Universidade Eduardo Mondlane

Faculdade de Ciências

Departamento de Matemática e Informática

TRABALHO DE LICENCIATURA

Requisitos Não Funcionais No Desenho de Modelo de Dados Para Desenvolvimento de Sistema de Informação

AUTOR:

João António Rodrigues M. R. Tembe

Maputo, Outubro de 2002

Universidade Eduardo Mondlane

Faculdade de Ciências

Departamento de Matemática e Informática

TRABALHO DE LICENCIATURA

Requisitos Não Funcionais No Desenho de Modelo de Dados Para Desenvolvimento de Sistema de Informação

SUPERVISOR:

dr. Fernando Comolo

AUTOR:

João António Rodrigues M. R. Tembe

Maputo, Outubro de 2002

AGRADECIMENTOS

Agradeço em particular ao dr. Fernando Comolo, não só pela supervisão deste trabalho, mas também pelas suas contribuições para a extensão das minhas capacidades de análise, descrição e reflexão, assim como na coordenação e organização técnica.

Aos colegas, pela permanente compreensão, paciência e colaboração demonstradas nas diversas ocasiões de consultas.

A todos, que directa ou indirectamente contribuíram, quer em apoio em correcões tanto linguística assim como de contexto, como pelo encorajamento na realização deste trabalho.

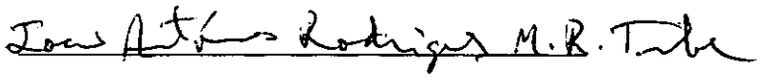
À minha família, cuja compreensão e paciência, facilitaram de algum modo, na realização e finalização deste trabalho.

DECLRARAÇÃO DE HONRA

Eu, João António Rodrigues M. R. Tembe, declaro por minha honra, que este trabalho foi por mim realizado sob orientação do meu supervisor, servindo-me das fontes ao longo do mesmo referenciadas e que não foi submetido para outro grau que não seja a tese de *Licenciatura em Informática* na Universidade Eduardo Mondlane.

Maputo, Outubro de 2002

O Autor



(João António Rodrigues M. R. Tembe)

DEDICATÓRIA

Dedico este trabalho, aos meus familiares e aos meus amigos, com eles aprendi a ter fé e esperança.

Resumo

O desenvolvimento de sistemas de informação necessita que se utilize modelos conceptuais que lidem com aspectos que vão além de entidades e actividades. Em particular, pesquisas recentes apontam que esses modelos conceptuais necessitam modelar metas de forma a capturar intenções que fundamentam situações complexas que ocorrem em uma organização. Uma classe em particular dessas metas é chamada de requisitos não funcionais (RNF) que precisam ser capturados e analisados desde os primeiros momentos do desenvolvimento do software. Erros provocados por não se lidar convenientemente, ou simplesmente não lidar com RNFs são apontados como estando entre os mais caros e difíceis de corrigir. Essa tese procura abordar dois aspectos de como se lidar com RNFs: como analisar RNFs e como integrá-los aos modelos conceptuais. Para isso, propômos uma estratégia que trata da análise dos RNFs ainda no início do processo de desenvolvimento de software e de como integrar os RNFs analisados aos modelos conceptuais. A tese foi validada através de exemplos ilucidativos ao longo do trabalho e um estudo de caso em anexo que apontam, como esperávamos, que o uso desta estratégia pode levar a ganhos na qualidade do modelo conceptual final bem como a um processo de produção de software mais produtivo.

ACRÓNIMOS

RFs – Requisitos Funcionais

RFNs – Requisitos Não Funcionais

Developer – Analista/Programador

DEA – Diagrama Entidade Associação

UML – Unified Modeling Language

Índice Pag.

Índice de Figuras Pag.

1- Introdução

1.1 - Motivação

A crescente complexidade dos sistemas de software e o aumento da exigência de qualidade por parte dos clientes vêm impulsionando o mercado a cada dia produzir mais softwares que atendam, não somente as funcionalidades exigidas, mas também a aspectos não funcionais exigidos pelos clientes tais como: custo, confiabilidade, segurança, manutenabilidade, usabilidade, portabilidade, performance, rastreabilidade de informações entre outros. Estes aspectos não funcionais devem ser tratados como requisitos não funcionais (RNF) do software. Devem ainda ser tratados desde o início do seu desenvolvimento[Chung 95] estendendo este tratamento por todo o ciclo de vida do software.

Requisitos não funcionais vêm sendo citados em vários processos de desenvolvimento de software, dentre outras formas como restrições e condições de contorno, porém sempre de maneira quando muito secundária e altamente informal do ponto de vista da análise de requisitos. Este tipo de tratamento faz com que, quando tratados, estes requisitos sejam frequentemente contraditórios, difíceis de serem considerados durante o desenvolvimento de software e difíceis de serem validados. O facto destes requisitos terem sido mal analisados ou não analisados tem causado uma série de histórias de insucessos, incluindo a desativação de sistemas pouco após terem sido implantados [Finkelstein 96]. Estudos apontam estes requisitos como estando entre os mais caros e difíceis de corrigir [Cysneiros 99].

Segundo Finkelstein [Finkelstein 96], os erros na fase da análise de requisitos são extremamente caros de reparar. Os erros citados concentram-se na fase inicial de estudos.

Destes, 41% referem-se à definição de requisitos, 28% a projecto lógico e varia em torno de 6% a 5% os demais factores como: dados, interface, ambiente e pessoas, ficando a documentação com 2%.

1.2 – Objectivos

1.2.1 – Objectivo Geral

- Determinar o impacto dos Requisitos Não Funcionais (RNFs) no modelo de dados Diagrama Entidade Associação (DEA).

1.2.2 – Objectivos Específicos

- Estudar a teoria de requisitos para software
- Qualificar o requisito pela sua característica funcional
- Apresentar uma estratégia de tratamento de Requisitos não Funcionais (RNFs).
- Avaliar a aplicabilidade dos RNFs no modelo de dados (DEA).

1.3 - Visão Geral da Solução Proposta

Apresentaremos nesta tese um processo para se lidar com requisitos não funcionais desde as etapas iniciais do processo de desenvolvimento de software. A integração dos RNFs, analisados durante as fases iniciais do desenvolvimento do software, com os modelos conceptuais gerados a partir dos requisitos funcionais deverá permitir obtermos um modelo conceptual que enfoque os requisitos funcionais e não funcionais ao mesmo tempo, obtendo-se assim modelos mais completos e com os impactos da satisfação dos requisitos não funcionais já analisados, resultando em uma melhoria da qualidade final do produto sob a óptica, tanto do cliente quanto do *developer*, bem como em um menor tempo de entrega de produto e menores custos de manutenção.

Para atingir este objetivo propômos uma estratégia que se baseia no uso de RNF para a análise do modelo conceptual gerado através dos requisitos funcionais, isto é como o modelo será modificado para satisfazer um determinado RNF. Uma vez na posse dos modelos funcionais e não funcionais propôremos uma estratégia para integrar ambos os modelos de forma a obtermos um modelo conceptual consolidado que reflicta ambas as visões.

Finalmente, como essa estratégia precisa ser apoiada por um método a ser utilizado na especificação de software, iremos usar a UML [Fowler 97] [Jacobson 99] [Rumbaugh 99] em seus diagramas de classe. Esta pretende representar nesses diagramas as conseqüências das operacionalizações dos RNFs encontrados na visão não funcional, bem como estabelecer uma ligação para os modelos dessa visão que possibilite a rastreabilidade desses RNFs. Essa rastreabilidade é de grande auxílio no tratamento do

aspecto evolutivo do software, uma vez que podemos nos reportar às fontes que originaram a inclusão nos modelos conceptuais de uma determinada classe, operação, atributo ou mesmo mensagem oriunda da operacionalização de um RNF.

1.4 - Organização

Esta tese é composta por seis capítulos, onde o capítulo dois irá abordar conceitos gerais da engenharia de requisitos, incluindo um detalhamento dos conceitos e problemas relacionados a RNFs, bem como uma visão geral do grafo de RNFs e da UML.

O capítulo três irá mostrar uma visão geral da estratégia proposta para lidar com os RNFs da análise à finalização do modelo conceptual.

O capítulo quatro mostrará o grafo de RNF, como obtê-los a partir dos RNFs, uma sistematização de processo para identificação de interdependências.

O capítulo cinco faz um estudo de caso.

O capítulo seis apresentará as principais conclusões e recomendações deste trabalho e o capítulo sete a bibliografia utilizada e consultada.

2- Conceitos Básicos

2.1 - Engenharia de Requisitos

A engenharia de requisitos procura sistematizar o processo de definição de requisitos. Essa sistematização é necessária porque a complexidade dos sistemas exige que se preste mais atenção ao correcto entendimento do problema antes do *comprometimento de uma solução* [Leite 94]. Uma boa definição para requisitos pode ser encontrada em Leite, 94 e é mostrada a seguir.

"***Requisito:*** *Condição necessária para a obtenção de certo objectivo, ou para o preenchimento de certo objectivo.*"

Pesquisas sobre a aquisição de requisitos são valorosas por duas razões: primeiramente, na perspectiva da engenharia de software, a aquisição de requisitos é talvez a mais crucial parte do processo de desenvolvimento de software. Estudos [Boehm 84] indicam que, quando só detectados depois do software implementado, erros em requisitos de software são até 20 vezes mais caros de corrigir que qualquer outro tipo de erro.

Para a engenharia de requisitos é fundamental que o engenheiro de software delimite os contornos do macrosistema em que a definição de software ocorrerá, ou seja, delimitar o Universo de Informações do sistema (UdI)[1].

É importante ressaltar que o UdI sempre existe. O UdI não depende do modelo que estamos a utilizar. Mesmo que o macrosistema não esteja bem definido, sempre podemos,

---

[1] *"Universo de Informações é o contexto geral no qual o software deverá ser desenvolvido e operado. O UdI inclui todas as fontes de informação e todas as pessoas relacionadas ao software. Essas pessoas são também conhecidas como os atores desse universo. O UdI é a realidade circunstanciada pelo conjunto de objetos definidos pelos que demandam o software" [Leite 94].*

e devemos, estabelecer os limites de nossa actuação. Quanto mais bem delineado um UdI, maiores são as chances de um software bem definido.

A aquisição de requisitos é um processo que está constantemente em evolução, ou seja, toda vez que o UdI muda, o modelo resultante do processo de aquisição de requisitos deve mudar junto. Esta realidade é muitas vezes não considerada através do que se procurou determinar como um congelamento dos requisitos. Se por um lado a constante evolução do UdI pode gerar custos altos no desenvolvimento, uma vez que temos de estar a actualizar os nossos requisitos, por outro lado o congelamento dos requisitos em um determinado momento não deve produzir efeitos contrários, aumentando consideravelmente o custo do produto final que decorrem de alterações que serão demandadas para a aceitação do software.

Em muitas organizações, os requisitos são escritos como parágrafos em linguagem natural e suplementados por diagramas [Sommerville98]. A linguagem natural é a única notação que é compreendida por todos os leitores potenciais (ex. clientes, usuários, engenheiros de software, engenheiros de requisitos) dos requisitos. Uma vez que dificilmente teremos à disposição recursos que permitam satisfazer todos os requisitos dos utilizadores [Berry98], requisitos podem ser classificados como desejáveis ou obrigatórios, utilizando-se aqui de um enfoque voltado para a necessidade de priorizar requisitos. Os requisitos podem ser ainda classificados como estáveis, mudam mais lentamente, ou voláteis, mudam mais rapidamente. Esta classificação auxilia a actividade de gerência de requisitos, uma vez que possibilita antecipar mudanças prováveis de requisitos. Por fim, uma outra classificação que tem tido bastante aceitação na

comunidade acadêmica [Sommerville98] [Mamani99] [Cysneiros99], divide os requisitos em requisitos funcionais (RFs) e requisitos não funcionais (RNFs).

2.2 - Requisitos Não Funcionais

A complexidade de um software é determinada em parte por sua funcionalidade, ou seja, o que o sistema faz, e em parte por requisitos gerais que fazem parte do desenvolvimento do software como custo, performance, confiabilidade, manutenabilidade, portabilidade, custos operacionais entre outros [Chung 00]. Estes requisitos podem ser chamados de requisitos não funcionais. Esta denominação foi utilizada por Roman [Roman 85], sendo os RNFs também conhecidos como atributos de qualidade [Keller 90], restrições [Roman 85], objectivos [Mostow 85] entre outros. Recentemente, o termo requisitos não funcionais vem se firmando como nomenclatura corrente no meio acadêmico.

