YUAN, J. J. Aspect Browser: Tool Support for Managing Dispersed Aspects. In: WORKSHOP ON MULTI-DIMENSIONAL SEPARATION OF CONCERNS IN OBJECT-ORIENTED SYSTEMS, OOPSLA, 1999, Colorado. **Proceedings...** [S.l : s.n.], 1999. p. 1-6. Disponível em: <http://www.cs.ubc.ca/- ~murphy/multd-workshop-oopsla99>. Acesso em: dez. 2002.

[GRU 2001a] GRUNDY, J.; PATEL, R. Developing Software Components with the UML, Enterprise Java Beans and Aspects. In: AUSTRALIAN SOFTWARE ENGINEERING CONFERENCE, ASWSEC, 2001, Canberra. **Proceedings...** Canberra: IEEE Computer Society Press, 2001. p.127-136.

[GRU 2001b] GRUNDY, J.; HOSKING, J. Software Tools. In: **Wiley Encyclopedia of Software Engineering**. [S.l.]: Wiley and Sons, 2001. Disponível em: < http://www.cs.auckland.ac.nz/~john-g/papers/se2e2001.pdf>. Acesso em jun. 2002.

[GRU 99] GRUNDY, J. Aspect-oriented Requirements Engineering for Component-based Software Systems. In: SYMPOSIUM ON REQUIREMENTS ENGINEERING, RE, Limerick, 1999. **Proceedings...** Limerick: IEEE Computer Society Press, 1999. p. 84-91. Disponível em: <http://www.cs.auckland.ac.nz/~john-g/papers/re99.ps.gz>. Acesso em: dez. 2001.

[HAR 2000] HARRISON, W.; OSSHER, H.; TARR, P. Software engineering tools and environments: A roadmap. In: INTERNATIONAL CONFERENCE ON SOFTWARE ENGINEERING, ICSE, 22., 2000, Limerick. **Proceedings...** New York: ACM Press, 2000. p. 261 – 277.

[HAR 93] HARRISON, W.; OSSHER, H. Subject-oriented programming (A critique of pure objects). **ACM SIGPLAN Noticies**, New York, v. 28, n. 10, p. 411-428, 1993.

[HER 2000] HERRERO, J. L. et al. Introducing Separation of Aspects at Design Time. In: WORKSHOP OF ASPECTS AND DIMENSIONS OF CONCERNS, 2000, Sophia Antipolis and Cannes. **Proceedings...** [S.l.: s.n.], 2000. p. 1-7. Disponível em: <http://trese.cs.utwente.nl/Workshops/adc2000/papers/Herrero.pdf>. Acesso em: out. 2002.

[HIG 99] HIGHLEY, T. J.; LACK, M.; MYERS, P. **Aspect Oriented Programming**: a critical analysis of a new programming paradigm. Virginia: University of Virginia, 1999. (Technical Report CS-99-29). Disponível em: <ftp://ftp.cs.virginia.edu/pub/techreports/CS-99-29.pdf>. Acesso em: dez. 2002.

[HIL 99] HILLIARD, R. Aspects, Concerns, Subjects, Views, ...*. In: WORKSHOP ON MULTI-DIMENSIONAL SEPARATION OF CONCERNS IN OBJECT-ORIENTED SYSTEMS, 1999, Denver. **Proceedings...** [S.l.: s.n.], 1999. p. 1-4. Disponível em: <www.cs.ubc.ca/~murphy/multid-workshop-oopsla99/position-papers/ws08-hilliard.pdf>. Acesso em: set. 2002.

[HOF 93] HOFMANN, H. F. **Requirements Engineering**: a survey of methods and tools. Zürich: Institut für Informatik der Universität Zürich, 1993. (Technical Report, ifi-93.05). Disponível em:<ftp://ftp.ifi.unizh.ch/pub/techreports/TR-93/ifi-93.05.ps.gz>. Acesso em: jan. 2001.

[HSI 94] HSIA, P. et al. Formal Approach to Scenario Analysis. **IEEE Software**, Los Alamitos, v. 11, n. 2, p. 33–41, Mar. 1994.

[HÜR 95] HÜRSCH, W. L.; LOPES, C. V. **Separation of Concerns**. Boston: Northeastern University, 1995. (Technical Report NU-CCS-95-03). Disponível em: <ftp://ftp.ccs.neu.edu/pub/people/lieber/crista/techrep95/index.html>. Acesso em: set. 2001.

[HYP 2002a] MULTI-DIMENSIONAL Separation of Concerns using Hyperspaces. Disponível em: <http://www.research.ibm.com/hyperspace/>. Acesso em: maio 2002.

[HYP 2002b] HYPER/J – Home Page. Disponível em: <www.alphaworks.ibm.com/tech/hyperj>. Acesso em: maio 2002.

[IEE 2000] INSTITUTE OF ELECTRICAL AND ELECTRONIC ENGINEERING. **IEEE Std. 1471-2000**: practice for architectural description of software-intensive systems. New York, 2000.

[ISO 92] INTERNATIONAL ORGANIZATION FOR STANDARDIZATION. **ISO 9126**: quality characteristics and guidelines for their use. Geneva, 1992.

[JAC 2003] JACOBSON, I. Use Cases and Aspects – Working Seamlessly Together. **Journal of Object Technology**, [S.l.], v. 2, n. 4, p. 7-28, July/Aug. 2003.

[JAC 99] JACOBSON, I.; RUMBAUGH, J.; BOOCH, G. **The unified software development process**. New York: Addison-Wesley, 1999.

[JAC 92] JACOBSON, I. et al. **A Use Case Driven Approach**: Object-Oriented Software Engineering. Boston: Addison-Wesley, 1992.

[JED 2002] JEDUC – Home Page. Disponível em: <http://www.inf.ufrgs.br/jeduc>. Acesso em: nov. 2002.

[JMA 2002] JMANGLER – HomePage. Disponível em: <http://javalab.iai.uni-bonn.de/research/jmangler/>. Acesso em: maio 2002.

[JRX 2001] JREFLEX – Home Page. Disponível em: <http://www.inf.ufrgs.br/~silviacb/JReflex/index.html>. Acesso em: jan. 2001.

[KAI 99] KAINDL, H. Difficulties in the Transition from OO Analysis to Design. **IEEE Software**, Los Alamitos, v. 16, n. 5, p. 94-102, Sept./Oct. 1999.

[KAS 98] KASBEKAR, M.; NARAYANAN, C.; DAS, C. R. Using Reflection for Checkpoint Concurrent Object Oriented Programs. In: WORKSHOP ON REFLECTION PROGRAMMING IN C++ AND JAVA, 1998, Vancouver. **Proceedings...** [S.l. : s.n.], 1998. p. 71-75. Disponível em: <http://www.csg.is.titech.ac.jp/~chiba/oopsla98/proc/kasbekar.pdf>. Acesso em: mar. 2000.

[KEL 90] KELLER, S.E. et al. Specifying Software Quality Requirements with Metrics. In**: Tutorial System and Software Requirements Engineering**. Washington: IEEE Computer Society Press, 1990. p.145-163.

[KIC 2001] KICZALES, G. et al. Getting Started with AspectJ. **Communications of the ACM**, New York, v. 44, n. 10, p. 59-65, Oct. 2001.

[KIC 97] KICZALES, G. et al. Aspect-Oriented Programming. In: EUROPEAN CONFERENCE FOR OBJECT-ORIENTED PROGRAMMING, ECOOP, 1997, Finland. **Proceedings...** Berlin: Springer-Verlag, 1997. p. 220-242.

[KIC 91] KICZALES, G.; RIVIÈRES, J. BOBROW, G. **The Art of the Metaobject Protocol**. Cambridge: MIT Press, 1991.

[KON 92] KONTONYA, G.; SOMMERVILLE, I. Viewpoints for Requirements Definition. **Software Engineering Journal**, Herts, v. 7, n. 6, p. 375-387, 1992.

[LAD 2003] LADDAD, R. **AspectJ in Action**: Practical Aspect-Oriented Programming. Greenwich: Manning Publications Company, 2003.

[LAD 2002a] LADDAD, R. **I want my AOP ! Part 1**. Disponível em: <http://www.javaworld.com/javaworld/jw-01-2002/jw-0118-aspect.html>. Acesso em: fev. 2002.

[LAD 2002b] LADDAD, R. **I want my AOP ! Part 2.** Disponível em: <http://www.javaworld.com/javaworld/jw-03-2002/jw-0301-aspect2.html>. Acesso em: fev. 2002.

[LAF 2003] LAFFERTY, D.; CAHILL, V. Language-Independent Aspect-Oriented Programming. In: OBJECT ORIENTED PROGRAMMING SYSTEMS LANGUAGES AND APPLICATIONS, OOPSLA, 18., 2003, Anaheim. **Proceedings...** New York: ACM Press, 2003. p. 1–12.

[LAP 95] LAPRIE, J-C. Dependable Computing: Concepts, Limits, Challenges. In: INTERNATIONAL SYMPOSIUM ON FAULT-TOLERANT COMPUTING, 25., 1995, Pasadena. **Proceedings...** Los Alamitos: IEEE Computer Society Press, 1995. p. 42-54.

[LAP 92] LAPRIE, J-C. Dependability: Basic concepts and terminology. In: DEPENDABLE COMPUTING AND FAULT TOLERANCE, 1992, Vienna. **Proceedings...** New York: Springer-Verlag, 1992, p.265-275.

[LAR 2000] LARMAN, C. **Aplicando UML e Padrões:** uma Introdução à Análise e Projeto Orientados a Objetos. Porto Alegre: Bookman, 2000.

[LEI 2001] LEITE, J. C. S. P. Gerenciando a Qualidade de Software com Base em Requisitos. In: ROCHA, A. R.C. et al. **Qualidade de Software**: teoria e prática. São Paulo: Prentice-Hall, 2001. p. 238-246.

[LEI 95] LEITE, J. C. S. P. **Engenharia de Requisitos**. Rio de Janeiro: Pontifícia Universidade Católica do Rio de Janeiro, 1995. Notas de Aula da Disciplina "Engenharia de Requisitos".

[LEI 93] LEITE, J.C.S.P.; FRANCO, A.P.M. A Strategy for Conceptual Model Acquisition. In: INTERNATIONAL SYMPOSIUM ON REQUIREMENTS ENGINEERING, 1993, SanDiego. **Proceedings...** Los Alamitos: IEEE Computer Society Press, 1993. p. 243-246.

[LEI 91] LEITE, J.C.S.P.; FREEMAN, P. A. Requirements Validation Through Viewpoint Resolution. **IEEE Transactions on Software Engineering**, New York, v. 17, n. 1, p. 1253 – 1269, 1991.

[LEE 99] LEE, J. et al. Developing Distributed Software Systems by Incorporating Meta-Object Protocol (diMOP) with Unified Modeling Language (UML). In: INTERNATIONAL SYMPOSIUM ON AUTONOMOUS DECENTRALIZED SYSTEMS, 4., 1999, Tokyo. **Proceedings...** Los Alamitos: IEEE Computer Society Press, 1999. p. 65-72.

[LEM 2000] LEMOS, R. **Confiança no Funcionamento**: Terminologia em Português. 2000. Disponível em: <http://www.cs.kent.ac.uk/people/staff/rdl/CoF/>. Acesso em: mar. 2003.

[LEM 99] LEMOS, R. **Confiança no Funcionamento:** Terminologia em Português. 1999. Disponível em: <http://www.cs.kent.ac.uk/people/staff/rdl/CoF/glossario.html>. Acesso em: mar. 2003.

[LIE 2001] LIEBERHERR, K.; ORLEANS, D.; OVLINGER, J. Aspect-Oriented Programming with Adaptive Methods. **Communications of the ACM**, New York, v. 44, n. 10, p.39-42, Oct. 2001.

[LIE 98] LIEBERHERR, K. **Connections between Demeter/Adaptive Programming and Aspect-Oriented programming**. 1998. Disponível em: <http://www.ccs.neu.edu/home/lieber/connection-to-aop.html>. Acesso em fev. 2001.

[LIE 96] LIEBERHERR, K. **Adaptive Object-Oriented Software**: the Demeter method with propagation patterns. Boston: PWS Publishing Company, 1996. Disponível em: <ftp://ftp.ccs.neu.edu/pub/people/lieber/book/aoos.PDF>. Acesso em: nov. 2001.

[LIE 94] LIEBERHERR, K. Component Enhancement: An Adaptive Reusability Mechanism for Groups of Collaborating Classes. **Communications of the ACM**, New York, v. 37, n. 5, p. 94-101, 1994.

[LIE 88] LIEBERHERR, K. J.; HOLLAND, I.; RIEL, A. J. Object-oriented programming: an objective sense of style. **ACM SIGPLAN Noticies**, New York, v. 23, n. 10, p. 323-334, 1988.

[LIN 93] LINDSTROM, D.R. Five Ways to Destroy a Development Project. **IEEE Software**, Los Alamitos, v. 10, n. 5, p. 55-58, Sept./Oct. 1993.

[LIO 2002] LIONS, J. M.; SIMONEAU, D.; PITETTE, G.; MOUSSA, I. Extending OpenTool/UML Using Metamodeling: An Aspect Oriented Programming Case Study. In: WORKSHOP ON ASPECT-ORIENTED MODELING WITH UML, 2002, Dresden. **Proceedings...** Dresden: [s.n.], 2002. p. 1-6. Disponível em: <http://lglwww.epfl.ch/workshops/uml2002/papers/lions.pdf>. Acesso em: jul. 2002.

[LIS 95] LISBÔA, M. L. B. **MOTF:** Meta-objetos para Tolerância a Falhas**.** 1994. Tese (Doutorado em Ciência da Computação) - Instituto de Informática, UFRGS, Porto Alegre.

[LIS 98] LISBÔA, Maria Lúcia Blanck. A New Trend on the Development of Fault-Tolerant Applications: Software Meta-level Architectures. In: INTERNATIONAL WORKSHOP ON DEPENDABLE COMPUTING AND ITS APPLICATIONS, IFIP, 1998, Johannesburg. **Proceedings...** [S.l. : s.n.], 1998.

[LOP 97] LOPES, C. V.; KICZALES, G. **D**: A Language Framework for Distributed Programming. Boston: Northeastern University, 1997. (Technical Report SPL97-010 P9710047). Disponível em: <http:// www.ccs.neu.edu/pub/people/crista/thp.ps>. Acesso em: nov. 2001.

[LOR 2001] LORENZ, D. H.; VLISSIDES, J. Designing components versus Objects: A transformational Approach. In: INTERNATIONAL CONFERENCE ON SOFTWARE ENGINEERING, ICSE, 23., 2001, Toronto. **Proceedings...** Los Alamitos: IEEE Computer Society Press, 2001. p. 253-262.

