VINICIUS MIANA BEZERRA

TDRRC - TÉCNICA PARA DOCUMENTAÇÃO E RECUPERAÇÃO DE REQUISITOS NO CÓDIGO-FONTE ATRAVÉS DO USO DE ANOTAÇÕES

São Paulo

2011

VINICIUS MIANA BEZERRA

TDRRC - TÉCNICA PARA DOCUMENTAÇÃO E RECUPERAÇÃO DE REQUISITOS NO CÓDIGO-FONTE ATRAVÉS DO USO DE ANOTAÇÕES

Tese apresentada à Escola Politécnica da Universidade de São Paulo para a obtenção do título de Doutor em Engenharia

Área de Concentração:
Engenharia Elétrica

Orientadora:
Prof. Dr. Selma Shin Shimizu Melnikoff

São Paulo

2011

FICHA CATALOGRÁFICA

Bezerra, Vinicius Miana
TDRRC – técnica para documentação e recuperação de requisitos no código-fonte através do uso de anotações / V.M. Bezerra. -- São Paulo, 2011.
p.

Tese (Doutorado) - Escola Politécnica da Universidade de São Paulo. Departamento de Engenharia de Computação e Sistemas Digitais.

1. Metodologia e técnicas de computação 2. Engenharia de software 3. Processo de software 4. Processos fundamentais de desenvolvimento de software I. Universidade de São Paulo. Escola Politécnica. Departamento de Engenharia de Computação e Sistemas Digitais II. t.

Dedico este trabalho a Natalia e ao Felipe por me fazerem lembrar o que é importante.

AGRADECIMENTOS

À minha orientadora, Prof. Dr. Selma Shin Shimizu Melnikoff, pela oportunidade concedida de realizar este trabalho, orientação, apoio e incentivo.

Ao meu orientador de mestrado, Prof. Dr. Kechi Hirama, por toda sua ajuda e apoio para a realização deste trabalho, assim como pelos valiosos comentários realizados durante a qualificação que contribuíram para a conclusão desta tese.

Ao Prof. Dr. João José Neto pela grande contribuição oferecida no exame de qualificação, cujos comentários foram fundamentais para a conclusão desta tese.

Aos colegas professores e à Universidade Presbiteriana Mackenzie pelo incentivo, em especial aos professores Dr. Vilmar Pedro Votre e Dr. Arnaldo Vallim.

À minha irmã pelo apoio sem o qual não poderia realizar este trabalho.

À minha esposa pela infinita paciência.

Aos amigos do curso pelas horas de estudo e lazer em conjunto.

E a todos que direta ou indiretamente, contribuíram para a realização deste trabalho.

RESUMO

Manter os documentos de requisitos atualizados e recuperar os requisitos de um *software* são desafios enfrentados por desenvolvedores no seu dia a dia durante o desenvolvimento, a manutenção e a evolução de sistemas. Embora existam técnicas para gestão de requisitos, muitas vezes estas técnicas não são aplicadas, os requisitos não são atualizados e a única fonte de informação confiável sobre um *software* passa a ser seu código-fonte. Esta tese apresenta a TDRRC, uma técnica para a documentação e recuperação dos requisitos no código-fonte usando anotações. A TDRRC possibilita a reengenharia de requisitos sem que haja uma interrupção no desenvolvimento e permite que os requisitos sejam documentados em ambientes de desenvolvimento ágil. A TDRRC contribui para a redução dos problemas relacionados à atualização dos requisitos, pois o desenvolvedor responsável pelo programa passa a ser responsável pela documentação e atualização dos requisitos no código-fonte que ele escreve e mantém. Este trabalho apresenta também formas de aplicar a TDRRC na reengenharia de requisitos, em métodos ágeis e na gestão de requisitos, assim como a sua aplicação em um estudo de caso.

ABSTRACT

Keeping requirements documents updated and recovering requirements of a software are common challenges faced by developers on their day to day activities. Although there are many requirements management techniques, usually these techniques are not applied, requirements are not updated and the only reliable source of information about a software becomes its source code. This thesis presents TDRRC, a technique that can be used to document and retrieve requirements from the source code using annotations. Applying TDRRC, it is possible to reengineer the requirements of a software without interrupting its development. Also requirements can be documented in a agile environment. TDRRC also contributes to minimize requirements documents update issues as the developer will be clearly responsible for documenting and updating the requirements in the source code he is programming. This thesis also presents how to apply the technique in a requirement reengineering project, in a agile development environment and in a requirements management process. Finally a case study is presented.

LISTA DE ILUSTRAÇÕES

FIGURA 1 – METODOLOGIA UTILIZADA.....10

FIGURA 2 – EXEMPLO DE CASO DE USO.....23

FIGURA 3 – ESTILOS DAS LINGUAGENS FORMAIS (HARRY, 1997).....26

FIGURA 4 – META-MODELO DOS CASOS DE USO (RUI, 2007).....28

FIGURA 5 – MODELO CONCEITUAL DA ENGENHARIA REVERSA (CANFORA ET AL., 2008).....43

FIGURA 6 – MODELO DE GESTÃO DE MUDANÇAS (FAHMI ET AL., 2007).....46

FIGURA 7 – SINTAXE DOS TIPOS ANOTAÇÃO.....55

FIGURA 8 – EXEMPLO DE DECLARAÇÃO DE UM TIPO ANOTAÇÃO.....56

FIGURA 9 – EXEMPLO DE DECLARAÇÃO DE TIPO DE ALVO.....57

FIGURA 10 – EXEMPLO DE DECLARAÇÃO DE TEMPO DE RETENÇÃO.....57

FIGURA 11 – SINTAXE DA ANOTAÇÕES.....58

FIGURA 12 – EXEMPLO DE UMA ANOTAÇÃO.....58

FIGURA 13 – CARDINALIDADE DE UMA ANOTAÇÃO.....59

FIGURA 14 – EXEMPLO DE UMA ANOTAÇÃO EM ELEMENTOS DE UM PROGRAMA.....60

FIGURA 15 – EXEMPLO DE UMA ANOTAÇÃO EM UM PACOTE.....61

FIGURA 16 – EXEMPLO DE PROCESSADOR DE ANOTAÇÕES.....63

FIGURA 17 – VISÃO GERAL DA TDRRC.....68

FIGURA 18 – EXEMPLO DE META-MODELO DE REQUISITOS.....72

FIGURA 19 – EXEMPLO DE TIPO ANOTAÇÃO DO META-MODELO.....73

FIGURA 20 – EXEMPLO DE PROCESSADOR DE ANOTAÇÕES.....74

FIGURA 21 – EXEMPLO DE CÓDIGO-FONTE ANOTADO.....76

FIGURA 22 – META-MODELO DE CASOS DE USO APÓS REMOÇÃO DO NÍVEL DE EVENTOS.....85

FIGURA 23 – META-MODELO DE CASOS DE USO APÓS REMOÇÃO DE ATOR, OBJETIVO, TAREFA E USUÁRIO.....86

FIGURA 24 – META-MODELO DE CASOS DE USO ADAPTADO PARA A TDRRC 87

FIGURA 25 – EXEMPLO DE META-MODELO COM ASSOCIAÇÃO.....90

FIGURA 26 – TIPO ANOTAÇÃO MANTENDO ASSOCIAÇÃO DO META-MODELO 90

FIGURA 27 – ANOTAÇÕES COM REDUNDÂNCIA.....90

FIGURA 28 – TIPO ANOTAÇÃO SEM ASSOCIAÇÃO.....91

FIGURA 29 – ANOTAÇÕES SEM REDUNDÂNCIA.....91

FIGURA 30 – TIPO ANOTAÇÃO SERVIÇO.....92

FIGURA 31 – TIPOS ANOTAÇÃO CONDIÇÃO E CONTEXTO.....93

FIGURA 32 – TIPO ANOTAÇÃO EPISÓDIO.....93

FIGURA 33 – PSEUDO CÓDIGO DA RECUPERAÇÃO DO MODELO DE REQUISITOS.....96

FIGURA 34 – MODELO DE FAHMI ET AL. (2007) ADAPTADO PARA A TDRRC 105

FIGURA 35 – VISÃO GERAL DOS MÓDULOS DO SGVD.....115

FIGURA 36 – PROCESSO DE DESENVOLVIMENTO COM APLICAÇÃO DA TDRRC.....122

FIGURA 37 – DEMANDA DE REGISTRO DE EVENTOS PARA RENOVAÇÃO DE DOMÍNIOS.....125

FIGURA 38 – CASO DE USO REGISTRO DE DOMÍNIOS GERADO PELAS ANOTAÇÕES.....126

FIGURA 39 – CASOS DE USO DE COMPRA DE DOMÍNIOS GERADO PELAS ANOTAÇÕES.....127

FIGURA 40 – CASOS DE USO DE RENOVAÇÃO DE DOMÍNIOS GERADO PELAS ANOTAÇÕES.....127

FIGURA 41 – PRIMEIRAS LINHAS DO MÉTODO DE CONGELAMENTO DE DOMÍNIO ANTES DA REFATORAÇÃO.....130

FIGURA 42 – MÉTODO DE CONGELAMENTO DE DOMÍNIO APÓS REFATORAÇÃO E UM EPISÓDIO ANOTADO.....131

FIGURA 43 – DEMANDA DA PROMOÇÃO NOVOS DOMÍNIOS.....134

FIGURA 44 – PRÉ-CONDIÇÕES DO CASO DE USO COMPRA DE DOMÍNIOS..134

FIGURA 45 – CASOS DE USO DE COMPRA DE DOMÍNIOS COM INFORMAÇÃO DE RASTREABILIDADE.....136

LISTA DE TABELAS

TABELA 1 - MATRIZ DE RASTREABILIDADE.....39

TABELA 2 - MODELO DE REQUISITOS DA CALCULADORA.....77

TABELA 3 - OPERAÇÕES PARA AJUSTE NO META-MODELO.....81

TABELA 4 - PONTOS DE SCRUM ENTREGUES NOS SPRINTS.....137

LISTA DE ABREVIATURAS E SIGLAS

API – *Application Programming Interface*

APT - *Annotation Processing Tool*

BNF - *Backus Normal Form* ou *Backus–Naur Form*

CCS - *Calculus of Communicating Systems*

CRM – *Customer Relationship Management*

CSP - *Communicating Sequential Processes*

DNS - *Domain Name Server*

E-BNF – BNF Extendida

EJB - *Enterprise Java Beans*

ERP – *Enterprise Resource Planner*

IEEE – *Institute of Electrical and Electronics Engineers*

JEE - *Java Enterprise Edition*

JMS - *Java Messaging Service*

JSP - *Java Server Pages*

LSI – *Latent Semantic Indexing*

NA – Não se aplica

NC – Não Conhecido

OMG – *Object Management Group*

RMI - *Remote Method Invocation*

SOA – *Service Oriented Architecture*

TDRRC - Técnica para Documentação e Recuperação de Requisitos no Código-fonte

UML – *Unified Modeling Language*

VDM - *Vienna Development Method*

SUMÁRIO

Pág.

1 INTRODUÇÃO.....1

1.1 Motivação.....1

1.2 Objetivo.....4

1.3 Justificativa.....5

1.4 Metodologia de Trabalho.....10

1.5 Estrutura do Trabalho.....12

2 REPRESENTAÇÃO DE REQUISITOS FUNCIONAIS.....14

2.1 Principais Representações de Requisitos.....14

2.2 Casos de Uso.....21

2.3 Métodos Formais.....24

2.4 Um meta-modelo de Casos de Uso.....27

2.5 Considerações Finais.....31

3 GESTÃO DE REQUISITOS DE SOFTWARE.....34

3.1 Gestão de Requisitos.....34

3.2 Rastreabilidade de Requisitos.....39

3.3 Reengenharia de Requisitos.....41

3.4 Gestão de Requisitos Integrada à Engenharia Reversa.....46

3.5 Considerações Finais.....49

4 MECANISMO DE ANOTAÇÕES.....51

4.1 Introdução ao Mecanismo de Anotações.....51

4.2 Tipos Anotação.....54

4.3 Anotações....58

4.4 Processamento de anotações....61

4.5 Considerações Finais....64

5 TDRRC - TÉCNICA PARA DOCUMENTAÇÃO E RECUPERAÇÃO DE REQUISITOS....67

5.1 Visão geral da TDRRC....67

5.2 Artefatos da TDRRC....70

5.3 Atividades da TDRRC....77

5.4 Aplicação da TDRRC....100

5.5 Considerações Finais....109

6 ESTUDO DE CASO....111

6.1 Descrição do contexto....111

6.2 Aplicação da TDRRC....120

6.3 Situações Típicas....123

6.4 Análise dos Resultados Obtidos....137

6.5 Considerações Finais....141

7 CONSIDERAÇÕES FINAIS....143

7.1 Conclusão....143

7.2 Contribuições....147

7.3 Trabalhos Futuros....149

APÊNDICE I – GRAMÁTICA DAS ANOTAÇÕES EM JAVA....159

1 INTRODUÇÃO

Este capítulo apresenta a motivação, o objetivo, a justificativa, a metodologia do desenvolvimento deste trabalho e a sua estrutura.

1.1 MOTIVAÇÃO

Em geral, no contexto de desenvolvimento e manutenção de *software,* podem-se destacar as seguintes situações:

- Um processo de desenvolvimento ágil, em que a documentação dos requisitos é vista como um processo lento e de pouco valor e que a documentação formal tira agilidade do desenvolvimento (CAO; RAMESH, 2008);
- Manutenção de *software*, em que a documentação de requisitos não existe ou está desatualizada, fato que ocorre na maioria dos projetos de desenvolvimento de *software* depois de um certo tempo de uso (FAHMI et al., 2007);
- Manutenção de *software,* em que técnicas de gestão de requisitos foram aplicadas e um repositório de requisitos foi criado.

Ao desenvolver um sistema usando um processo ágil, os desenvolvedores concentram-se em entender as necessidades do usuário , representá-las de forma informal e descartável e implementá-las (PAETSCH et al., 2003). A constante reengenharia faz parte de um ambiente de desenvolvimento ágil, onde a refatoração é a principal técnica empregada. Seguindo estes processos, os desenvolvedores deveriam passar 50% do seu tempo programando novas funções e 50% refatorando código (SNEED, 2008; LAM; SHANKARARAMAN, 2002). Em função da constatação de que, com a exceção do código-fonte, todos os outros artefatos acabam ficando desatualizados e só servem para tornar o desenvolvimento mais lento, os métodos

ágeis não pregam a documentação formal dos requisitos (CAO; RAMESH, 2008; SNEED, 2008). Esta forma de trabalho permite que os requisitos sejam implementados mais rapidamente, porém perde-se o modelo de requisitos. Entre as vantagens trazidas pelos processos ágeis estão: menor tempo de desenvolvimento, maior contato entre o *stakeholder*[1] e a equipe de desenvolvimento e refatoração constante do *software* que, segundo seus criadores, permitiria impedir a degradação da arquitetura do *software* (CAO; RAMESH, 2008). Na prática, tudo funciona muito bem, quando se tem bons desenvolvedores com sólida experiência no desenvolvimento de *software* e sistemas de pequeno e médio porte. Ao utilizá-la em sistemas de grande porte com equipes heterogêneas, a arquitetura acaba se degradando de forma mais rápida que em um desenvolvimento tradicional, pois nos métodos ágeis os desenvolvedores têm um papel mais importante na definição da arquitetura do sistema. Além disso, devido à informalidade na captura de requisitos, tem-se o agravante de se possuir pouca informação sobre os requisitos do *software* (CAO; RAMESH, 2008).

Nos processos de desenvolvimento tradicionais, os requisitos são capturados e documentados com mais detalhes e, após uma sequência de atividades de análise e projeto, são implementados (ROYCE, 1998). É comum durante a execução destas atividades que estes requisitos implementados sejam distintos dos que foram documentados (LAMSWEERDE, 2008). Isto ocorre devido à falha na captura inicial dos requisitos, falha no processo de desenvolvimento e mudanças nos requisitos (KOTONYA; SOMMERVILLE, 1998). Com o passar do tempo, novos requisitos vão surgindo e as mudanças nos requisitos precisam ser gerenciadas para evitar que o problema se agrave (LAMSWEERDE, 2008). Na prática, ao usar processos de desenvolvimento tradicionais, muitas vezes, os documentos de requisitos não são atualizados, embora isso seja necessário. Quando o *software* já foi desenvolvido e os novos requisitos vem de uma evolução do *software*, o problema se torna ainda mais grave, pois muitas vezes os documentos iniciais já estão completamente

---

[1] O *stakeholder* é qualquer parte interessada no resultado do desenvolvimento, manutenção ou evolução do *software.* Pode ser, por exemplo, o cliente, usuário ou investidor.

desatualizados ou perdidos (LAMSWEERDE, 2008). Com o passar do tempo, ocorre a degradação da arquitetura, o custo de manutenção vai aumentando e o desenvolvimento de um novo *software* para substituir o antigo vai se tornando mais atrativo; porém devido à falta da documentação dos requisitos, o *software* passa a ser a única fonte das regras de negócio que foram implementadas ao longo dos anos (LIU, 2005).

Algumas organizações, porém, conseguem implantar, com sucesso, o gerenciamento de requisitos, tornando possível evoluir um *software* e manter seus documentos de requisitos atualizados e coerentes. O gerenciamento de requisitos é um processo contínuo durante o desenvolvimento e evolução de um *software* e engloba a documentação, análise, priorização, validação, controle de mudanças dos requisitos (KOTONYA; SOMMERVILLE, 1998). Visando aumentar a produtividade neste processo e facilitar a identificação de impactos das mudanças, foram criados mecanismos de rastreabilidade que permitem ligar um código-fonte ao requisito que o originou. O aumento de produtividade no gerenciamento de requisitos devido à aplicação dos mecanismos de rastreabilidade é mais visível em sistemas de grande porte e, por este motivo, a rastreabilidade é considerada fundamental para o sucesso de um processo de gerenciamento de requisitos deste tipo de sistema (WEBER et al., 2005; SCHWARZ et al., 2008; LAMSWEERDE, 2008). No entanto, apesar de trazer grandes avanços para a produtividade, é trabalhoso acessar as informações básicas dos documentos de requisitos, pois existe a necessidade de procurar e abrir documentos que estão em um repositório, o que poderia ser evitado durante a programação de uma alteração de um requisito se os mesmos estivessem registrados no código-fonte.

Como se pode perceber nestas três situações, enfrentam-se os seguintes desafios:

- Como manter os documentos de requisitos atualizados sem burocratizar o processo ágil, mantendo-o centrado no desenvolvedor;
- Como recuperar requisitos na manutenção de sistemas, quando os documentos de requisitos não existem ou estão desatualizados;

- Como permitir ao desenvolvedor atualizar e consultar o modelo de requisitos de forma mais ágil na manutenção de sistemas, quando as técnicas de gestão de requisitos foram aplicadas.

Pensando nestes desafios, estabeleceu-se o objetivo deste trabalho.