Os RNFs desempenham um papel crítico durante o desenvolvimento de sistemas, e erros devido a não aquisição ou a aquisição incorreta destes estão entre os mais caros e difíceis de corrigir, uma vez que o sistema tenha sido implementado [Davis 93].

Embora sem formalmente definir RNFs, alguns trabalhos propõem classificações destes. Em um relatório do *Rome Air Development Center* [Bowen 85], RNFs são classificados da seguinte maneira:

- orientados ao consumidor (ou critérios de qualidade de software), aqui se referindo a requisitos observáveis pelo cliente como eficiência, correção, amigabilidade e outros; e,

- atributos orientados tecnicamente (ou critérios de qualidade) associados mais a requisitos ligados ao sistema, como tratamento de erros e anomalias, completeza, processamento veloz em pontos críticos de tempo real e outros.

Duas diferentes abordagens são utilizadas em tratamento sistemático de RNFs. Elas podem ser classificadas como orientada a produto e orientada a processo [Chung 95]. A primeira aborda RNFs de forma a classificar o quanto um sistema satisfaz os RNFs que dele são requeridos. Por exemplo, medir a visibilidade de um software pode incluir, entre outras coisas, a medição de quantos desvios (branches) existem em um software. Isto pode ser obtido utilizando-se um critério como: "Não deve haver mais do que X desvios por 1000 linhas de código". Quase todos os trabalhos em RNFs utilizam esta abordagem, orientada a produto, a qual é bem descrita por Keller [Keller 90].

Boehm [Boehm 84] mostra que a qualidade geral do produto final de um software pode ser melhorada simplesmente tornando os *developers* conscientes de que critérios de qualidade deviam ser cumpridos. Também baseados em uma abordagem quantitativa sob a óptica da qualidade de software, Basili e Musa [Basili 91] propõem modelos e métricas para o processo de engenharia de software de uma perspectiva gerencial.

Já a abordagem orientada a processo, que é por nós utilizada nesta tese, advoga o desenvolvimento de estratégias para justificar decisões de desenho do modelo de dados. Ao contrário da abordagem orientada a produto, nesta abordagem procura-se racionalizar o processo de desenvolvimento propriamente dito em termos de RNFs [Chung 00]. Frequentemente, quando adicionamos um RNF a uma especificação de requisitos, forçamos tomadas de decisões que poderão afectar positiva ou negativamente outros RNFs, efeito esse visualizado como interdependências entre RNFs. Estas

interdependências podem ser positivas ou negativas, ou seja, um RNF pode influenciar positivamente outro RNF contribuindo assim para sua satisfação, bem como pode influenciar negativamente, ou seja, contribuir para que um destes RNFs não seja satisfeito ou satisfeito apenas parcialmente.

Ortogonalmente, podemos classificar o tratamento dispensado a RNFs como divididos em qualitativos e quantitativos. A maioria das abordagens orientada a produto é quantitativa, uma vez que elas propõem métodos quantitativos para medir o quanto um software satisfaz um dado RNF. As abordagens orientadas a processo são, por outro lado, inteiramente voltadas para o aspecto qualitativo, via de regra adotando ideias oriundas do raciocínio qualitativo [AI 84].

RNFs abordam importantes aspectos relacionados à qualidade de softwares. Eles são essenciais para que softwares sejam bem sucedidos. A não observância de RNFs pode resultar em: softwares com inconsistência e de baixa qualidade; clientes e *developers* insatisfeitos; tempo e custo de desenvolvimento além dos previstos devido à necessidade de se consertar softwares que não foram desenvolvidos sob a óptica da utilização de RNFs.

RNFs são geralmente subjectivos, uma vez que podem ser vistos, interpretados e conceptuados de forma diferente por diferentes pessoas. É verdade que os requisitos funcionais também sofrem do problema de diferentes conteúdos advindos de diferentes pontos de vista, porém, como os RNFs são por natureza mais abstratos e uma vez que RNFs são comumente relacionados de maneira breve e vaga este problema é potencializado [Chung 00].

Além disso, RNFs frequentemente interagem entre si, uma vez que a tentativa de satisfazer um RNF pode prejudicar ou ajudar a satisfazer outros RNFs. Como vários RNFs possuem efeitos de natureza global nos softwares, uma solução localizada pode não ser adequada.

Por todas estas razões, RNFs são difíceis de se lidar e vitais de serem tratados para que possamos obter softwares de qualidade.

2.3 - Requisitos Funcionais versus Requisitos Não Funcionais

Os *requisitos funcionais* são requisitos que expressam funções ou serviços que um software deve ou pode ser capaz de executar ou fornecer. As funções ou serviços são, em geral, processos que utilizam entradas para produzir saídas.

Os *requisitos não funcionais (RNFs)* são requisitos que declaram restrições, ou atributos de qualidade para um software e/ou para o processo de desenvolvimento deste sistema. Segurança, Precisão, Usabilidade, Performance e Manutenabilidade são exemplos de requisitos não funcionais.

A distinção entre o que é um requisito funcional e o que é um não funcional nem sempre é clara. Parte da razão advém para o facto de que os RNFs estão sempre relacionados a um requisito funcional [Chung 00]. A Grosso modo, partindo da definição acima podemos dizer que um requisito funcional expressa algum tipo de transformação que tem lugar no software, enquanto um RNF expressa como essa transformação irá se comportar ou que qualidades específicas ela deverá possuir.

Um exemplo de diferença entre requisito funcional e RNF pode ser visto a seguir.

Suponhamos que estejamos no domínio de laboratórios de análises clínicas: um requisito funcional desse sistema poderia ser expresso da seguinte forma:

> "O sistema deve fornecer uma entrada de dados que possibilite a designação de resultados a exames admitidos para um paciente por técnicos, supervisores e chefes".

Este mesmo requisito funcional poderá ter associado a ele o seguinte RNF:

> "Alguns exames deverão ter tratamento especial para a entrada de resultados. Para estes exames valores acima ou abaixo de pré-determinados valores só poderão ser digitados por chefes de secção".

É importante ressaltar que esse último parágrafo não exprime uma função do sistema, e sim, uma restrição a uma função existente, entrar resultados de exames. O que se vê aqui é que essa funcionalidade do sistema deverá ser restrita de forma que, quando utilizada por pessoas que não sejam chefes, será restrita à condição de que essas pessoas estejam digitando um valor que se encontre dentro de limites pré-estabelecidos.

É visível que este RNF irá demandar um processo de segurança no tratamento de acesso a módulos do software, bem mais complexo daquele que seria implantado se apenas o requisito funcional tivesse sido especificado. A implementação de um processo de acesso a módulos, que critique não apenas a permissão de acesso ao se entrar em um módulo de software, mas também o conteúdo do que está sendo digitado dentro do módulo, é consideravelmente mais complexa. A tardia ou não consideração deste requisito certamente implicaria num grande "retrabalho".

2.4 - Classificação de RNFs

Algumas classificações dos RNFs podem ser encontradas na literatura. Mamani propõe a classificação de RNFs apresentada na Figura 2.1 [Mamani 99] enquanto a Figura 2.2 apresenta a classificação de RNFs proposta por Sommerville [Sommerville 98] e a Figura 2.3 mostra uma classificação proposta por Boehm [Boehm 84] na qual ele define uma árvore mínima de padrões de qualidade que um software deveria apresentar.

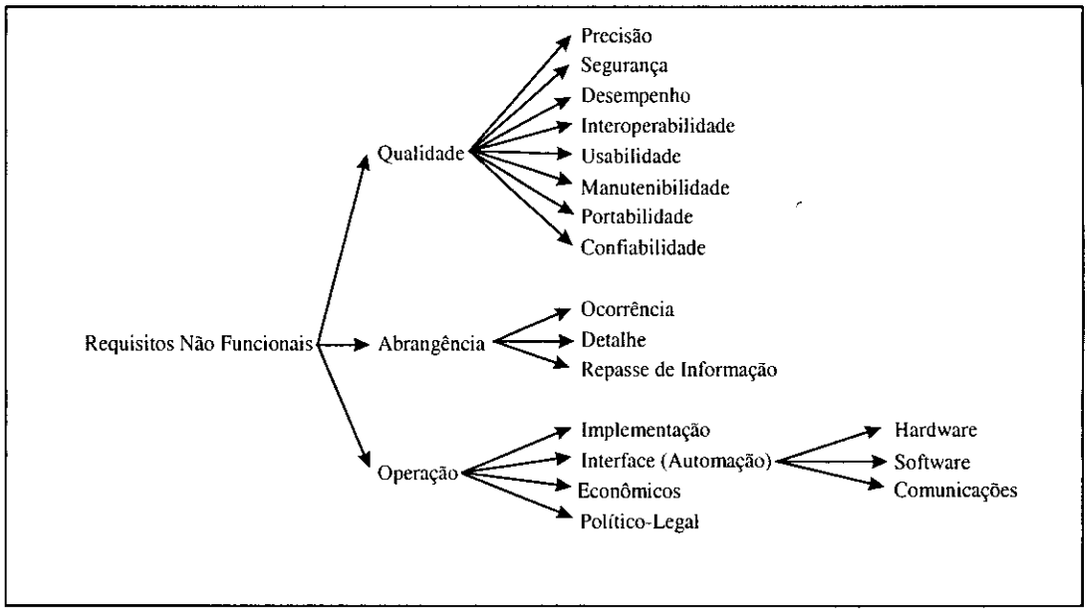


Figura 2.1 - Classificação de RNFs proposta por Mamani

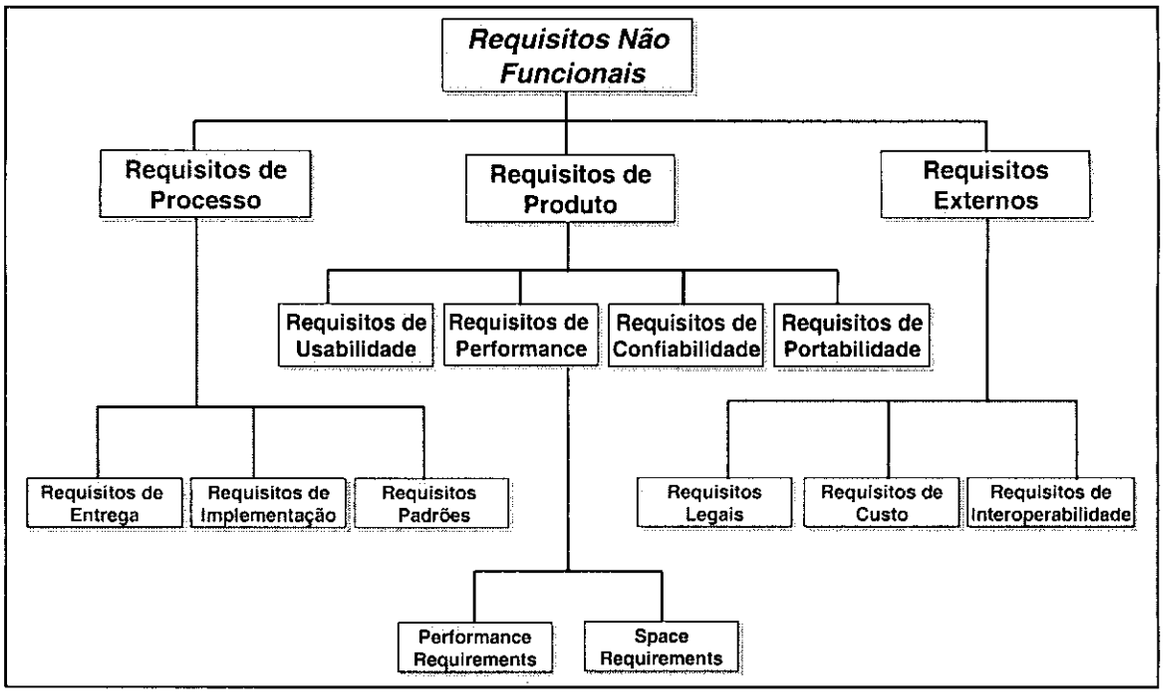


Figura 2.2 - Classificação de RNFs segundo Sommerville

Uma outra fonte que classifica alguns RNFs é o padrão internacional ISO 9126 [ISO9126]. Nesta norma são descritas seis características que definem a qualidade de

software: Funcionalidade, Confiabilidade, Usabilidade, Eficiência, Manutenabilidade e Portabilidade. Se observarmos as diversas definições de RNF, facilmente iremos considerar que estes requisitos de qualidade definidos na ISO 9126 são, em geral, RNFs. O item funcionalidade é, porém, uma excepção que certamente não poderá ser enquadrado como RNF.