[MAC 99] MACEDO, N. A. M. et al. Integrando requisitos não-funcionais aos requisitos baseados em ações concretas. In: WORKSHOP IBERO-AMERICANO DE ENGENHARIA DE REQUISITOS E AMBIENTES DE SOFTWARE, IDEAS, 1999, Costa Rica. **Proceedings...** Costa Rica: CYTED & Instituto Tecnológico de Costa Rita (TEC), 1999. p. 1-9.

[MAE 87] MAES, P. Concepts and experiments in computational reflection. **ACM SIGPLAN Notices**, New York, v. 22, n.12, p.147-155, 1987.

[MAL 2001] MALAN, R., BRENDEMEYER, D. **Defining Non-Functional Requirements.** 2001. Disponível em: <http://www.brendemeyer.com/papers.htm>. Acesso em: dez. 2002.

[MAM 99] MAMANI, Nestor A. **Integrando requisitos não funcionais aos requisitos baseados em ações concertas**. Dissertação (Mestrado) - PUC-Rio, Rio de Janeiro.

[McC 77] McCALL, J. et al. **Factors in Software Quality**. Sunnyvale: General Electric Company, 1977. (Technical Report RADC-TR-77-357).

[MEN 97] MENS, K. et al. Aspect-Oriented Programming Workshop Report. In: EUROPEAN CONFERENCE FOR OBJECT-ORIENTED PROGRAMMING, ECOOP, 1997, Finland. **Proceedings...** Berlin: Springer-Verlag, 1997. p. 483-496.

[MIC 2002] MICROSOFT Research – Intentional Programming Group. Disponível em: <ftp://ftp.research.microsoft.com/pub/tr/tr-95-52.ps>. Acesso em: jan 2002.

[MIL 96] MILI, H.; HARRISON, W.; OSSHER, H. Supporting subject-oriented programming in Smalltalk. In: TECHNOLOGY OF OBJECT-ORIENTED LANGUAGES AND SYSTEMS, TOOLS, 20., 1996, Santa Barbara. **Proceedings...** [S.l. : s.n.], 1996.

[MOR 95] MORDHORST, J.; DIJK, W. V. **Composition Filters in Smalltalk**. 1995. Dissertação (Mestrado) - Department of Computer Science, UT, Twente. Disponível em: <http://trese.cs.utwente.nl/publications/paperinfo/aksit.phd.pi.top.htm >. Acesso em: out. 2001.

[MOU 2001] MOURA, A. V. **Especificações em Z**: uma introdução. Campinas: Ed. da Unicamp, 2001. p. 23-34.

[MYL 92] MYLOPOULOS, J. et al. Representing and Using Non-functional Requirements: A Process-Oriented Approach. **IEEE Transactions on Software Engineering**, New york, v. 18, n. 6, p.483-497, 1992.

[NAU 69] NAUR, P. et al. Revised Report on the Algorithmic Language Algol 60. **Programming Systems and Languages**. New York: McGraw-Hill, 1969.

[NEN 2000] NENTWICH, C. et al. BOX: Browsing objects in XML. **Software Practice and Experience**, New York, v. 30, p. 1661-1676, 2000.

[NIE 2002] NIELSEN, J. **Homepage Usabilidade**: 50 websites desconstruídos. Rio de Janeiro: Campus, 2002.

[OMG 2001] UNIFIED Modeling Language. Disponível em: <http://www.uml.org/>. Acesso em: mar. 2001.

[ORL 2001] ORLEANS, D.; LIEBERHERR, K. DJ: Dynamic Adaptive Programming in Java. In: REFLECTION 2001: META-LEVEL ARCHITECTURES AND SEPARATION OF CROSSCUTTING CONCERNS, 2001, Tokyo. **Proceedings...** Berlin: Springer Verlag, 2001. p. 73-80.

[OSS 2001a] OSSHER, H.; TARR, P. Hyper/J: multi-dimensional separation of concerns for Java. In: INTERNATIONAL CONFERENCE ON SOFTWARE ENGINEERING, ICSE, 23., 2001, Toronto. **Proceedings...** New York: ACM Press, 2001. p. 734-737.

[OSS 2001b] OSSHER, H.; TARR, P. Using Multidimensional Separation of Concerns to (re)shape evolving software. **Communications of the ACM**, New York, v. 44, n. 10, p. 43-50, Oct. 2001.

[OSS 98] OSSHER, H. Operation-Level Composition: A Case in (Join) Point. In: WORKSHOP ON ASPECT ORIENTED SYSTEMS, 1998, Brussels. **Proceedings...** Berlin: Springer Verlag, 1998. p. 406-409.

[OSS 96] OSSHER, H. et al. Specifying Subject-Oriented Composition. **Theory and Practice of Object Systems**, New York, v. 2, n. 3, p. 179 – 202, 1996.

[OSS 95] OSSHER, H. et al. Subject-Oriented Composition Rules. **ACM SIGPLAN Noticies**, New York, v. 30, n.10, p. 235-250, 1995.

[PAC 2000] PACE, J. A. D.; TRILNIK, F.; CAMPO, M. R. How to Handle Interacting Concerns? In: Advanced Techniques for Separation of Concerns, ASOC, 2000, Minneapolis. **Proceedings...** New York: ACMP Press, 2000. Disponível em: <citeseer.nj.nec.com/551559.html>. Acesso em: dez. 2001.

[PER 2002] PEREGO, Cássia Alves; LISBÔA, Maria Lúcia Blanck; BERTAGNOLLI, Silvia de Castro. A Migração de Pascal para Java: Problemas e Propostas de Soluções. In: WORKSHOP SOBRE EDUCAÇÃO EM COMPUTAÇÃO, WEI, 2002, Florianópolis. **Anais...** Porto Alegre: SBC, 2002. p. 147-156.

[PIM 97] PIMENTA, M. S. **Tarefa**: Une approche pour l'Ingénierie des Besoins des Systèmes Interactifs. 1997. Tese (Doutorado) - Université Toulose, Toulose.

[PIN 96] PINHEIRO, F. A. C.; GOUGUEN, J. A. An Object-Oriented Tool for Tracing Requirements. **IEEE Software**, Los Alamitos, v. 13, n. 2, p. 52-64, Mar. 1996.

[POP 2001] POPOVICI, A. The Impact of Aspect-Oriented Programming on Future Application Design. Zurich: ETH Zurich, 2001. (Techinical Report: Information and Communication Research Group Report). Disponível em: <http://www.vs.inf.ethz.ch/edu/WS0001/UI/slides/ui_08AspectOP.pdf >. Acesso em: fev. 2003.

[POT 94] POTTS, C.; TAKAHASHI, K. ANTON, A. Inquiry-Based Requirements Analysis. **IEEE Software**, Los Alamitos, v. 11, n. 2, p. 21-32, Mar. 1994.

[PRE 97] PREE, W. Component-Based Software Development—A New Paradigm in Software Engineering? **Software—Concepts and Tools**, [S. l.], v. 18, n. 4, p. 169–174, 1997.

[PRS 2001] PRESSMAN, R. **Software Engineering**: A practitioner's Approach. 5th ed. New York: McGrawHill, 2001.

[PRY 2000] PRYOR, J. L. **Dynamic Multilevel Separation of Concerns**. 2000. Thesis Proposals. Universidad Nacional del Centro de la Pcia. de Buenos Aires, Facultad de Ciencias Exactas, ISISTAN Research Institute.

[PRY 2002] PRYOR, J. L.; PACE, A. D.; CAMPO, M. Reflecting on Separation of Concerns. **Revista de Informática Teórica e Aplicada**, Porto Alegre, v. 9, n. 1, p. 7-36, 2002.

[RAN 98] RANDELL, B. Dependability - a Unifying Concept. In: COMPUTER SECURITY, DEPENDABILITY, AND ASSURANCE: FROM NEEDS TO SOLUTIONS, 1998, York. **Proceedings...** Washington: IEEE Computer Society Press, 1998. p. 16-26.

[RAS 2003] RASHID, A.; MOREIRA, A.; ARAÚJO, J. Modularisation and Composition of Aspectual Requirements. In: INTERNATIONAL CONFERENCE ON ASPECT-ORIENTED SOFTWARE DEVELOPMENT, 2., 2003, Boston. **Proceedings...** New York: ACM Press, 2003. p. 11-20.

[REE 2002] REED, P. R. **Developing Applications with Java and UML**. New York: Addison-Wesley. 2002.

[REG 95] REGNELL, B.; KIMBLER, K. WESSLÉN. A. Improving the Use Case Driven Approach to Requirements Engineering, In: INTERNATIONAL SYMPOSIUM ON REQUIREMENTS ENGINEERING, 2., 1995, York. **Proceedings...** [S. l.]: IEEE Computer Society Press, 1995. p.40-47.

[ROC 2001] ROCHA, A. R. C.; MALDONADO, J. C.; WEBER, K. C. **Qualidade de Software:** Teoria e Prática. São Paulo: Prentice-Hall, 2001.

[ROM 85] ROMAN, G. C. A Taxonomy of Current Issues in Requirements Engineering. **IEEE Computer**, Los Alamitos, v. 18, n. 4, p. 14-23, Apr, 1985.

[RUB 98] RUBIB, D. M. Uses of Use Cases. 1998. Disponível em: <http://www.softstar-inc.com>. Acesso em: abr. 2002.

[RUM 94] RUMBAUGH, J. Getting Started – Using Use Cases to Capture Requirements. **Journal of Object-Oriented Programming**, [S. l.], v. 7, n. 5, p. 8-23, Sept. 1994.

[SAP 2002] SAPIR, N.; TYSZBEROWICZ, S.; YEHUDAI, A. Extending UML with Aspect Usage Constraints in the Analysis and Design Phases. In: WORKSHOP ON ASPECT-ORIENTED MODELING WITH UML. 2002.. **Proceedings...** Dresden: [s.n.], 2002. p. 1-6. Disponível em: <http://lglwww.epfl.ch/workshops/uml2002/papers/tyszberowicz.pdf>.

Acesso em: maio 2003.

[SAW 2002] SAWYER, P.; SOMMERVILLE, I.; VILLER, S. **PREview**: tackling the real concerns of requirements engineering. Lancaster: Lancaster University, 2002. (Technical Report CSEG/5/96). Disponível em: <citeseer.ist.psu.edu/sawyer96preview.html>. Acesso em: set. 2003.

[SCH 98] SCHNEIDER, G.; WINTERS, J. P. **Applying Use Cases**: A Practical Guide. New York: Addison-Wesley, 1998.

[SHA 96] SHAW, M.; GARLAN, D. **Software Architecture**: perspectives on an emerging discipline. London: Prentice-Hall, 1996.

[SIL 94] SILVA-LEPE, I.; HÜRSH, W.; SULLIVAN, G. A Report on Demeter C++. **C++ Report**, [S. l.], v. 6, n. 2, p. 24-30, Nov. 1994.

[SMI 82] SMITH, B. **Reflection and Semantics in a Procedural Language**. Massachusetts: Laboratory for Computer Science, 1982. (Technical Report TR 272).

[SOM 2001] SOMMERVILLE, I. **Software Engineering**. 6th ed. Harlow: Addison-Wesley, 2001.

[SOP 2002] SUBJECT-Oriented Programming – Home Page. Disponível em: <http://www.research.ibm.com/sop/>. Acesso em: jan. 2002.

[STE 2003] STEIN, D.; HANENBERG, S. UNILAND, R. On Representing Join Points in the UML. In: WORKSHOP ON ASPECT-ORIENTED MODELING WITH UML, AOM, 2003, Boston. **Proceedings...** Boston: [s.n.], 2003. Disponível em: < http://lglwww.epfl.ch/workshops/uml2002/papers/stein.pdf>. Acesso em: maio 2003.

[STR 96] STROUD, R. J.; WU, Z. Using Metaobject Protocols to Satisfy Non-Functional Requirements. In: **Advances in Object-Oriented Metalevel Architectures and Reflection**. Boca Raton: CRC, 1996. 227p.

[STR 95] STROUD, R. J.; WU, Z. Using Meta-Object Protocol to Implement Atomic Data Types. In: EUROPEAN CONFERENCE FOR OBJECT-ORIENTED PROGRAMMING, ECOOP, 1995. **Proceedings...** Berlin: Springer-Verlag, 1995. p. 168-189.

[STR 92] STROUD, R. J. Transparency and Reflection in Distributed Systems. **ACM Operating Systems Review**, New York, v. 22, n. 2, p. 99-103, Apr. 1992.

[SUB 2001] SUBRAMANIAN, N. CHUNG, L. Software Architecture Adaptability: An NFR Approach. In: INTERNATIONAL WORKSHOP ON PRINCIPLES OF SOFTWARE EVOLUTION, IWPSE, 4., 2001, Vienna. **Proceedings...** New York: ACM Press, 2001. p. 52-61. Disponível em: <http://www.utdallas.edu/~chung/ftp/IWPSE.pdf>. Acesso em: fev. 2002.

[SUL 2001] SULLIVAN, G. T. Aspect-Oriented Programming Using Reflection and MetaObject Protocols. **Communications of the ACM**, New York, v. 44, n. 10, p. 39-42. Oct. 2001.

[SUZ 99a] SUZUKI, J.; YAMAMOTO, Y. Extending UML for Modeling Reflective Software Components. In: INTERNATIONAL CONFERENCE ON THE UNIFIED MODELING LANGUAGE, 2., 1999, Columbia. **Proceedings...** Berlin: Springer Verlag, 1999. p. 220-235. Disponível em: < http://www.yy.cs.keio.ac.jp/~suzuki/project/pub/uml99.pdf.zip>. Acesso em: ago. 2002.

[SUZ 99b] SUZUKI, J.; YAMAMOTO, Y. Extending UML with Aspects: Aspect Support in the Design Phase. In: ASPECT-ORIENTED PROGRAMMING WORKSHOP, 1999, Lisbon. **Proceedings...** Berlin: Springer-Verlag, 1999. p. 299-300. Disponível em: <http://www.yy.cs.keio.ac.jp/~suzuki/project/pub/ecoop99.pdf.zip>. Acesso em: nov. 2001.

[TAR 99] TARR, P.; OSSHER, H.; HARRISON, W.; SUTTON, Jr. S.M. N Degrees of Separation: Multi-Dimensional Separation of Concerns. In: INTERNATIONAL CONFERENCE ON SOFTWARE ENGINEERING, ICSE, 21., 1999, Los Angeles. **Proceedings...** New York: ACM Press, 1999. p. 107-119.