1.2 OBJETIVO

O objetivo desta tese é desenvolver uma técnica para representação dos requisitos funcionais diretamente no código-fonte, que foi denominada TDRRC (Técnica para Documentação e Recuperação de Requisitos no Código-fonte). Desta forma, esta técnica permite minimizar a desatualização dos requisitos e recuperar os documentos de requisitos a partir do código-fonte. Com o uso da TDRRC é possível, por meio da análise do código-fonte, determinar os requisitos que estão implementados em um dado *software* e os trechos de código que os implementam.

Para o desenvolvimento da TDRRC, foi utilizado o mecanismo de anotações que é um recurso de meta-modelagem presente em diversas linguagens de programação modernas. Por meta-modelagem, entende-se como a capacidade de criar modelos de modelos que, no caso específico da técnica, corresponde ao modelo de um modelo de requisitos (meta-modelo de requisitos) (BARGMEYER et al., 2000). Este recurso permite ao desenvolvedor, definir o modelo de requisitos que segue um meta-modelo de requisitos previamente definido.

Para especificar a TDRRC, foi utilizada a linguagem Java que possui o mecanismo de anotações desde a versão 5.0; esta linguagem foi escolhida por ser bastante utilizada e possuir um dos mecanismo de anotações mais completos e maduros (BLOCH, 2008). No entanto, a TDRRC pode ser aplicada em qualquer linguagem que tenha suporte ao mecanismo de meta-modelagem, sendo necessário fazer apenas os ajustes de sintaxe, pois os outros aspectos são tratados e ajustados durante a aplicação da TDRRC.

Com uso da TDRRC, cria-se um meta-modelo de requisitos e uma forma de modelar os requisitos usando este meta-modelo e o mecanismo de anotações. O

desenvolvedor passa a documentar os requisitos funcionais diretamente no código-fonte através de anotações. Uma vez que as anotações estejam feitas no código-fonte, um processador de anotações, pode extrair o modelo de requisitos e gerar documentos de requisitos. É possível também, através de um processador de anotações, sincronizar os requisitos com outros requisitos já existentes e armazenados em um repositório. Ao representar os requisitos diretamente no código-fonte, espera-se evitar problemas relacionados a requisitos desatualizados em relação ao que foi implementado e minimizar a necessidade de mecanismos de rastreabilidade. A TDRRC permite que os desenvolvedores documentem os requisitos de maneira gradual, com o conhecimento adquirido durante o processo de manutenção. Por esta característica, a TDRRC permite a recuperação gradual dos requisitos sem que haja a necessidade de um esforço concentrado de reengenharia de requisitos.

A TDRRC torna-se particularmente interessante nos processos ágeis, pois permite ao desenvolvedor, após sua reunião informal de captura de requisitos, documentar o que foi especificado no próprio código-fonte que vai ser implementado. O uso da TDRRC nesta situação permite que os documentos de requisitos passem a existir e se mantenham atualizados sem um grande aumento da burocracia evitada pelos processos ágeis.

1.3 JUSTIFICATIVA

Este trabalho se posiciona dentro de duas importantes áreas da Engenharia de *Software*:

- A Engenharia de Requisitos, que visa o estudo e a criação de métodos, técnicas e processos para lidar com os requisitos, englobando as atividades envolvidas na descoberta, elicitação, avaliação, especificação, documentação e evolução dos objetivos, funcionalidades, qualidade e restrições de um *software* (LAMSWEERDE, 2008; LAUESEN, 2002; ZAVE, 1997; KOTONYA; SOMMERVILLE, 1998).

- A Evolução de S*oftware,* que compreende as atividades realizadas após a implantação do *software*, visando a alteração do mesmo. Engloba várias atividades de modificação do código-fonte, onde a leitura e compreensão deste são necessárias para a realização de atividades como: aplicar atualizações de segurança, adicionar novas funcionalidades, corrigir erros e melhorar o desempenho (BENNETT; RAJLICH, 2000; CHAPIN et al., 2001; IEEE CCSE, 2004).

Na intersecção entre estas áreas, encontram-se a área de estudo deste trabalho. Nesta intersecção, encontram-se a Evolução de Requisitos, que é estudada pela Evolução de *Software*, e a Gestão de Requisitos, que é estudada pela Engenharia de Requisitos. Enquanto, na Gestão de Requisitos, o enfoque está em administrar a mudança; na Evolução de Requisitos está em determinar o que muda, como muda e por que muda. A TDRRC contribui com a Gestão de Requisitos; ao propor uma forma de documentar, armazenar e recuperar os requisitos; e serve de apoio à Evolução de Requisitos, pois com o registro dos requisitos no código-fonte e o seu armazenamento em sistemas de controle de versão fica mais fácil estudar e relacionar mudanças nos requisitos com mudanças no *software.* Ambas as áreas são bastante críticas para a evolução de *software,* pois para aplicar uma mudança ao *software* é necessário saber o que o *software* faz hoje e o que deveria fazer (LAMSWEERDE, 2008). A definição do que um *software* deveria fazer é um problema clássico de definição dos requisitos; no entanto, depois de algum tempo, descrever o que um *software* faz nem sempre é uma tarefa trivial, pois muitas vezes os documentos de requisitos não são atualizados depois que o *software* é implementado (FAHMI et al., 2007). Dentre as abordagens para resolver o problema de determinar o que o *software* faz hoje encontram-se a rastreabilidade e a engenharia reversa (FAHMI et al., 2007; CANFORA; PENTA, 2007; SCHWARZ et al., 2008; JERMAKOVICS et al., 2008).

Nas pesquisas da literatura sobre o tema, encontrou-se um grande número de publicações sobre propostas para a melhoria de mecanismos de rastreabilidade de requisitos. A rastreabilidade é a capacidade de associar um artefato do *software* a

outros artefatos correlatos, especialmente aqueles relacionados a requisitos. Através de ligações de rastreabilidade pode-se correlacionar um trecho de código ao requisito que demandou que este código fosse escrito e vice-versa (EGYED et al., 2009). A rastreabilidade é um fator importante para diminuição do risco na evolução de *software*, porém a grande maioria dos *software* não foi desenvolvida para prever rastreabilidade. Diversos estudos foram feitos para obter a rastreabilidade entre código e requisitos após o desenvolvimento ter sido concluído. Muitos deles utilizam uma técnica chamada *Latent Semantic Indexing* (LSI) que busca determinar o significado do texto de descrição dos requisitos e encontrar estes trechos no código-fonte; dentre estes destacam-se os trabalhos de Marcus e Maletic (2003), Jermakovics et al. (2008), De Lucia et al. (2008), Lormans e Deursen (2009) e Oliveto et al. (2010). Os trabalhos pesquisados que utilizam rastreabilidade de requisitos têm, como limitação, partir do pressuposto de que os documentos de requisitos existem e estão atualizados.

Os trabalhos em engenharia reversa permitem que se recuperem modelos de projeto, diagramas de classes, atividades, a partir do código-fonte de maneira automática e sem intervenção humana (FAHMI et al., 2007; CANFORA et al., 2008). A engenharia reversa não consegue, porém, recuperar de maneira automática artefatos de nível de abstração mais alto, entre eles, os requisitos de um sistema (YU et al., 2005; FAHMI et al., 2007). Por não poder ser recuperada de forma automática e por depender de intervenção humana, a extração dos requisitos a partir do código-fonte através da engenharia reversa é extremamente custosa e falha (FAHMI et al., 2007; CANFORA et al., 2008). Existem poucos trabalhos publicados tratando especificamente da engenharia reversa de requisitos. Na pesquisa bibliográfica realizada no presente trabalho, não foram encontrados artigos tratando do assunto mais recentes do que o artigo de Canfora et al. (2008). Canfora et al. (2008) apresenta o estado da arte da engenharia reversa e tem o trabalho de Yu et al. (2005) como única referência para engenharia reversa de requisitos. Yu et al. (2005) apresentam um *workshop* sobre o assunto realizado na décima-segunda conferência de engenharia reversa do IEEE (12th *Working Conference on Reverse Engineering*) e não trata o assunto com muita profundidade.

Na pesquisa realizada no presente trabalho, foram encontrados os seguintes trabalhos tratando da engenharia reversa de requisitos:

- Liu (2005): apresenta uma técnica semiótica para engenharia de requisitos que se baseia fortemente na utilização do *software* e entrevistas com usuários. Esta técnica, muitas vezes é a única opção, pois pode ser utilizada mesmo sem a existência do código-fonte, porém tem como limitação ser intensamente manual e sujeita a erros;

- Yang et al. (2006): apresenta um *framework* para engenharia reversa de requisitos não funcionais do *software.* Esta técnica foi capaz de recuperar esta informação de maneira automática, porém se limita à recuperação de requisitos não funcionais;

- Fahmi et al. (2007): apresenta um modelo de um ambiente onde a engenharia reversa de requisitos faz parte do processo de desenvolvimento de um *software* em evolução. Ele reforça a ausência de publicações com uma solução automática para o problema e a importância da resolução deste problema para evolução de *software*.

Ao invés de tentar recuperar um artefato de nível de abstração mais alto (requisitos) a partir de um artefato derivado e de nível de abstração mais baixo (código-fonte) como propõem as técnicas de engenharia reversa, este trabalho propõe que se inclua junto com o código-fonte de um *software*, as informações de requisitos. Com esta abordagem, o trabalho da engenharia reversa se torna mais fácil ou desnecessário, pois se o código-fonte passa a conter informações de nível de abstração mais alto, fica mais fácil recompor ou recuperar esta informação. Com isto, este trabalho contribui com a solução do problema da falta de documentos de requisitos dos sistemas citado por Canfora e Penta (2007), Schwarz et al. (2008), Fahmi et al. (2007), Yu et al. (2005) entre outros.

Este trabalho contribui também com uma proposta que simplifica ou elimina a necessidade de criar e manter as ligações de rastreabilidade (*trace-links*), pois informações que precisam ser conectadas passam a estar no próprio código-fonte.

Resumindo, este trabalho se justifica por propor uma abordagem que contribui para solução dos seguintes problemas citados por pesquisadores nas áreas de Engenharia Reversa de Requisitos, Evolução de Requisitos e Rastreabilidade de Requisitos:

- Ausência de documentos de requisitos atualizada (CANFORA; PENTA, 2007; FAHMI et al., 2007; YU et al., 2005);
- Dificuldade em encontrar o trecho de código que implementa um dado requisito (SCHWARZ et al., 2008);
- Dificuldade em determinar qual requisito está sendo implementado por um certo trecho de código (SCHWARZ et al., 2008);
- Dificuldade em determinar quais mudanças deverão ser feitas em um sistema para que ele atenda um certo requisito (CIRACI et al., 2007);
- Ausência de formas de verificar se um requisito foi implementado ou não (FAHMI et al., 2007);
- Dificuldade em verificar se houve mudanças em um requisito (FAHMI et al., 2007).

1.4 Metodologia de Trabalho

A metodologia de desenvolvimento deste trabalho está representada na figura 1.

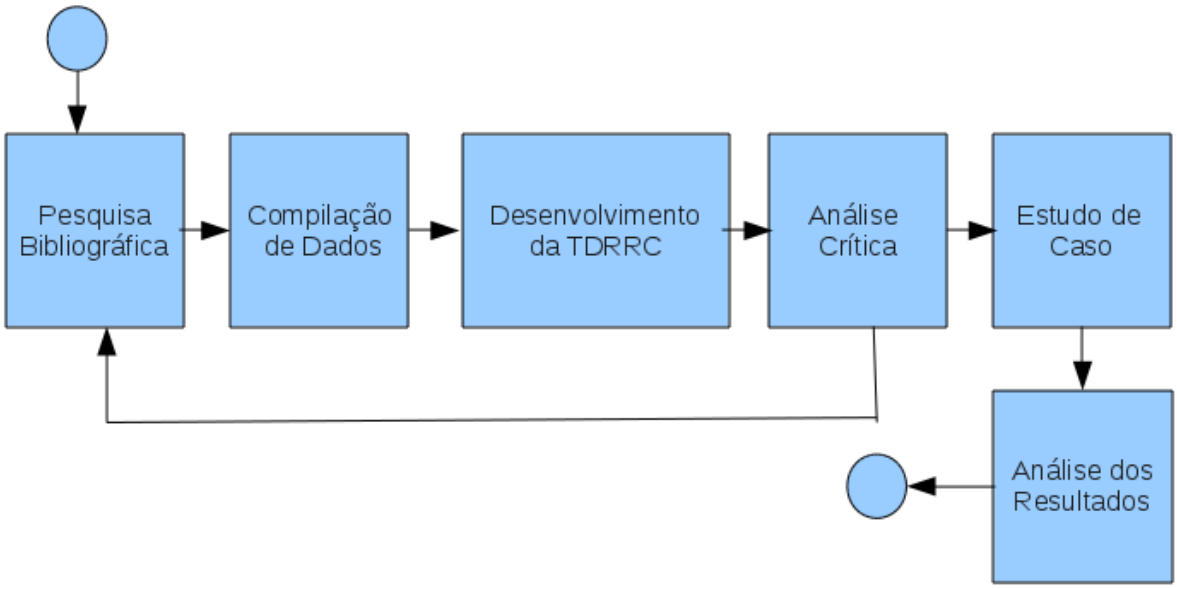

Figura 1 – Metodologia Utilizada

O trabalho teve como ponto de partida uma Pesquisa Bibliográfica, na qual as destacam-se as seguintes referências:

- Canfora e Penta (2007), Canfora et al. (2008) que apresentam as técnicas, um modelo conceitual e o estado da arte da Engenharia Reversa;
- Cao e Ramesh (2008) e Paetsch et al. (2003) que discutem a aplicação da Engenharia de Requisitos em métodos ágeis;
- Chung e Leite (2009) que apresenta requisitos não funcionais;
- De Lucia et al. (2008), Marcus e Maletic (2003), Lormans e Deursen (2009) e Oliveto et al. (2010) que apresentam técnicas para recuperação automática da rastreabilidade de requisitos;

- Fahmi et al. (2007) que propõe um processo de desenvolvimento iterativo baseado na Gestão de Requisitos;
- Gotel e Finkelstein (2002) e Ramesh (1998; 2005) que discutem a rastreabilidade de requisitos;
- Lam e Shankararaman (2002), Robertson e Roberston (2006) e Leffingwell (2006) que discutem a Gestão de Requisitos;
- Lamsweerde (2008), e Zave (1997) que colocam a Engenharia de Requisitos dentro de um contexto histórico e apresentam o estado da arte;
- Lauesen (2002), Kotonya e Sommerville (1998) que apresentam a Engenharia de Requisitos em geral e as formas de representar requisitos;
- Liu (2005) que define um processo e técnicas para a reengenharia de requisitos através de entrevistas com usuários, uso do *software*;
- Rui (2007) que apresenta uma proposta para refatoração de casos de uso e um meta-modelo de casos de uso usado para definir estas refatorações;
- Schwarz et al. (2008) que apresenta a rastreabilidade de requisitos dentro do contexto da Evolução de *Software;*
- Sneed (2008) que apresenta um panorama geral da Reengenharia de *Software*;
- Yang et al. (2006) que apresenta uma técnica para Engenharia Reversa de requisitos não funcionais;
- Yu et al. (2005) que apresenta os tópicos de interesse da Engenharia Reversa de requisitos.

Com os resultados da pesquisa, foi feita uma Compilação de Dados que sumarizou as informações pertinentes aos potenciais objetivos deste trabalho.

Com estes dados, foi realizado o Desenvolvimento da TDRRC e uma técnica candidata foi detalhada, documentada e avaliada.

Foi feita então uma Análise Crítica, onde a TDRRC foi confrontada com outras referências para avaliar o seu potencial de atingir o objetivo deste trabalho e trazer uma contribuição significativa. Enquanto o resultado da análise não foi satisfatório, o processo foi reiniciado com uma ampliação da Pesquisa Bibliográfica, uma nova Compilação de Dados, um aprimoramento no Desenvolvimento da TDRRC e uma nova Análise Crítica.

Uma vez que os resultados da Análise Crítica foram satisfatórios, foi realizado um Estudo de Caso onde a TDRRC foi aplicada.

Finalmente foi feita uma Análise dos Resultados da aplicação da TDRRC no Estudo de Caso.

1.5 Estrutura do Trabalho

A divisão de capítulos foi feita como da seguinte forma:

O capítulo 1, Introdução, apresenta a introdução a este trabalho com sua motivação, objetivo, justificativa, metodologia e a estrutura descrita.

O capítulo 2, Representação de Requisitos Funcionais, apresenta as formas mais usuais de descrever requisitos funcionais, mostra com mais detalhes as técnicas mais relevantes, e finaliza com a descrição de um meta-modelo para representação de casos de uso que é usado como exemplo na TDRRC.

O capítulo 3, Gestão de Requisitos de *Software*, apresenta um processo de gestão de requisitos, assim como as técnicas para rastreabilidade de requisitos e os trabalhos na área de engenharia reversa de requisitos. Estes conceitos são importantes para a integração da técnica dentro de um processo de gestão de requisitos.

O capítulo 4, Mecanismo de Anotações, apresenta o funcionamento do mecanismo de meta-modelagem presente em diversas linguagens de programação modernas e usa como base a implementação destes mecanismos na linguagem JAVA. Neste capítulo são discutidas também as outras opções para meta-modelagem e representação dos requisitos no código-fonte que são comparadas com o mecanismo de anotações.

O capítulo 5, TDRRC – Técnica para documentação e recuperação de requisitos, apresenta uma descrição da TDRRC e detalha seus artefatos, suas atividades. Apresenta, também, exemplos dos artefatos e a aplicação da TDRRC dentro dos contexto de reengenharia de requisitos, dos métodos ágeis e integrada à gestão de requisitos.

O capítulo 6, Estudo de Caso, apresenta um estudo de caso, onde a TDRRC foi aplicada em um ambiente de desenvolvimento ágil e os resultados obtidos.

O capítulo 7, Considerações Finais, apresenta as conclusões, as contribuições e a discussão de possíveis evoluções deste trabalho.

2 REPRESENTAÇÃO DE REQUISITOS FUNCIONAIS

Os requisitos funcionais constituem a maior parte dos requisitos de um sistema (KOTONYA; SOMMERVILLE, 1998) e os requisitos não funcionais, na maioria dos casos, não estão ligados diretamente a um trecho de código e sim a características gerais do sistema, processo de desenvolvimento ou ambiente operacional (CHUNG; LEITE, 2009).

Este capítulo apresenta as principais representações de requisitos funcionais encontradas na literatura. A seguir, apresenta com mais detalhes a representação de requisitos em casos de uso e os métodos formais. Os casos de uso são apresentados com mais detalhes por serem a forma mais utilizada pelas empresas para representar os requisitos, pela sua facilidade de uso, pelo tratamento sistemático das tarefas do usuário e por ser um padrão importante dentro da indústria de *software.* Os métodos formais são apresentados com mais detalhes para permitir um maior entendimento das vantagens e desvantagens da escolha dos casos de uso que é uma forma de representação não formal. Também neste capítulo é apresentado um meta-modelo para representação de requisitos funcionais baseado em casos de uso. Finalmente, são tecidas as considerações finais sumarizando os conceitos apresentados no capítulo.