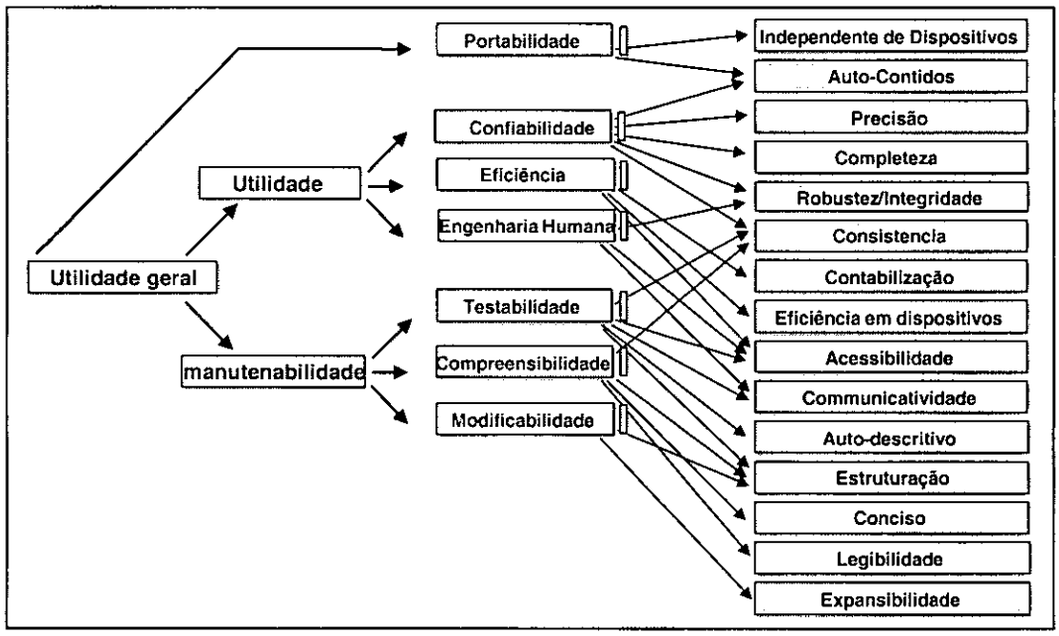


Figura 2.3 - Árvore de Características de Qualidade de Software (Boehm 84)

Proporemos aqui uma taxonomia que classifica os RNFs em primários e específicos. RNFs primários são aqueles mais abstratos que representam propriedades como: Confiabilidade, Rastreabilidade, Precisão, Segurança. Já os RNFs específicos são decomposições que se seguem aos RNFs primários e tendem a diminuir o nível de abstração de um determinado RNF, e portanto, atingir um nível de granularidade no tratamento da informação mais detalhado.

Um exemplo desta classificação seria o RNF Primário *Confiabilidade* que pode ser decomposto em validação, autorização e entrega do software.

RNFs primários podem ser decompostos em outros RNFs secundários até que se chegue ao que Chung [Chung 00] chama de operacionalização dos RNFs. Uma operacionalização de um RNF é uma acção ou informação que irá garantir a satisfação do RNF. Por exemplo, no caso de um sistema de informação para laboratórios de análises clínicas, a entrega do resultado ao paciente possui um RNF de *confidencialidade* que seria seu RNF primário. Para que possamos detalhar o que isso implica, temos de decompôr este RNF em um RNF específico de *segurança operacional* que por sua vez poderia ser decomposto na operacionalização *solicitar carteira de identidade*. Ou seja, para satisfazer o RNF *confidencialidade* teríamos de prevêr uma acção no sistema, que garantisse que o funcionário que entregou o resultado solicitou a carteira de identidade ao paciente antes de entregar-lhe o resultado.

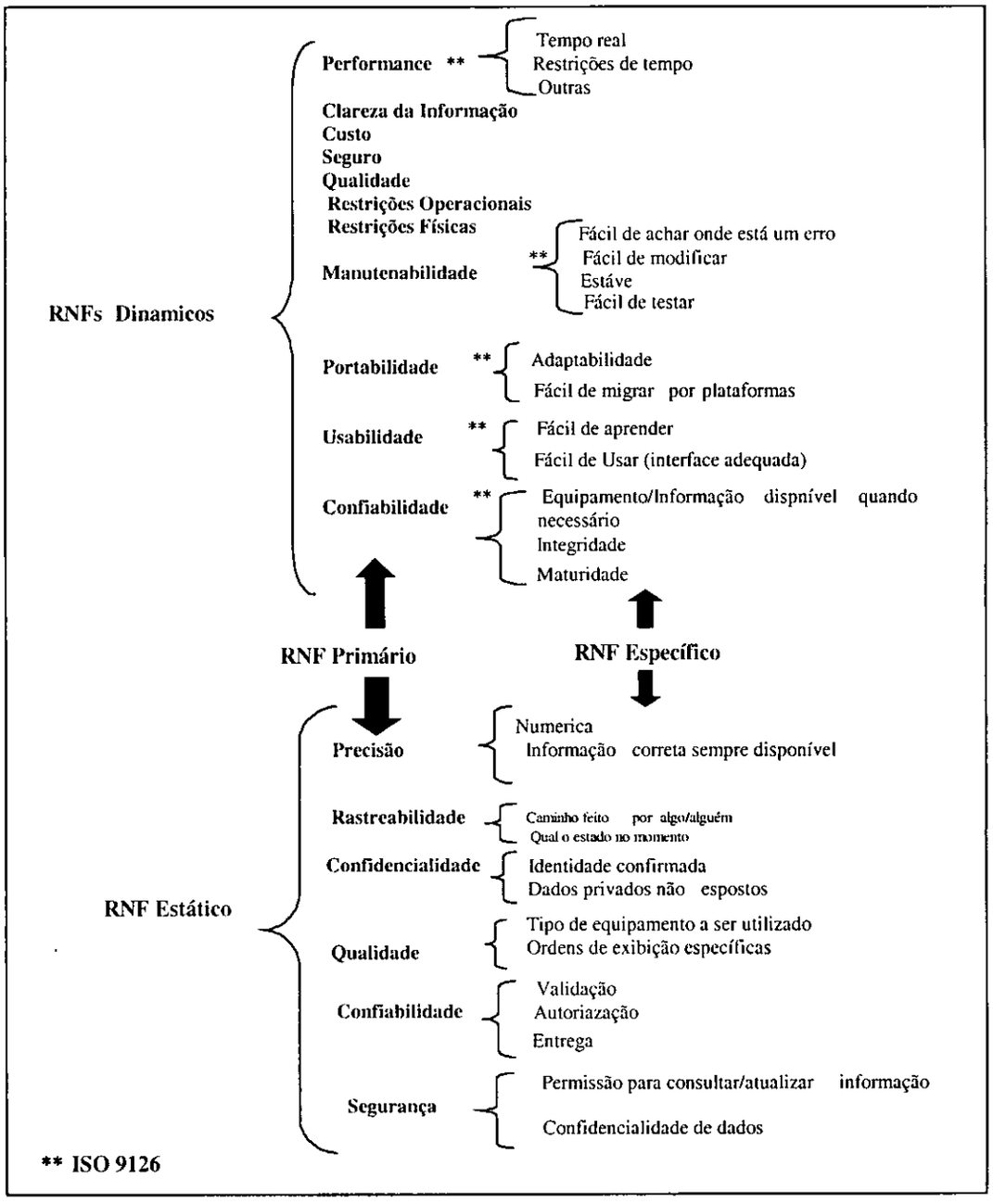


Figura 2.4 – Uma Taxonomia para RNFs

Ortogonalmente, separamos os RNFs em estáticos e dinâmicos. RNFs estáticos são aqueles que, quando presentes, **normalmente requerem o uso de dados** para validá-los como, por exemplo: segurança, precisão e rastreabilidade. RNFs dinâmicos, por outro lado, usualmente representam conceitos mais abstratos, **que normalmente envolvem**

acções ou critérios de qualidade como, por exemplo: Qualidade, Performance, Manutenabilidade, Restrições Operacionais. Alguns RNFs podem ser tanto estáticos quanto dinâmicos dependendo do contexto do domínio que estejam inseridos. A Figura 2.4 mostra um resumo desta taxonomia que também pode ser utilizada como um checklist para a aquisição de RNFs.

A lista apresentada na figura acima não pretende ser completa, mas sim, um ponto de partida para cada *developer* desenvolver seu próprio conhecimento a respeito de RNFs.

2.6 - Cenários

Nesta tese utilizaremos a abordagem proposta por Leite [Leite 97], onde os pontos centrais desta proposta são [Breitman 98]:

- Cenários descrevem situações que ocorrem no macrosistema e suas relações com o sistema;
- Cenários evoluem durante o processo de desenvolvimento de software;
- Cenários são descritos em linguagem natural.

O modelo de cenários visa representar aspectos comportamentais dos requisitos e faz parte de uma proposta maior, o *Requirements Baseline* [Leite97]. A Figura 2.5

mostra um diagrama de entidades e relacionamentos para a estrutura de cenários proposta por Leite [Leite97], enquanto a Figura 2.6 explica cada uma das entidades do cenário, mostradas na Figura 2.5, bem como a sintaxe[1] utilizada para descrever estas entidades.

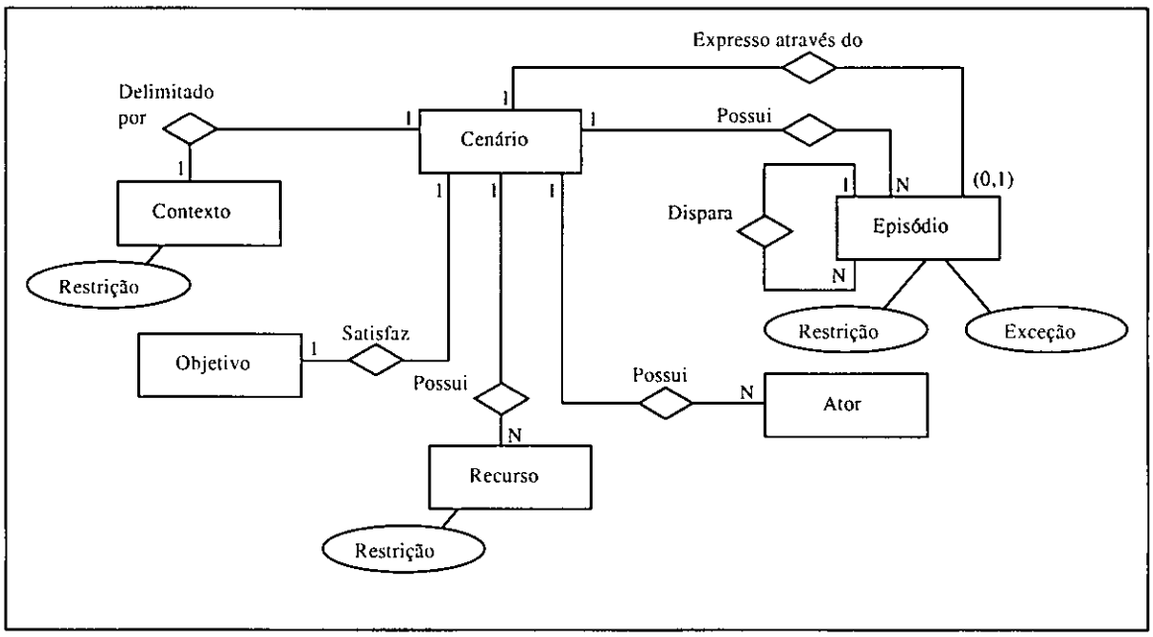


Figura 2.5 – Diagrama Entidade-Associação Descrevendo Cenários

O relacionamento "*expresso através do*" entre as entidades cenário e episódio, indica que um episódio pode ser expresso ou explicado através de um cenário, possibilitando a decomposição de cenários em sub-cenários. O atributo restrição indica, via de regra, requisitos não funcionais que se referem às entidades contexto, recurso e episódio do modelo DEA. O atributo excepção indica falta ou mau funcionamento de um recurso necessário para se atingir o objetivo do cenário e deve ser representado por frases curtas ou pequenos parágrafos.