[TRA 99] TRAN, Q.; CHUNG, L. Tool Support for Dealing with Non-Functional Requirements. In: APPLICATION-SPECIFIC SYSTEMS AND SOFTWARE ENGINEERING TECHNOLOGY, ASSET, 1999, Dallas. **Proceedings...** [S.l. : s.n.], 1999. Disponível em: <http://www.utdallas.edu/~chung/ftp/tran.ps>. Acesso em: fev. 2002.

[TUR 98] TURNER, C. R.; FUGGETTA, A.; LAVAZZA, L.; WOLF, A. L. Feature Engineering. In: INTERNATIONAL WORKSHOP ON SOFTWARE SPECIFICATION AND DESIGN, 9., 1998, Ise-Shima. **Proceedings...** Los Alamitos: IEEE Computer Society Press, 1998. p. 162–164.

[VRA 2001] VRANIC, V. Multiple software development paradigms and multi-paradigm software development, In: INFORMATION SYSTEMS MODELLING, 2001, Czech Republic. **Proceedings...** [S.l. : s.n.], 2001. p. 191-196. Disponível em: <citeseer.nj.nec.com/499157.html>. Acesso em: jul. 2002.

[WAL 2002] WALKERDINE, J. et al. Dependability Properties of P2P Architectures. In: PEER TO PEER, P2P, 2002, Linkoping. **Proceedings...** Los Alamitos: IEEE Computer Society Press, 2002. p.173-174.

[WIC 99] WICHMAN, J. C. **ComposeJ:** The Development of a Preprocessor to Facilitate Composition Filters in the Java Language. 1999. Dissertação (Mestrado) - University of Twente, Twente. Disponível em: <http://www.ccs.neu.edu/home/dougo/thesis/020313/proposal.ps>. Acesso em jan. 2001.

[WIE 99] WIEGERS, K. **Software Requirements**. New York: Microsoft Press, 1999.

[WIN 2003] WIN, B. D.; JOOSEN, W.; PIESSENS, F. AOSD & Security: a practical assessment. In: WORKSHOP ON SOFTWARE ENGINEERING PROPERTIES OF LANGUAGES FOR ASPECT TECHNOLOGIES, SPLAT, 2003, Boston. **Proceedings...** [S.l. : s.n.], 2003. p. 1-6.

[WIN 2001] WIN, B. D.; VANHAUTE, B.; DECKER, B. D. Security Through Aspect-Oriented Programming. In: **Advances in Network and Distributed Systems Security**. [S. l.]: Kluwer Publishing, 2001. P. 125-138.

[YEH 84] YEH, R. et al. **Software Requirements: New Directions and Perspectives**. London: Prentice-Hall, 1984.

[ZAK 2002] ZAKARIA, A., A.; HOSNY, H.; ZEID, A. A UML Extension for Modeling Aspect-Oriented Systems. In: INTERNATIONAL WORKSHOP ON ASPECT-ORIENTED MODELING WITH UML, AOM, 2003, Boston. **Proceedings...** Boston: [s.n.], 2003. Disponível em: <http://lglwww.epfl.ch/workshops/uml2002/papers/zakaria.pdf>. Acesso em: maio 2003.

ANEXO A - BNF PARA ASPECTOS

A BNF é um formalismo extensamente aceito para a definição formal da sintaxe, de modo que a definição fique precisa e sem ambigüidades [NAU 69]. Assim, após analisar as definições encontradas na literatura [HIG 99, KIC 2001, LAD 2002b, LAD 2003, LAF 2003] e utilizar-se da lista de discussão brasileira [ABR 2003] sobre AOP foi definida uma BNF que estabelece algumas regras para a definição de um vocabulário comum na área de AOSD (Figura A1.1).

separation_of_concerns ::= separação de <concerns>
concerns ::= sem tradução | interesses | conceitos | preocupações | responsabilidades
scattered ::= espalhado | disperso
tangled ::= entrelaçado | emaranhado | mesclado
crosscutting_concerns ::= <concerns> (multi-dimensionais | ortogonais | transversais | entrelaçados | sobrepostos | dispersos) | entrelaçamentos
aspect ::= aspecto
joinpoint ::= ponto de (combinação | composição | junção)
pointcut ::= ponto de atuação | transversal | ponto de corte
advice ::= sem tradução
introduction ::= propriedades ortogonais estáticas
aspect_weaver ::= (combinador | compilador | entrelaçador | processador) de aspectos

Figura A.1: Vocabulário para AOSD

ANEXO B - AOSD EM ASPECTJ

Na literatura, é possível encontrar diversos trabalhos [ASJ 2002, HIG 99, KIC 2001, LAD 2002b, LAD 2003, LAF 2003] que apresentam a sintaxe de AspectJ, bem como exemplos. Nesse sentido, resolveu-se criar este anexo cujo objetivo principal é apresentar os principais elementos de AspectJ e suas respectivas semânticas.

AB.1 Aspecto (*aspect*)

Em AspectJ um aspecto é similar a uma classe, pois é composto por diversos membros: atributos, métodos, pontos de corte e *advices*.

Todo aspecto tem um modificador de acesso (determina sua visibilidade) e um nome que descreve seu objetivo. Para realizar a sua declaração utiliza-se a palavra chave `aspect`. A sintaxe para a definição de um aspecto em AspectJ é esquematizada pela Figura B.1.

```
[modificador_acesso] aspect [nome_aspecto]
[extends classe_ou_aspect_abstrato]
[implements lista_interfaces]
[<especificador_associação>(pointcut)]{
        // ... corpo do Aspecto
}
```

Figura B.1: Definição de Aspectos em AspectJ

A partir da sintaxe apresentada na Figura A2.1 e de alguns trabalhos encontrados na literatura [ELR 2001a, ELR 2001b, HIG 99, KIC 2001, LAD 2002b, LAD 2003] pode-se afirmar um aspecto caracteriza-se por:

(i) herdar de outras classes ou outros aspectos (somente dos abstratos);
(ii) implementar interfaces;
(iii) podem ser abstratos, neste caso os pontos de combinação não são ativados;
(iv) podem possuir atributos e métodos; e
(v) não podem ser instanciados diretamente.

O corpo de um aspecto é usado para a declaração e implementação de seus membros. Para a definição de atributos e métodos deve-se adotar a mesma sintaxe e semântica do modelo orientado a objetos. Já os pontos de corte e *advices,* elementos pertencentes somente ao modelo orientado a aspectos, possuem uma sintaxe e semântica próprias.

AB.2 Pontos de Combinação (*joinpoints*)

Os pontos de combinação compreendem pontos relevantes na execução de um programa. Em AspectJ os pontos de combinação encontrados são todos dinâmicos. Os pontos de combinaçãos podem ser classificados em três categorias distintas [KIC 2001, LAD 2003, LAF 2003] conforme esquematiza a Tabela B.1.

Tabela B.1: Categorias e Tipos de Pontos de Combinação

Categoria	Tipo	Descrição
Execução	Execução de método	inserção de ações entrelaçadas no corpo do próprio método.
	Execução de construtor	inserção de ações entrelaçadas no corpo do próprio construtor.
	Execução de inicializador estático	representa a carga da classe pelo carregador de classes (*class loader*) incluindo a inicialização da parte estática.
	Manipulador de exceção	usado para obrigar que certas políticas de tratamento de exceções sejam respeitadas.
	Inicialização de objeto	usado para capturar a inicialização de um objeto desde a chamada implícita/explícita ao construtor da superclasse até o término da execução do construtor chamado.
Chamada	Chamada de método	relacionado a outras partes do programa (geralmente outros métodos) que estão ativando (chamando) método.
	Chamada de construtor	relacionado a outras partes do programa (geralmente outros construtores) que estão ativando (chamando) construtor.
	Pré-inicialização de objeto	pré-inicialização de objeto: é usado quando é necessário chamar o construtor da super classe de forma explícita, ou seja, com o auxílio da palavra reservada super.
Acesso a atributos	Referência a um atributo	usado para garantir que os objetos serão inicializados corretamente antes de seu uso.
	Atribuição de valor para um atributo	utilizado quando deseja-se obrigar que o valor de um atributo encontre-se em um limite de valores considerado válido.

Deve-se observar que usando-se AspectJ somente a leitura/escrita de atributos pode ser monitorado, ou seja, o acesso a variáveis locais não pode ser definido.

Para que um ponto de combinação seja, efetivamente, implementado ele deve ser associado a um ponto de corte (*pointcut*). Um ponto de corte [ASJ 2002, LAD 2002b, POP 2001] corresponde a uma coleção de pontos de combinação.

B.3 Pontos de Corte (*pointcut*)

Em AspectJ existem dois tipos de pontos de corte: (i) anônimo, assim como uma classe anônima, é definido no local em que será usado (parte de um advice ou dentro de outro ponto de corte); e (ii) rotulado, esse tipo de ponto de corte pode ser referenciado a partir de vários outros pontos no código, tornando-se assim uma unidade reutilizável.

Os pontos de corte rotulados são os mais usados para AOP e sua sintaxe é esquematizada na Figura A2.2. O nome do ponto de corte é apresentado à esquerda do sinal de pontuação dois-pontos (":"), enquanto que sua definição é descrita à direita.

```
pointcut nome_ponto_corte(args) : definição_ponto_corte
```

Figura B.2: Definição de Ponto de Corte em AspectJ

Convém observar que a definição de um ponto de corte é dada por um conjunto de pontos de combinação. AspectJ fornece alguns operadores para unir, em um mesmo ponto de corte, diversos pontos de combinação [ZAK 2002]:

- Operador && (*and*): causa a seleção de pontos de combinação unindo ambos pontos de corte;
- Operador || (*or*): causa a seleção de pontos de combinação unindo tanto um como o outro ponto de corte;
- Operador ! (*not*): causa a seleção de todos os pontos de combinação exceto os descritos pelo ponto de corte em questão.

Alguns exemplos de pontos de combinação que irão formar a definição de um ponto de corte, usando os operadores enumerados anteriormente, são descritos na Tabela B.2.

Tabela B.2: Exemplos de Pontos de Corte

Exemplo	Descrição
Conta	Classe Conta.
*Conta	Qualquer classe que Termine com Conta.
java.*.Date	Classe Date em qualquer subpacote.
javax..*	Qualquer classe dentro do pacote javax, ou de seus subpacotes diretos.
javax..*Model+	Qualquer classe dentro do pacote javax que possua pacotes, diretos ou indiretos, cujo nome termine em Model.
!String	Todas as classes exceto a classe String.
Button || JButton	Classe Button ou JButton.
public void Container.paint(Graphics g)	O método paint na classe Container que tem acesso público, não possui retorno e tem como argumento um objeto do tipo Graphics.
public void Component.set*(*)	Todos os métodos públicos na classe Component que iniciam com *set* e possuem um único argumento.
public void Component.*()	Todos os métodos públicos na classe Component que retornam void e não possuem argumentos.
public * Component.*()	Todos os métodos públicos na classe Component que retornam qualquer tipo e não possuem argumentos.
!public void Component.*()	Todos os métodos não públicos na classe Component que retornam void e não possuem argumentos.
public static void Teste.main(String args[])	O método de classe público main() na classe Teste.
* Component+.*(..)	Todos os métodos públicos na classe Component ou em suas subclasses.
* java.awt.Component.enable(..)	Qualquer método enable() na classe Component, independente os tipo de acesso e número de parâmetros.
* *.*(..) throws IOException	Qualquer método que propague uma IOException.
public Conta.new()	O construtor público sem parâmetros.
public Conta+.new(..)	Construtores públicos da classe e das subclasses de Conta.
public *Conta.new(..)	Qualquer construtor público das classes que terminam com Conta.
private double Conta.saldo	O atributo saldo da classe Conta.
* Conta.*	Todos os atributos da classe Conta, independente do tipo de acesso, do tipo de dado ou do nome.
!public static * meupacote..*.*	Todas as variáveis de classe do pacote meupacote e de seus subpacotes diretos ou indiretos.
public !final *.*	Todos os atributos não *final* de qualquer classe.

Para cada categoria de ponto de combinação existe um respectivo ponto de corte que captura a funcionalidade indicada. A sintaxe de cada um desses pontos de corte pode ser encontrada na Tabela B.3.

Tabela B.3: Categorias e Tipos de Pontos de Combinação

Categoria	Tipo	
Execução	Execução de método	execution(AssinaturaMétodo)
	Execução de construtor	execution(AssinaturaConstrutor)
	Execução de inicializador estático	staticinicialization(AssinaturaTipo)
	Manipulador de exceção	handler(AssinaturaTipo)
	Inicialização de objeto	initialization(AssinaturaConstrutor)
Chamada	Chamada de método	call(AssinaturaMétodo)
	Chamada de construtor	call(AssinaturaConstrutor)
	Pré-inicialização de objeto	preinitialization(AssinaturaConstrutor)
Acesso a atributos	Referência a um atributo	get(AssinaturaAtributo)
	Atribuição de valor para um atributo	set(AssinaturaAtributo)

No Código 1, é possível encontrar a definição de um ponto de corte, em AspectJ, relacionado com a chamada de construtores. Esse ponto de corte define que ele será ativado sempre que (i) qualquer construtor, (ii) em qualquer classe, (iii) com qualquer número de parâmetros for invocado.

(1) CÓDIGO

```
1.  public aspect Monitorar{
2.      pointcut callConstructor() : call(*.new(..));
3.       // *  = chamada de construtor de qualquer classe
4.       // .. = chamada de construtor com qualquer número de parâmetros
5.       // continua ....
6.  }
```

Existem outros tipos de ponto de corte, tais como *cflow*, *cflowbellow*, *within*, *withcode*, *target*, *this*, *args* e *if*. Todos eles possuem além de uma semântica uma sintaxe própria. Esses elementos não serão explorados neste trabalho, uma vez que não compreendem o foco do trabalho.

Após entender o mecanismo de definição de um ponto de corte o próximo passo consiste em determinar em quais momentos eles deverão ser ativados. Para essa finalidade, é necessário definir pelo menos um advice.

B.4 Advice

O *advice* é um elemento que define "o que fazer". Ele compreende um código, similar a um método, o qual propicia uma forma para expressar os entrelaçamentos (pontos de combinação) que são capturados em pontos de corte [ASJ 2002, LAD 2003].

Um *advice* pode pertencer a uma das seguintes categorias [ASJ 2002, KIC 97, POP 2001]: antes (*before*), depois (*after*) e simultaneamente (*around*) a execução de um ponto de combinação.

Essas categorias determinam o momento em que o ponto de combinação será executado. O *advice before* determina que será imediatamente antes do ponto de combinação especificado, enquanto o *after* especifica que será executado após, e o *around* que executará tanto antes quanto após [LAD 2002a, SAP 2002].