2.1 Principais Representações de Requisitos

As representações de requisitos de um sistema correspondem a um conjunto de especificações, criados, na maioria das vezes, antes da implementação deste sistema e descrevem aquilo que deveria ser implementado (KOTONYA; SOMMERVILLE, 1998; ROBERTSON; ROBERTSON, 2006). Estas especificações são descrições do objetivo dos seus usuários em relação ao sistema, seu comportamento para atingir este objetivo, as restrições na sua operação, as informações do domínio da aplicação, propriedades do sistema e atributos, a forma de utilização, armazenamento, computação, transmissão e atualização dos dados,

etc. (LAUESEN, 2002; ROBERTSON; ROBERTSON, 2006; KOTONYA; SOMMERVILLE, 1998). Dentro do conjunto de requisitos, os requisitos funcionais são aqueles que especificam o comportamento que o sistema deve ter para ser útil aos seus usuários e os dados necessários para que esse comportamento seja obtido (ROBERTSON; ROBERTSON, 2006). Estas especificações correspondem aos requisitos funcionais e são usualmente descritas através de abstrações, ou seja, são expressas de uma maneira que seja tecnologicamente neutra para evitar influenciar o projeto da solução e, idealmente, devem ser uma expressão pura das necessidades do negócio (ROBERTSON; ROBERTSON, 2006).

Ao descrever a especificação de um requisito, pode-se utilizar diferentes representações que podem ser mais ou menos adequadas ao tipo de requisito e ao público-alvo do requisito. Por exemplo, a representação mais adequada ao usuário final pode não conter informação suficiente, ou não estar organizada da maneira mais adequada ao desenvolvedor.

Ao escolher a representação de um requisito, devem-se observar as seguintes características:

- Facilidade de verificação pelo usuário: indica a facilidade com que o usuário final pode entender a representação e, portanto, verificar se o requisito representado está correto (LAUESEN, 2002);

- Facilidade de verificação pelo desenvolvedor: indica a facilidade com que o desenvolvedor pode entender a representação, verificar se o requisito representado está correto e entender o que deverá ser implementado (LAUESEN, 2002);

- Precisão da representação: indica o quanto a representação permite avaliar se o requisito está consistente e correto de forma automatizada (KOTONYA; SOMMERVILLE, 1998);

- Possibilidade da definição do ambiente: indica se a representação permite a inclusão do contexto de funcionamento do sistema e dos elementos externos

com que o sistema interage (KOTONYA; SOMMERVILLE, 1998; LAUESEN, 2002);

- Flexibilidade: indica se a representação permite que os requisitos estejam temporariamente incompletos e que seja adaptável às mudanças que ocorrem nos requisitos durante a evolução do sistema (KOTONYA; SOMMERVILLE, 1998);

- Facilidade de integrar outras abordagens: as representações devem permitir o mapeamento de sua estrutura com elementos equivalentes em outras representações de requisitos, pois, muitas vezes, uma representação pode ser melhor que outras, dependendo do tipo do requisito. Dado que pode existir mais do que uma representação do mesmo requisito, é importante poder relacionar os diferentes modelos (KOTONYA; SOMMERVILLE, 1998);

- Facilidade de comunicação: as representações devem permitir que os requisitos e o seu entendimento sejam comunicados e discutidos (KOTONYA; SOMMERVILLE, 1998);

- Existência de suporte de ferramentas: é importante que existam ferramentas que auxiliem na representação dos requisitos e auxiliem na avaliação de inconsistências e incorreções (KOTONYA; SOMMERVILLE, 1998).

Várias representações de requisitos foram utilizadas e experimentadas e a importância de criar representações padronizadas de requisitos foi sendo percebida com a evolução da indústria de *software* e com o surgimento da Engenharia de Requisitos como uma disciplina. Ao padronizar a representação de um requisito, tornou-se possível avaliá-la quanto às caraterísticas descritas anteriormente e facilitou-se o trabalho do responsável pela especificação de um sistema ao definir uma forma de representar as informações coletadas, muitas vezes acompanhada com um método para a coleta destas informações (LAUESEN, 2002; ROBERTSON; ROBERTSON, 2006).

A seguir, estão listadas, as principais representações de requisitos que foram sendo utilizadas ao longo dos anos (LAUESEN, 2002):

- Diagrama de contexto: especifica o ambiente em que se encontra o *software* e os elementos externos (usuários e sistemas) com os quais ele interage. O diagrama de contexto representa todo o sistema como um único processo e constitui o diagrama de nível mais alto da análise estruturada (YOURDON, 1991). Este diagrama é útil para definição do escopo do sistema e permite especificar as interfaces do sistema;

- Lista de eventos: também utilizada pela análise estruturada, especifica uma lista dos estímulos que ocorrem no ambiente externo e que o *software* deve ser capaz de tratar (YOURDON, 1991). É de grande importância para o desenvolvedor, pois lista todos os eventos que devem ser tratados pelo *software* e serve para verificar se os tratamentos destes eventos foram implementados. Tem como principal desvantagem o fato de alguns eventos não terem significado do ponto de vista do usuário e envolver decisões de projeto, trazendo dificuldade em ser verificada pelo usuário final;

- Descrição de Funções: especifica em formato textual o que o *software* deve ser capaz de fazer. É bastante atrativa aos usuários finais que conseguem verificar se o *software* faz aquilo que ele deseja. Do ponto de vista do desenvolvedor, facilita o seu trabalho, pois muitas vezes as funções especificadas são facilmente traduzidas em funções do *software*. Quando a elaboração é mal conduzida, pode levar à criação de um número grande e desnecessário de funções, pois a existência de uma função não garante que um objetivo de negócio seja atingido. Quando fora de um contexto, a lista de funções não leva em consideração os objetivos de negócio do sistema, os quais podem acabar por ser negligenciados;

- Telas e protótipos: especificam a interface visual do *software* através de imagens, descrições ou animações e o comportamento dos elementos da interface visual. São bastante atrativas ao usuário final e aos

desenvolvedores, pois são fáceis de serem verificadas. O uso desta representação deve ser apoiado por uma outra representação que define o escopo do sistema. Sem isso, os usuários tendem, com a visualização das telas e protótipos, desejar novos requisitos que não faziam parte do escopo inicial, o que pode levar a um custo maior do que foi orçado;

- Descrição de tarefas: especifica as tarefas do usuário através de um texto contendo, tipicamente, o nome da tarefa, o objetivo da tarefa, os eventos que acionam esta tarefa, as pré-condições para a execução da tarefa, os passos envolvidos na execução da tarefa e possíveis variações nesta execução. A especificação de um sistema através de tarefas é fácil de ser verificada pelo usuário e pelo desenvolvedor, assim como permite especificar variações e complexidade da tarefa. Tem como principal desvantagem, a dificuldade apresentada pelos desenvolvedores em projetar o *software* a partir da descrição de uma tarefa, pois o mapeamento não é direto. Um outro problema é que nem toda função de um sistema é uma tarefa de usuário;

- Funções definidas a partir de descrições de tarefas: a representação por funções pode ser bastante atrativa aos desenvolvedores, já que muitas vezes pode ser mapeada facilmente em funções do *software*. Por outro lado, a descrição de tarefas é mais amigável ao usuário, pois descreve as tarefas que este quer realizar usando o *software*. A representação de funções definidas a partir de descrições de tarefas mescla estas duas representações buscando inicialmente descrever as tarefas e depois descrever as funções que participam na execução de cada tarefa. Ao contrário da simples descrição de funções, nesta representação é necessário que todas as funções necessárias para a execução de todas as tarefas do usuário sejam definidas;

- Tarefas e Suporte: especificam as tarefas como na descrição de tarefas e também um texto sobre o possível suporte para a implementação desejada para cada um dos passos da tarefa. Esta forma de representação é indicada quando se deseja especificar detalhes de implementação nos requisitos. A

representação é fácil de ser verificada pelo usuário e pelo desenvolvedor e permite especificar variações e complexidade da tarefa. Para especificar requisitos que não consistem de tarefas, esta representação enfrenta os mesmos problemas de todas as outras baseadas em tarefas e, além disso, é mais trabalhosa que a simples descrição de tarefas;

- Cenários: especificam um caso ilustrando uma ou mais tarefas do usuário e serve para ajudá-lo a compreender melhor os requisitos. Não se tratam propriamente de uma representação de requisitos, mas permite visualizar os requisitos já documentados;

- Lista de Tarefas Boas: apresenta a mesma característica da descrição de tarefas. No entanto, ao invés de especificar tarefas sem nenhum critério, inclui somente as tarefas que atendam as seguintes condições:

    - Fechada: A tarefa fechada é aquela que finaliza atingindo um objetivo significativo para o usuário;

    - Sessão Agrupada: é uma a sessão de trabalho que contém um conjunto de pequenas tarefas fechadas e que são desempenhadas juntas em uma mesma atividade;

    - Sem especificar implementação: as tarefas não devem detalhar como a implementação deve ser feita, pois isto é trabalho dos desenvolvedores.

  Ao seguir estas condições para selecionar as tarefas a serem descritas, evita-se especificar uma série de tarefas que pouco agregam ao entendimento do sistema;

- Tarefas de Alto-Nível: trata-se da descrição de tarefas de negócio, ao invés de tarefas de usuário. Raramente são usadas para especificação de requisitos;

- Casos de Uso: especifica uma série de atividades e suas variações que um sistema deve executar, a partir de uma solicitação de um usuário, denominado ator. É a forma de especificação de requisitos mais usada atualmente e faz parte da UML e do processo unificado (KOBRYN, 1999; KRUCHTEN et al., 2002);

- Tarefas com Dados: descrevem as tarefas, os dados visíveis ao usuário necessários para a execução dos passos da tarefa e as especificações de telas que seriam capazes de apresentar esta informação. Esta representação especifica apenas os dados que o usuário vê, pois a especificação da organização dos dados é deixada para o projeto e implementação do sistema. Da mesma forma, as especificações de telas descrevem apenas que informação e controles devem estar contidos na tela. Esta representação é fácil de se verificar pelo usuário e pelo desenvolvedor, e permite a rastreabilidade dos requisitos na implementação;

- Modelos de Fluxo de Dados: faz parte da análise estruturada e, através de um diagrama, descreve o fluxo de dados de um sistema através dos processos que transformam os seus dados de entrada em dados de saída (PRESSMAN, 2009; YOURDON, 1991). Foram amplamente usados antes da difusão das técnicas orientadas a objetos. Sua principal fraqueza é não servir para descrever tarefas do usuário com muitas variações;

- Padrões de produto usados como requisitos: ao invés de definir requisitos, declara-se que o *software* deve ser aderente a um certo padrão de produto reconhecido pela indústria. É um tipo de requisito muito importante, mas traz uma falsa sensação de segurança ao transferir a responsabilidade pelo requisito para a organização responsável pelo padrão. Deve ser usado em conjunto com outro tipo de especificação, para garantir que as necessidades do usuário sejam atendidas e, ao mesmo tempo, a solução esteja de acordo com padrões da indústria;

- Métodos formais: diferente de todas as formas descritas anteriormente, métodos formais especificam os requisitos de forma precisa não havendo margem para dúvidas. A precisão vem do uso de formalismos matemáticos para a especificação dos requisitos. Por outro lado, são difíceis de serem compreendidos e validados pelo usuário e por muitos dos desenvolvedores.

Das representações citadas nesta seção, os Casos de Uso se destacam por oferecer uma forma sistemática de elicitar requisitos através das tarefas realizadas por cada tipo de usuário (ator), por fornecerem suporte a variações nas tarefas dos usuários (através de fluxos alternativos) e serem fáceis de entender tanto por usuários quanto por desenvolvedores. Estas características levaram os Casos de Uso a fazerem parte da UML, um padrão consagrado pela indústria e pela OMG, e a se tornarem a forma mais utilizada e de aplicação mais ampla na indústria de *software* para representar requisitos (LAUESEN, 2002; BOOCH et al., 1999; KOBRYN et al., 1999; JACOBSON et al., 1999). Por estes motivos, os Casos de Uso serão detalhados na próxima seção.

2.2 CASOS DE USO

Os Casos de Uso são usados para especificar o que um novo *software* deve fazer ou o que um *software* já existente faz (AMBLER, 2000, BOOCH et al., 1999). Para fazer esta especificação, as funções do *software* são definidas a partir de interações entre o usuário ou outros sistemas e o *software*. Essas interações são chamadas Caso de Uso e devem representar uma transação do negócio (JACOBSON et al., 1999). Neste contexto, o sistema é visto como uma caixa-preta que provê Casos de Uso, que são formas como o sistema pode ser usado (ERIKSSON et al., 1998, ENGELS et al., 2000).

A visão geral do *software* é provida por um modelo de Casos de Uso, onde são apresentados todos os Casos de Uso do *software,* a relação entre eles e os atores que participam de cada Caso de Uso (BOOCH et al., 1999; JACOBSON et al., 1999). Um ator representa um papel desempenhado por um grupo de usuários que

tem um certo objetivo ao interagir com o *software.* Um Caso de Uso descreve possíveis comportamentos do *software* através de cenários. O cenário principal é o caso de sucesso, ou seja, aquele em que tudo transcorre conforme o esperado e deve corresponder à grande maioria das transações. Os outros cenários apresentam situações em que alguma barreira ou dificuldade pode provocar comportamentos específicos no *software* ou até mesmo impedir a realização da transação. O comportamento de um *software* em um cenário de um Caso de Uso é descrito por uma sequência de atividades que devem se realizar a partir de um evento ou estímulo realizado por um ator (MEDVIDOVIC et al., 2002; KRUCHTEN et al., 2002).

Ao organizar os requisitos de um software baseado em transações de negócio, que são claramente reconhecíveis e validáveis por desenvolvedores e usuários, os Casos de Uso facilitam a identificação, a elicitação e o acompanhamento da evolução dos requisitos durante todo o ciclo de vida do projeto e do *software* (RUI, 2007).

Ao se desenvolver um projeto orientado por Casos de Uso, como propõe o processo unificado[2], estes passam a orientar a especificação, modelagem e programação das transações de negócio, e com isso, servem como base para rastreabilidade entre todas as etapas do projeto e servir de apoio na manutenção e evolução do *software* (RUI, 2003; 2007).

O modelo de Casos de Uso é constituído por Diagramas de Caso de Uso e por uma especificação ou descrição do Caso de Uso. Os Diagramas de Caso de Uso são parte da UML e a figura 2 apresenta um diagrama de casos de uso seguindo o padrão UML (BOOCH et al., 1999).

---

[2]O Processo Unificado é um processo de desenvolvimento de *software* iterativo e incremental bastante utilizado pela indústria no desenvolvimento de sistemas orientados a objeto.

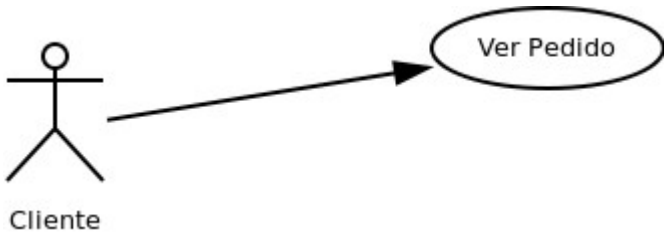

Figura 2 – Exemplo de Caso de Uso

Como pode ser visto, o *software* tem o ator “Cliente” que participa do caso de uso “Ver Pedido”. Além do relacionamento entre o ator e o caso de uso, os diagramas UML de casos de uso permitem relacionar casos de uso entre si. Os relacionamentos permitidos pela UML entre casos de uso são:

- <<include>>: permite a inclusão de um Caso de Uso em outro e deve ser usado quando existe no sistema um conjunto de passos comuns a vários casos de uso. Cria-se um Caso de Uso com estes passos comuns e, depois, usa-se o relacionamento <<include>> para referenciar os Casos de Uso incluídos, evitando reescrever os passos;

- <<extend>>: permite a extensão de um Caso de Uso e deve ser utilizado quando se deseja acrescentar um comportamento no Caso de Uso estendido em determinadas condições. A extensão é usada para definir a partir de um Caso de Uso base, um comportamento opcional que o *software* pode ter, mas que não é obrigatório para atingir o objetivo do Caso de Uso;

- herança: permite a criação de um Caso de Uso genérico que tem comportamento, requisitos, restrições e suposições comuns a vários Casos de Uso. Os Casos de Uso específicos herdam as características do Caso de Uso genérico e descrevem aquilo que eles tem de diferente.

Estes relacionamentos permitem uma reutilização de Casos de Uso e, com isso, aumentam a produtividade na especificação de sistemas.

A especificação ou descrição do Caso de Uso é, tipicamente, realizada na forma textual, mas pode ser feita também através de diagramas de sequência, atividade ou

interação (ERIKSSON et al., 1997). Não existe uma forma rígida para especificar um Caso de Uso, mas espera-se que esta especificação contenha os atores, o evento que dá início ao Caso de Uso, as pré-condições, a sequência de atividades que devem ocorrer e as pós-condições. Usualmente define-se um padrão para esta especificação dentro de um projeto onde decide-se o que vai ser feito através de uma descrição textual e o que vai ser feito através de diagramas (BOOCH et al., 1999; JACOBSON et al., 1999).

Este padrão garante uma certa uniformidade na descrição de um Caso de Uso, e pode deixá-lo mais preciso, dependendo do grau de formalismo desejado. Esta flexibilidade no grau de formalismo é também um dos fatores do sucesso dos Casos de Uso (RUI, 2003; 2007). A precisão que pode ser oferecida pelos Casos de Uso é suficiente para a maioria dos projetos, mas não alcança a precisão dada quando se aplicam formalismos matemáticos. Nos poucos projetos que a precisão da especificação dos requisitos é o atributo mais importante, a alternativa são os métodos formais e por este motivo, eles serão detalhados na próxima seção.

2.3 Métodos Formais

Os métodos formais surgem da ideia de que um *software* é construído baseado em um pequeno número de conceitos, capazes de especificar, desenvolver e verificar sistemas através de uma sintaxe matematicamente formal (KOTONYA; SOMMERVILLE, 1998; HARRY, 1997; CLARK, 1996). Os métodos formais surgiram como uma resposta à seguintes limitações encontradas na representação de requisitos em linguagem natural (HARRY, 1997):

- Interpretação: ao contrário da linguagem natural, os métodos formais não estão sujeitos a ambiguidade, pois descrevem o funcionamento do sistema usando uma formulação matematicamente precisa;

- Processamento da especificação: por possuir uma sintaxe bem definida, as representações em métodos formais podem ser processadas por programas

de computador que podem fazer simplificações, validações de maneira automática.