[1] + indica composição. n{ x }m indica no mínimo **n** e no máximo **m** ocorrências de x. ( ) indica agrupamento. | indica **ou**. [x] indica que x é opcional.

- **Título:** título do cenário, que serve também para identificá-lo. Em caso de sub-cenário, o título é o mesmo do episódio que o sub-cenário explica, sem as restrições e exceções.
  Sintaxe: frase | ([Ator | Recurso] + verbo + predicado)

- **Objetivo:** meta que o cenário pretende atingir. O cenário descreve o alcance de um objetivo.
  Sintaxe: [sujeito] + verbo + predicado

- **Contexto:** localização geográfica/temporal e estado inicial do cenário
  Sintaxe: localização + estado, onde
  localização é nome,
  estado é ([ator | recurso] + verbo + predicado + 0{restrições}n) +
  0{ ([e | ou]+ [ator | recurso] + verbo + predicado + 0{restrições}n) }n

- **Recursos:** meios de suporte, dispositivos e outras entidades passivas utilizadas pelos atores do cenário
  Sintaxe: 1{nome + restrições}n , onde
  restrições é ("**Restrição:** deve ter" + 1{RNF específico}n)

- **Atores:** pessoas ou estruturas organizacionais que tem um papel no cenário
  Sintaxe: 1{nome}n

- **Episódios:** ações que detalham o cenário e fornecem seu comportamento
  Sintaxe: episódios :: = < série >
  série :: = < sentença > | < série > < sentença >
  < sentença > :: = < sentença seqüencial > | < sentença não seqüencial > |
  < sentença condicional > | < sentença optativa >
  < sentença seqüencial> :: = <sentença episódio>
  < sentença não seqüencial > :: = # < série > #
  < sentença condicional > :: = Se < condição > , então < sentença episódio >
  < sentença optativa > :: = [< série >]
  < sentença episódio > :: = [ator | recurso] + verbo + predicado + restrição +0{exceção}n,
  onde:
  restrição é ("**Restrição:** " + 1{recurso }n + "deve(m) ter "
  + 1{RNF específico}n
  + ", sendo a estratégia de satisfação "+
  estratégia de satisfação)

Figura 2.6 - Esquema para Descrição de Cenários

2.7 - Grafo de RNFs

A aquisição de RNFs demanda o uso de uma notação para representá-los de alguma forma para que a informação não se perca no tempo. Além disso, esta notação deve de alguma forma possibilitar que lidemos com os RNFs de uma maneira organizada e que facilite a lidarmos com as interdependências. Como propômos que os RNFs sejam analisados desde as primeiras etapas do processo de desenvolvimento de software, necessitamos de uma maneira de representar estes RNFs.

Para representarmos os RNFs iremos utilizar o grafo de RNFs proposto por Chung [Chung 00]. Segundo o *Framework* apresentado por Chung, RNFs devem ser representados como *metas* a serem satisfeitas. Cada meta será decomposta em subsequentes submetas até encontrarmos o que são chamadas de operacionalizações. Uma operacionalização corresponde a acções ou atributos que claramente identifiquem o que é necessário para satisfazer a meta principal.

Chung divide a decomposição dos RNFs em tipo e tópico. Na decomposição por **tipo** são usados fundamentalmente os métodos de decomposição existentes no *framework*, que indicam quais são as decomposições possíveis para um determinado RNF. No caso do RNF Segurança, por exemplo, Integridade, Confidencialidade e Disponibilidade podem ser possíveis decomposições desta meta. A Figura 2.7, extraída de [Chung 00], mostra um exemplo da decomposição do RNF *Segurança* aplicado à conta corrente como parte de um estudo de caso de automatização bancária. Neste exemplo pode-se ver que O RNF *Segurança* quando analisado sob a óptica de contas correntes apresenta 3 possíveis

decomposições *em integridade, confidencialidade e disponibilidade.* Percebe-se ainda, que a submeta *integridade* é novamente decomposta em submetas de *completeza* e *precisão.* As outras submetas não são aqui decompostas para tornar o grafo simples de ser visualizado ao nível de exemplo. Em uma abordagem completa todas seriam decompostas.

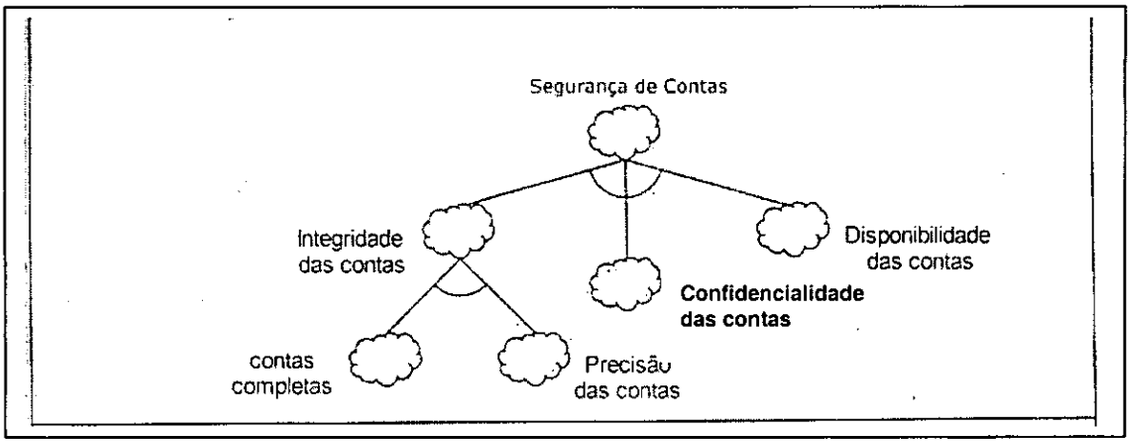


Figura 2.7– Exemplo de Decomposição do Grafo de RNF

A decomposição por **tópicos** trata da decomposição estrutural do problema. Por exemplo, no exemplo acima, o RNF Seguro aplicado a contas poderia ter sido decomposto em contas pessoa física e jurídica.

A Figura 2.8, extraída de [Chung 00], mostra um exemplo de refinamento até encontrarmos as necessárias e suficientes operacionalizações. Nesta figura podemos observar que, desta vez, a submeta decomposta é a de *confidencialidade.* Esta submeta é refinada na submeta *necessidade de autorização para acesso a informações sobre a conta.* Vemos então, que esta submeta é novamente decomposta em *Validar Acesso*, *Identificar Usuários* e *Acesso de Usuário Autenticado*, onde esta última é ainda decomposta em *Use P.I.N., Comparar assinatura* e *Requer Identificação Adicional.* No

contexto do desenvolvimento deste sistema, estes últimos refinamentos foram considerados suficientes para expressar as acções e dados necessários para satisfazer ao RNF *Segurança*. O momento de parar a decomposição é uma decisão pessoal e não existem regras fixas que determinem um limite.

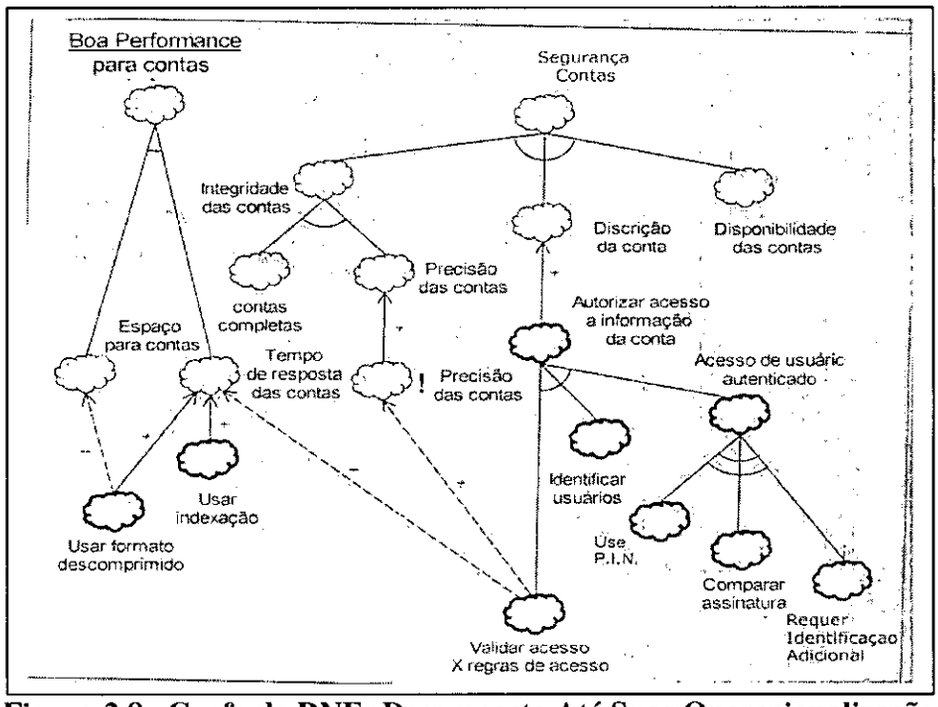


Figura 2.8– Grafo de RNFs Decomposto Até Suas Operacionalizações

É importante observar que todas as três submetas de *Autorizar Acesso a Informação da Conta* estão ligadas por um arco. Isto denota que as três são contribuições do tipo E, ou seja, para que a submeta seja satisfeita todas as três submetas terão de ser satisfeitas.

Por sua vez, a submeta de *Acesso de usuário Autenticado* possui outras três submetas que são ligadas por um arco duplo que desta feita denota uma contribuição do tipo OU, ou seja, se uma das três submetas for satisfeita a submeta imediatamente superior será satisfeita.

Neste novo grafo aparece ainda um outro RNF *Boa Performance* que também é decomposto até acharmos as operacionalizações: *Usar formato descomprimido* e *Usar indexação.*

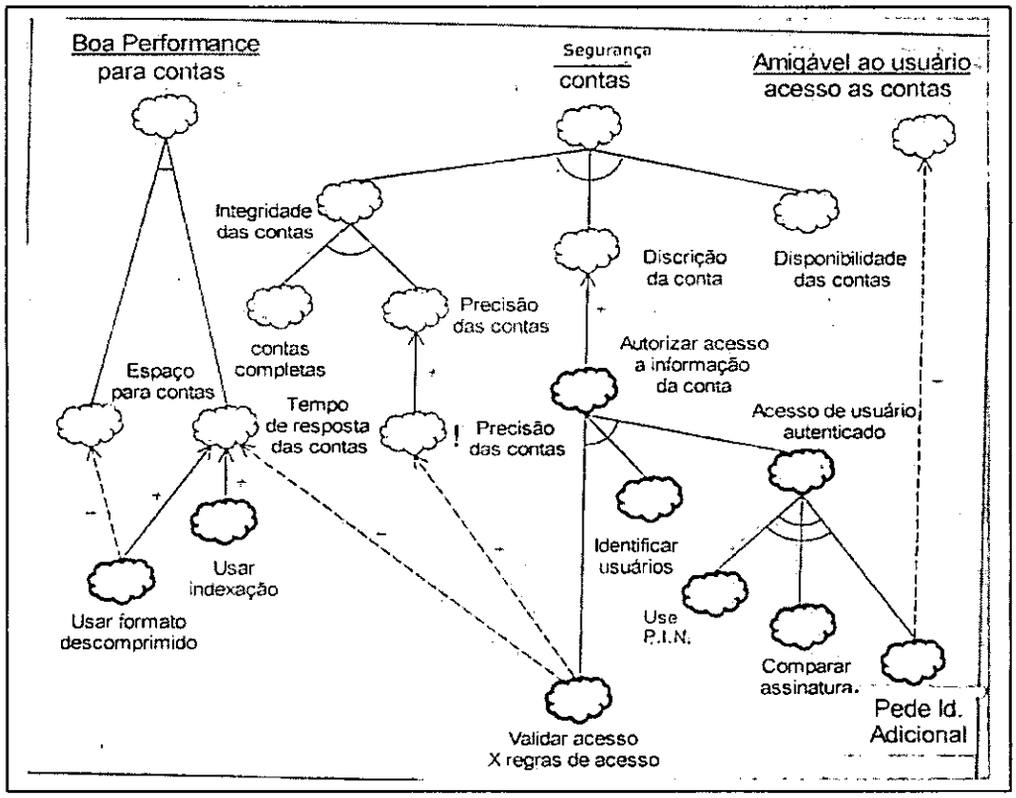


Figura 2.9– Exemplo de Grafo com Interdependências

Uma análise mais detalhada deste grafo irá revelar algumas interdependências entre submetas, algumas positivas representadas com um sinal + e outras negativas, representadas com o sinal -. A figura 2.9 mostra estas interdependências. Nela, podemos observar que a operacionalização *Validar Acesso X regras de elegibilidade* contribui positivamente para a satisfação da submeta *Precisão* e negativamente para a Submeta *Tempo de Resposta das contas* do RNF *Boa Performance* para contas correntes. Podemos ainda observar um impacto negativo da operacionalização *Pede Id. Adicional*

no RNF *Amigável ao Utilizador* no acesso a contas correntes. Esta contribuição negativa é representada como uma possível futura contribuição negativa e adveio da consulta a uma base de conhecimentos de interdependências entre RNFs.