Por exemplo, para determinar as ações realizadas antes da execução de todos os construtores em uma aplicação seria necessário utilizar o *advice before*. Ou seja, seria determinado ao programa "execute este código antes da chamada do construtor", conforme descrito no fragmento de código apresentado pelo Código 2 – linhas 5 a 7.

(2)	CÓDIGO
1.	`Public aspect Monitoring{`
2.	`    pointcut callConstructor() : call(*.new(..));`
3.	`     // * = chamada de construtor de qualquer classe`
4.	`    // .. = chamada de construtor com qualquer número de parâmetros`
5.	`   before( ) : callConstructor(){`
6.	`                System.out.println("Advice = Antes de chamar o construtor");`
7.	`   }`
8.	`   after( ) : callConstructor(){`
9.	`                System.out.println("Advice = Após chamar o construtor");`
10.	`   }`
11.	`}`

Pode-se concluir, então, que um *advice* é um procedimento que está associado estaticamente a algum ponto de corte.

Além dos conceitos vistos nas seções anteriores, é possível encontrar outros conceitos relacionados com AOSD e AspectJ. Porém eles não serão apresentados neste trabalho. Informações mais detalhadas sobre todos os tópicos pertinentes a AspectJ e a AOSD pode podem ser encontrados em [ASJ 2002, ELR 2001a, HIG 99, KIC 2001, LAD 2003, LAD2002a, LAD2002b, LAF 2003].

ANEXO C - CHECKLIST DEFINIDA

Conforme indicado no capítulo 6 (seção 6.3) foram definidas *checklists* para três áreas pertencentes ao domínio de tolerância a falhas: desempenho, confiabilidade e segurança. Para a construção de cada *checklist* foi realizada uma análise detalhada da literatura [ALB 2002, AVI 2001, LAP 92, LAP 95, LEM 99, LEM 00, RAN 98] chegando-se às questões descritas a seguir.

C.1 Checklist para Desempenho

Esta *checklist* aborda questões relativas ao desempenho de aplicações, tais como latência, capacidade de processamento, capacidade de carga, modos e recursos para operação. A Tabela A3.1 apresenta as principais indagações encontradas na literatura relacionadas com esses itens.

Tabela C.1: Checklist para Desempenho

	R	P	Restrição
Desempenho			
Latência			
Existe um intervalo de tempo durante o qual a resposta de qualquer evento deve ser executada ?	❍		
Capacidade de Processamento			
Existe alguma restrição para capturar, validar ou armazenar dados/transações ?	❍		
Existe alguma restrição na capacidade de processamento ?	❍		
Capacidade			
Existe algum limite na aplicação em termos de número de transações/clientes/usuários simultâneos ?	❍		
Há alguma restrição quanto ao número de clientes por item de dado ?	❍		
Há alguma restrição ao tamanho (bytes) dos dados ?	❍		
Modos			
Se houver a possibilidade de parte dos recursos ficarem pendentes o sistema deverá atuar com capacidade reduzida ?	❍		
Caso ocorra sobrecarga no sistema será necessário penalizar alguns requisitos relacionados com o tempo ?	❍		
Recursos			
Existem restrições com relação à memória, disco (HD), processador, comunicação via rede ?	❍		
Existem restrições relacionadas com qualquer dispositivo de hardware, tanto de entrada quanto de saída ?	❍		

C.2 Checklist para Confiabilidade

A *checklist* descrita na Tabela C.2 apresenta algumas questões essenciais para o desenvolvimento de sistemas confiáveis, levando em consideração itens como disponibilidade, confiabilidade e segurança[31].

Tabela C.2: Checklist para Confiabilidade

	R	P	Restrição
Confiabilidade			
Disponibilidade			
Existe uma percentagem do tempo em que o sistema deve estar disponível ?	❍		
Existem condições em que o sistema ficará indisponível ?	❍		
Confiabilidade			
É necessário o uso de algum mecanismo de redundância ?	❍		
Existe a possibilidade de se utilizar alguma técnica de tolerância a falhas ?	❍		
O sistema deve apresentar tratamento/prevenção/tolerância a falhas ?	❍		
O sistema deve apresentar tratamento/prevenção/tolerância a erros ?	❍		
Em algum momento deve-se pensar em restaurar/recuperar o estado do sistema ?	❍		
É necessário que os dados sejam replicados ?	❍		
Segurança			
Caso ocorra algum erro/falha no sistema pode ocasionar a perda de dados fundamentais para a sobrevivência da organização ?	❍		
Caso ocorra algum erro/falha no sistema pode ocasionar a perda de vidas humanas ?	❍		
Caso ocorra algum erro/falha no sistema pode ocasionar a perda de recursos financeiros?	❍		

[31] Segurança relacionada a falhas acidentais – do inglês safety

C.3 Checklist para Segurança

Outra *checklist* desenvolvida para este trabalho é a que engloba os atributos fundamentais relacionados com segurança[32] de software. Ela aborda questões que envolvem integridade, autorização, autenticação, auditoria, etc., como destaca a Tabela C.3.

Tabela C.3: Checklist para Segurança

	R	P	Restrição
Segurança			
Integridade			
A modificação em qualquer dado é permitida somente para usuários autorizados ?	❍		
O sistema deve ser protegido de falhas intencionais[33] ?	❍		
Confidencialidade			
Para acessar o sistema o usuário deve estar autorizado ?	❍		
O sistema/serviço/informação deve estar protegido contra o acesso de intrusos ?	❍		
O sistema deve oferecer controle de acesso (login/logout – logon/logoff) ?	❍		
É necessário determinar a real identidade do usuário logado ?	❍		
É necessário registrar as atividades/ações do usuário no sistema ?	❍		
Existe a necessidade de cifrar algum dado do sistema ?	❍		
O sistema deve realizar algum controle sobre as ações que são utilizadas ?	❍		

[32] Segurança relacionada com falhas intencionais – do inglês *security*
[33] Por exemplo, vírus, hackers, cavalos-de-tróia, etc.

ANEXO D - LRNFS

Conforme mencionado neste trabalho, o léxico é um dos artefatos fundamentais do método FRIDA. Esse léxico, diferente dos demais encontrados na literatura, compreende um conjunto de RNFs bem conhecidos, os quais foram organizados e encontram-se descritos por regras da BNF. Observa-se que como o relacionamento do léxico com os RNFs é altamente dependente, desse modo resolveu-se denominá-lo de LRNFs – ***L****éxico de* ***R****equisitos* ***N****ão* ***F****uncionais*.

D.1 Entrada do Léxico

Neste trabalho os RNFs foram classificados em três domínios genéricos: desempenho, segurança e confiabilidade. Para a construção do léxico destes domínios alguns trabalhos foram analisados [ALB 2002, AVI 2001, LAP 92, LAP 95, LEM 99, LEM 2000, RAN 98] e a partir deles a entrada do léxico foi defina (Figura D.1).

<RNF_generico> ::= <desempenho> | <segurança> | <confiabilidade>
<desempenho> ::= <latência> | <cap_de_processamento> | <capacidade> | <modos> | <recursos>
<segurança> ::= <confidencialidade> | <integridade>
<confiabilidade> ::= <impedimentos> | <atributos> | <meios> | <disponibilidade>

Figura D.1: BNF para RNFs Genéricos

D.2 Léxico para o Contexto Desempenho

O desempenho (Figura D.2), ou eficiência, de um sistema é determinado comparando-se o nível de desempenho do software e a quantidade de recursos utilizados sob determinadas condições. Geralmente, ele encontra-se relacionado a recursos de hardware, capacidade de processamento ou de carga, entre outros.

<desempenho> ::= <latência> | <capacidade_de_processamento> | <capacidade> | <modos> | <recursos>
<latência> ::= tempo de resposta | em <n> <unidade_tempo> | em <n> velocidade
<cap_de_processamento> ::= <n>ciclos por <unidade_tempo> | <n>operações por <unidade_tempo>
<capacidade> ::= tamanho <unidade_capacidade> | <unidade_capacidade> número | <unidade_capacidade> utilização
<modos> ::= capacidade reduzida | sobrecarga
<recursos> ::= processador | comunicação de rede | barramento | memória | disco | RAM |<de> | <ds>
<unidade_tempo> ::= ms | milisegundos| segundos | hora | dia | mês | ano | semana
<unidade_capacidade> ::= máximo | mínimo | menos | mais | menor | maior
<de> ::= mouse | teclado | mesa digitalizadora | microfone | touchscreen
<ds > ::= crt | monitor | impressora | plotadora | alto-falante
<n> ::= <n> | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

Figura D.2: BNF para RNFs Relacionados com Desempenho

A4.3 Léxico para o Contexto Confiabilidade

Vários autores [AVI 2001, LAP 92, LAP 95, LEM 99, LEM 2000, RAN 98] apresentam uma classificação para os elementos envolvidos com a confiabilidade[34] de um sistema. A partir destes trabalhos foi construído um léxico voltado ao domínio da confiabilidade, o qual é ilustrado na Figura D.3.

<confiabilidade> ::= <impedimentos> | <meios> | <atributos>
<impedimentos> ::= fallha | erro | defeito
<meios> :: = <meios_falha> falha | <tolerância_falhas>
<meios_falha> ::= prevenção | supressão | previsão | prevenção | diagnóstico | tratamento
< **tolerância_falhas** > ::= replicação | recuperação | ponto de salvaguarda | restauração | credibilidade | severidade | compensação | fail <fail> | <sistema> do sistema | <erro> de erro | <serviço> de um serviço | serviço correto
<fail> ::= -safe | -salient | -stop
<sistema> ::= comportamento | componente | criticalidade | integridade | organização | usuário | especificação | estrutura | estado
<erro> ::= compensação | contenção | detecção | mascaramento | processamento | recuperação
<serviço> ::= execução | interrupção | fornecimento | restauração | especificação | credibilidade
<atributos> ::= <disponibilidade> | <confiabilidade> | <segurança>
<disponiblidade> ::= <modulo> <n> <unidade_tempo>/<unidade_tempo>indisponível | <n>%do tempo disponível | disponível <n> X <n> <unidade_tempo>
<confiabilidade> ::= <unidade_recuperação> | <unidade_recuperação> em <unit_tempo> | <impedimentos> <unidade_recuperação> em <unidade_capacidade> em <unidade_tempo> | <unidade_recuperação> em <unidade_capacidade> em <unidade_tempo>
<unidade_recuperação> ::= recuperação | restauração | cópia | resposta
<segurança> ::= acidente | infortúnio | morte | dano | catástrofe | prejuízo | risco

Figura D.3: BNF para RNFs Relacionados com Confiabilidade

D.4 Léxico para o Contexto Segurança

Segundo Avizienis [AVI 2001], além dos atributos de confiabilidade descritos anteriormente, é possível enquadrar a segurança como um atributo necessário para se desenvolver um sistema confiável. Nesse sentido, foi construído um léxico para segurança, o qual encontra-se dividido em dois subconjuntos: confidencialidade e integridade (Figura D.4).

<segurança> ::= <confidencialidade> | <integridade>
<confidencialidade>::= autoriza<sufx> | autentica<pos> | controle acesso | cifrar| auditoria | contabilidade | login | logon | logout | logoff
<pos> ::= r | do | ção
<integridade>::= <unidade_unsec> <afeta> | <unidade_unsec> <afeta> <modulo> | <unidade_sec> <afeta> | <unidade_sec> <afeta> <modulo>
<unidade_unsec> ::= negado | prejuízo | não autorizado | ataque | vírus | dano | intrusão | verme | não identificado | malicioso | maligno
<unidade_sec> ::= protegido | autoriza<pos> | | secreto | privilégio | confidencial | privacidade | aprovar
<afeta> ::= apagar | excluir | eliminar | alterar | melhorar | aperfeiçoar | modificar
<modulo> ::= método | função | classe | processo | procedimento | dado | produto | sistema | aplicação | informação

Figura D.4: BNF para RNFs Relacionados com Segurança

[34] do inglês *dependability* – siginificando o quanto é possível confiar no funcionamento do sistema.

ANEXO E - MATRIZ E REGRAS

A identificação e resolução de conflitos é muito importante na definição de um sistema baseado em RNFs. Logo, em FRIDA essa é uma etapa tão importante que foram definidos dois artefatos para a identificação de conflitos: regras e matriz de conflitos, conforme apresentam as próximas seções.

E.1 Regras para Adição de Novos Conflitos

As regras, descritas na Figura A5.1, definem que dois requisitos, por exemplo r_1 e r_2, encontram-se em conflito quando alterações em r_1 causam impacto em r_2. Por exemplo, quanto maiores as restrições de segurança de um sistema maior será o tempo de resposta para o usuário. Ou ainda, quanto menor o nível de confiabilidade pode-se afirmar que menor será o nível de segurança. Caso o nível de confidencialidade do sistema seja muito alto a performance sofrerá um impacto negativo; por outro lado a confiabilidade será maximizada.

(1) alteração: $V \rightarrow V$
alteração (x) = x + ε
onde: V é o conjunto de valores,
$x \in V$ e $x \neq \perp$,
$\varepsilon \in V$ e $x \neq \perp$,
<v, +, ⊥> é um grupo

(2) r_i altera ⇔ alteração (x_{ri})
onde: r_i é o requisito *i*,
x_{ri} é o valor agregado,
alteração é definida em (1)

(3) r_i impacta r_j ⇔ ((r_i altera) ⇒ (r_j altera))
onde: $r_i \neq r_j$,
altera é definida em (2)

(4) r_i conflitante r_j ⇔ r_i impacta r_j
onde: $r_i \neq r_j$,
impacta é definida em (3)

Figura E.1: RNFs Conflitantes: regras

Em resumo, essas regras são adotadas para aprovar ou rejeitar conflitos entre RNFs. Para facilitar a compreensão desses conflitos resolveu-se utilizar uma matriz de conflitos [BOE 96].

A5.2 Matriz de Conflitos

A matriz de conflitos (Tabela A5.1) é uma forma de estabelecer os relacionamentos conflitantes entre RNFs. A análise da matriz deve ser realizada conforme indicado pelos seguintes passos:

- Passo 1: cada linha é, necessariamente, combinada com cada coluna;
- Passo 2: para cada combinação, anteriormente, identificada aplicar as regras; e
- Passo 3: analisar a resposta resultante da aplicação das regras. Se a resposta de todas as regras é verdadeira, então um conflito é identificado. Caso contrário os RNFs não são conflitantes.