Um método é considerado formal, se tem uma base matemática sólida, tipicamente dada por uma linguagem de especificação formal. Esta base provê meios de definir, de maneira precisa, conceitos como consistência, integridade, especificação, implementação e correção. Com isso, o método formal provê meios de provar que uma especificação é implementável, que o sistema está correto e quais propriedades o sistema tem, sem a necessidade de implementá-lo (CLARK, 1996).

Uma linguagem de especificação formal é composta de três componentes principais <Syn, Sem, Sat> (KOTONYA; SOMMERVILLE, 1998; CLARK, 1996):

- Sintaxe (Syn): define o domínio sintático da linguagem e é derivada de uma teoria de notações e cálculo de predicados;
- Semântica (Sem): define o domínio semântico da linguagem que vai ser usado para descrever o sistema;
- Relações (Sat): define regras que indicam os objetos que satisfazem propriamente a especificação.

Uma linguagem formal pode ser categorizada como construtiva, em que um modelo matemático do sistema é criado, ou comportamental, em que o sistema é descrito a partir de suas propriedades, entradas e saídas e tratado como uma caixa-preta. Dessas categorias, foram criadas um grande número de estilos de especificações, linguagens e métodos formais, todos tendo como fundamento a matemática discreta (CLARK, 1996; HARRY, 1997). Estes estilos foram classificados por Harry (1997) da forma apresentada na figura 3.

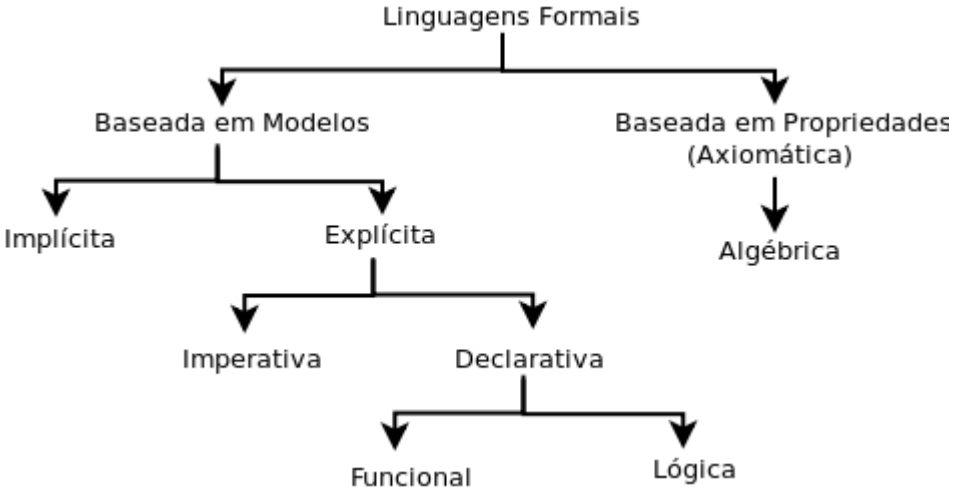

Figura 3 – Estilos das Linguagens Formais (HARRY, 1997)

Nas linguagens baseadas em modelos, o sistema é representado por um modelo matemático abstrato e suas funções são expressas utilizando coleções de tipos de dados. Eles podem fazer esta definição explicitamente ou implicitamente ou combinando estes dois estilos. Na definição implícita, descreve-se qual o resultado das funções, sem descrever como a função é executada. Na definição explícita, descreve-se como a função é executada de maneira imperativa ou declarativa. Ao usar o estilo imperativo, que é o mesmo utilizado na maioria das linguagens de programação, descreve-se a especificação através de ações que atuam sobre variáveis. Ao usar o estilo declarativo, descreve-se as manipulações de dados através de equações ou relações. Neste grupo, encontra-se o estilo funcional, em que se define o sistema a partir de funções e o estilo lógico, em que se usam predicados lógicos. Alguns exemplos de linguagens baseadas em modelos são Z, VDM, ML, Miranda e Haskell (CLARK, 1996; HARRY, 1997).

Nas linguagens baseadas em propriedades, o sistema é definido indiretamente a partir de uma série de axiomas que definem propriedades que o sistema deve satisfazer. Predicados de primeira ordem são usados para definir pré-condições e pós-condições das funções em termos de axiomas. Se estes axiomas são descritos utilizando equações, então o estilo é chamado de algébrico. Entre as linguagens com capacidade de suportar o estilo algébrico se encontram: Larch, OBJ3, CSP, CCS e LOTOS (CLARK, 1996; HARRY, 1997).

Os métodos formais tem as seguintes desvantagens em relação a outras formas de especificação (KOTONYA; SOMMERVILLE, 1998; CLARK, 1996):

- São focados principalmente em funções e dados, sendo difícil utilizá-los para aspectos comportamentais de um problema;
- O uso dos métodos formais garante que os requisitos estão corretos, mas não garante que eles sejam implementados corretamente;
- Métodos formais são difíceis de serem compreendidos pelo usuário final e por muitos desenvolvedores.

Devido ao alto custo das ferramentas necessárias para sua utilização, os métodos formais são geralmente usados apenas no desenvolvimento de sistemas de alta-confiabilidade, no qual há alta probabilidade das falhas conduzirem para a perda da vida ou sério prejuízo (CLARK, 1996).

2.4 UM META-MODELO DE CASOS DE USO

Um meta-modelo é um modelo que especifica a sintaxe de uma linguagem de modelagem (BORONAT; MESEGUER, 2008). Um meta-modelo de requisitos especifica uma sintaxe para representação de requisitos. Esta sintaxe pode ser definida com diferentes níveis de detalhes, permitindo que a representação do requisito tenha uma estrutura mais ou menos rígida dependendo do objetivo a ser atingido com o modelo. Um meta-modelo de Casos de Uso é um modelo que especifica uma estrutura capaz de descrever um Caso de Uso (RUI, 2007).

Rui (2007) desenvolveu um meta-modelo para casos de uso que formaliza a estrutura de Casos de Uso sem se preocupar com formalismos semânticos. Este meta-modelo foi criado com o objetivo de permitir a refatoração de Casos de Uso e ser utilizado num contexto de evolução de *software*. Mais especificamente, Rui (2007) utilizou o conceito de refatoração que é usado para designar mudanças feitas no código-fonte, com o objetivo de deixá-lo mais fácil de se ler e manter, e aplicou-o

nos Casos de Uso. Com isso, Rui (2007) definiu operações de refatoração de Casos de Uso, formalizando várias transformações que um Caso de Uso pode sofrer, sem que o seu significado seja alterado. Para definir estas transformações Rui (2007) usou o meta-modelo de Casos de Uso apresentado na figura 4.

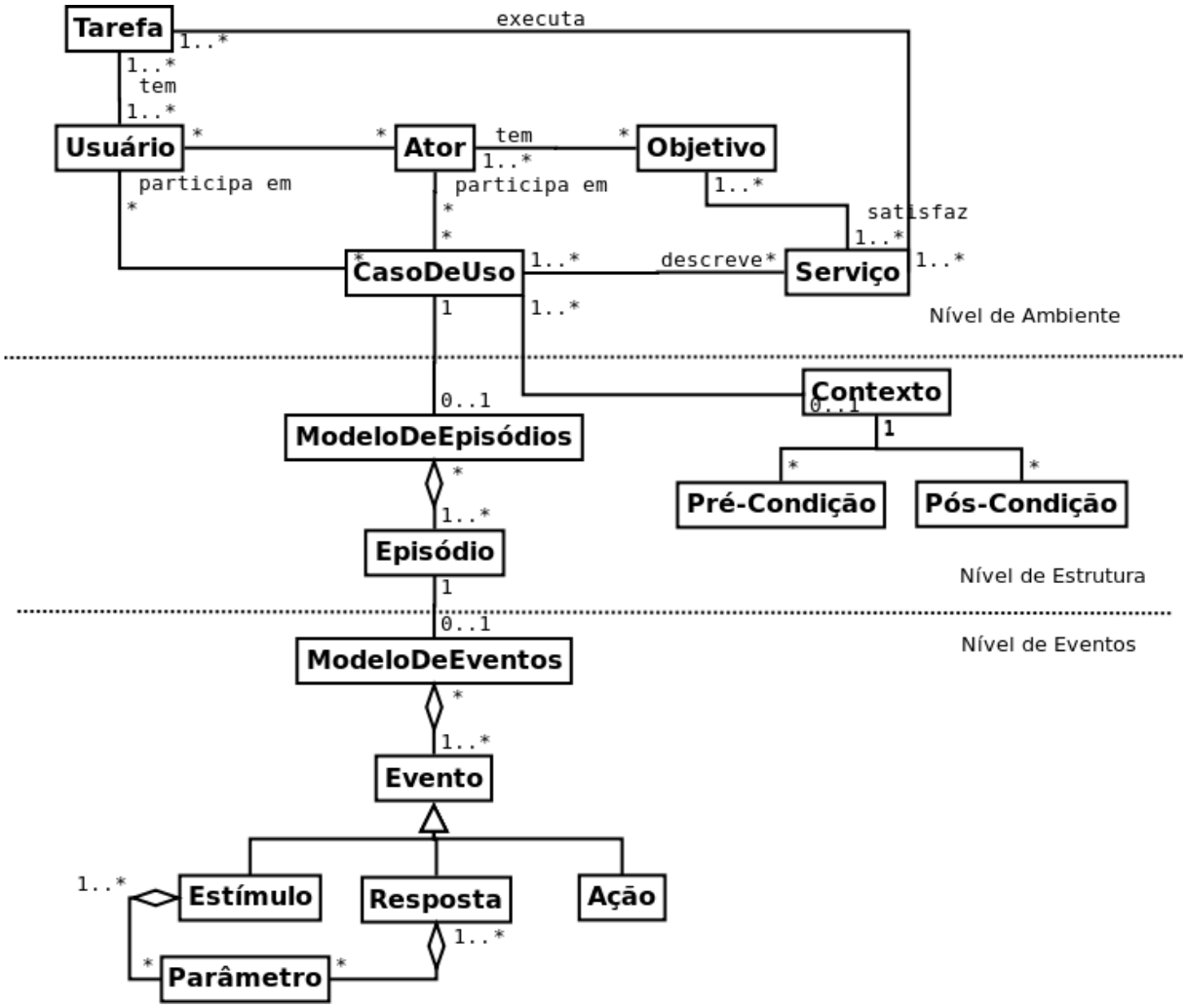

Figura 4 – Meta-Modelo Dos Casos de Uso (RUI, 2007)

O meta-modelo de Rui (2007) apresenta três diferentes níveis:

- Nível de Ambiente: corresponde ao relacionamento do Caso de Uso com as entidades externas ao sistema;
- Nível de Estrutura: corresponde à definição da estrutura interna do Caso de Uso;

- Nível de Eventos: corresponde à definição de eventos individuais.

O nível de ambiente é representado pelas seguintes entidades (RUI, 2007):

- Objetivo: modela uma meta, alvo ou propósito a ser atingido ou satisfeito com a execução de uma transação de negócio;

- Caso de Uso: modela uma situação de uso do sistema, em que um ou mais serviços do *software* são usados por um ou mais usuários com um ou mais objetivos em mente. Um Caso de Uso representa uma transação de negócio e permite descrever o comportamento do *software* sem a necessidade de especificar sua estrutura interna;

- Ator: modela o papel desempenhado por um usuário do *software* e representa uma categoria de usuários que apresentam comportamento similar ao utilizar o *software.* Um ator pode ter vários objetivos ao usar o *software* e, para atingir estes objetivos, participa em vários Casos de Uso;

- Usuário: é uma instância de um ator neste modelo. Usuários podem ser seres humanos ou outros sistemas e dispositivos que se comunicam com o *software*. Um usuário pode aparecer como várias instâncias de diferentes atores dependendo do contexto. O usuário tem uma ou mais tarefas que ele precisa realizar e ele as executa participando em um ou mais casos de uso;

- Serviço: é uma função oferecida ao usuário para que ele possa satisfazer um ou mais objetivos que tenha;

- Tarefa: é uma atividade que é realizada por um ou mais usuários através da execução de um serviço.

O nível de estrutura é representando pelas seguintes entidades (RUI, 2007):

- Episódio: consiste de uma das partes coerentes em que um Caso de Uso pode ser dividido. Os episódios revelam a estrutura interna de um Caso de Uso e descrevem o seu funcionamento. O mesmo episódio pode ocorrer em diferentes Casos de Uso;
- Modelo de Episódios: é uma abstração que representa o conjunto de todos os episódios de um Caso de Uso;
- Contexto: demarca o escopo de um Caso de Uso e define suas Pré-Condições e Pós-Condições. Um contexto pode estar relacionado a um ou mais casos de uso;
- Pré-Condição: é uma propriedade do ambiente ou do sistema que precisa ser atingida para que o Caso de Uso possa ser executado. Uma pré-condição está sempre relacionada a um contexto;
- Pós-Condição: é uma propriedade do ambiente ou do sistema que é atingida no final da execução de um Caso de Uso. Uma pós-condição está sempre relacionada a um contexto.

O nível de eventos é representado pela entidade Evento e seus três sub-tipos: Estímulo, Resposta e Ação (RUI, 2007):

- Evento: é uma ocorrência significativa que tem um lugar no tempo e espaço. A diferença entre um Evento e um Episódio é que o Evento é instantâneo (tem duração igual a zero) enquanto um Episódio tem uma duração;
- Modelo de Eventos: análogo ao modelo de episódios, é uma abstração criada por Rui (2007) para representar o conjunto de todos os eventos de um certo episódio;

- Estímulo: é um dos sub-tipos de Evento e consiste de uma mensagem do usuário ao sistema. Pode conter parâmetros;
- Resposta: é um dos sub-tipos de Evento e consiste de uma mensagem do sistema ao usuário. Pode conter parâmetros;
- Parâmetro: representa dados que podem ser transmitidos em um estímulo ou resposta;
- Ação: um dos sub-tipos de Evento e consiste de uma mensagem interna ao sistema, onde não existe comunicação entre o sistema e o usuário.

O modelo de Rui (2007) define em mais detalhes a estrutura dos Casos de Uso e elimina vários graus de liberdade que a especificação de Casos de Uso permite. Os elementos que fazem parte desta especificação e as relações entre eles ficam limitadas àqueles estabelecidos no meta-modelo. Por ter sido concebido para o seu trabalho na área de refatoração de Casos de Uso, o modelo leva em conta a reutilização de episódios ao permitir que um mesmo episódio possa estar relacionado a mais de um modelo de episódios (BEZERRA; MELNIKOFF, 2010).

Rui (2007), ao definir um meta-modelo para os Casos de Uso, contribui para maior formalização dos Casos de Uso, tornando sua estrutura mais homogênea e precisa.

2.5 Considerações Finais

O capítulo mostrou que existem diversas representações de requisitos funcionais de *software,* apresentou os Casos de Uso e os métodos formais e um meta-modelo de Casos de Uso.

A escolha da representação de requisitos é uma questão importante, pois dependendo do tipo de representação escolhida, algumas caraterísticas do *software* ficam mais evidentes e claramente especificadas do que outras. Desta forma, para representar os requisitos de um *software* adequadamente, devem-se combinar as

diferentes representações que sejam mais adequadas às características do *software* que se deseja especificar.

A especificação de requisitos através de Casos de Uso oferece um tratamento sistemático das tarefas do usuário, é fácil de usar e é um padrão importante dentro da indústria de *software.* Os Casos de Uso organizam os requisitos de um *software* baseado em transações de negócio, que são reconhecíveis e validáveis por desenvolvedores e usuários, com isso facilitam a identificação, a elicitação e o acompanhamento da evolução dos requisitos durante todo o ciclo de vida do projeto e do *software* (RUI, 2007). Por todas estas características os Casos de Uso se tornaram a forma mais utilizada pelas empresas para representar requisitos e fazem parte da maioria dos processos de desenvolvimento orientado a objetos. Nos métodos ágeis, os requisitos são representados por histórias e é possível transformar uma história em um caso de uso. Desta forma, a utilização de casos de uso é também viável para estes métodos.

Os métodos formais, por se basearem em formalismos matemáticos, conseguem garantir a integridade e consistência dos requisitos, com isso estão menos sujeitos as subjetividades e imprecisões do que as outras representações de requisitos. Porém, devido ao seu alto custo, tem seu uso limitado praticamente a sistemas críticos.

O meta-modelo de Casos de Uso apresentado neste capítulo foi o de Rui (2007). Rui (2007) definiu os elementos de um Caso de Uso de forma mais detalhada, tornando a sintaxe do meta-modelo mais rigorosa do que a descrição da estrutura proposta na UML. Desta forma, os elementos de um Caso de Uso e os relacionamentos entre eles são definidos de forma explícita. A utilização do meta-modelo de Rui (2007) na especificação de requisitos traz como vantagem a criação de uma especificação com um formato mais uniforme e uma estrutura melhor definida, o que facilita o seu processamento por programas de computador que podem validá-las, extrair informações ou, no caso específico da tese de Rui (2007) definir operações de

refatoração que permitem organizar melhor os requisitos sem alterar a sua semântica.

3 GESTÃO DE REQUISITOS DE SOFTWARE

A gestão de requisitos é um processo sistemático para identificar, documentar, organizar e rastrear os requisitos variáveis de um sistema (JACOBSON et al., 1999).

Este capítulo inicialmente apresenta a gestão de requisitos, os fatores que levam o *software* a mudar, as principais formas para a identificação de requisitos, os meios utilizados para armazenamento dos requisitos e o processo de gestão das mudanças no *software.* A seguir, apresenta a rastreabilidade de requisitos, técnica fundamental para a gestão de requisitos, pois permite a análise de impacto de uma possível mudança em um requisito. Ainda neste capítulo é apresentada a reengenharia de requisitos, que permite recuperar, melhorar a qualidade ou reorganizar os documentos de requisitos. Também neste capítulo é apresentado um modelo iterativo de gestão de requisitos integrado à engenharia reversa de requisitos, onde a cada iteração os documentos de requisitos são produzidos a partir da engenharia reversa do código-fonte. Finalmente, são tecidas as considerações finais sumarizando os conceitos apresentados no capítulo.

3.1 GESTÃO DE REQUISITOS

As mudanças nos requisitos ocorrem a todo momento, são inevitáveis e não estão diretamente relacionadas com algum problema na engenharia de requisitos (KOTONYA; SOMMERVILLE, 1998; LEFFINGWELL; WIDRIG, 2006). Elas se devem principalmente aos seguintes fatores (KOTONYA; SOMMERVILLE, 1998; LEFFINGWELL; WIDRIG, 2006; POLO, 2002):

- Mudança no problema: o problema que o *software* resolve se alterou devido a mudanças na economia, regulamentação governamental, mercado, preferência do consumidor ou qualquer outro fator externo à organização;

- Erros, conflitos e inconsistências nos requisitos: ocorrem devido a problemas na captura dos requisitos; podem demorar a ser percebidos, sendo descobertos, muitas vezes, só depois que o *software* estiver em uso;

- Maior conhecimento do usuário sobre o sistema: com a captura dos requisitos ou o uso do sistema, o usuário passa a ter um entendimento maior sobre suas necessidades e percebe que os requisitos demandados não eram exatamente os que ele necessitava;

- Problemas técnicos, de cronograma ou custo: após capturados os requisitos e durante a implementação do *software*, problemas técnicos podem aparecer, inviabilizando prazos e custos esperados;

- Mudança de prioridades: durante o desenvolvimento do sistema, as prioridades do usuário podem mudar, fazendo com que certos requisitos mais importantes percam importância e outros novos tenham mais urgência;

- Mudança de ambiente: sistema operacional, equipamento, rede ou qualquer outro elemento do ambiente, em que o sistema está instalado, podem ser alterados e demandam mudanças para manter a compatibilidade;

- Mudanças organizacionais: a organização que utiliza o sistema pode sofrer mudanças na sua estrutura e nos seus processos, gerando impactos na forma de utilizar o sistema e, consequentemente, novos requisitos.