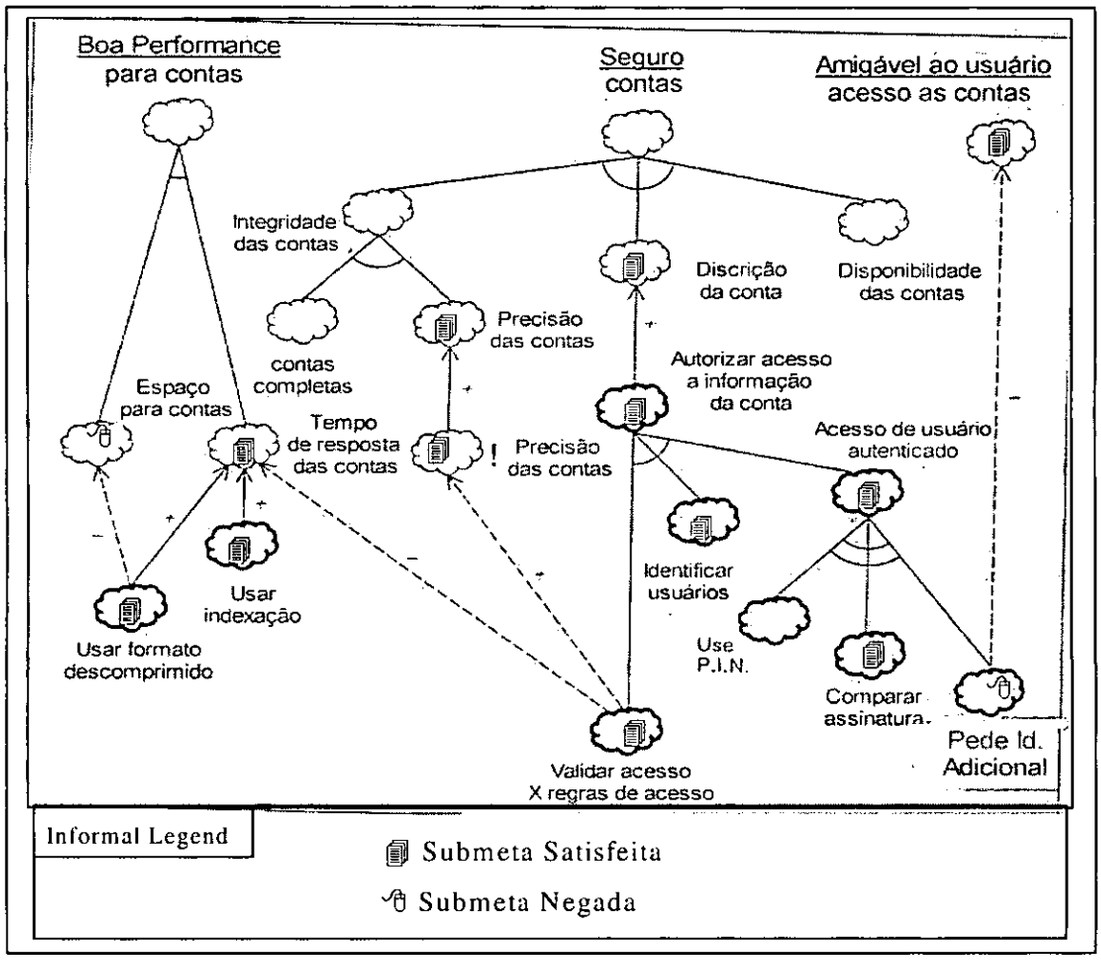


Figura 2.10 – Grafo de RNFs com Interdependências Analisadas

Uma vez que detectamos interdependências negativas temos que lidar com os conflitos que elas representam, ou seja, avaliar quais metas e submetas deverão não ser satisfeitas ou satisfeitas apenas parcialmente, ou ainda, se apesar das contribuições negativas todas as metas e submetas podem ser mantidas. A Figura 2.10 mostra o grafo acima após o engenheiro de requisitos ter tomado as decisões necessárias. Neste grafo o símbolo √

representa que estas operacionalizações foram escolhidas para serem implementadas enquanto o X, representa operacionalizações que foram rejeitadas devido a seus impactos negativos em outras metas de maior importância relativa.

3 - Unified Modeling Language

A Unified Modeling Language (UML) é uma linguagem de modelagem visual de propósito geral que é utilizada para especificar, visualizar, construir e documentar artefatos de um software. Ela captura decisões e conhecimentos sobre o software a ser construído. A UML é para ser utilizada para compreender, desenhar, configurar, manter e controlar informações sobre o software sendo desenvolvido. Em momento algum ela é atrelada com algum método de desenvolvimento, ciclos de vida de sistema ou domínios específicos, e portanto, pretende suportar a maioria dos processos de desenvolvimento de software orientados a objeto existentes [Rumbaugh 99].

Rumbaugh [Rumbaugh 99] divide a UML em áreas que por sua vez contêm visões, onde uma visão é um subconjunto de construtores da UML que representam um aspecto de um sistema. A UML fica então dividida em três áreas: estrutural, comportamento dinâmico e gerência de modelos.

A classificação estrutural descreve coisas existentes nos sistemas e seus relacionamentos com outras coisas. Classificadores incluem, entre outros, classes, ator, componentes, interface, tipo de dado e nós, podendo ser encontrados nas visões: estática, casos de uso e de implementação.

A área de comportamento dinâmico descreve o comportamento de um sistema no tempo. Comportamento pode ser descrito como uma série de fotografias de mudanças no

desenho do sistema retirados da visão estática. Esta área inclui as visões: máquinas de estado, actividade e interação.

A gerência de modelos descreve a organização dos modelos propriamente ditos através de unidades hierárquicas. Um pacote é uma unidade organizacional genérica para modelos. Pacotes especiais incluem modelos e subsistemas. A visão de gerência cruza as outras visões e organiza-as objectivando o trabalho de desenvolvimento e controle de configuração.

Nesta tese estaremos especialmente voltados para as visões estáticas.

3.1 - Visão Estática

A visão estática modela tanto conceitos pertinentes ao domínio da aplicação quanto conceitos inventados como parte da solução proposta para a implementação desta aplicação [Rumbaugh 99]. Esta visão é denominada de estática por não refletir os comportamentos dependentes de tempo do software, os quais são descritos em outras visões. Os principais participantes desta visão estão agrupados no diagrama de classes e são classes e seus relacionamentos, a saber: associação, generalização e vários outros tipos de dependência. A visão estática é mostrada através do diagrama de classes.

Um diagrama de classes descreve os tipos de objectos de um sistema e vários tipos de relacionamentos estáticos que existem entre eles. Existem dois tipos de relacionamentos que se destacam [Fowler 97]:

- Associações: um cliente pode alugar um certo número de cassetes de vídeos;
- Subtipos: Uma enfermeira é um tipo de pessoa.

Diagramas de classe mostram também atributos e operações de uma classe e as restrições que se aplicam como os objectos são interligados.

Fowler [Fowler 97] destaca ainda que um diagrama de classes pode ser construído utilizando-se três diferentes perspectivas:

1. Conceptual: representa basicamente conceitos do domínio em estudo e em geral é independente de linguagem ou arquitetura de implementação;
2. Especificação: representa um modelo já preocupado com a implementação do problema sob forma de um software, onde soluções de desenho são implementadas como, por exemplo, o uso de padrões;
3. Implementação: Aqui todos os detalhes de tratamento de classes são utilizados e o desenho passa a ser fortemente dependente de software e arquitetura utilizados.

Classes são descritas utilizando-se um retângulo com três diferentes compartimentos onde são mostrados respectivamente, o nome da classe, seus atributos e as operações desta classe. A representação de atributos e operações é comumente suprimida das classes excepto em um único diagrama.

Relacionamentos entre classes são representados como caminhos conectando duas classes (ou uma classe consigo mesma). Os diferentes tipos de relacionamentos são distinguidos pela textura da linha e adereços nos caminhos e terminações destes [Rumbaugh 99]. A Figura 3.1, extraída de [Fowler 97], mostra um exemplo onde aparecem os diversos componentes do diagrama de classes. É necessário ressaltar que o que aparece entre { }

são restrições impostas ao modelo, que podem obter a forma tanto de uma expressão formal quanto de uma sentença em linguagem natural.

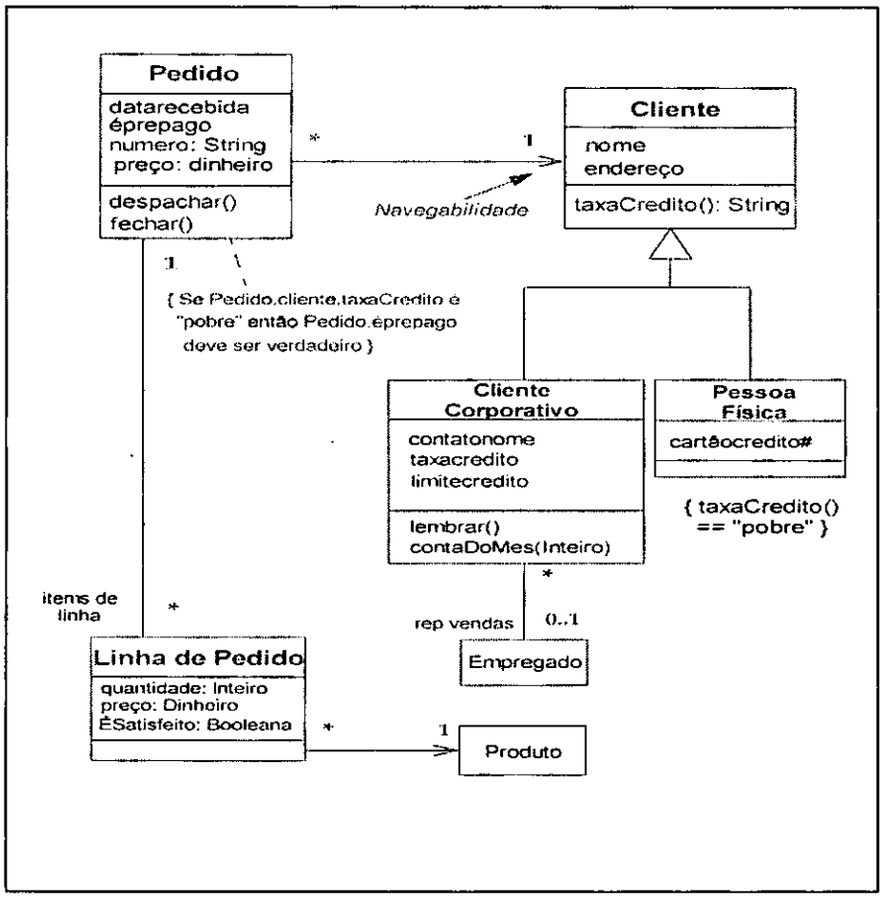


Figura 3.1 – Exemplo de um Diagrama de Classes em UML

Podemos vêr no exemplo mostrado na figura 3.1 que o diagrama lá mostrado indica uma associação entre as classes pedido e cliente onde o lado da associação referente ao cliente possui uma multiplicidade 1 e a de pedido uma multiplicidade *, indicando que um pedido tem de vir de apenas um cliente, mas um cliente pode realizar diversas ordens. As multiplicidades mais comuns são 1, *, 0..1 (podemos ter zero ou um). Esta associação especificamente é representada por uma seta que indica a navegabilidade desta

associação. Neste caso significaria que um pedido tem a responsabilidade de dizer a quais clientes ele pertence, enquanto um cliente não tem a capacidade de dizer quais pedidos possui.