Tabela E.1: Conflitos Parciais entre RNFs

	Latência	Cap.Process.	Capacidade	Modos	Recursos	Confidencialidade	Integridade	Availability	Reliability	Safety
Latência								✓	✓	✓
Cap.Process.	✓		✓	✓				✓	✓	✓
Capacidade	✓	✓		✓				✓	✓	✓
Modos	✓	✓	✓			✓	✓	✓	✓	✓
Recursos	✓	✓	✓	✓		✓	✓	✓	✓	✓
Confidencialidade	✓	✓	✓	✓			✓		✓	✓
Integrity	✓	✓	✓	✓		✓		✓	✓	✓
Availability									✓	✓
Reliability	✓	✓	✓	✓	✓			✓		✓
Safety	✓	✓	✓	✓	✓			✓	✓	

Nota-se que essa matriz apresenta um conjunto limitado de conflitos[35], mas nada impede que ela sofra atualização quando o analista achar conveniente. Isso pode ocorrer principalmente quando o conhecimento do analista, sobre o domínio do problema, permite que este estabeleça outros conflitos antes não identificados. Deve-se observar que para a adição de novos conflitos na base de conhecimento, é obrigatório o uso das regras de semântica definidas na seção A5.1. Caso isso não seja levado em consideração pode-se levar a base para um estado inconsistente.

[35] Em Chung [CHU 00] é possível encontrar um catálogo com alguns conflitos existentes.

ANEXO F - ESTEREÓTIPOS ATRAVÉS DE CORES

F.1 A importância no Uso de Cores

O uso de estereótipos baseados em descrições textuais é muito importante, visto que possibilita associar restrições ou adicionar novos elementos na UML. Porém, no método FRIDA somente o uso desse recurso não foi suficiente, porque a visibilidade dos elementos estereotipados não ficava muito clara e em algumas vezes tornava-se até confusa.

Optou-se então, pelo uso agregado de descrições textuais e cores. O uso de cores permite representar relações simbólicas, chamar e direcionar a atenção do desenvolvedor e também enfatizar a importância de alguns elementos estruturais. Percebeu-se que não bastava enfatizar apenas as palavras, mas o elemento estrutural como um todo.

Além disso, a cor é um elemento fundamental em qualquer processo de comunicação e pode portanto ser adotada para atingir determinados objetivos [FER 99, NIE 2002].

F.2 As Cores Selecionadas

Basicamente, as cores foram selecionadas com o objetivo de auxiliar na identificação de estruturas, que de alguma forma relacionam-se com o método FRIDA:

(i) Diagrama de casos de uso:

Elemento estereotipado: caso de uso

Cor: vermelha

Justificativa da seleção da cor – abordagem teórica: segundo Nielsen [NIE 2002] a cor vermelha é uma das que possui maior impacto no usuário, pois possui uma conotação de sangue e fogo. Ela demonstra-se eficiente quando se deseja chamar a atenção para algum elemento, ou para advertir algum problema [FER 99].

Justificativa da seleção da cor – abordagem prática: no método FRIDA o uso da cor vermelha consiste no destaque que se pretende dar aos casos de uso que possuem RNFs associados, mas não elicitados. O objetivo principal é advertir o analista de que ele pode estar esquecendo de algum requisito importante para a aplicação.

(ii) Diagrama de classes

Elemento estereotipado: classe no diagrama de classes (aspecto)

Cor: amarela

Justificativa da seleção da cor – abordagem teórica: é uma cor incandescente e encontra-se, geralmente, associada com o sol [FER 99, NIE 2002]. Como essa cor constitui-se na "matiz mais clara" do sistema RGB, é ideal para indicar atividade, ou seja, é recomendada para indicar se um elemento está ativo ou não.

Justificativa da seleção da cor – abordagem prática: como no método FRIDA o foco são os RNFs e os aspectos são usados para modelar esses requisitos resolveu-se adotar a cor amarela, na tentativa de demonstrar que esses elementos devem ser sempre monitorados e acompanhados, ou seja sempre devem estar ativos na mente do projetista.

ANEXO G – CÓDIGO GERADO

A construção de um software é uma atividade essencial no processo de desenvolvimento, porque sem ela não há software propriamente dito. Existem diversos mecanismos que podem auxiliar o desenvolvedor, durante a fase de codificação, entre os quais pode-se destacar a geração de código.

No método FRIDA a geração automática de código foi desenvolvida com a finalidade de automatizar parte da fase de implementação no ciclo de vida do software. Basicamente, o protótipo FRIDA propicia a geração de código de três elementos estruturais: interfaces, classes e aspectos.

G.1 Código Gerado: interface

Na linguagem Java, o termo interface é usado para caracterizar classes abstratas, que permitem apenas a declaração de dados e de métodos. Interfaces não podem ser instanciadas; elas devem ser implementadas por uma classe que se responsabiliza pela codificação dos métodos declarados na interface.

No protótipo FRIDA, uma interface é criada dentro de um diagrama de classes, conforme esquematiza a Figura G.1.

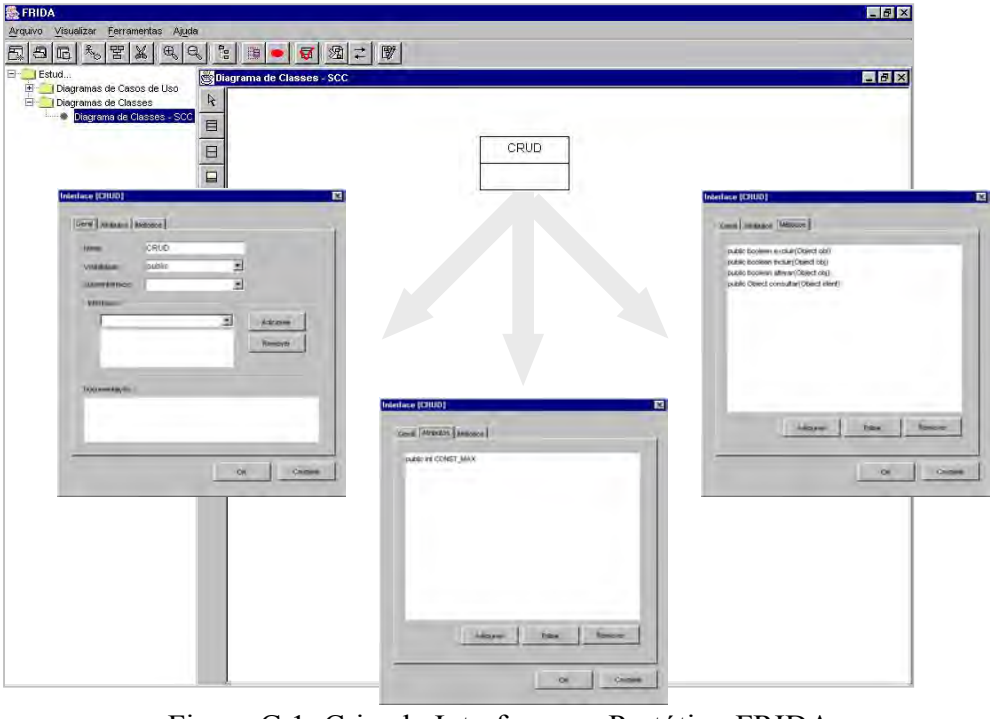

Figura G.1: Criando Interfaces no Protótipo FRIDA

A partir das informações introduzidas na ferramenta é possível gerar o código correspondente a uma `interface` na linguagem Java. O código correspondente as descrições ilustradas na Figura G.1 é esquematizado no Código 1.

(1)	CÓDIGO

```
1.  /**
2.    *Interface: CRUD
3.    *@see
4.    *author
5.  */
6.  public interface CRUD{
7.          // Atributos Gerados
8.         public int CONST_MAX=0;
9.
10.         // Métodos Gerados
11.        public boolean incluir(Object obj);
12.        public Object consultar(Object ident);
13.        public boolean excluir(Object obj);
14.        public boolean alterar(Object obj);
15. }
```

G.2 Código Gerado: classe

Uma classe encapsula definições de tipos de dados e o conjunto de operações julgadas pertinentes, importantes e úteis para todas as suas instâncias.

Para definir uma classe utilizando-se o protótipo FRIDA deve-se criar em primeiro lugar um diagrama de classes, para após poder criar a classe. Toda classe é constituída de diversas informações, as quais encontram-se esquematizadas na Figura A7.3.

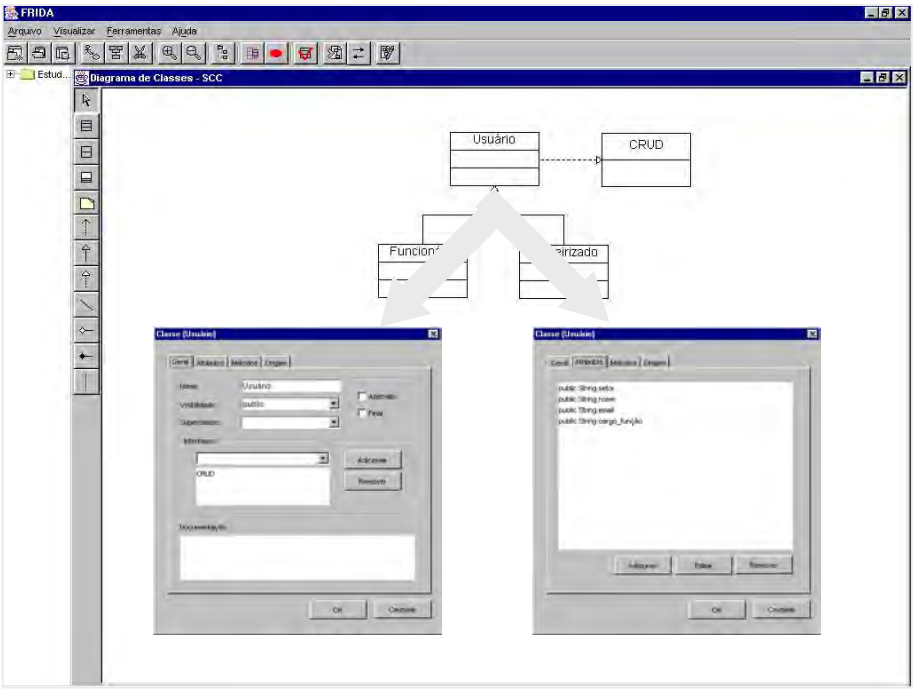

Figura G.2: Criando Classes no Protótipo FRIDA

Caso a classe implemente alguma interface, previamente definida no diagrama de classes do protótipo, esses métodos serão gerados automaticamente (Código 2 – linhas 14 a 18). Caso algum atributo possua escopo privado o gerador irá gerar, automaticamente, um par de métodos get/set para o atributo em questão (Código 2 – linhas 19 e 20).

(2)	CÓDIGO
1.	/**
2.	*Classe: Usuário
3.	*@see
4.	*author
5.	*/
6.	
7.	public class Usuário implements CRUD{
8.	// Atributos Gerados
9.	public String setor;
10.	public String nome;
11.	private String email;
12.	public String cargo_função;
13.	
14.	// Métodos Gerados
15.	public boolean incluir(Object obj){return true;}
16.	public Object consultar(Object ident){return null;}
17.	public boolean excluir(Object obj) {return true;}
18.	public boolean alterar(Object ident){return true;}
19.	public void setEmail(String nV) {email = nV;}
20.	public String getEmail() {return email;}
21.	}

G.3 Código Gerado: aspecto

No protótipo FRIDA um aspecto pode ser construído de duas formas: (i) explicitamente, onde o programador inclui o aspecto no diagrama de classes, bem como suas propriedades, e (ii) automaticamente. Neste último caso, o protótipo se encarrega de examinar as dependências do aspecto e gera seu código correspondente usando Java e AspectJ. A Figura G.3 ilustra a ativação do gerador de código embutido no protótipo FRIDA.

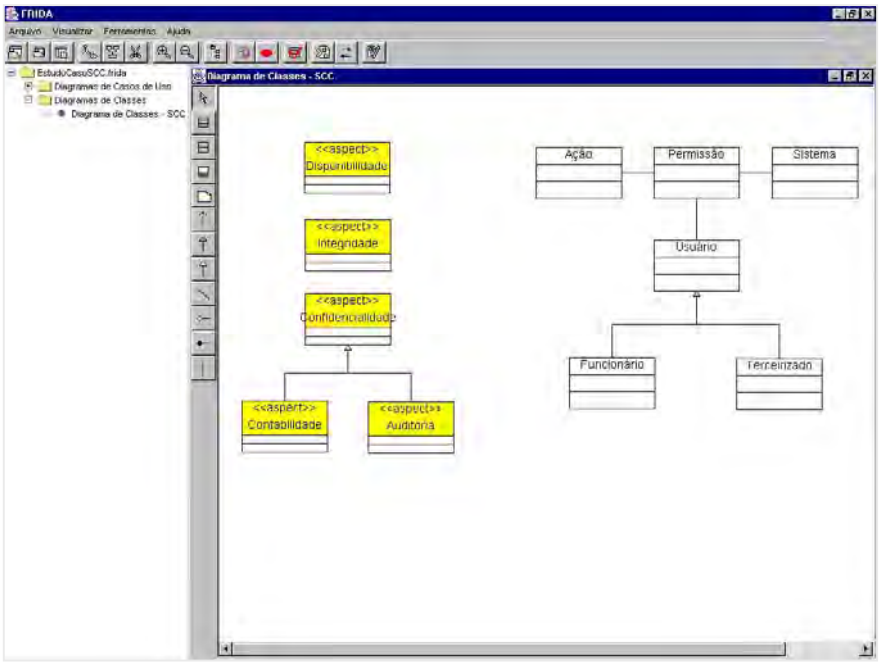

Figura G.3: Criando Aspectos no Protótipo FRIDA

O Código 3 ilustra o código gerado pelo protótipo FRIDA para o aspecto Auditoria, representado na Figura G.3. O primeiro elemento gerado é o aspecto `Confidencialidade` (linhas 17 a 19), o qual deve ser abstrato (aspectos podem herdar somente de outros aspectos abstratos). Outro elemento essencial para que o aspecto alcance seus objetivos é a definição da classe que irá conter o código não funcional (linhas 21 a 23), bem como a sua associação com o aspecto (linha 8). Sem esse vínculo os *advices* do aspecto não conseguem realizar nenhum tipo de implementação.

Os métodos gerados automaticamente são provenientes de duas fontes: (i) interfaces que o aspecto implementa (semelhante às classes) e (ii) restrições associadas com o aspecto, onde para cada restrição existe um respectivo método responsável por estabelecer de forma visível essa restrição.