Dentro de um cenário de mudanças, onde cada necessidade de mudança gera novos requisitos, cada requisito deve possuir algum tipo de identificador único (KOTONYA; SOMMERVILLE, 1998). As três principais formas de identificação de requisitos são (KOTONYA; SOMMERVILLE, 1998):

- Numeração dinâmica: através do uso de processadores de texto e referências cruzadas que são atualizadas automaticamente quando o texto é reorganizado;

- Identificador no banco de dados: quando uma nova necessidade de mudança é identificada, ela imediatamente é registrada em um banco de dados de requisitos e recebe um identificador fornecido por este banco de dados;

- Identificação simbólica: baseada em alguma classificação de acordo com o aspecto do sistema que o requisito trata. Tem como desvantagem o fato de que, às vezes, alguns requisitos são difíceis de ser classificados, sejam por envolverem múltiplos aspectos ou algum aspecto ainda não classificado.

Um outro aspecto importante na gestão dos requisitos é o armazenamento dos requisitos e, geralmente, os seguintes meios são utilizados (KOTONYA; SOMMERVILLE, 1998):

- Documento de requisitos: o uso de um único documento tem como vantagem permitir que todos os requisitos fiquem em um único lugar, é fácil de produzir novas versões e de se acessar. Por este motivo, este é o meio preferido para sistemas de pequeno e médio porte. Porém quando se tem sistemas de grande porte o uso de um documento traz dificuldades, pois só pode ser editado por uma única pessoa por vez e, se fica muito extenso, suas facilidades de busca estão limitadas àquilo que o processador faz e a rastreabilidade precisa ser mantida em um arquivo externo;

- Banco de dados: para um *software* de maior porte e mais complexo, a alternativa é manter os requisitos em banco de dados. Com esta abordagem, é possível a navegação entre requisitos dependentes, o armazenamento de informação de rastreabilidade e o uso de facilidades de busca que são difíceis de se implementar usando um arquivo texto, mesmo com os recursos dos processadores de texto modernos.

Mudanças nos requisitos têm se tornado uma das principais fontes de erro em um *software* com impacto direto na sua qualidade, por isso a aplicação de técnicas para gerenciar estas mudanças são primordiais para a longevidade do *software.* (BAOJIAN, 2011). A gestão das mudanças envolve as seguintes ações (BERSOFF, 2009; CATER-STEEL, 2008; KOTONYA; SOMMERVILLE, 1998):

- Identificação do evento: envolve determinar que fator causou a mudança, ou seja, se foi uma mudança no problema, erro, conflito, inconsistência nos requisitos, maior conhecimento do usuário sobre o sistema, mudança de prioridades, mudança de ambiente, mudança organizacional, problema técnico, de cronograma ou custo;

- Análise da mudança: envolve a especificação da mudança, usando em geral, técnicas para elicitação de requisitos; a avaliação do impacto da mudança e a avaliação do prazo e custo para a sua execução. Com a análise concluída deve estar claro o que deve ser feito e qual seu impacto, prazo e custo;

- Aprovação da mudança: envolve a priorização da execução da mudança, que pode ser feita de maneira formal ou informal, por uma pessoa ou por um comitê dependendo das práticas da organização. Nesta priorização as mudanças são aprovadas ou rejeitadas e as mudanças aprovadas são ordenadas de acordo com critérios como criticidade, importância, impacto, prazo e custo;

- Execução da mudança: envolve a execução da mudança de acordo com um cronograma definido em função da sua prioridade. A mudanças rejeitadas podem ser arquivadas ou descartadas.

A gestão de mudanças está prevista tanto em processos tradicionais quanto em métodos ágeis, porém a forma como estas ações são executadas são distintas principalmente no que tange à documentação das mudanças (LEFFINGWELL; WIDRIG, 2006). Processos tradicionais defendem a documentação detalhada das

mudanças, enquanto métodos ágeis apoiam a análise no entendimento da mudança através do diálogo face a face entre usuários e desenvolvedores (CAO; RAMESH, 2008).

Além de seguir as ações descritas anteriormente, algumas medidas são necessárias para a gestão das mudanças. Entre essas medidas se incluem:

- Uso do controle de versão para requisitos: embora pareça óbvio, a maioria das organizações, apesar de adotarem controle de versão para código-fonte, não adotam controle de versão para requisitos (LEFFINGWELL; WIDRIG, 2006; LAM; SHANKARARAMAN, 2002);

- Estabelecimento de um único canal para controlar as mudanças: deve ser criado um único canal para aprovação de mudanças de tal forma que o impacto das mudanças e potenciais conflitos entre elas possam ser analisados e resolvidos em função das prioridades definidas (LEFFINGWELL; WIDRIG, 2006; CATER-STEEL, 2008);

- Registro de todas as solicitações de mudança (RFC – *Request for Change*): para resolver conflitos entre mudanças e tomar decisões em relação a aprovação e priorização de uma mudança, é importante saber o tipo de mudança que foi solicitada e a razão desta mudança (LEFFINGWELL; WIDRIG, 2006; CATER-STEEL, 2008).

No processo de gerenciamento das mudanças, é necessário analisar além da documentação, pois algumas mudanças no *software* podem ter impactos que só podem ser previstos com a análise da arquitetura ou, em muitos casos, do código-fonte (RUI, 2007; CIRACI et al., 2007). Neste contexto, o conhecimento e o entendimento dos requisitos implementados atualmente pelo *software* são fundamentais (BENNETT; RAJLICH, 2000). Além disso, para saber a quais requisitos está associado um trecho do código-fonte, deve-se utilizar técnicas de rastreabilidade.

3.2 Rastreabilidade de Requisitos

A rastreabilidade é definida como o grau com que se pode estabelecer uma relação entre dois artefatos do desenvolvimento de *software* (IEEE 610-1991, 1991; RAMESH et al., 2005). Para haver rastreabilidade deve haver um relação entre dois artefatos que são criados dentro de um processo de desenvolvimento de *software* e um artefato deve ser necessário ao processo de desenvolvimento e existir como consequência do outro artefato (LEFFINGWELL; WIDRIG, 2006).

A rastreabilidade de requisitos é, então, o grau com que se pode estabelecer uma relação entre um requisito do *software* e algum dos artefatos produzidos no desenvolvimento do *software* que implementa este requisito.

A relação de rastreabilidade entre dois artefatos pode ser feita de diversas maneiras e não existe nenhum padrão para a sua representação ou construção (LEFFINGWELL; WIDRIG, 2006). Mais ainda, apesar de ser aplicada há vários anos, existe pouco consenso sobre qual informação deve ser capturada e usada em uma relação de rastreabilidade (RAMESH et al., 2005; EGYED et al., 2009).

Entre as formas usadas para representar as relações de rastreabilidade, se encontra a matriz de rastreabilidade que indica o relacionamento entre dois tipos de artefatos e pode ser vista na tabela 1 (LEFFINGWELL; WIDRIG, 2006).

Tabela 1 - Matriz de rastreabilidade

	Classe A	Classe B	Classe C
Caso de Uso 1	X	X	
Caso de Uso 2		X	X
Caso de Uso 3	X		

Como pode ser visto, a matriz de rastreabilidade, neste caso, relaciona casos de uso com classes do *software.* No exemplo, o Caso de Uso 1 é implementado pelas classes A e B. O exemplo é bastante simples e supõe que o *software* tenha apenas três casos de uso e três classes. Em um *software* de grande porte, as matrizes de rastreabilidade ficam grandes, o que gera um grande esforço para que estas sejam mantidas atualizadas; ao mesmo tempo, elas são importantes para a análise de impacto de uma modificação (KOTONYA; SOMMERVILLE, 1998; LEFFINGWELL; WIDRIG, 2006).

Para lidar com grandes matrizes de rastreabilidade, existem ferramentas que permitem automatizar parte do trabalho de atualização e oferecem recursos que permitem ver o histórico de um relacionamento de rastreabilidade, ou seja que alterações foram feitas no requisito e o seu impacto no artefato (LEFFINGWELL; WIDRIG, 2006).

Na prática, para a maioria das equipes de desenvolvimento, a implementação de um sistema com aplicação de técnicas de rastreabilidade é extremamente complicado, seja porque as ferramentas são custosas e difíceis de usar, seja porque ao não usar ferramentas existe um grande trabalho manual envolvido para manutenção dos relacionamentos de rastreabilidade (SCHWARZ et al., 2008; RAMESH, 1998; GOTEL; FINKELSTEIN, 2002).

Devido ao grande custo para manter a informação de rastreabilidade atualizada, recomenda-se o uso de políticas de rastreabilidade para definir o nível de rastreabilidade que deve ser mantido de acordo com a criticidade do *software* (KOTONYA; SOMMERVILLE, 1998).

O custo, muitas vezes, acaba inviabilizando o uso das técnicas de rastreabilidade e a própria atualização dos requisitos. Quando as técnicas de rastreabilidade não foram aplicadas, a documentação de requisitos não existe ou está desatualizada e,

se for necessário recuperar essa informação, deve-se aplicar as técnicas de reengenharia de requisitos.

3.3 Reengenharia de Requisitos

A reengenharia de *software* consiste da revisão e da reestruturação de um *software* ou de seus artefatos e serve para minimizar os efeitos da degradação crescente do *software* durante sua evolução. A reengenharia de *software* tem como objetivo o aumento da qualidade do *software* através da recuperação de documentos, reestruturação da sua arquitetura, reescrita do código, etc (SNEED, 2008).

Neste contexto, a reengenharia de requisitos é a aplicação das técnicas de reengenharia de *software* com o objetivo de recuperar, melhorar a qualidade ou reorganizar os documentos de requisitos de um sistema (SNEED, 2008; FAHMI et al., 2007).

Dentre as técnicas de reengenharia de *software* aplicáveis à reengenharia de requisitos as seguintes se destacam (SNEED, 2008):

- Refatoração: é uma técnica que consiste da reestruturação do código-fonte e é empregada mais usualmente em códigos-fonte com linguagens orientadas a objeto. Através da refatoração, busca-se reestruturar classes, métodos, heranças e associações, deixando o código mais fácil de compreender e, consequentemente, de manter. A refatoração é considerada, pelos métodos ágeis, inexorável, pois não se espera que o projeto original do *software* contenha as classes e abstrações corretas;

- Engenharia Reversa: a engenharia reversa consiste da recuperação de artefatos de mais alto nível de abstração utilizando artefatos de baixo nível de abstração. Ela tem como objetivo extrair vários tipos de informação de um *software* existente, esteja este na forma binária ou código-fonte, e utilizar esta informação na compreensão do sistema (YU et al., 2005; CANFORA et al.,

2008). A engenharia reversa é considerada um pré-requisito de qualquer projeto de reengenharia, uma vez que os artefatos de mais alto nível são importantes para compreensão do *software* e, portanto, necessários para a sua alteração. A engenharia reversa também é recomendada como uma tecnologia chave para o suporte da evolução de *software* quando a única fonte confiável de informações é o código-fonte (CANFORA; PENTA, 2007). A compreensão do que o sistema faz e como está estruturado é fundamental para o sucesso na evolução de um *software* (FAHMI et al., 2007).

A aplicação das técnicas de refatoração na reengenharia de requisitos é o resultado do trabalho de Rui (2003, 2007). Rui (2003, 2007) propôs uma técnica para refatoração de casos de uso, permitindo a reestruturação dos casos de uso sem que a sua semântica seja alterada.

A engenharia reversa de requisitos é um processo que busca, através da análise de artefatos de nível mais baixo de um *software* (código-fonte e/ou binários em geral), criar o modelo de requisitos do mesmo (CANFORA et al., 2008). Ela é a principal técnica utilizada na reengenharia de requisitos e é fundamental para a compreensão e/ou manutenção de um *software,* quando a documentação não é confiável (IEEE 1219-1998, 1998; CANFORA; PENTA, 2007; CANFORA et al., 2008). Segundo estimativas, mais de 50% do esforço de manutenção é despendido na compreensão do programa (CANFORA et al., 2008; FAHMI et al., 2007).

Um modelo conceitual da engenharia reversa está representado por um diagrama de classes na figura 5 (CANFORA et al., 2008).

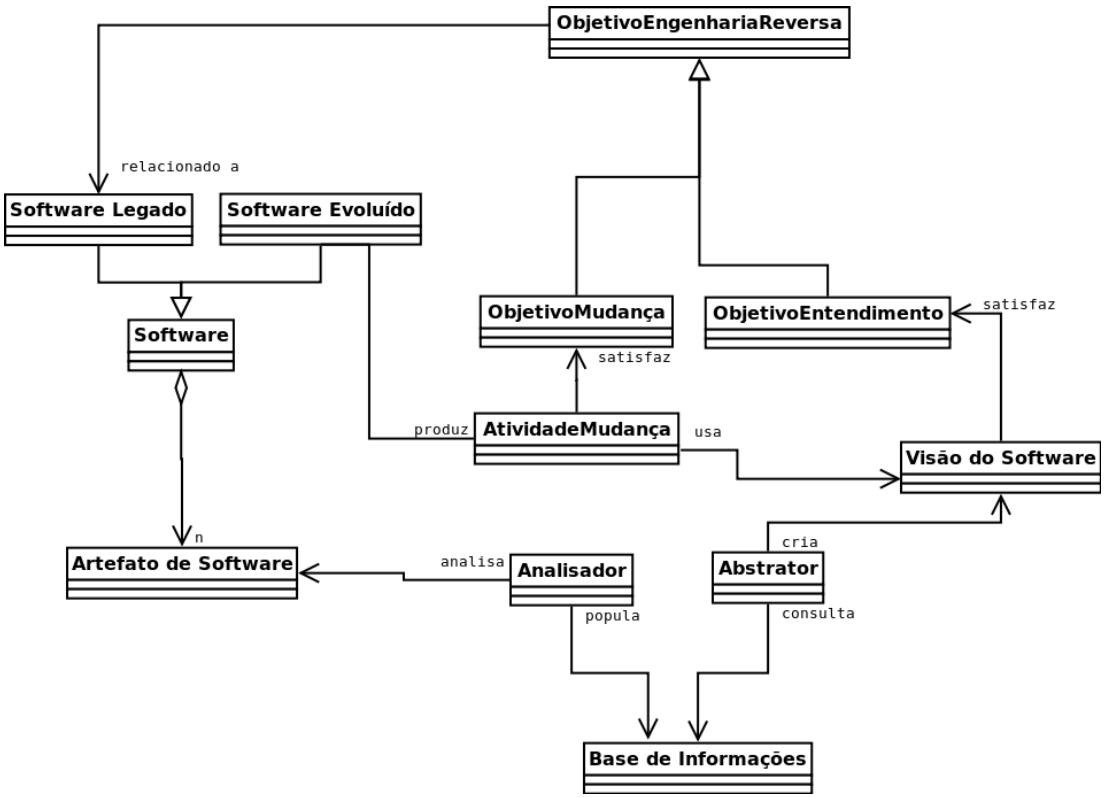

Figura 5 – Modelo Conceitual da Engenharia Reversa (CANFORA et al., 2008)

Segundo este modelo, a engenharia reversa é realizada para atingir o objetivo de um engenheiro de *software* (classe ObjetivoEngenhariaReversa) para resolver algum problema relacionado a algum *software* legado (classe *Software* Legado) e transformá-lo em um *software* evoluído (classe *Software* Evoluído). Este objetivo pode ser o maior entendimento do *software* (classe ObjetivoEntendimento) ou uma mudança que torne possível realizar sua evolução (classe ObjetivoMudança). Um *Software* (classe *Software)* possui vários artefatos (classe Artefato de *Software*), dentre estes artefatos, se encontram código-fonte e os binários. A engenharia reversa pode extrair informações do código-fonte ou dos binários, quando o código-fonte não está disponível. Esta extração é realizada pelo Analisador (classe Analisador) e alimenta uma base de informações sobre o sistema legado (classe Base de Informações). O Abstrator (classe Abstrator) consultando esta base de informações consegue gerar visões de alto nível do *software* (classe Visão do

Software). Entre estas visões se encontram: arquiteturas, diagramas UML, requisitos e ligações de rastreabilidade entre código-fonte e documentação de alto nível. As visões são importantes para satisfazer o objetivo de entendimento do *software* e são usadas na Atividade de Mudança do *software* (classe AtividadeMudança) que é realizada para satisfazer o objetivo de mudança, produzindo um *software* evoluído (CANFORA et al., 2008).

Dentro deste modelo, destacam-se duas etapas na engenharia reversa de requisitos (CANFORA et al., 2008):

- Extração de informação (realizada pela classe Analisador): a extração da informação analisa diversos artefatos do sistema e captura dados;
- Abstração (realizada pela classe Abstrator): usa os dados extraídos para criar uma visão ou documentação voltada para o engenheiro de *software*.

As técnicas atuais de engenharia reversa permitem a construção de analisadores e abstratores bastante eficientes, permitindo em algumas ferramentas, a visualização de um diagrama de classes que é resultado da engenharia reversa de um código-fonte instantaneamente (CANFORA et al., 2008).

Canfora e Penta (2007) apresentam o estado da arte na engenharia reversa, deixando claro que a engenharia reversa é fundamental para o entendimento de código já desenvolvido, dando um grande apoio à manutenção de *software.* Dentro desta pesquisa, pode-se perceber que a geração automática de modelos de requisitos, ainda é um assunto sendo pesquisado e sem respostas.

Na busca destas respostas, Yu et al. (2005), ao pesquisar o estado da arte, apresentou os seguintes tópicos de interesse para a engenharia reversa de requisitos:

- Recuperação de requisitos não funcionais de um sistema;

- Recuperação dos objetivos e propósitos de um sistema;
- Recuperação de cenários e diagramas de atividades de um sistema;
- Recuperação de casos de uso de um sistema;
- Confronto de requisitos e implementação, permitindo detectar diferenças entre o especificado e o implementado;
- Recuperação da rastreabilidade dos requisitos.