Uma associação pode ainda ter um papel associado a ela, como por exemplo, a classe empregado da Figura 3.1 que possui o papel de representante de vendas.

É possível ainda, verificar na Figura 3.1 que a classe cliente é uma generalização das sub-classes cliente corporativo e pessoa física. Isto significa que as sub-classes irão herdar os atributos (nome e endereço) e operações (taxaCredito()) contidos na classe cliente, além de ter os seus próprios atributos e operações.

3.2- Visão de Interação

Esta visão descreve sequências de trocas de mensagens entre papéis que espelham o comportamento do software. Um classificador do papel é uma descrição de um objeto que desempenha uma actividade particular dentro de uma interação, que o distingue de outros objectos da mesma classe [Rumbaugh 99]. Um classificador possui identidade, estado, comportamento e relacionamentos. A UML define diversos tipos de classificadores entre eles: classes, actor, componentes, interface, tipo de dado e nós.

A visão de interação é mostrada através de dois diagramas que enfocam aspectos diferentes: o diagrama de seqüências e o diagrama de colaborações.

3.2.1 - O Diagrama de Sequência

O diagrama de sequência mostra um conjunto de mensagens dispostas de maneira temporal, ou seja, sequencialmente no tempo. Cada classificador é mostrado através de uma linha da vida do objeto, a qual é representada através de uma linha vertical.

Mensagens são mostradas como setas entre linhas da vida. Um diagrama pode mostrar um cenário, ou seja, a história individual de uma transação.

Cada mensagem possui pelo menos um nome de mensagem, sendo possível ainda incluir-se argumentos e mensagens de controle. É possível ainda representar-se auto-delegações que são mensagens enviadas de um objeto para ele próprio [Fowler 97].

Existem dois tipos de controles pré-definidos para serem utilizados no diagrama de sequência, *condição* e *marcador de interação.*

Condição é um controle utilizado para indicar que aquela mensagem só será enviada caso a condição expressa seja verdadeira, onde a condição estará especificada entre colchetes.

Marcadores de interação são controles que estabelecem que uma mensagem é enviada mais de uma vez, o que caracteristicamente ocorrerá quando quisermos atuar sobre uma colecção.

A Figura 3.2 mostra um exemplo de um diagrama de sequência extraído de [Fowler 97] onde podem ser vistos exemplos de condição e interação.

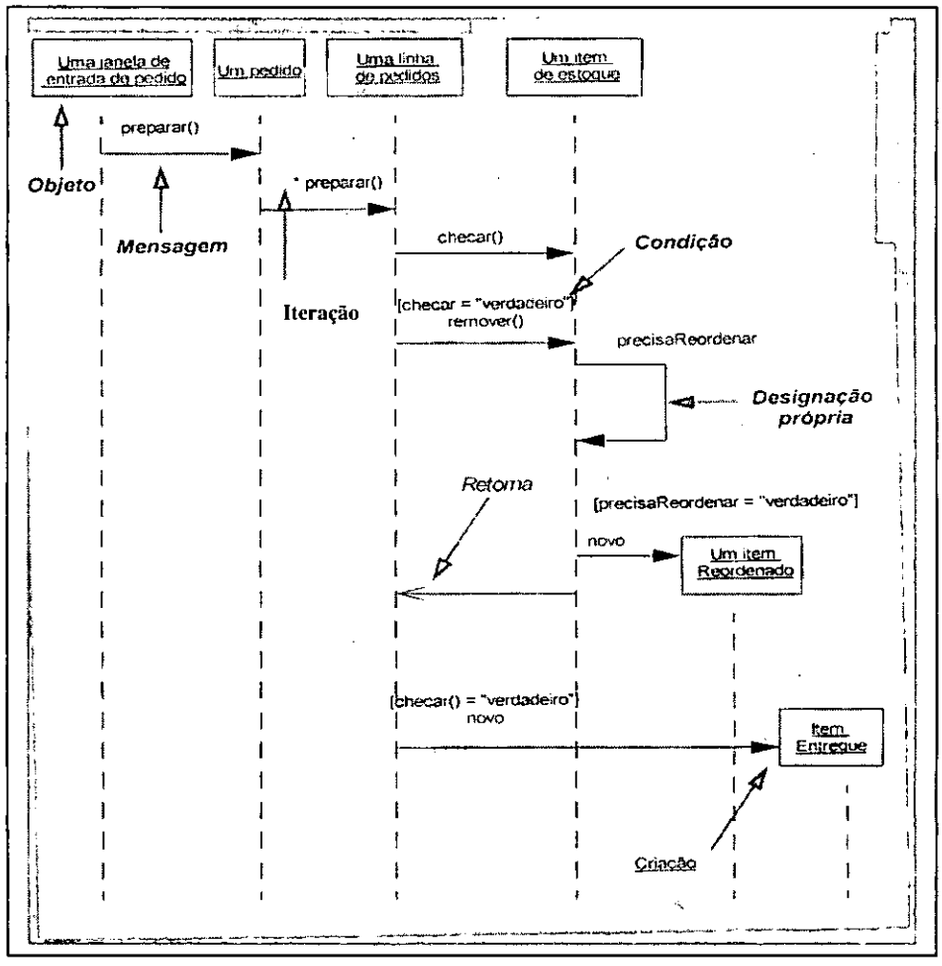


Figura 3.2– Exemplo de um Diagrama de Seqüência

3.2.2 - Diagrama de Colaborações

Em um diagrama de colaborações, ilustrado na Figura 3.3, os objectos são mostrados como ícones. Da mesma forma que no diagrama de sequência, as setas indicam as mensagens que são trocadas dentro de um dado cenário, sendo que no presente caso, a sequência é indicada por meio da numeração das mensagens.

O uso da numeração, se por um lado dificulta a visualização de sequências, por outro favorece que outros detalhes sejam mostrados mais facilmente devido a liberdade de diagramar [Rumbaugh 99].

Vários tipos de numeração podem ser utilizados, inclusive qualquer um que seja criado por um *developer*, contanto que a semântica da numeração utilizada fique bem clara para todos os envolvidos no projeto.

Os controles de condição e interação relatados na secção anterior são aplicáveis também no diagrama de colaborações.

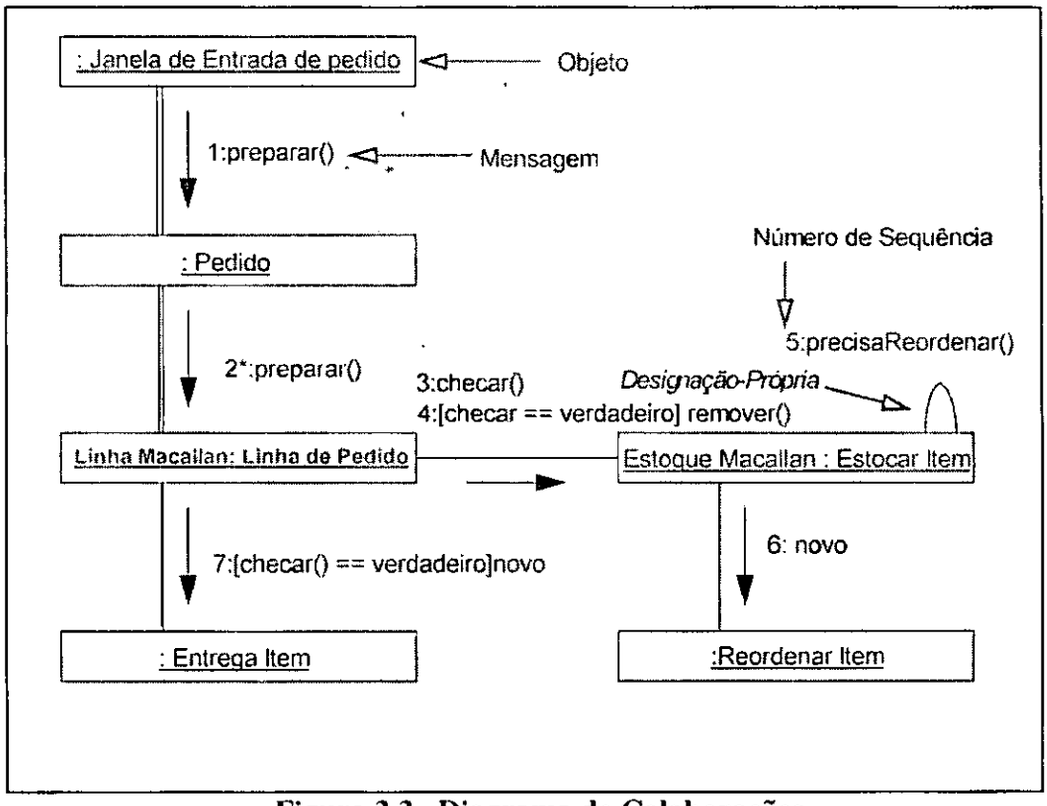


Figura 3.3 –Diagrama de Colaborações

4 - Lidando com RNFs: Da Análise ao Modelo Conceptual

A estratégia proposta nesta tese não tem por objectivo enfocar um RNF específico, mas sim, demonstrar como identificar e representar RNFs de uma maneira geral, proposta por Mylopoulos [Mylopoulos 92].

O simples facto de abordar e representar RNFs durante a etapa de aquisição de requisitos já ser um grande passo para a melhoria da qualidade do software obtido, a utilização de uma estratégia que permita a integração destes RNFs aos modelos que reflectem os requisitos funcionais contribui para que estes RNFs estejam presentes no desenho do sistema desde suas primeiras versões, diminuindo a possibilidade de futuros conflitos entre as visões funcional e não funcional do problema.

Para que possamos lidar com os RNFs durante grande parte do ciclo de vida do software, propômos uma estratégia que além de especificar uma maneira de se lidar com RNFs e representá-los, também os integre ao modelo conceptual que expressa os requisitos funcionais. Desta maneira, podemos obtêr um processo sistematizado para obtêr e representar RNFs e também para analisá-los, seja na óptica das interdependências entre eles como de suas interdependências com requisitos funcionais.

A estratégia propõe o uso da UML como âncora para todo o processo. A UML será utilizado para a construção tanto do modelo funcional como do não funcional facilitando,

desta maneira, a posterior integração de ambos os modelos. Em conseqüência, a construção tanto do grafo de RNFs (artefato usado para representação dos RNFs), bem como do diagrama de classes (artefato usado para representação num primeiro momento dos requisitos funcionais e posteriormente de todos os requisitos) deverá ser feita tomando-se o uso de símbolos do léxico como restrição para a nomeação de tipos de RNFs no grafo e das classes no diagrama de classes. Desta forma, uma entrada comum no léxico irá servir como ligação entre o modelo não funcional e o funcional. Isto facilita a análise de, em que parte do modelo conceptual um RNF irá impactar, ou seja, a correcta ligação entre o modelo não funcional e o funcional.

A estratégia proposta pode ser encarada como um meta-modelo que pode ser instanciado para ser utilizado tanto com o modelo entidade-associação quanto com o modelo orientado a objetos.

É importante ressaltar que a estratégia aqui proposta não depende do tipo de método utilizado no processo de desenvolvimento de software bem como não está atrelada a nenhum tipo de ciclo de vida de sistemas específico, sendo, portanto uma estratégia de uso geral.