(3)	CÓDIGO
1.	`/**`
2.	`  *Aspecto: Auditoria`
3.	`  *@see`
4.	`  *author`
5.	`*/`
6.	`public aspect Auditoria extends Confidencialidade{`
7.	`    //vincula com a classe que implementa a ação de Auditoria`
8.	`    Auditoria auditoria = new Auditoria();`
9.	
10.	`    //Métodos Gerados Automaticamente`
11.	`    //Método vinculado com a restrição 1 no template`
12.	`    public void metodoRestricao1(){`
13.	`       //Deve auditar informações relacionadas com o usuário, sistema e a ação`
14.	`    }`
15.	`}`
16.	
17.	`abstract aspect Confidencialidade{`
18.	
19.	`}`
20.	
21.	`class Auditoria{`
22.	`   //implementa as regras de Auditoria`
23.	`}`

ANEXO H - PROTÓTIPO FRIDA

H.1 Introdução

Fowler em [FOW 2000] demonstra claramente que a construção de um sistema deve adotar três perspectivas:

1. perspectiva conceitual: relacionada com os conceitos do domínio do problema, geralmente independente de linguagem de programação e de decisões arquitetônicas;
2. perspectiva de especificação: esquematiza um modelo preocupado com a implementação do problema, no qual as soluções de projeto devem ser aplicadas;
3. perspectiva de implementação: apresenta todos os detalhes das classes e demais artefatos relacionados ao projeto; nesta perspectiva o projeto é fortemente influenciado pela linguagem e pela arquitetura do sistema.

Desse modo, neste trabalho o protótipo FRIDA além de enfatizar os principais aspectos do método, também tenta contemplar essas perspectivas.

H.2 Interface Principal

A interface principal do protótipo encontra-se dividida em duas partes principais (Figura H.1): uma área destinada à visualização dos diagramas criados (Figura H.1 - ❶), e uma área de seleção e manipulação de componentes visuais (Figura H.1 - ❷). Esta última área é destinada à construção dos diagramas de casos de uso e/ou classes, bem como para a ativação dos artefatos pertencentes ao método FRIDA.

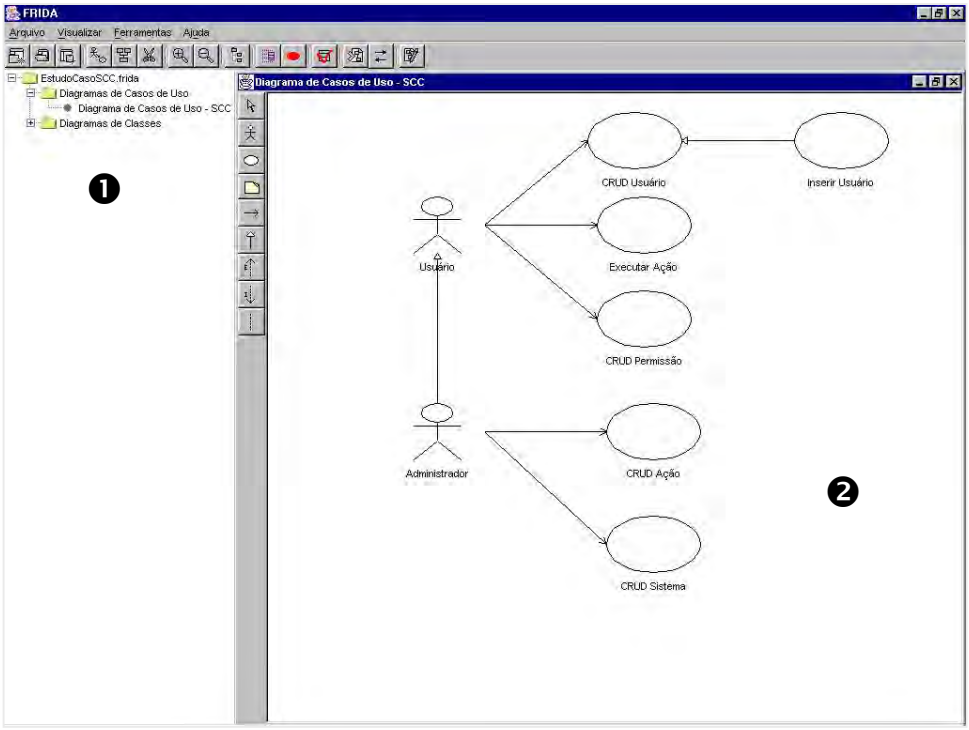

Figura H.1: Interface Principal

Conforme pode-se observar na Figura H.1 a tela principal do ambiente possui um conjunto de botões/menus que oferecem acesso às diversas funcionalidades disponibilizadas pelo método FRIDA (Tabela H.1).

	Manipulação de Projetos
	Ativar e construir diagrama de casos de uso
	Ativar e construir diagrama de classes
	Ativar a checklist para RNFs globais
	Ativar o processamento do léxico
	Exibir informações obtidas a partir do processamento do léxico
	Rastreabilidade da propagação das mudanças nos casos de uso
	Ativar criação de aspectos
	Ativar ligações entre classes e aspectos
	Geração de código

Tabela H.1: Ferramentas Disponibilizadas por FRIDA

H.3 Ativando o Desenhador de Casos de Uso

Para ativar o desenhador de casos de uso o analista seleciona o botão criar diagrama de caso de uso (Figura H.2 - ❶). Após, cada diagrama criado é adicionado a uma árvore que contém todos os diagramas que compõem a solução (Figura H.2 - ❷). Para desenhar o diagrama basta selecionar o elemento do diagrama de casos de uso desejado na barra de ferramentas (Figura H.2 - ❸).

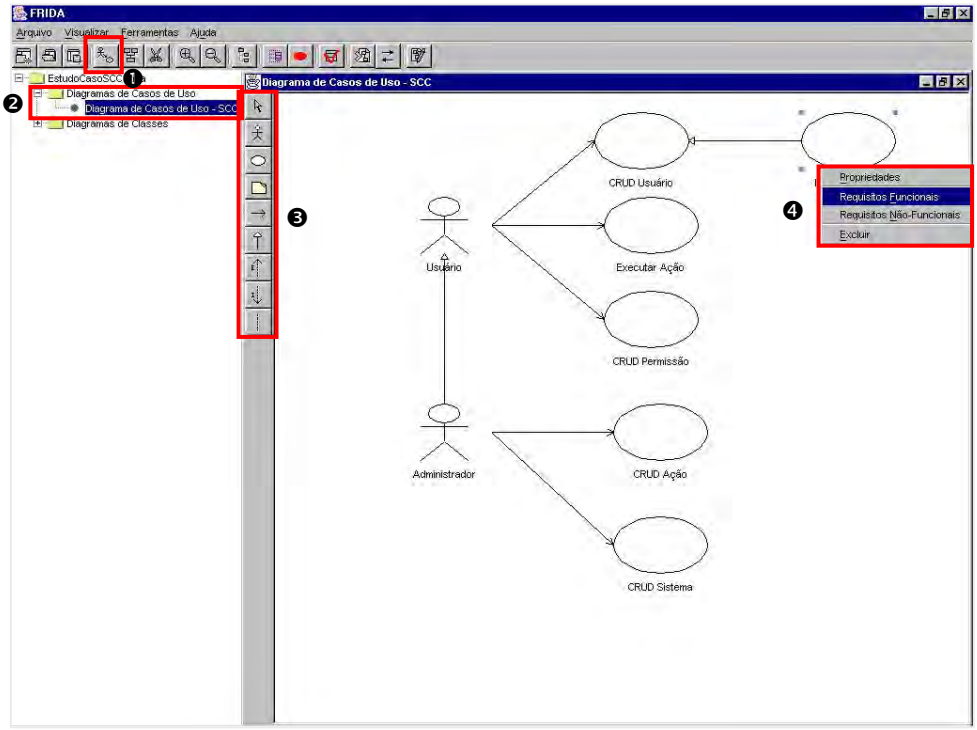

Figura H.2: Desenhador de Diagramas de Casos de Uso

Cada item do diagrama pode ser especificado em detalhes através dos botões existentes na barra de ferramentas. Por exemplo, para um ator é possível caracterizá-lo com um nome e descrevê-lo por meio de uma documentação, além de possibilitar sua exclusão do diagrama. No caso do item e relacionamento é permitido associar restrições ou excluir o mesmo do diagrama.

Quando o elemento do diagrama de casos de uso é um caso de uso o protótipo proporciona a sua caracterização (nome e documentação), bem como a associação de um *template* que descreve a funcionalidade (Figura A8.2 - ❹).

H.4 Ativando a Checklist

Para ativar a *checklist* deve-se analisar o nível do RNF que será elicitado. Caso o seu nível seja global a ativação ocorre através do item de interface ❶ da Figura H.3. Por outro lado, se o nível é parcial a *checklist* é exibida pressionando-se o item de interface ❷ da Figura H.3.

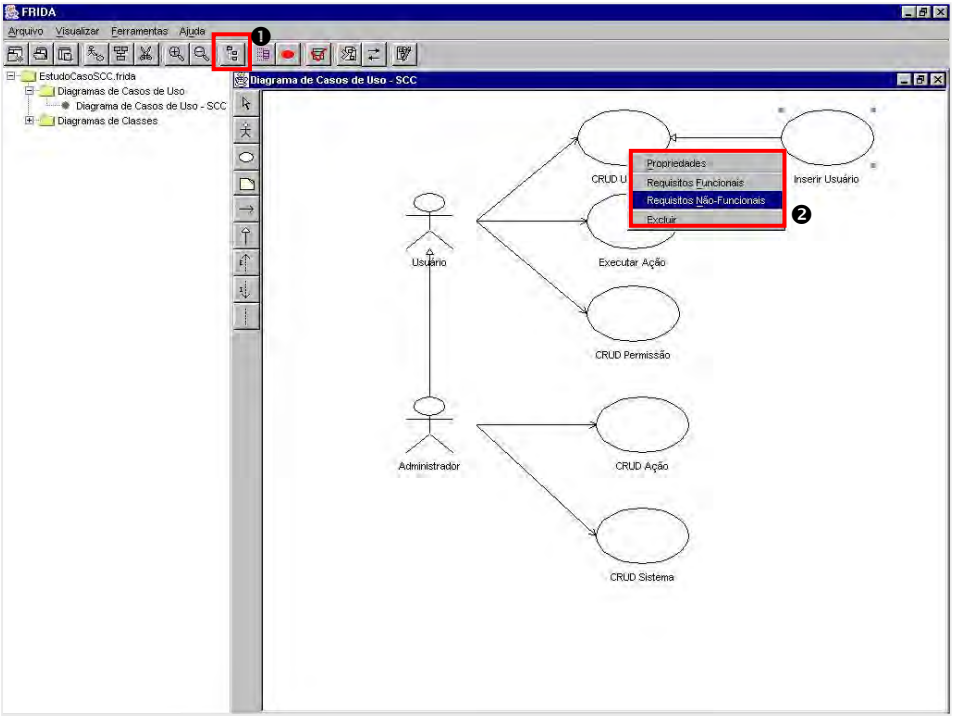

Figura H.3: Ativando a Checklist

H.5 Ativando o Léxico

A ativação do léxico ocorre usando-se o item de interface ❶ da Figura H.4. Neste momento o léxico é construído e seu processamento é desencadeado. Todo e qualquer caso de uso que apresente novos RNFs associados é transformado em um caso de uso estereotipado (Anexo 6).

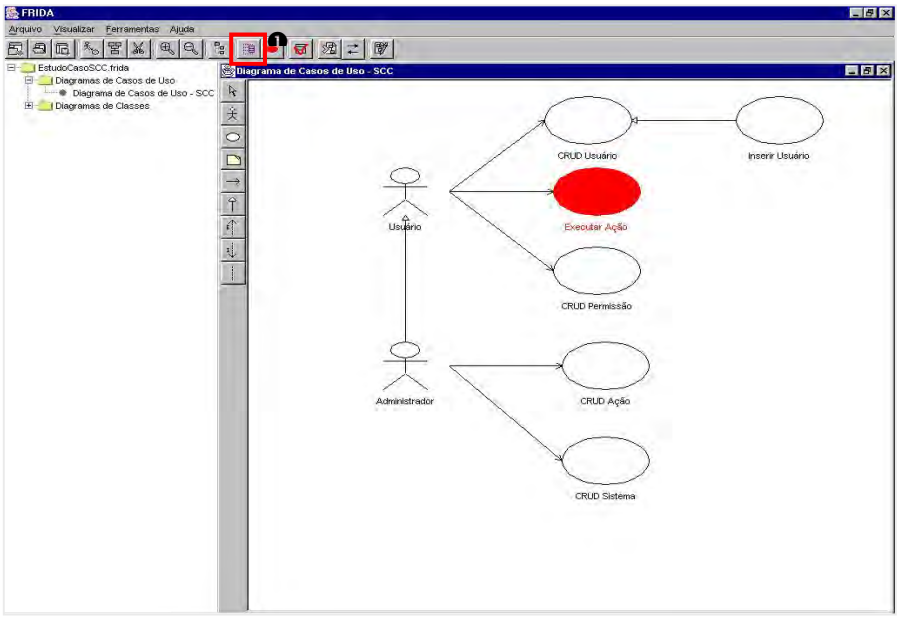

Figura H.4: Processando o Léxico

Para proceder com a verificação dos resultados deve-se ativar o item de interface ❶ da Figura H.5.

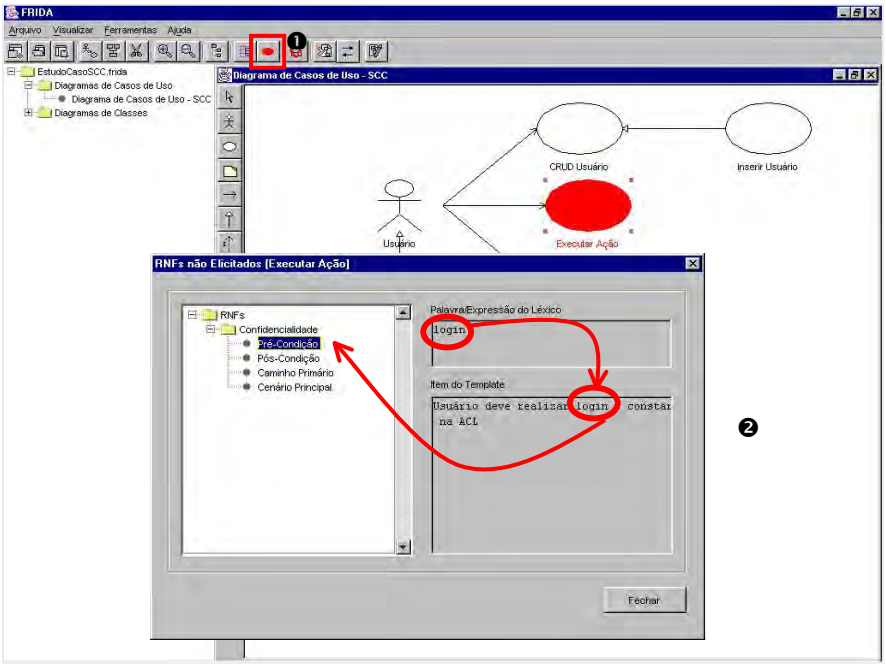

Figura H.5: Verificando Saída do Processamento do Léxico

Com base no exemplo apresentado o processamento do léxico gera como saída: (i) a lista de RNFs elicitados automaticamente; (ii) o item em que o RNF foi encontrado, no caso do exemplo no item pré-condição do *template* foi identificada a sentença "Usuário deve realizar login e constar na ACL"; (iii) no léxico existe a palavra *login* como pertencendo ao domínio da confidencialidade; (iv) como existe uma combinação possível é gerado um caso de uso estereotipado, caso contrário o caso de uso permaneceria conforme determinam as regras da UML.