Destes tópicos, a recuperação de casos de uso ainda representa um grande desafio e, na falta de meios automatizados de recuperá-los, Liu (2005) apresenta um trabalho para engenharia reversa de requisitos baseado em técnicas semióticas. O seu trabalho propõe um método baseado em investigação, inspeção do sistema e entrevista com os usuários para a recuperação dos requisitos. Ele contribui, ao propor uma abordagem metodológica para a engenharia reversa de requisitos, mas esta abordagem, por ser totalmente manual, tem custo muito elevado e só deve ser usada em sistemas que já não podem ser evoluídos e precisam ser substituídos.

A pesquisa de trabalhos subsequentes mostra que o estado da arte pouco foi alterado após Yu et al. (2005), com destaques para o trabalho de Yang et al. (2006) com recuperação de requisitos iniciais, Fahmi et al. (2007) com um processo de evolução de *software* baseado na engenharia reversa de requisitos, Canfora et al. (2008) com um modelo conceitual da engenharia reversa e Marcus e Maletic (2003), Jermakovics et al. (2008), De Lucia et al. (2008), Lormans e Deursen (2009) e Oliveto et al. (2010) com técnicas e ferramentas para recuperação da rastreabilidade dos requisitos usando técnicas de LSI (*Latent Semantic Indexing*).

A pesquisa do estado do arte da engenharia reversa de requisitos revelou uma crescente importância deste tema na evolução de sistemas e a escassez de

soluções que permitam o uso extensivo da mesma, principalmente em função da falta de soluções para automatização.

3.4 Gestão de Requisitos Integrada à Engenharia Reversa

A engenharia reversa de requisitos, embora seja também usada na manutenção de *software*, é um processo utilizado predominantemente para recuperar requisitos de um *software* legado, quando se quer construir um novo para substituí-lo (FAHMI et al., 2007). Por ser um processo manual e custoso, a aplicação da engenharia reversa de requisitos a partir do código-fonte em outras situações não foi muito estudada (FAHMI et al., 2007).

A despeito dos custos de uma eventual aplicação, Fahmi et al. (2007) estudou a integração da engenharia reversa de requisitos dentro de um ciclo de um processo iterativo de gestão de mudanças e desenvolveu o processo apresentado na figura 6.

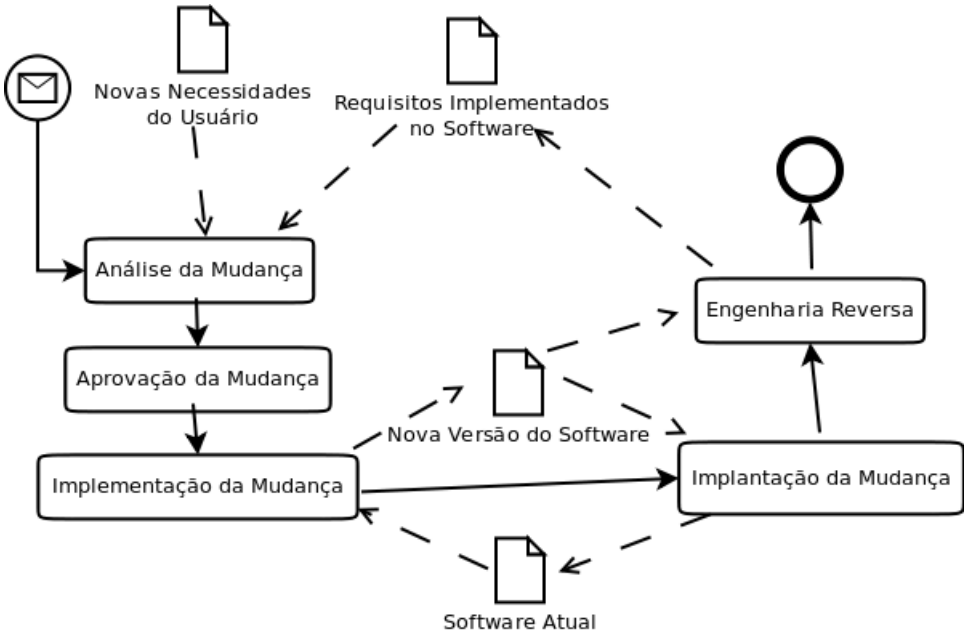

Figura 6 – Modelo de Gestão de Mudanças (FAHMI et al., 2007)

Seguindo este modelo, a engenharia reversa de requisitos passa a fazer parte da gestão das mudanças. No modelo de Fahmi et al. (2007), a análise da mudança

deve usar como fontes de informação os novos requisitos (Novas Necessidades do Usuário) e os requisitos que já estão implementados no *software* (Requisitos Implementados no *Software).* Apoiado nestas informações, pode-se analisar a mudança solicitada e o impacto no *software* existente. Com base nos resultados é feita a aprovação ou não da mudança (Aprovação da Mudança) e, no caso da aprovação é feita a Implementação da Mudança. A Implementação da Mudança transforma o *Software* Atual em uma Nova Versão do *Software* e, com a Implantação da Mudança, é feita a instalação e disponibilização da Nova Versão do *Software* para os seus usuários, fazendo com que esta nova versão se torne o *Software* Atual. Terminando a iteração, a Engenharia Reversa gera um documento com os Requisitos Implementados no *software* atualizado, o qual será utilizado na próxima mudança.

Conforme pode ser visto neste modelo, Fahmi et al. (2007) propõe que se use a documentação de requisitos recuperada a partir do código-fonte, pois na maioria dos sistemas a documentação de requisitos não está corretamente atualizada. Na análise dos requisitos, os documentos com os requisitos implementados devem ser usados para verificar a integridade e consistência do *software* após a implementação dos novos requisitos. Com isso, garante-se um melhor entendimento do que é redundante, do que deve ser mantido e do que pode ser reutilizado.

Fahmi et al. (2007) propõe também a comparação dos modelos de requisitos implementados entre duas versões para o estudo da evolução do *software*, onde pode-se verificar o que é comum, o que foi adicionado e o que foi excluído na nova versão do *software* (FAHMI et al., 2007).

De acordo com os resultados da pesquisa de Fahmi et al. (2007), sua proposta tem as seguintes vantagens:

- Permite aproveitar um grande número de requisitos de um *software* legado em um *software* novo;

- Permite fornecer informações do que está implementado para análise de requisitos em um desenvolvimento iterativo;
- Permite, após a engenharia reversa dos requisitos, verificar se todos os requisitos foram implementados;
- Permite, através de comparação entre versões dos requisitos, identificar mudanças nos requisitos.

Fahmi et al. (2007) reconhece que para que seu modelo possa ser aplicado é necessário o desenvolvimento de tecnologias que possibilitem que a engenharia reversa de requisitos seja realizada com custos e prazos menores do que é possível com as tecnologias atuais. Para que o seu modelo seja viável, Fahmi et al. (2007) defende fortemente a importância de formas automatizadas de fazer a engenharia reversa de requisitos a partir do código-fonte. Ele defende que a análise de requisitos; além de cuidar da organização, interpretação, entendimento e classificação dos requisitos de *software*; deve também lidar com a completude, consistência e viabilidade dos requisitos. Em sua proposta, na análise de requisitos em um sistema em evolução, a engenharia reversa de requisitos deve ser contínua e automatizada para que estes possam ser confrontados aos novos requisitos do *software*. Independente da existência de documentos de requisitos atualizados, a engenharia reversa de requisitos permitiria verificar em alto nível, o que realmente foi implementado.

O modelo de Fahmi et al. (2007) deixa esta questão em aberto, pois ele não fornece nenhuma solução para a engenharia reversa de requisitos ou qualquer técnica ou método para recuperar os requisitos implementados em um código-fonte. Ele reconhece que, no seu levantamento do estado da arte, muito pouco foi feito nesta área e reforça a importância deste trabalho, como um passo futuro para o sucesso na implantação de seu modelo.

3.5 Considerações Finais

A gestão de requisitos é fundamental para a longevidade de um *software.* Através dela, as mudanças que ocorrem nos requisitos de um *software* são controladas e realizadas somente após a análise de impacto, custo e prazo. Para que a gestão destas mudanças tenha sucesso, é fundamental que os documentos de requisitos estejam atualizados e que seja possível identificar os impactos da mudança de um requisito.

Dentro deste contexto, a rastreabilidade dos requisitos é uma técnica fundamental, pois permite associar um requisito a outros artefatos aos quais este requisito está relacionado, facilitando a análise do impacto de uma possível alteração em um requisito. No entanto, a rastreabilidade exige um grande trabalho manual ou ferramentas sofisticadas, tendo um grande custo para implantação e manutenção.

Em função deste cenário, o destino da maioria dos sistemas é não ter documentos de requisitos atualizados e pouca rastreabilidade. Nesse momento, passa a ser necessária a aplicação de técnicas para engenharia reversa de requisitos para poder recuperar estes documentos e ter condições de analisar com maior clareza as consequências das mudanças em um *software* ou para poder construir um novo *software* quando o custo de manutenção do legado passou a não compensar mudanças no mesmo.

A pesquisa da literatura mostrou que a engenharia reversa de requisitos é uma questão importante e citada pelos trabalhos Yu et al. (2005), Fahmi et al. (2007), Liu (2005), Canfora e Penta (2007) e Canfora et al. (2008), onde Canfora et al. (2008) é o trabalho relevante mais recente sobre o assunto. Ao mesmo tempo, poucas soluções concretas existem, principalmente devido ao nível de abstração muito mais elevado dos requisitos em relação ao código-fonte e a uma série de sutilezas que impedem que estes sejam determinados a partir do processamento automático de códigos-fontes.

Fahmi et al. (2007), no seu trabalho, mostra um modelo em que engenharia reversa de requisitos permitiria a gestão de requisitos de forma mais eficiente. A proposta de Fahmi et al. (2007), por não apresentar uma forma para fazer a engenharia reversa de requisitos, deixa esta questão em aberto.

4 MECANISMO DE ANOTAÇÕES

O mecanismo de anotações é um recurso disponível em diversas linguagens de programação modernas que permite a associação de meta-dados a elementos de um programa, sem afetar o seu funcionamento (BLOCH, 2008; SIERRA, 2009). Os meta-dados são dados que descrevem outros dados (BARGMEYER et al., 2000); neste contexto, a anotação é um meta-dado, que contém dados que descrevem o elemento do programa ao qual está associada.

Entre as linguagens de programação que permitem anotações se encontram JAVA, .NET[1], Ruby (BLOCH, 2008; SIERRA, 2009). A linguagem JAVA foi escolhida para o desenvolvimento da técnica TDRRC no objetivo deste trabalho e, por isso, este capítulo apresenta o mecanismo de anotações utilizando os exemplos e a sintaxe da linguagem JAVA

Este capítulo apresenta inicialmente uma introdução ao mecanismo de anotações. A seguir, apresenta os Tipos Anotação e as Anotações, com suas respectivas sintaxes e características. Também neste capítulo é apresentado o processador de anotações e o seu funcionamento. Finalmente, são tecidas as considerações finais, sumarizando os conceitos apresentados no capítulo.

4.1 Introdução ao Mecanismo de Anotações

As anotações podem ser usadas sempre que se deseja acrescentar informações adicionais em um código-fonte que possam ser facilmente processadas. Para permitir este processamento, as anotações são escritas de acordo com uma estrutura de dados previamente definida. Com a incorporação dos mecanismos de anotações na linguagem, sua sintaxe foi estendida para fornecer suporte à definição da estrutura de dados de uma anotação e a anotação de um elemento de um

[1]Em .NET, o mecanismo é chamado de mecanismo de atributos, mas a sintaxe em C# é praticamente a mesma.

programa. As anotações podem ser feitas nos seguintes elementos de um programa: pacotes, tipos, construtores, métodos, atributos, argumentos de métodos e variáveis locais. Também faz parte da infraestrutura de um mecanismo de anotações, uma biblioteca para processamento de anotações que permite o desenvolvimento rápido de programas capazes de extrair e utilizar as informações contidas nas anotações (BLOCH, 2008).

A extensão na linguagem permite que o programa contendo anotações possa compilar e armazenar as informações destas no binário, porém as anotações não afetam a maneira como um programa é executado.

Quanto ao momento de processamento, as anotações podem ser processadas durante a compilação, durante a execução ou separadamente. Ao processar as anotações durante a compilação, torna-se possível ao compilador gerar avisos ao encontrar certas anotações em certas condições. Um exemplo é a anotação @deprecated, nativa da linguagem JAVA que permite indicar que um método que foi descontinuado e a chamada a ele deve ser substituída por uma chamada ao método novo. Desta maneira; ao invés de remover o método antigo e criar um novo, gerando erros de compilação; um desenvolvedor pode anotar o método com a anotação @deprecated, o que não impedirá a compilação, mas gerará avisos de que o método foi descontinuado para que o desenvolvedor possa, na sua melhor conveniência, decidir quando substituir a chamada ao método antigo pela chamada ao método novo (SIERRA, 2009).

As anotações podem estar visíveis em tempo de execução e o próprio programa em execução pode processar e ler as informações nelas contidas. Para que isso seja possível, as anotações são carregadas para a memória pela máquina virtual JAVA e podem ser acessadas através da API de reflexão da linguagem JAVA. Esta API permite, através da programação, acessar a estrutura de um programa e listar suas classes e seus respectivos construtores, métodos e atributos. Se algum destes elementos possui uma anotação, a API permite acessá-las e obter todas as

informações que foram anotadas (FLANAGAN, 2005; BLOCH, 2008; FLANAGAN, 2005).

Quando processadas separadamente da compilação e execução, as anotações permitem gerar documentação, código e relatórios. Nesta situação, um processador de anotações é executado e extrai as informações das anotações para gerar a documentação, código ou relatório desejado. Por exemplo, pode-se gerar um relatório com todas as classes do sistema e seus autores. Para isso, pede-se aos desenvolvedores que coloquem uma anotação Autor nas classes que eles criarem. Desenvolve-se então um processador de anotações que processa estas anotações e monta o relatório.

A entidade básica do mecanismo de anotações é o Tipo Anotação. A Anotação é, portanto, uma instância de um Tipo Anotação. A Anotação é criada durante a programação, quando o desenvolvedor a insere no código-fonte. Diferentemente de um objeto, uma Anotação não pode ser instanciada em tempo de execução (FLANAGAN, 2005).

O Tipo Anotação é definido através dos seguintes elementos (FLANAGAN, 2005):

- nome – que identifica o Tipo Anotação e deve ser único dentro de um mesmo pacote;
- atributos – que definem a estrutura do Tipo Anotação, possuem um nome, um valor padrão opcional e um tipo que pode ser qualquer um dos tipos primitivos, *String*, *Class*, tipos enumerados, Tipos Anotação ou vetores de qualquer um dos tipos citados anteriormente. Não são aceitos vetores de vetores;
- tipo de alvo – que define os tipos de elemento de um programa ao qual uma Anotação do Tipo Anotação em questão pode ser associada. Os tipos de alvos possíveis são: pacotes, tipos (classes, interfaces, tipos enumerados e

anotações), atributos, métodos, construtores, argumentos de métodos e variáveis locais ;

- tempo de retenção – que especifica por quanto tempo a informação de uma Anotação do Tipo Anotação em questão é mantida, ou seja, de acordo com a retenção, a anotação pode ser descartada pelo compilador, ir para o binário da classe e ser ignorada pela máquina virtual, ou ir para o binário e ser carregada para a memória junto com o executável.

Quando um Tipo Anotação não define atributos, as anotações deste tipo são chamadas de Anotações Marcadoras e servem para informar que o elemento de programação anotado pertence ao mesmo grupo de todos os outros elementos que tenham esta mesma anotação. Pode ser usada por exemplo, para associar um estereótipo a uma classe.

Com os conhecimentos básicos sobre o mecanismo de anotações, pode-se aprofundar no Tipo Anotação.

4.2 TIPOS ANOTAÇÃO

A sintaxe de um tipo anotação pode ser descrita pela expressão em E-BNF (BNF estendida) da figura 7. A BNF estendida é uma metalinguagem que usa um conjunto de expressões regulares estendidas para representação de suas regras gramaticais (RAMOS et al., 2009). Uma gramática mais completa da sintaxe de anotações em JAVA em E-BNF é apresentada no APÊNDICE I – Gramática das Anotações em JAVA.

```
<TipoAnotação> ::= @interface <Identificador>  <CorpoTipoAnotação>

<CorpoTipoAnotação> ::=  { <DeclaraçãoAtributosAnotação> }

<DeclaraçãoAtributosAnotação> ::=  <DeclaraçãoAtributoAnotação>
<DeclaraçãoAtributosAnotação>

<DeclaraçãoAtributoAnotação> ::= <Tipo>  <NomeAtributo>() <ValorPadrão>;

<Tipo> ::= <Identificador>

<NomeAtributo> ::= <Identificador>

<ValorPadrão> ::= default <ValorAnotação>
```

Figura 7 – Sintaxe dos Tipos Anotação

Conforme pode ser visto na figura 7, para declarar um Tipo Anotação usa-se a palavra reservada *@interface* seguida do nome do tipo e da declaração dos seus atributos entre chaves. Ao se declarar um tipo usando *@interface,* este tipo implicitamente herda da classe *Annotation* que define as características de um Tipo Anotação (SIERRA, 2009).

Os atributos de um Tipo Anotação são declarados um a um separados por ponto e vírgula. A declaração do atributo inicia-se com o tipo do atributo, seguida do nome do atributo, um par de parênteses e opcionalmente da palavra reservada *default* e um valor padrão para o atributo Por exemplo, para definir um atributo cor do tipo *String* com valor padrão vermelho, declara-se da seguinte forma: *String* cor() *default* "Vermelho" (SIERRA, 2009; BLOCH, 2008). Estes atributos não podem, por definição, ter argumentos nem levantar exceções (FLANAGAN, 2005). Caso o atributo seja único e tenha nome *value*, é permitido ao criar uma anotação, suprimir o nome do atributo e escrever diretamente o valor (SIERRA, 2009). A figura 8 mostra

a declaração de um Tipo Anotação com essas características e a sintaxe resumida de uma anotação deste tipo.

```
// Declaração de um tipo anotação
public @interface Autor {
    String value();
}
// Anotacao usando a sintaxe completa
@Autor(value=”Vinicius”)
// Anotacao Usando sintaxe resumida
@Autor(”Vinicius”)
```

Figura 8 – Exemplo de declaração de um Tipo Anotação

Uma outra particularidade dos atributos de um Tipo Anotação está no uso de tipos genéricos que só pode ser feita para restringir os tipos de retorno de um atributo do tipo *Class*. Todos os outros usos de genéricos não são aceitos (FLANAGAN, 2005).