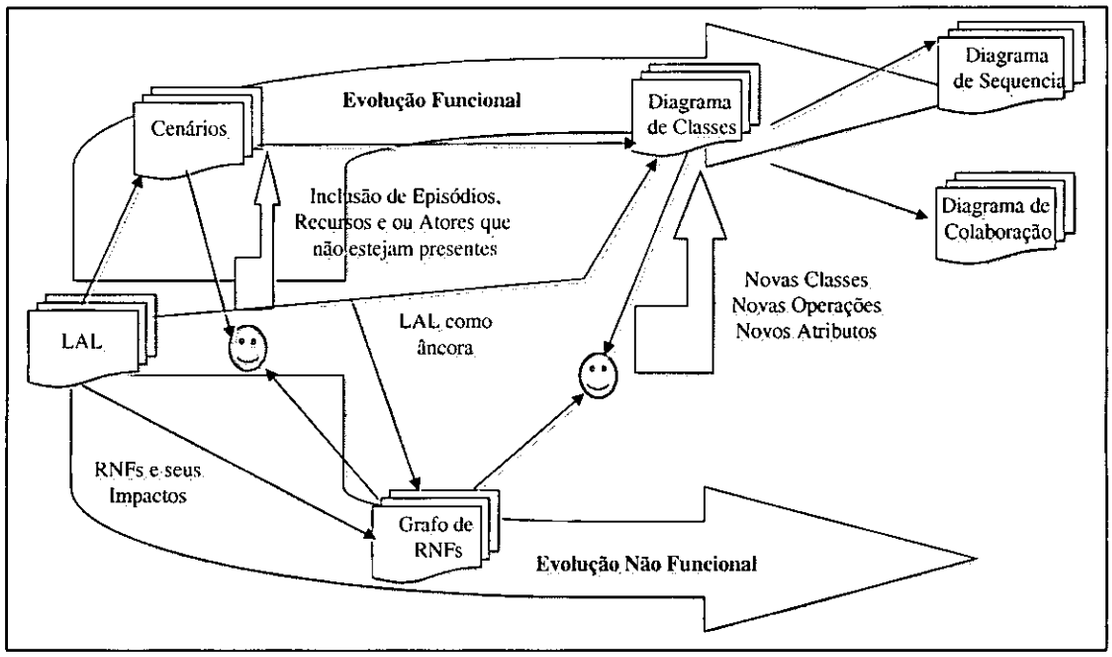


Figura 4.1 – Detalhamento da Estratégia Proposta

Na nossa proposta, analisamos os requisitos evoluindo sob dois ciclos distintos: o funcional e o não funcional. O detalhamento da estratégia pode ser visto na Figura 4.1. Nesta figura pode-se observar que o ciclo funcional seria composto pelos modelos de cenários, de classes, de seqüência e de colaboração, enquanto o ciclo não funcional seria composto basicamente do grafo de RNFs ou diagrama de classes.

5 - Estudo de Caso

5.1 – Dados do Sistema

A ONG Link é uma organização não governamental que regista e coordena as actividades das ONG's que operam ou que pretendem operar em Moçambique e reporta junto do governo ao Ministério de Negócios Estrangeiros e Cooperação.

Nas cheias de Março de 2001, muitas ONG's Nacionais e Estrangeiras em resposta ao pedido do Governo Moçambicano acorreram em apoio às vítimas nas regiões afectadas ou não pelas enchurradas. Para melhor coordenar as actividades desenvolvidas pelas ONG's era necessásio um SI que pudesse responder as múltiplas questões em tempo real. Por conseguinte o registo da ONG, se é Nacional ou Estrangeira, Zonas geográficas de Intervenção, Áreas e Projectos que foram ou estavam sendo realizados em cada Sector/subsector incluindo os montantes envolvidos e os respectivos Doadores, Meios de Comunicação disponíveis/usados, Meios de Mobilidade disponíveis/usados, localização dos Escritórios e Armazens, Pessoal empregue e Bens e Seviços disponibilizados.

O estudo que foi feito permitiu desenvolver modelos funcionais e não funcionais que resultaram em 14 classes para o modelo funcional e 15 novas classes para o modelo não funcional resultante da aplicação da estratégia proposta, do ponto de vista de análise tendo em conta os RNF's durante o ciclo de vida do Software. A tabela 5.1 ilustra os resultados obtidos durante o estudo de caso.

	Classes
Modelo conceptual original	14
Derivadas da aplicação da estratégia	15
Variação Percentual	13,81

Tabela 5.1- Resultados

Este estudo exibe todos RNF's achados para o estudo de caso e o detalhamento de grande parte das classes que sofreram alterações por conta de satisfazer RNFs, bem como os diagramas alterados.

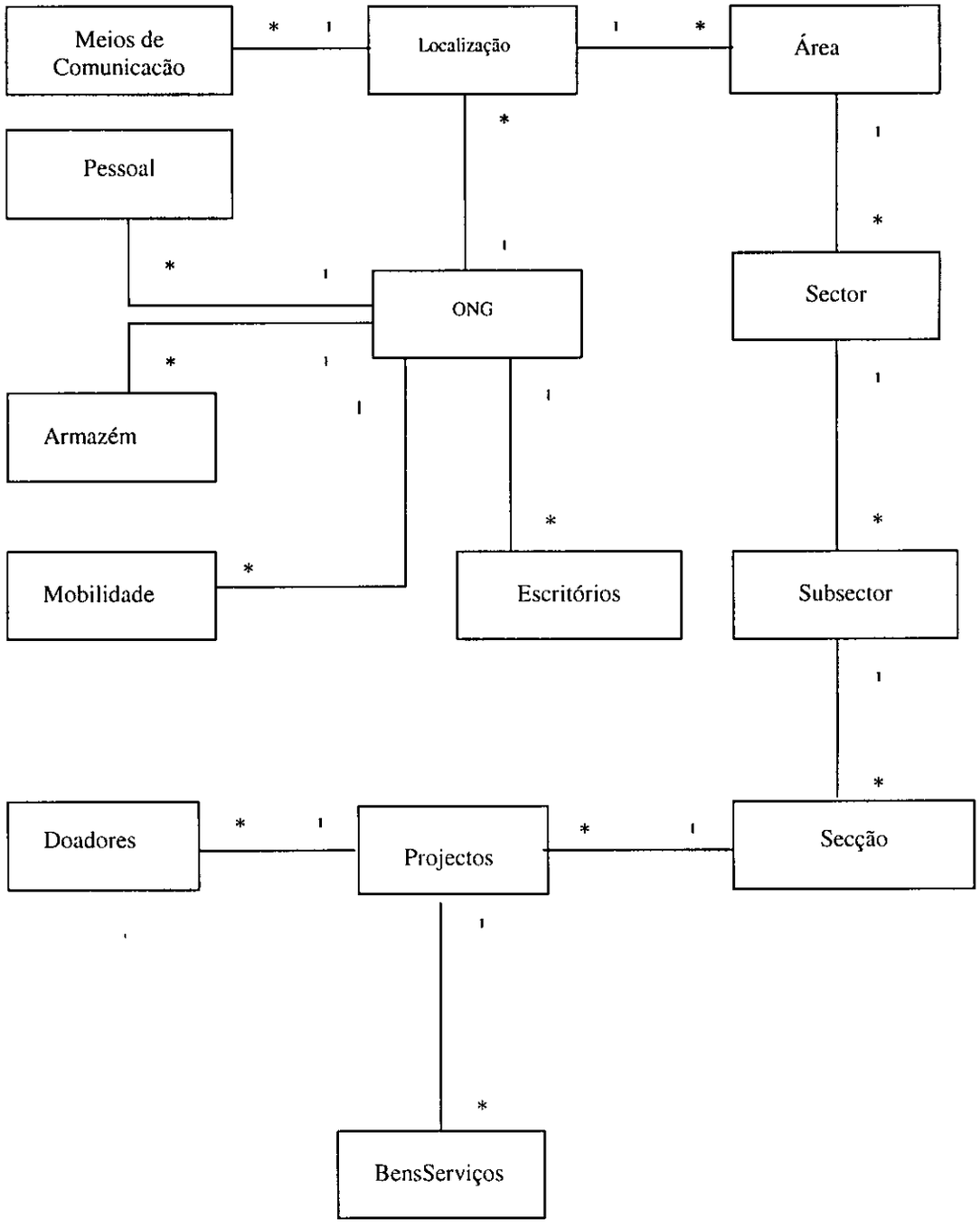
Lista de RNF's

1. O acesso ao sistema será feito apenas por utilizadores com a devida permissão
2. O sistema deverá registar as operações dos utilizadores
3. A Parametrização e Cadastro
4. O Sistema deverá ser de fácil uso

Actores do UDI:

1. Administrador do Sistema.
2. Utilizadores do Sistema.

5.2. Visão Funcional



5.3. Visão Não Funcional

- Possui **RNF Segurança**
- Possui **RNF Perfomance**
- Possui **RNF Reusabilidade**
- Possui **RNF Manutenabilidade**

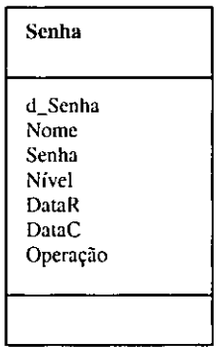
RNF Segurança: Permite-nos fazer o histórico das actividades dos utilizadores no sistema, para além de só utilizadores autorizados é que tem acesso.

RNF Perfomance: É desejável que o sistema seja flexível em termos de tempo de resposta em relação aos pedidos dos utilizadores.

RNF Reusabilidade: a parametrização permite-nos fazer uso dos mesmos dados (dados mestres).

RNF Manutenabilidade: a parametrização permite que o sistema seja de fácil uso e de baixo custo para a sua manutenção (ex: uma mudança de câmbio ou outro item evita que se reescreva o código de programa)

Para satisfazer o **RNF Segurança** foi necessário criar a classe **Senha**

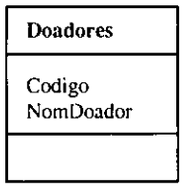


Esta classe permite que só pessoas autorizadas pelo administrador do sistema tenham acesso a determinados níveis no sistema (ex: introduzir dados, correr relatórios, visualizar paineis, parametrização e codificação,...) e por conseguinte satisfaz o **RNF Perfomance e RNF Reusabilidade.**

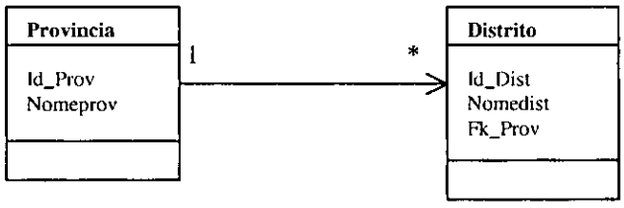
- Para Satisfazer o **RNF Manutenabilidade** adicionamos várias classes dentre os quais:
  - Paracodigos; Doadores; Província e Distritos; Área1, Sectore1s, Subsectores1 e Secção1.

Paracodigo

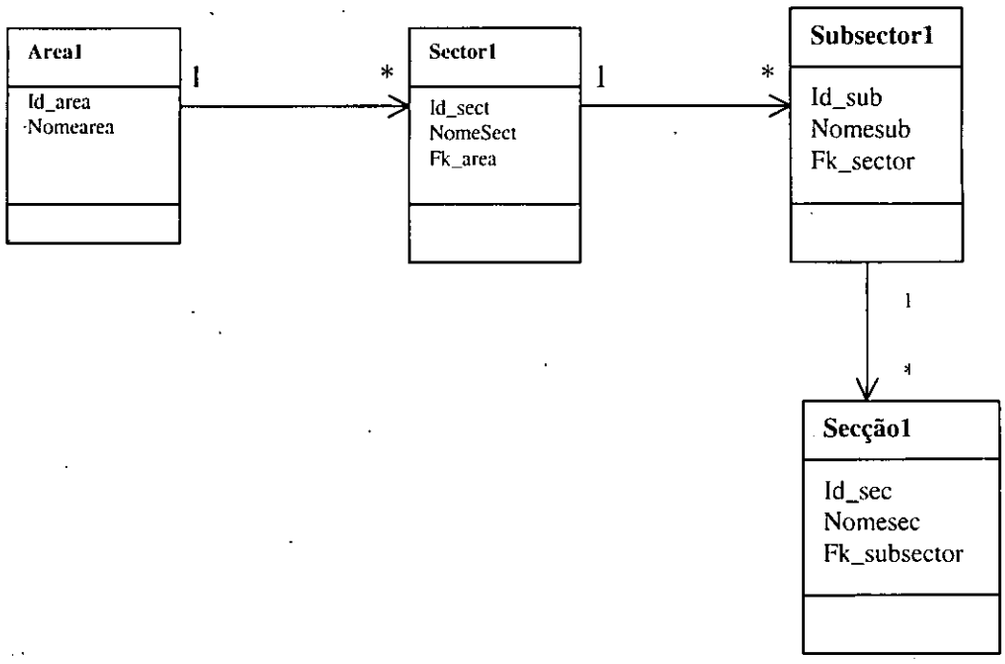
Doadores



Cadastrar Ongs : Esta classe permite introduzir novas Ong's e codificá-las, facilitando deste modo a intrudução e consultas de dados. O mesmo acontece com a classe Doadores.