H.6 Ativando e Criando o Diagrama de Classes

O projetista pode ativar do desenhador de diagramas de classe como ilustra a Figura H.6 - ❶. Após, a criação do diagrama ele é incluído no projeto atual, ou seja ele é adicionado a uma árvore que contém todos os diagramas da solução (Figura H.6 - ❷). Para desenhar o diagrama basta selecionar o qualquer elemento válido (Figura H.6 - ❸).

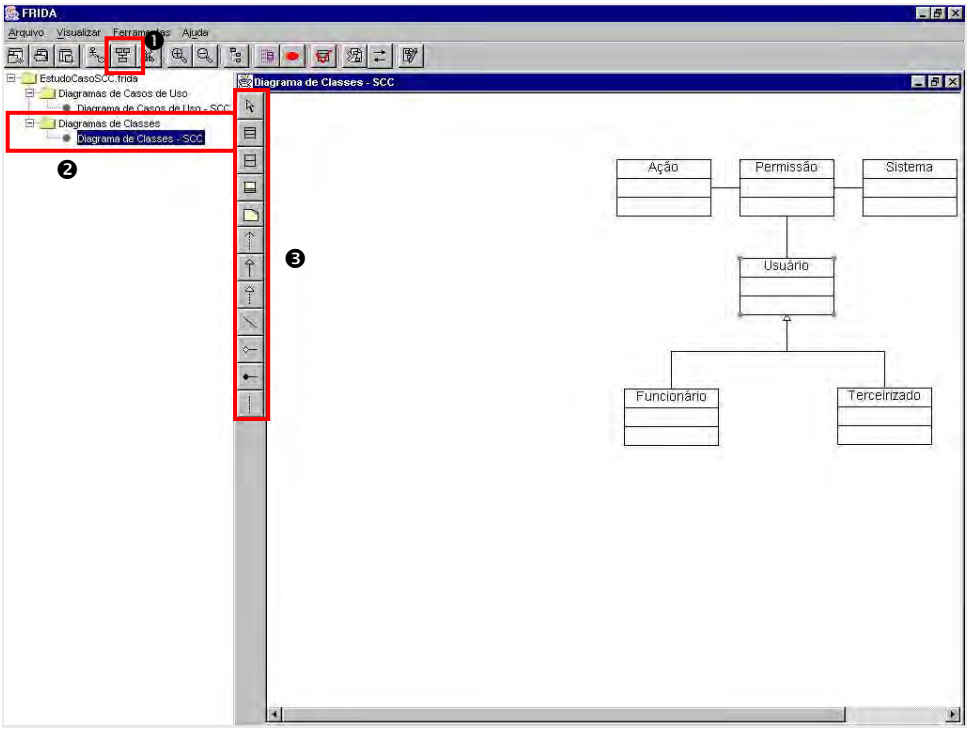

Figura H.6: Ativando e Criando um Diagrama de Classes

Deve-se observar que no protótipo desenvolvido é possível criar alguns tipos de elementos estruturais: classes, interfaces e aspectos, bem como descrevê-los em detalhe. Na Figura H.7 - ❶, pode-se observar como são definidas as características gerais da classe, bem como seus atributos e operações (respectivamente parte ❷ e parte ❸ da Figura H.7).

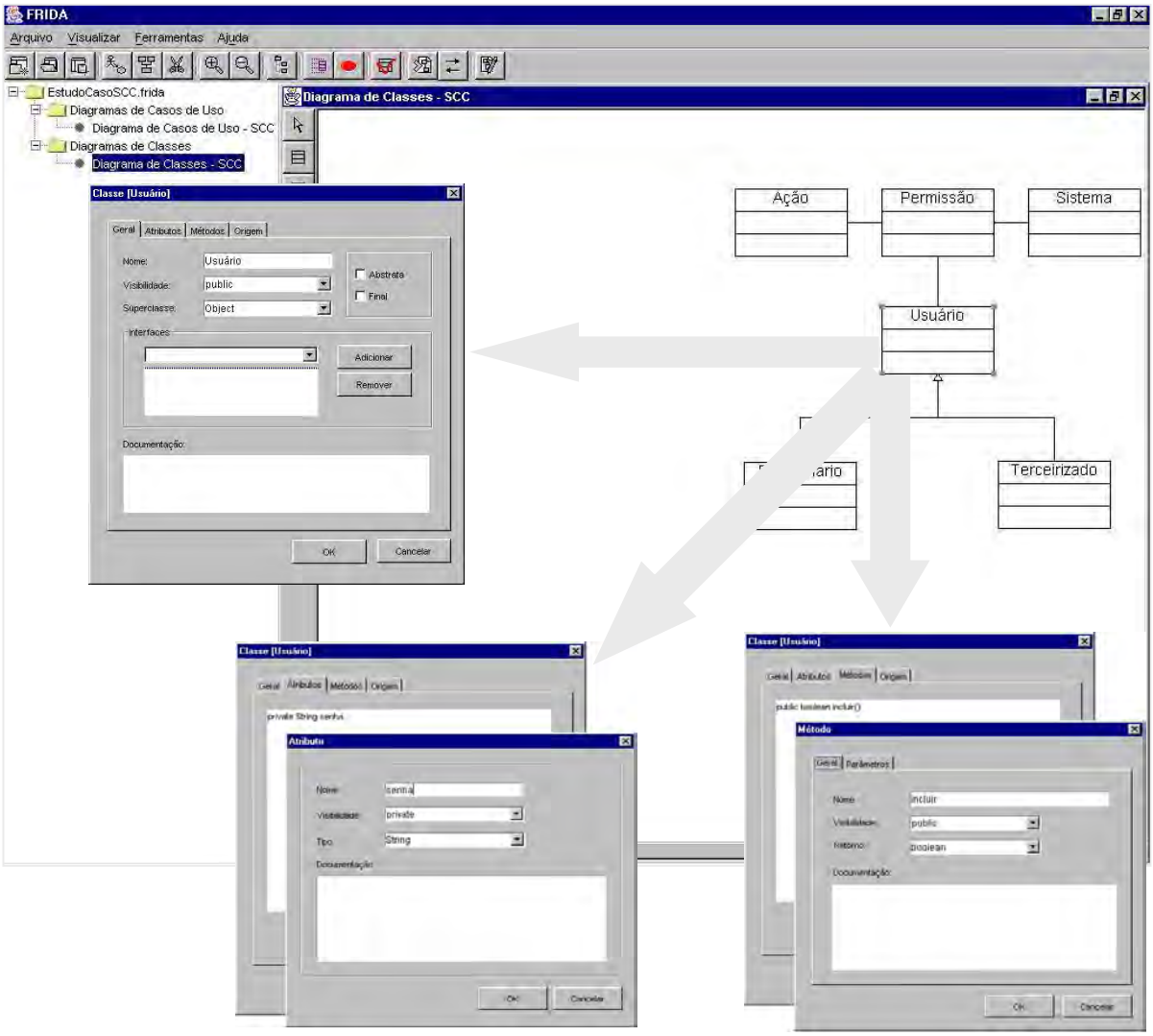

Figura H.7: Especificando em Detalhes uma Classe

H.7 Confirmando a Verificação das Mudanças

Uma vez realizada alguma alteração na especificação de requisitos, o protótipo se encarrega de marcar as classes que foram de certa forma afetadas por essa modificação. Para desmarcar as classes, ou seja, confirmar que as alterações foram verificadas é necessário seguir alguns passos, os quais encontram-se esquematizados na Figura H.8.

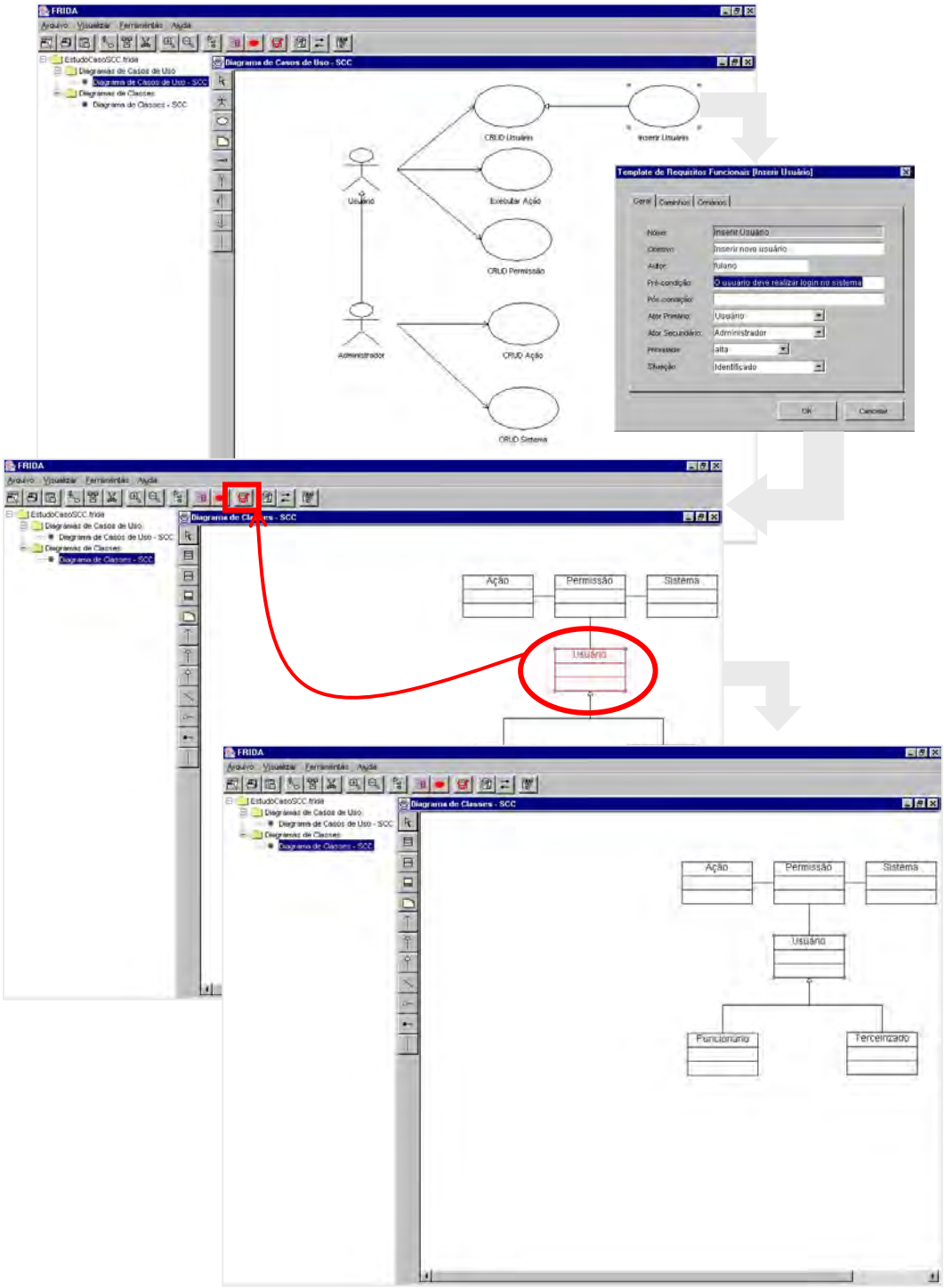

Figura H.8: Confirmando Verificações nas Classes

H.8 Detalhando os Aspectos

Conforme apresentado no Capítulo 6, os aspectos são gerados automaticamente através do protótipo FRIDA pressionando-se o botão ❶ da Figura H.9. Todo e qualquer aspecto gerado possui associado um conjunto de propriedades que podem ser determinadas utilizando-se a janela ilustrada por ❷ na Figura H.9.

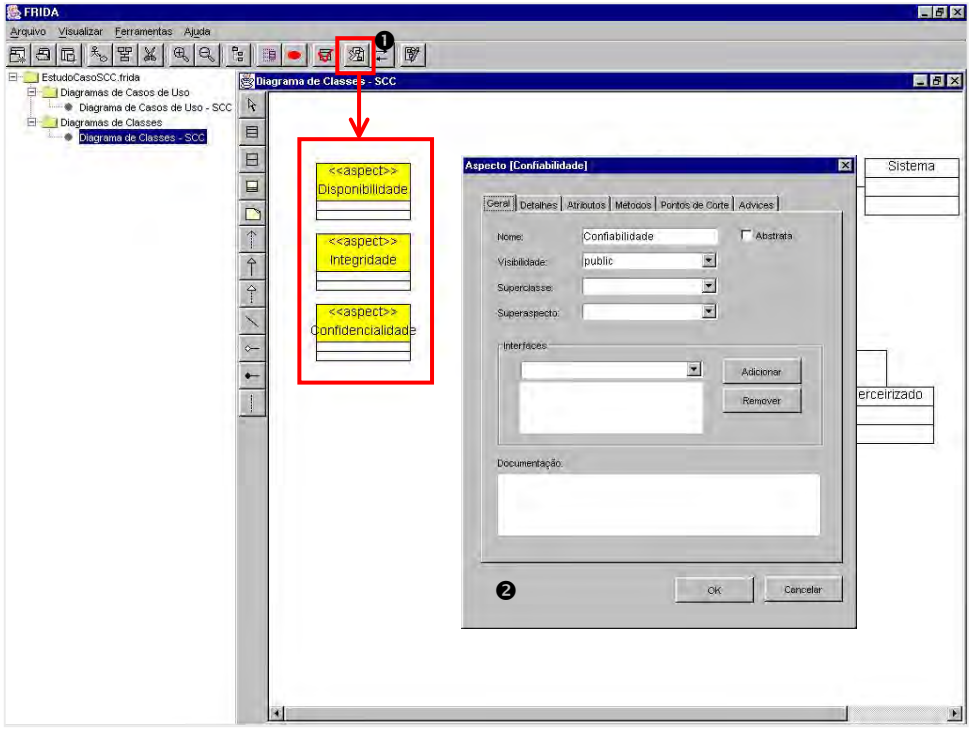

Figura H.9: Gerando Aspectos Dinamicamente

Essa janela (item 2 Figura H.9) foi denominada janela de propriedades do aspecto e ela encontra-se dividida em abas:

(i) aba geral – Figura H.10: utilizada para apresentar informações gerais sobre o aspecto, tais como o seu nome, a visibilidade, sua hierarquia de herança, entre outros.