Para definir o tipo de alvo de um Tipo Anotação deve se usar uma anotação que faz parte da linguagem JAVA chamada *Target.* Esta anotação admite um único valor que é um vetor do tipo enumerado *ElementType.* O *ElementType* pode assumir os seguintes valores (SIERRA, 2009):

- *ANNOTATION_TYPE* – quando o alvo é um Tipo Anotação;
- *CONSTRUCTOR* – quando o alvo é um construtor;
- *FIELD* – quando o alvo é um atributo, incluíndo constantes enumeradas;
- *LOCAL_VARIABLE* – quando o alvo é uma variável local;
- *METHOD* – quando o alvo é um método;
- *PACKAGE* – quando o alvo é um pacote;

- *PARAMETER* – quando o alvo é um argumento de método;
- *TYPE* – quando o alvo é uma classe, interface, tipo anotação ou tipo enumerado;

A figura 9 define como tipo de alvo para o Tipo Anotação Autor, os métodos e os tipos (classe, interface, tipo anotação ou tipo enumerado).

```
@Target({ElementType.METHOD,ElementType.TYPE})
public @interface Autor {
    String value();
}
```

Figura 9 – Exemplo de declaração de tipo de alvo

Para definir o tempo de retenção de um Tipo Anotação, deve-se usar uma anotação do tipo *Retention.* Esta anotação pode assumir os seguintes valores do tipo enumerado *RetentionPolicy: RUNTIME* (para retenção em tempo de execução), *CLASS* (para retenção no binário, porém descartada em tempo de execução) e *SOURCE* (para retenção somente no código-fonte). A figura 10 define que as anotações do Tipo Anotação Autor serão mantidas em tempo de execução.

```
@Retention(RetentionPolicy.RUNTIME)
public @interface Autor {
    String value();
}
```

Figura 10 – Exemplo de declaração de tempo de retenção

Uma vez compreendido o tipo anotação é necessário conhecer melhor as anotações.

4.3 ANOTAÇÕES

A sintaxe das anotações pode ser descrita pela expressão em E-BNF da figura 11.

```
<Annotações>::= <Anotação> (<Annotações>)*

<Anotação> ::= @ <NomeTipo> ( <(>

                  <ValorAnotação> |

( ( <Identificador> = <ValorAnotação> (, (Identificador
=)? ValorAnotação )* )

             <)> )*
```

Figura 11 – Sintaxe da anotações

Conforme pode ser visto, uma anotação consiste do símbolo arroba @, seguido do nome do Tipo Anotação e um conjunto de valores separados por vírgula e entre parênteses. Cada valor deve ser precedido de um identificador que é o nome de um dos atributos declarado no Tipo Anotação. Os atributos podem aparecer em qualquer ordem e podem ser omitidos se o atributo tiver um valor padrão. Os valores podem ser uma constante, uma anotação ou um vetor (BLOCH, 2008). A figura 12 mostra o exemplo de uma anotação.

```
@Anotacao(nomeMembroString=”valor”,
nomeMembroInteiro=1,nomeMembroVetor={1,2,3})
```

Figura 12 – Exemplo de uma Anotação

Quanto à cardinalidade, cada elemento do programa pode ter várias anotações, porém somente uma única anotação de cada Tipo Anotação (SIERRA, 2009). A figura 13 mostra uma classe com duas anotações de Tipos Anotação distintos (AnotacaoA e AnotacaoB) e uma classe com duas anotações do mesmo Tipo Anotação (AnotacaoA), que por este motivo não compila.

```
@AnotacaoA(valor = 1)
@AnotacaoB(valor = 2)
public class Ok {
}

@AnotacaoA(valor = 1)
@AnotacaoA(valor = 2)
public class NaoOk {
}
```

Figura 13 – Cardinalidade de uma anotação

Os atributos de uma anotação devem ter valores que não sejam nulos e sejam um valor constante em tempo de compilação. Seu tipo deve ser compatível com o tipo declarado no Tipo Anotação (SIERRA, 2009).

Quanto aos alvos, existem algumas regras que devem ser notadas (FLANAGAN, 2005):

- Do ponto de vista sintático, as anotações podem ser associadas aos seguintes elementos de um programa: pacotes, tipos, construtores, métodos, atributos, parâmetros e variáveis locais;
- A anotação deve ser feita logo antes do alvo que se deseja anotar. A figura 14 mostra um exemplo de anotação em uma classe, em um atributo e em um método;

```
@Anotacao(nomeMembroString=”valor”,
nomeMembroInteiro=1,nomeMembroVetor={1,2,3})
public class ClasseAnotada {

    int atributoSemAnotacao;

    @AnotacaoAtributo(nomeMembro=”valor”)
    int atributoAnotado;

    public void metodoSemAnotacao() {
    }

    @AnotacaoMetodo(nomeMembro=”valor”)
    public void metodoAnotado() {
    }
}
```

Figura 14 – Exemplo de uma Anotação em elementos de um programa

- Para anotar um pacote é necessário criar um arquivo com nome package-info.java que pode conter uma anotação java-doc opcional, todas as

anotações do pacote e a declaração do pacote. A figura 15 mostra um exemplo de anotação em pacote. Quando o programa é compilado, um arquivo com nome package-info.class é gerado e este contém uma declaração de interface com o conteúdo das anotações;

```
/* O nome deste arquivo é package-info.java */
@AnotacaoDePacote(nomeMembroString="valor",
nomeMembroInteiro=1)
package br.usp.pacote.anotado;
```

Figura 15 – Exemplo de uma Anotação em um pacote

- Anotações feitas em variáveis locais ou argumentos de métodos aparecem como modificadores para este elemento do programa. Como o formato do binário não tem espaço para guardar informações referentes à variáveis locais, a retenção destas anotações é sempre limitada ao código-fonte. Já os argumentos de métodos podem ser acessados por reflexão e ter retenção no binário ou em tempo de execução.

Uma vez compreendidas as anotações, é importante compreender como estas são processadas e quais ferramentas podem ser utilizadas para este processamento.

4.4 Processamento de anotações

O processamento das anotações é realizado por um utilitário chamado apt. O apt pode ser executado em linha de comando que faz parte do pacote de desenvolvimento da linguagem JAVA. Este utilitário é capaz de executar um ou mais processadores de anotações baseado nas anotações presentes em um conjunto de códigos-fonte JAVA pré-definido (SIERRA, 2009).

Para utilizar o apt, deve-se passar os seguintes parâmetros (SIERRA, 2009):

- Diretório em que se encontra o código-fonte anotado;

- Diretório em que se encontram processadores de anotação a serem utilizados;
- Diretório em que se deve gerar a saída do processamento das anotações.

O utilitário apt trabalha da seguinte forma:

- Inicialmente o utilitário lê todo o código-fonte e determina quais anotações estão presentes;
- A seguir, ele busca quais processadores de anotações disponíveis são capazes de processar as anotações encontradas;
- Um a um os processadores são chamados para processar as anotações do código-fonte;
- Se estes processadores gerarem novo código-fonte, o utilitário é executado novamente de forma recursiva até que não haja mais código-fonte sem processar.

Com isso, para utilizar o apt, deve-se desenvolver um processador de anotações. Os processadores de anotação são classes JAVA que devem herdar da classe abstrata *AbstractProcessor*. Esta classe contém um método chamado *process* que é chamado pelo utilitário e pode processar um ou mais Tipos Anotações. A figura 16 mostra um processador de anotações que lista todas as anotações encontradas em um código-fonte, todas as classes e todas as classes com a anotação Autor (figura 9) e os seus respectivos autores, extraídos da anotação.

```
@SupportedAnnotationTypes(value= {"Autor"})
@SupportedSourceVersion(SourceVersion.RELEASE_6)
public class ProcessadorDeAnotacoes extends AbstractProcessor
{
   @Override
   public boolean process(Set<? extends TypeElement> anotacoes,
RoundEnvironment env) {
      System.out.println("Anotacoes encontradas");
      for(TypeElement t : anotacoes) {
         System.out.println(t.getSimpleName());
      }
      System.out.println("Classes encontradas");
      for (Element c : env.getRootElements()) {
         System.out.println(c.getSimpleName());
      }
      System.out.println("Classes com Anotacao Autor");
      for(Element c:env.getElementsAnnotatedWith(Autor.class))
      {
         Autor a = c.getAnnotation(Autor.class);
         System.out.println("Elemento ="+c.getSimpleName()+
             ", autor="+a.value());
      }
      return true;
   }
}
```

Figura 16 – Exemplo de processador de anotações

Como pode ser visto, o classe ProcessadorDeAnotacoes tem duas anotações:

- *@SupportedAnnotationTypes* – que indica quais Tipos Anotação este processador é capaz de processar. No exemplo, ele processa o Tipo Anotação Autor;

- *@SupportedSourceVersion* - que indica qual a versão do código-fonte mínima que o processador é capaz de processar. No exemplo, ele processa código-fonte JAVA versão 6 ou superior.

O método *process* tem dois argumentos: as anotações encontradas no código-fonte e o ambiente de processamento (classe *RoundEnvironment).* Com o ambiente de processamento é possível listar todas as classes encontradas no código-fonte (método *getRootElements*) e também listar os elementos do programa com uma anotação específica (método *getElementsAnnotatedWith*). Os elementos do programa são retornados através do tipo *Element* que é a classe abstrata que representa um elemento de um programa que pode ser anotado. O tipo *Element* possui vários métodos que permitem extrair informação sobre: o tipo de elemento que ele representa: classe, método, atributo, pacote ou argumento de método; o nome do elemento; as anotações do elemento; o arquivo e linha de código em que ele se encontra e os outros elementos com os quais ele está relacionado, por exemplo dado um método pode-se obter o elemento classe ao qual este método pertence ou os elementos que representam os argumentos deste método.

Através de uma série de APIs oferecidas pela linguagem JAVA, pode-se escrever um programa para extrair informações sobre as anotações e sobre o elemento de programação anotado (SIERRA, 2009). Com o desenvolvimento de processadores de anotações, pode-se gerar código, documentação e relatórios a partir de um código-fonte que foi previamente anotado.

4.5 Considerações Finais

O capítulo apresentou os conceitos, regras e a sintaxe para o uso e criação de anotações usando a linguagem JAVA. Conforme pode ser visto, as anotações constituem um recurso para acrescentar informações a elementos de um programa, sejam estes elementos um pacote, um tipo, um método, um atributo ou até mesmo

uma variável local. As informações adicionadas aos elementos de um programa seguem uma regra sintática de construção que é definida pelo Tipo Anotação. O Tipo Anotação torna-se, portanto, um importante recurso para criação dos meta-modelos das informações que podem ser associadas ao elemento de um programa. As informações adicionadas aos elementos de um programa podem ser processadas posteriomente, inclusive em tempo de execução. O uso das anotações permite a criação de elementos sintáticos que são capazes de adicionar novas semânticas a elementos de um programa, aumentando o poder de expressão da linguagem.

Para acrescentar informações em um código-fonte, além do uso de anotações, também são possíveis outras abordagens, dentre as quais destacam-se:

- Uso de comentários usando uma sintaxe bem definida e a criação de um processador destes comentários;
- Expansão da linguagem com a definição de novas palavras reservadas e especificação de compiladores para a linguagem expandida;
- Uso dos próprios mecanismos da linguagem para modelar os requisitos, criando classes e métodos que nunca seriam de fato executados, servindo apenas para representar os requisitos.

Quando comparado com a criação de uma sintaxe para ser usada nos comentários, as anotações tem a vantagem de poderem ser validadas pelo próprio compilador da linguagem e terem o processamento mais fácil em função de todas as ferramentas já desenvolvidas para o seu processamento. Por outro lado, ao criar uma nova sintaxe, não se estaria restrito ao que o mecanismo de anotações permite fazer. De maneira geral, a sintaxe de anotações é bastante flexível e, em função das razões já citadas anteriormente, o uso de anotações é mais vantajoso.

Quando comparado com a expansão da linguagem, assim como a sintaxe para comentários, tem-se muito mais flexibilidade para definir uma solução; por outro lado, o código-fonte passa a poder ser compilado apenas por um compilador proprietário, o que traz sérias limitações a sua aplicação. Por esta e pela mesma razão da sintaxe em comentários, esta opção acaba tendo um custo muito alto e por isso é menos vantajosa.

Quando comparado com a opção de criar estruturas não executáveis dentro do código-fonte, as anotações constituem uma solução mais elegante, pois fazem exatamente isso e com o suporte da linguagem de programação, não havendo razão para este caminho ser considerado.

Como as anotações podem fazer parte inclusive do código binário executável, ao utilizá-las, tem-se ainda a vantagem de permitir a recuperação das informações nelas contidas, ainda que o único artefato disponível do *software* seja seu executável.

5 TDRRC - TÉCNICA PARA DOCUMENTAÇÃO E RECUPERAÇÃO DE REQUISITOS

Este capítulo apresenta a TDRRC, através da sua descrição, do detalhamento dos seus artefatos, da especificação das suas atividades e de exemplos de utilização.

O capítulo também apresenta a aplicação da TDRRC dentro de um processo de desenvolvimento no contexto da reengenharia de requisitos, integrada à gestão de requisitos e em métodos ágeis.

5.1 VISÃO GERAL DA TDRRC

A TDRRC possui 3 atividades que permitem definir a representação de requisitos, o meta-modelo, os tipos anotação e o processador de anotações que são necessários para a sua aplicação dentro de um processo de desenvolvimento. Para estas atividades atribuiu-se o papel de Projetista da TDRRC.

A utilização da TDRRC constitui-se da atividade de Programação e Anotação do Código-Fonte, desempenhada pelo Desenvolvedor e da Geração do Modelo de Requisitos, desemepenhada por um Stakeholder. Ambas as atividades devem ser inseridas dentro de um processo de desenvolvimento de software onde a TDRRC é aplicada. As atividades envolvidas na definição e utilização da TDRRC estão representadas na figura 17.

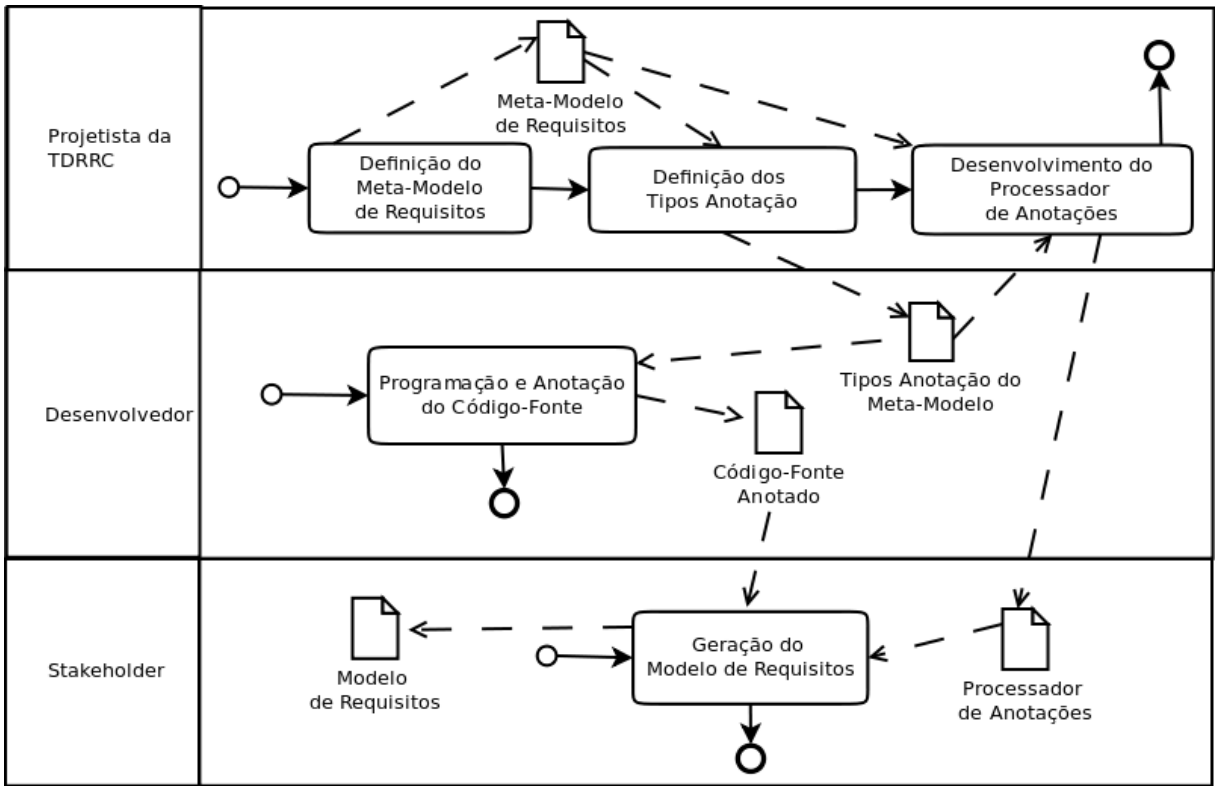

Figura 17 – Visão Geral da TDRRC

Considerou-se neste processo, três papeis:

- Projetista da TDRRC, que é o responsável por definir e criar a infra-estrutura para aplicação da TDRRC;
- Desenvolvedor, que aplica a TDRRC para documentar os requisitos no código-fonte durante a programação do *software*;
- *Stakeholder*, que aplica a TDRRC para gerar o Modelo de Requisitos através da execução do Processador de Anotações. Pode ser qualquer membro da equipe, incluindo os desenvolvedores e os projetistas, ou qualquer pessoa interessada em obter o Modelo de Requisitos.

Durante a aplicação da TDRRC, os seguintes artefatos são gerados:

- Meta-Modelo de Requisitos, que define a forma como um requisito é representado;
- Tipos Anotação do Meta-Modelo, que permitem a representação dos requisitos seguindo o Meta-Modelo de Requisitos;
- Código-Fonte Anotado, que consiste de um código-fonte de um *software* com anotações que representam os requisitos que estão implementados neste código-fonte;
- Processador de Anotações, que é um programa capaz de processar um Código-Fonte Anotado e gera o Modelo de Requisitos deste código-fonte baseado nas suas anotações;
- Modelo de Requisitos, que é uma representação de requisitos feita na forma definida pelo Meta-Modelo de Requisitos e gerada pelo Processador de Anotações.

Durante a aplicação da TDRRC, várias atividades são realizadas, as quais foram agrupadas de acordo com o papel responsável pela sua execução.

O Projetista da TDRRC realiza as seguintes atividades:

- Definição do Meta-Modelo de Requisitos: consiste da definição da estrutura para a representação dos requisitos, necessária para criação dos Tipos Anotação do Meta-Modelo;
- Definição dos Tipos Anotação: consiste da definição de um conjunto de tipos anotação que representa o Meta-Modelo de Requisitos e permite anotar o código-fonte com a informação dos requisitos;

- Desenvolvimento do Processador de Anotações: consiste da implementação de um programa capaz de gerar o Modelo de Requisitos do *software*, a partir do Código-Fonte Anotado com os Tipos Anotação do Meta-Modelo.

O Desenvolvedor realiza a seguinte atividade:

- Programação e Anotação do Código-fonte: consiste da programação do *software* com o trabalho extra de anotar as informações dos requisitos no código-fonte. Esta atividade só pode ser realizada após o Projetista da TDRRC finalizar a Definição dos Tipos Anotação.

O *Stakeholder* realiza a seguinte atividade:

- Geração Modelo de Requisitos: consiste da execução do Processador de Anotações com o Código-Fonte Anotado como entrada e produz como saída o Modelo de Requisitos em um formato adequado para ser consultado.