Cadastrar Províncias e Distritos. As classes província e distrito, para além de facilitar o cadastro das mesmas, onde as Ong's intervém, a relação impede-nos de cometer erros do tipo Província de Maputo, Distrito de Dondo, quer dizer só podemos digitar um distrito corespondente a respectiva província, minimizando deste modo os erros na introdução de dados e evitando os custos na correção.

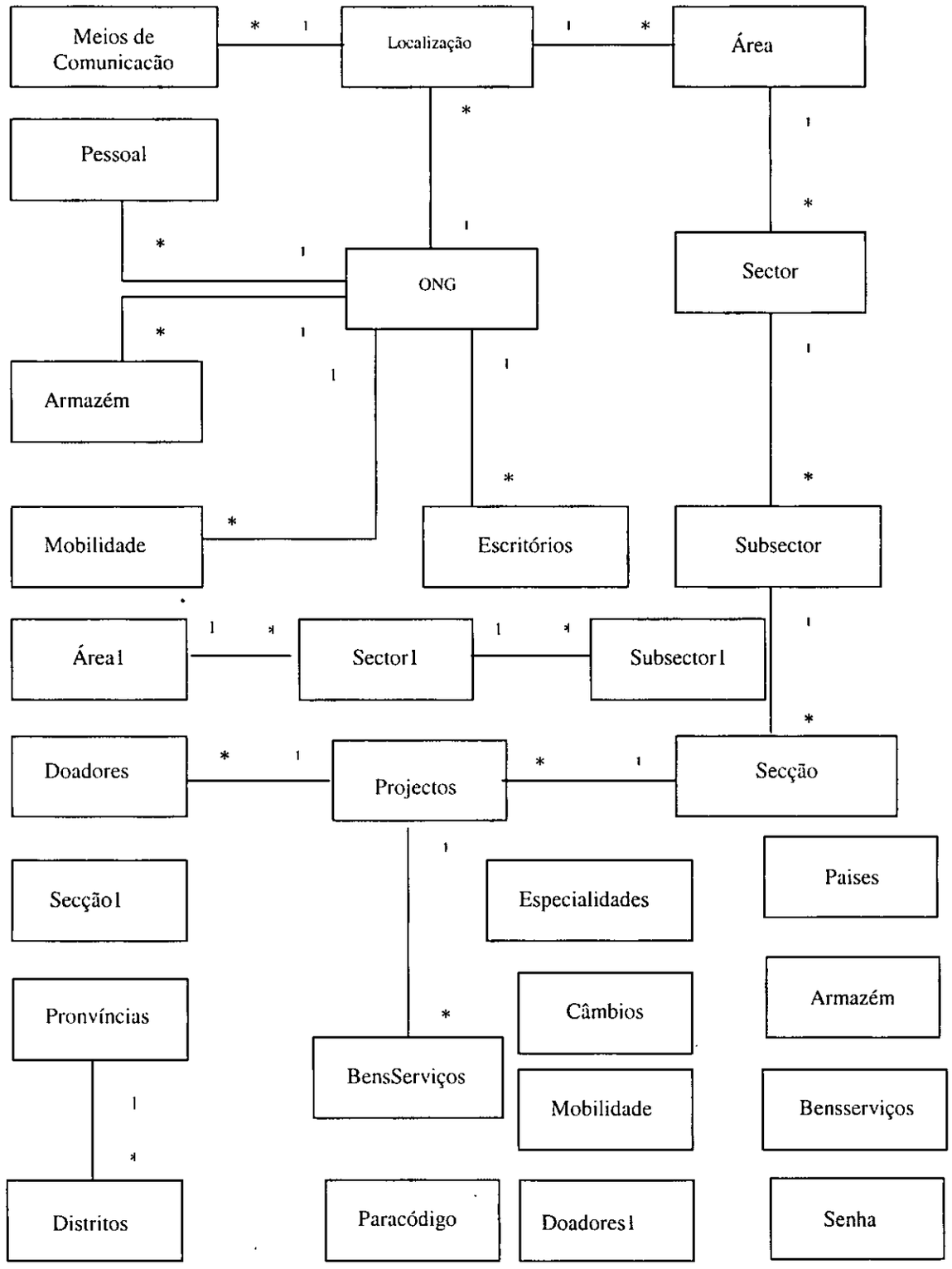


Cadastrar Area1, Sector1, Subsector1 e Secção1. Em relação a estas quatro classes é similar a situação acima.

A inclusão destas classes também satisfaz o **RNF Reusabilidade e Custo**

Outras classes: Paises, Armazém, Bensserviços, Câmbios, Mobilidades, Especialidades.

5.4. Visão Funcional e Não Funcional



6 – Conclusão e Recomendações

6.1. Conclusão

Nos últimos anos temos observado um substancial desenvolvimento da engenharia de requisitos. Como parte desse desenvolvimento foi mostrado que uma correcta análise de requisitos é vital para obtermos softwares de qualidade. Dentro do processo de análise de requisitos, tem sido dada uma grande atenção fundamentalmente para os requisitos funcionais. Actualmente a engenharia de requisitos apresenta várias propôstas para consistentemente lidar com os requisitos funcionais de um software, enquanto os requisitos não funcionais têm recebido pouca atenção. Entretanto, alguns autores vêm destacando os RNFs como sendo de crucial importância para a obtenção de softwares que atendam as expectativas dos clientes[Cysneiros 99].

Igualmente importante é estabelecer como os requisitos analisados irão guiar o resto do processo de desenvolvimento de software. Grande parte dos aspectos de qualidade do software é estabelecida durante o desenho do software[Yu 00]. Enorme parte dos aspectos de qualidade de um sistema é expressa por RNFs que, muitas vezes, são também denominados "atributos de qualidade de um software" [Boehm, 84]. Da mesma forma que os requisitos funcionais os RNFs estão em constante evolução e é importante não apenas sermos capazes de detectar essa evolução, mas também de avaliar suas consequências sobre o software em questão.

Mostramos durante a tese uma estratégia que estabelece como se analisar e tratar RNFs desde o início do processo de desenvolvimento de software. Propômos ainda um processo sistematizado de integração destes RNFs ao desenho do sistema, independentemente do método utilizado no desenvolvimento do software.

Dentro dessa estratégia apresentamos uma proposta de como produzir o que chamamos de visão não funcional do software, a qual representa o ciclo evolutivo dos RNFs, sua representação em modelos não funcionais. Mostramos então como esses RNFs irão restringir o modelo conceptual do software de forma a obtermos ao final um modelo conceptual mais completo que expresse não apenas os requisitos funcionais, mas também os não funcionais.

6.2. Recomendacões

1. Abordagem mais aprofundada dos RNFs nos conteúdos das disciplinas de Análise de Sistemas e Fundamentos de Base de Dados.
2. Considerar os RNFs para a avaliação de qualidade e aceitação de Sistemas de Informação.
3. Durante todo o processo de Análise e Desenho, deve-se prestar mais atenção não só aos RFs assim como também aos RNFs.

7– Bibliografia

7.1 – Bibliografia Referenciada

[AI 94] *Artificial Intelligence,* An International Journal, Special Volume on Qualitative Reasoning about Physical Systems, Vol. 24, No. 1-3. Dec. 1984.

[Berry 98] Berry, Daniel e Lawrence, Brian. *Requirements Engineering*. IEEE Software, March/April 1998, pp. 26-29

[Boehm 84] Boehm, Barry e In, Hoh. *"Identifying Quality-Requirement Conflict"s*. IEEE Software, March 1996, pp. 25-35

[Bowen 85] T. P. Bowen *et al,* "Specification of software quality attributes", Rep. RADC-TR-85-37, Rome Air Development Center, Grifiss Air Force Base, NY, Feb 1985

[Breitman 98] Breitman, K.K. *"Evolução de Cenários"* Tese de Doutorado submetida na PUC-Rio em Maio de 2000.

[Chung 95] Chung, L., Nixon, B. *"Dealing with Non-Functional Requirements: Three Experimental Studies of a Process-Oriented Approac*h" Proc. 17th Int. Con. on Software Eng. Seatle, Washington, April pp: 24-28, 1995.

[Chung 00] Chung, L., Nixon, B., Yu, E. and Mylopoulos,J. *"Non-Functional Requirements in Software Engineering"* Kluwer Academic Publishers 1999.

[Cysneiros 99] Cysneiros, L.M. and Leite, J.C.S.P. *'Integrating Non-Functional Requirements into data model"* 4th International Symposium on Requirements Engineering – Ireland June 1999.

[Finkelstein 96] Finkelstein, A. and Dowell J. *"A comedy of Errors: The London Ambulance Service Case Study"* Proceedings of the Eighth International Workshop on Software Specification and Design, IEEE Computer Society Press pp 2-5 1996.

[Fowler 97] Fowler, M. and Kendall Scott *"UML Destilled"* Addison-Wesley Inc, 1997

[ISO 9126] ISO9126 *"Information Technology – Software Product Evaluation – Quality Characteristics and Guidelines for their Use"* International Organization for Standardization, Geneva, 1992.

[Jacobson 99] Jacobson, I., Booch, G., Rumbauh, J. *"The Unified Software Development Process"* Addison-Wesley Inc, 1999.

[Keller 90] Keller, S.E. et al *"Specifying Software Quality Requirements with Metrics"* in Tutorial System and Software Requirements Engineering IEEE Computer Society Press 1990 pp:145-163

[Leite 97] Leite, J.C.S.P. et.al. " *Enhancing a Requirements Baseline with Scenarios.*" Requirements Engineering Journal, 2(4):184-198, 1997.

[Mamani 99] Mamani, Nestor A. *"Integrando requisitos não funcionais aos requisitos baseados em ações concertas"*, Dissertação de mestrado, Departamento de informática, PUC-RIO, Maio, 1999.

[Roman 85] Roma, G. *"A Taxonomy of Current Issues in Requirements Engineering"*, IEEE Computer Vol. 18, No. 4 Apr. 1985, pp:14-23

[Rumbaugh 99] Rumbaugh, J., Jacobson, I. and Booch,G. *"The Unified Modeling Language Reference Manual"* , Addison-Wesley, 1999.

[Sommerville 98] Sommerville, I. and Sawyer, P. *"Requirements Engineering – A Good Practice Guide"*. John Wiley & Sons, 1997.

[Yu 00] D. Gross, E. Yu *"From Non-Functional Requirements to Design through Patterns"* Proceedings of the 6th International Workshop on Requirements

7.2 – Bibliografia Consultada

[Chung 00] Chung, L., Nixon, B., Yu, E. and Mylopoulos,J. *"Non-Functional Requirements in Software Engineering"* Kluwer Academic Publishers 1999.

[Cysneiros 99] Cysneiros, L.M. and Leite, J.C.S.P. *'Integrating Non-Functional Requirements into data model"* 4th International Symposium on Requirements Engineering – Ireland June 1999.

[Fowler 97] Fowler, M. and Kendall Scott *"UML Destilled"* Addison-Wesley Inc, 1997

[Leite 97] Leite, J.C.S.P. et.al. " *Enhancing a Requirements Baseline with Scenarios.*" Requirements Engineering Journal, 2(4):184-198, 1997.