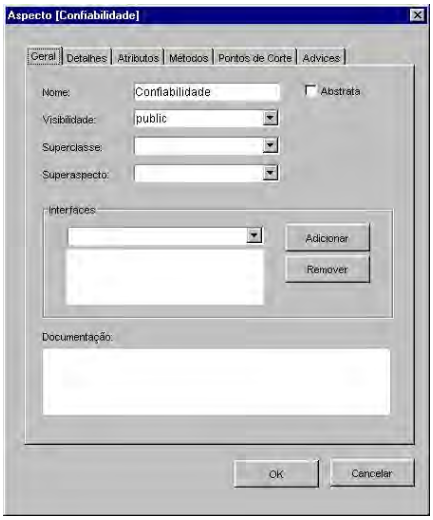

Figura H.10: Informações Gerais sobre um Aspecto

(ii) aba detalhes – Figura H.11: essa aba compreende o *template* definido na seção 6.8, responsável por integrar a visão funcional com a não funcional.

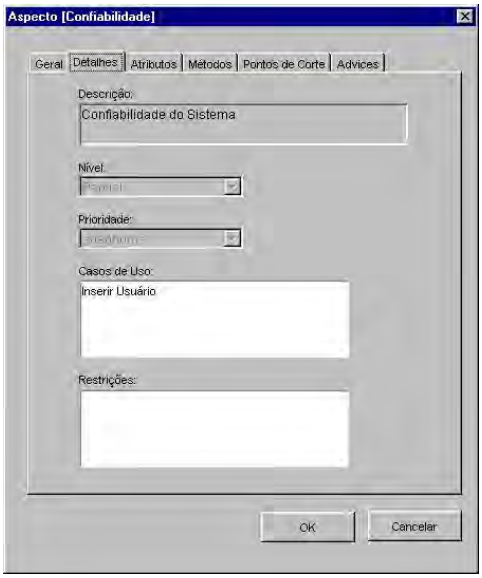

Figura H.11: Informações Detalhadas sobre um Aspecto

(iii) aba pontos de corte - Figura H.12: apresenta em detalhes um ponto de corte, o qual caracteriza-se por possuir um nome, uma visibilidade e seus parâmetros. Observa-se que, um ponto de corte pode ser aplicado a alguns elementos, no caso do método FRIDA apenas a atributos, métodos e construtores.

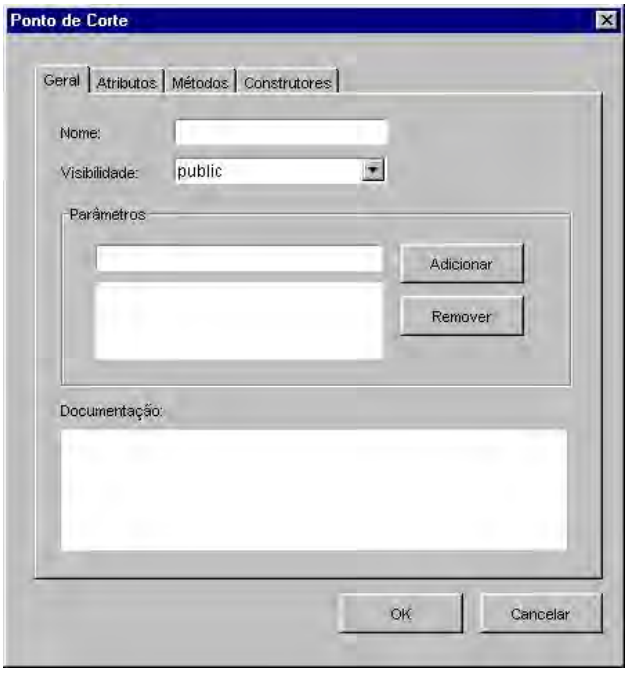

Figura H.12: Determinando Pontos de Corte para um Aspecto

No caso de pontos de corte para atributos as informações que são relevantes compreendem (Figura H.13):

- Filtro: *get/set* determinando se o ponto de corte irá capturar acessos de leitura/escrita ao atributo, respectivamente;
- Acesso, Modificador e Tipo: qual tipo de acesso[36], qual modificador[37] e qual tipo de dado o atributo, cujo ponto de corte irá capturar, foi definido;
- Classe: a classe em que o atributo foi especificado;
- Nome: compreende o nome do atributo.

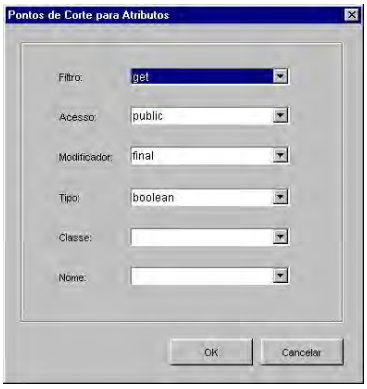

Figura H.13: Ponto de Corte para Atributos

Quando o ponto de corte está relacionado com construtores alguns detalhes devem ser observados (Figura H.14):

- Filtro: *call/execution* determinando se o ponto de corte irá capturar chamada/execução do construtor, respectivamente;
- Acesso: qual tipo de acesso[4] o construtor deve possuir para que o ponto de corte capture sua chamada ou execução;
- Classe: a classe em que o construtor foi definido;
- Parâmetros: que o construtor deve possuir de forma que o ponto de corte seja ativado.
- Exceções: que o construtor pode propagar, e que podem ativar o ponto de corte.

Figura H.14: Ponto de Corte para Construtores

[36] public, private, protected, !public, !private, !protected
[37] final, static, static final

Alguns dos possíveis filtros de ponto de corte que podem ser ativados para métodos são em função de (Figura H.15):

- Filtro: *call/execution* para controlar a chamada/execução de um método;
- Acesso: qual tipo de acesso[4] o método deve possuir de forma que o ponto de corte capture sua chamada/execução;
- Tipo de Retorno: qual tipo de retorno deve possuir o método que será capturado pelo ponto de corte;
- Nome: o nome do método que será controlado pelo ponto de corte que está sendo definido;
- Classe, Parâmetros e Exceções: ativação idêntica a de construtores.

Figura H.15: Ponto de Corte para Métodos

(iv) aba *advices* – Figura H.16: determina os pontos em que um ponto de corte deve ser ativado. São descritos basicamente por: tipo de retorno, o contexto (*before, after* e *around*), além dos parâmetros, exceções e ponto de corte ao qual o *advice* que está sendo especificado pertence.

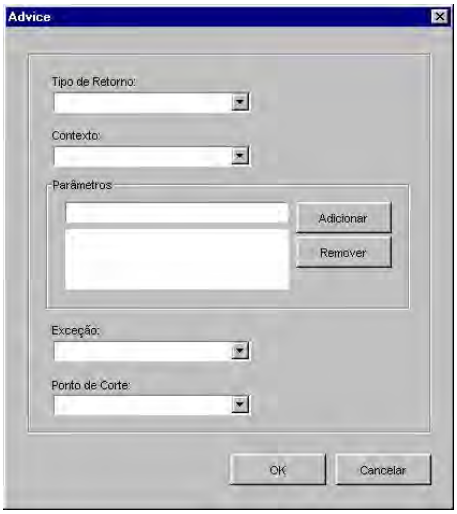

Figura H.16: Advices de um Ponto de Corte

Outras duas abas (atributos e métodos) encontram-se vinculadas a aspectos. Em virtude delas terem sido previamente apresentadas (seção A8.6) decidiu-se não apresentá-las novamente nessa seção.

ANEXO I – XML GERADO

No desenvolvimento de software é indispensável um modelo que possibilite a troca de informações, porque é necessário uma forma homogênea para a comunicação entre as ferramentas de desenvolvimento. O XML é um formato homogêneo que possibilita a troca de informações entre diferentes tipos de ferramentas de desenvolvimento proporcionando interoperabilidade, reusabilidade e produtividade [SUZ 99a].

XML é uma linguagem de descrição de dados que foi padronizada pela W3C (***W**orld **W**ide **W**eb **C**onsortium*), a qual fornece um padrão de dados que pode ser usado para codificar o conteúdo, a semântica e a estrutura de qualquer elemento.

No método FRIDA, todos os artefatos são armazenados em um arquivo no formato XML. Esse formato foi utilizando visando-se: (i) aumentar interoperabilidade com outras ferramentas de desenvolvimento; e (ii) possibilitar o salvamento e a carga de modelos UML em um formato comum. Além disso, outro ponto importante é destacado por Nentwich [NEN 2000] é que pode-se assumir que o conteúdo dos documentos de engenharia de software podem ser representados em XML. A Figura I.1 ilustra parte do XML gerado pelo protótipo FRIDA.

```
<?xml version="1.0" encoding="UTF-8"?>
<java version="1.4.0" class="java.beans.XMLDecoder">
<object class="ufrgs.inf.frida.UseCaseDiagramGraph">
  <void method="addNode">
   <object id="ActorNode0" class="ufrgs.inf.frida.ActorNode">
    <void property="name">
     <void property="size">
      <int>4</int>
     </void>
     <void property="text">
      <string>Usuário</string>
     </void>
    </void>
   </object>
   <object class="java.awt.geom.Point2D$Double">
    <double>130.0</double>
    <double>110.0</double>
   </object>
  </void>
<void method="addNode">
   <object id="UseCaseNode0" class="ufrgs.inf.frida.UseCaseNode">
    <void property="documentation">
     <void property="justification">
      <int>1</int>
     </void>
    </void>
    <void property="name">
     <void property="size">
      <int>4</int>
     </void>
     <void property="text">
```

```
          <string>CRUD Usuário</string>
         </void>
        </void>
       </object>
       <object class="java.awt.geom.Point2D$Double">
        <double>340.0</double>
        <double>20.0</double>
       </object>
      </void>
      <void method="addNode">
       <object id="UseCaseNode1" class="ufrgs.inf.frida.UseCaseNode">
        <void property="documentation">
         <void property="justification">
          <int>1</int>
         </void>
        </void>
        <void property="name">
         <void property="size">
          <int>4</int>
         </void>
         <void property="text">
          <string>Executar Ação</string>
         </void>
        </void>
       </object>
       <object class="java.awt.geom.Point2D$Double">
        <double>350.0</double>
        <double>110.0</double>
       </object>
      </void>
      <void method="addNode">
       <object id="UseCaseNode2" class="ufrgs.inf.frida.UseCaseNode">
        <void property="documentation">
         <void property="justification">
          <int>1</int>
         </void>
        </void>
        <void property="name">
         <void property="size">
          <int>4</int>
         </void>
         <void property="text">
          <string>CRUD Permissão</string>
         </void>
        </void>
       </object>
       <object class="java.awt.geom.Point2D$Double">
        <double>350.0</double>
        <double>210.0</double>
       </object>
      </void>
      <void method="addNode">
       <object id="UseCaseNode3" class="ufrgs.inf.frida.UseCaseNode">
        <void property="documentation">
         <void property="justification">
          <int>1</int>
         </void>
        </void>
        <void property="name">
         <void property="size">
          <int>4</int>
         </void>
         <void property="text">
          <string>Inserir Usuário</string>
         </void>
        </void>
```

```
        </object>
        <object class="java.awt.geom.Point2D$Double">
         <double>560.0</double>
         <double>20.0</double>
        </object>
       </void>
       <void method="addNode">
        <object id="UseCaseNode4" class="ufrgs.inf.frida.UseCaseNode">
         <void property="documentation">
          <void property="justification">
           <int>1</int>
          </void>
         </void>
         <void property="name">
          <void property="size">
           <int>4</int>
          </void>
          <void property="text">
           <string>CRUD Ação</string>
          </void>
         </void>
        </object>
        <object class="java.awt.geom.Point2D$Double">
         <double>360.0</double>
         <double>330.0</double>
        </object>
       </void>
      <void method="connect">
       <object class="ufrgs.inf.frida.ClassRelationshipEdge">
        <void property="endArrowHead">
         <object class="ufrgs.inf.frida.ArrowHead" field="TRIANGLE"/>
        </void>
       </object>
       <object idref="ActorNode1"/>
       <object idref="ActorNode0"/>
      </void>
      <void method="connect">
       <object class="ufrgs.inf.frida.ClassRelationshipEdge">
        <void property="endArrowHead">
         <object class="ufrgs.inf.frida.ArrowHead" field="TRIANGLE"/>
        </void>
       </object>
       <object idref="UseCaseNode3"/>
       <object idref="UseCaseNode0"/>
      </void>
      <void method="connect">
       <object class="ufrgs.inf.frida.ClassRelationshipEdge">
        <void property="endArrowHead">
         <object class="ufrgs.inf.frida.ArrowHead" field="V"/>
        </void>
       </object>
       <object idref="ActorNode0"/>
       <object idref="UseCaseNode0"/>
      </void>
      <void method="connect">
       <object class="ufrgs.inf.frida.ClassRelationshipEdge">
        <void property="endArrowHead">
         <object class="ufrgs.inf.frida.ArrowHead" field="V"/>
        </void>
       </object>
       <object idref="ActorNode0"/>
       <object idref="UseCaseNode1"/>
      </void>
      <void method="connect">
       <object class="ufrgs.inf.frida.ClassRelationshipEdge">
        <void property="endArrowHead">
```

```
          <object class="ufrgs.inf.frida.ArrowHead" field="V"/>
         </void>
        </object>
        <object idref="ActorNode0"/>
        <object idref="UseCaseNode2"/>
       </void>
       <void method="connect">
        <object class="ufrgs.inf.frida.ClassRelationshipEdge">
         <void property="endArrowHead">
          <object class="ufrgs.inf.frida.ArrowHead" field="V"/>
         </void>
        </object>
        <object idref="ActorNode1"/>
        <object idref="UseCaseNode4"/>
       </void>
       <void method="connect">
        <object class="ufrgs.inf.frida.ClassRelationshipEdge">
         <void property="endArrowHead">
          <object class="ufrgs.inf.frida.ArrowHead" field="V"/>
         </void>
        </object>
        <object idref="ActorNode1"/>
        <object idref="UseCaseNode5"/>
       </void>
       <void property="name">
        <string>Diagrama de Casos de Uso - SCC</string>
       </void>
      </object>
     </java>
```

Figura I.1: XML Gerado pelo Protótipo FRIDA