Uma vez apresentada a visão geral da TDRRC, é importante conhecer com mais detalhes os seus artefatos.

5.2 Artefatos da TDRRC

Esta seção descreve os artefatos da TDRRC:

- Meta-Modelo de Requisitos;
- Tipos Anotação do Meta-Modelo;
- Processador de Anotações;
- Código-Fonte Anotado;
- Modelo de Requisitos.

5.2.1 Meta-Modelo de Requisitos

O Meta-Modelo de Requisitos é um artefato criado pelo Projetista da TDRRC e especifica uma sintaxe para representação de requisitos. Esta sintaxe pode ser definida com diferentes níveis de detalhes, permitindo que a representação do requisito tenha uma estrutura mais ou menos rígida dependendo do objetivo a ser atingido. Um Meta-Modelo de Requisitos foi apresentado com detalhes na seção 2.4.

Em princípio, qualquer Meta-Modelo de Requisitos pode ser usado como base para a TDRRC, porém muitas vezes são necessárias modificações para torná-lo compatível com a representação no código-fonte e o uso de anotações. Os problemas de compatibilidade se devem ao fato de que nem todos elementos do Meta-Modelo de Requisitos podem ser associados a algum trecho de código-fonte e também às limitações na modelagem de anotações. Para minimizar estes problemas deve-se preferir definir o Meta-Modelo de Requisitos usando representações que são baseadas em atividade ou tarefas do sistema, pois estas, em geral, são fáceis de mapear no código-fonte e, portanto, adequadas para o uso na TDRRC.

A figura 18 apresenta um exemplo simples para facilitar o entendimento do artefato.

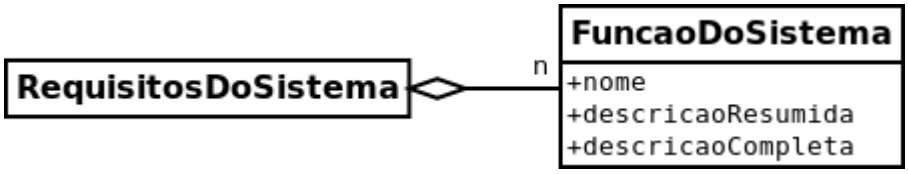

Figura 18 – Exemplo de Meta-Modelo de Requisitos

Conforme pode ser visto, o Meta-Modelo de Requisitos deste exemplo define os requisitos do sistema a partir das funções do sistema. Cada função do sistema tem um nome, uma descrição resumida e uma descrição completa.

5.2.2 Tipos Anotação do Meta-Modelo

Os Tipos Anotação do Meta-Modelo são artefatos criados pelo Projetista da TDRRC e permitem a representação, no código-fonte, de um Modelo de Requisitos baseado no Meta-Modelo de Requisitos. Para que isso seja possível, a estrutura definida pelos tipos anotação criados deve ser compatível com o Meta-Modelo de Requisitos. Os tipos anotação de JAVA, que são utlizados na TDRRC, foram apresentados na seção 4.2.

Ao modelar, através de tipos anotação, devem-se levar em consideração as seguintes limitações:

- Os atributos podem ser apenas dos tipos: primitivo, *String*, *Class*, enumeração, tipos anotação ou vetor de qualquer um dos tipos citados anteriormente;
- Não existe alocação dinâmica ou estruturas recursivas com tipos anotação;
- Cada tipo anotação será usado para anotar um trecho do código-fonte, portanto deve existir uma associação clara entre este tipo anotação e um trecho do código-fonte.

Em função destas limitações, a criação dos tipos anotação traz um grande desafio ao projetista da TDRRC que pode necessitar mudar o Meta-Modelo de Requisitos de maneira significativa para a aplicação da TDRRC.

Usando como base o Meta-Modelo de Requisitos da figura 18, a figura 19 apresenta o Tipo Anotação do Meta-Modelo correspondente.

```
public @interface FuncaoDoSistema {
    String nome();
    String descricaoResumida();
    String descricaoCompleta();
}
```

Figura 19 – Exemplo de Tipo Anotação do Meta-Modelo

A entidade RequisitosDoSistema não foi transformada em tipo anotação, pois não tem mapeamento explícito para o código-fonte. Para recuperar os requisitos do sistema e criar a entidade RequisitosDoSistema, listam-se todas as anotações FuncaoDoSistema encontradas no código-fonte.

5.2.3 Processador de Anotações

O Processador de Anotações é um programa para processar um Código-Fonte Anotado com as anotações dos Tipos Anotação do Meta-Modelo. Usando as bibliotecas para processamento de anotações, descritas na seção 4.4, o Processador de Anotações varre o código-fonte, encontra as anotações e cria o Modelo de Requisitos a partir delas.

O Modelo de Requisitos pode ser gerado em diversos formatos de acordo com a demanda e disponibilidade de recursos dos usuários da TDRRC. A geração de modelos mais sofisticados implica em custos maiores.

A figura 20 mostra um trecho do código-fonte de um Processador de Anotações para gerar o Modelo de Requisitos, a partir do Tipo Anotação da figura 19.

```
@Override
public boolean process(Set<? extends TypeElement> anotacoes,
RoundEnvironment env) {
   // A entidade requisitos do sistema do meta-modelo de
   // requisitos
   RequisitosDoSistema req = new RequisitosDoSistema();
   System.out.println("Anotacoes FuncaoDoSistema");
   for(Element
c:env.getElementsAnnotatedWith(FuncaoDoSistema.class))
   {
      // Recupera funcao do sistema
      FuncaoDoSistema f =
          c.getAnnotation(FuncaoDoSistema .class);
      // Adiciona funcao do sistema aos requisitos do sistema
      req.adicionarFuncaoDoSistema(f);
      System.out.println("Encontrada funcao ="+f.nome());
   }
   // A partir daqui o modelo de requisitos foi recuperado,
   // basta gerar os documentos no formato desejado.
   geraDocumento(req);
   return true;
}
```

Figura 20 – Exemplo de Processador de Anotações

Em função dos ajustes feitos para permitir a criação dos Tipos Anotação do Meta-Modelo, é necessário, à vezes, adaptar o Modelo de Requisitos gerado pelo Processador de Anotações com potenciais perdas semânticas.

No exemplo da figura 20, devido à simplicidade do Meta-Modelo de Requisitos e ao fato de se usar uma representação baseada em funções do sistema, não há perdas semânticas e um modelo de objetos que representa o Modelo de Requisitos é criado. A partir daí, o processador pode gerar um arquivo com o formato mais adequado para o usuário da TDRRC, usando as informações do Modelo de Requisitos recuperado.

5.2.4 Código-Fonte Anotado

Código-Fonte Anotado é o código-fonte de um *software* no qual foi aplicada a TDRRC e, por esta razão, ele possui o Modelo de Requisitos anotado e associado aos seus elementos que implementam estes requisitos.

Como as anotações não afetam a semântica do código-fonte, pode-se anotar um *software* já existente sem alterar o seu funcionamento.

A figura 21 mostra um trecho de Código-Fonte Anotado de uma calculadora. Cada função do sistema tem uma anotação FuncaoDoSistema e, com isso, é possível recuperar os requisitos implementados neste código-fonte ao executar o Processador de Anotações.

```
public class CalculadoraInt {
   private static final int SOMA = 1,SUBTRACAO = 2;

   public static void main(String[] args) {
      Scanner input = new Scanner( System.in );
      System.out.println( "Escolha 1-Soma\n 2-Subtração\n");
      int operaçao = Integer.parseInt(input.nextLine());
      System.out.print( "Informe o primeiro número: ");
      int numero1 = Integer.parseInt(input.nextLine());
      System.out.print( "Informe o segundo número: ");
      int numero2 = Integer.parseInt(input.nextLine());
      switch(operacao) {
         case SOMA: somar(numero1,numero2); break;
         case SUBTRACAO: subtrair(numero1,numero2); break;
      }
   }

   @FuncaoDoSistema(nome="Somar",descricaoResumida="Soma",
   descricaoCompleta="Esta função permite somar dois inteiros")
   public void somar(int numero1, int numero2) {
      System.out.println("A soma e"+(numero1+numero2));
   }

   @FuncaoDoSistema(nome="Subtrair",descricaoResumida=
   "Subtração",descricaoCompleta="Esta função permite subtrair"
   +" dois inteiros. O primeiro número digitado é o minuendo.")
   public void subtrair(int numero1, int numero2) {
      System.out.println("A soma e"+(numero1+numero2));
   }
}
```

Figura 21 – Exemplo de Código-Fonte Anotado

5.2.5 Modelo de Requisitos

O Modelo de Requisitos é o produto final da TDRRC e consiste do documento de requisitos gerado pelo Processador de Anotações ao processar um Código-Fonte Anotado. O Modelo de Requisitos é uma instância do Meta-Modelo de Requisitos.

A tabela 2 mostra o Modelo de Requisitos gerado por um Processador de Anotações com o Código-Fonte Anotado da figura 21.

Tabela 2 - Modelo de Requisitos da Calculadora

Requisitos do Sistema		
Nome	Descrição Resumida	Descrição Completa
Somar	Soma	Esta função permite somar dois inteiros
Subtrair	Subtração	Esta função permite subtrair dois inteiros. O primeiro número digitado é o minuendo.

As informações das duas funções anotadas no código-fonte foram extraídas e apresentadas em um formato de fácil leitura.

5.3 Atividades da TDRRC

Esta seção descreve as atividades da TDRRC:

- Definição do Meta-Modelo de Requisitos;
- Definição dos Tipos Anotação;
- Desenvolvimento do Processador de Anotações;
- Programação e Anotação do Código-Fonte;
- Geração do Modelo de Requisitos.

A descrição das atividades é ilustrada com exemplos para facilitar o seu entendimento.

5.3.1 Definição do Meta-Modelo de Requisitos

A representação de requisitos no código-fonte só traz benefícios se estes requisitos puderem ser validados e extraídos. Para isso, a representação dos requisitos deve seguir um modelo bem definido e detalhado em que cada um dos seus elementos tenha um lugar bem definido no código-fonte. Portanto, os requisitos no código-fonte não podem ser representados de maneira aleatória utilizando-se comentários em formato textual com descrições dos requisitos. Para que o modelo seja bem definido, é necessário definir um Meta-Modelo de Requisitos, no qual se baseia a modelagem dos requisitos.

A definição do Meta-Modelo de Requisitos envolve os seguintes passos:

1. Escolha da representação de requisitos;
2. Detalhamento do meta-modelo de acordo com a granularidade desejada;
3. Ajustes no meta-modelo para torná-lo mapeável no código-fonte e compatível com o mecanismo de anotações.

Conforme visto no capítulo 2, existem diversos tipos de representações de requisitos e cada uma delas traz vantagens e desvantagens, dependendo do domínio do problema e de questões técnicas e culturais. A escolha da representação mais adequada para a TDRRC não depende somente destes fatores, mas também da facilidade de se mapear os elementos desta representação no código-fonte. Por exemplo, representações de requisitos baseadas em descrição de atividades e tarefas do sistema podem permitir o mapeamento de cada atividade ou tarefa do sistema em funções ou métodos do código-fonte.

O detalhamento do meta-modelo consiste da definição de sua estrutura através de um número maior de partes, com o objetivo de permitir que o Processador de Anotações, que faz a leitura do Modelo de Requisitos, consiga recuperar a informação em um formato que facilite o seu processamento. Por exemplo, se a representação dos requisitos for modelada como sendo um único campo de texto contendo toda informação de requisitos, o processador poderia extrair esta informação, mas não poderia processar o seu conteúdo, se não fizer uso de técnicas sofisticadas para interpretar a descrição escrita em linguagem natural. Se, por outro lado, a representação de um requisito contiver dois campos: usuários (atores) e descrição separados por vírgula, o processador é capaz de listar atores, encontrar a descrição dos requisitos de um ator, etc. Quanto mais detalhado for o meta-modelo, mais elementos ele terá e menor será a granularidade da informação.

Na Definição do Meta-Modelo de Requisitos, o Projetista da TDRRC deve procurar detalhar o máximo possível para tornar o modelo mais facilmente processável; mas, ao mesmo tempo, se o meta-modelo tiver uma estrutura muito complexa, a tarefa de anotar o código-fonte pode se tornar muito trabalhosa, afetando a produtividade do Desenvolvedor. A definição da granularidade do meta-modelo não é uma ciência exata e o Projetista da TDRRC deve aumentar ou diminuir esta granularidade de acordo com o perfil da equipe, o tipo do projeto, a sua experiência e resultados obtidos na aplicação da TDRRC.

O projetista da TDRRC deve levar em conta que este meta-modelo deve ser representado por tipos anotação e mapeado no código-fonte, devendo portanto considerar que:

- Vários elementos de um requisito não tem um trecho do código-fonte específico com o qual estão relacionados, não tendo portanto um único ponto para se fazer anotação;

- As anotações não permitem criar estruturas recursivas, alocação dinâmica, referências a outras anotações e vetores de vetores;
- Algumas entidades representam elementos externos ao *software* ou abstrações que não tem mapeamento ou significado no código-fonte.

Com estas considerações, o projetista da TDRRC deve ajustar o meta-modelo, tomando decisões que podem trazer perdas semânticas, redundâncias ou ambiguidades. Estas decisões devem ser avaliadas ante as alternativas possíveis para os ajustes. Trata-se também de uma avaliação subjetiva, em que o Projetista da TDRRC poderia, em um caso extremo, chegar à conclusão que não vale à pena aplicar a TDRRC, pois o benefício de ter o requisito documentado diretamente no código-fonte não compensa a perda de informações no Modelo de Requisitos a ser adotado em função do Meta-Modelo de Requisitos final.

O ajuste do Meta-Modelo de Requisitos é feito através de uma ou mais operações nas suas entidades que não são mapeáveis no código-fonte ou que possuem uma estrutura sem suporte do mecanismo de anotações. As operações que podem ser feitas estão apresentadas na tabela 3.

A Revisão de Entidade consiste de uma revisão mais profunda da estrutura do Meta-Modelo de Requisitos e busca encontrar outras abstrações que permitam representar a mesma informação e que não tenham problemas de compatibilidade com o mecanismo de anotações. Por ser uma alteração em abstrações do meta-modelo, se não for feita com cuidado, pode acabar alterando de forma fundamental conceitos por trás da representação de requisitos escolhida inicialmente.

Tabela 3 - Operações para ajuste no meta-modelo

Nome	Descrição	Considerações
Revisão de Entidades	Consiste em alterar o meta-modelo revendo as abstrações utilizadas e substituindo-as por outras.	Nem sempre é possível rever. Uma revisão grande de abstrações pode tornar a representação do requisito inadequada.
Transferência de Informação.	Consiste da transferência de um ou mais atributos de uma entidade para outra.	Dependendo da cardinalidade da associação entre as entidades, pode trazer perda semântica, ambiguidade ou redundância.
Remoção de Entidade	Consiste da remoção de uma entidade do meta-modelo.	Traz a perda semântica da entidade removida, a não ser que seus atributos tenham sido transferidos para outra entidade.
Agregação de Atributos	Consiste da substituição de vários atributos por um único atributo do tipo *String* que deverá conter a informação.	Com a agregação, aumenta-se a granularidade e a informação que era processável, pois estava em vários atributos, passa a ser descrita em linguagem natural em um único atributo.

A Transferência de Informação é realizada antes da Remoção de Entidade para diminuir a perda semântica decorrente da retirada de uma entidade do meta-modelo. Desta maneira, transfere-se os atributos da entidade a ser removida para outras entidades com o qual esta estiver associada. Para transferir os atributos da

entidade a ser removida para uma entidade associada, deve-se levar em conta a cardinalidade do relacionamento:

- Se a cardinalidade do relacionamento é um para um não há problemas, pois não há redundância na informação e pode-se reconstruir a entidade removida trazendo o atributo de volta;
- Se a cardinalidade é de um (na entidade que transfere o atributo) para muitos (na entidade que recebe), o atributo, que antes aparecia uma única vez, passa aparecer várias vezes. Com a redundância podem ocorrer inconsistências, caso o valor do atributo nas várias instâncias, por algum erro, não seja igual;
- Se a cardinalidade do relacionamento é de muitos (na entidade que transfere o atributo) para um (na entidade que recebe), esta entidade deve receber não apenas um atributo, mas um vetor deste atributo e, de maneira geral, não há problemas para recuperação da informação;
- Se o relacionamento é de muitos para muitos pode haver uma perda semântica em função da ambiguidade em determinar a qual entidade removida pertencia qual valor do atributo, não sendo possível a recuperação da entidade.

A Remoção de Entidade consiste da retirada de uma entidade do meta-modelo. Ela é realizada quando a entidade não é mapeável no código-fonte, quando se deseja aumentar a granularidade do meta-modelo ou quando se deseja simplificar o modelo. A remoção de uma entidade não mapeável no código-fonte é justificada por evitar redundâncias e inconsistências que podem ser causadas ao colocar a entidade em lugares distintos no código-fonte. Antes de remover uma entidade deve-se avaliar a sua importância para o entendimento dos requisitos. Caso ela tenha pouca importância, pode-se decidir que a perda semântica é aceitável e simplesmente removê-la, simplificando o modelo. Caso se decida que a perda não é

aceitável, pode-se transferir a informação da entidade, através da operação de Transferência de Informação.

A Agregação de Atributos consiste em transformar vários atributos e entidades em um único atributo do tipo *String* e serve para aumentar a granularidade do meta-modelo, tornando o processo de anotá-lo mais simples e facilitando a Transferência de Informação. Sua principal desvantagem é transformar uma informação estruturada em um texto em linguagem natural que não é processável.

Uma vez apresentados os passos e considerações envolvidas na atividade Definição do Meta-Modelo de Requisitos, é apresentado um exemplo de realização desta atividade, onde os passos propostos foram seguidos.

O primeiro passo é a escolha da representação de requisitos. Para o exemplo, escolheu-se a representação de casos de uso pelos seguintes motivos:

- É a forma mais utilizada pelas empresas para representar os requisitos;
- É fácil de usar e oferece um tratamento sistemático das tarefas do usuário;
- É um padrão importante dentro da indústria de *software;*
- É baseada em um fluxo de atividades e permite que se mapeie atividades do sistema à métodos ou funções no código-fonte.

Uma vez escolhida a representação, o próximo passo é o detalhamento do meta-modelo de acordo com a granularidade desejada. Para a representação de casos de uso, um dos possíveis meta-modelos é o proposto pela UML. No meta-modelo da UML, os casos de uso são modelados através de um diagrama que está associado a um descrição textual de um fluxo de eventos, onde existe um fluxo principal e fluxos de extensão para descrever as exceções e os fluxos alternativos. A UML também propõe, como alternativa, a modelagem do comportamento do caso de uso através de diagramas de sequência ou de atividades (BOOCH et al., 1999). A forma como

estes diagramas deve ser usada fica a cargo do analista de requisitos. Como a UML é uma linguagem de modelagem que pode ser usada de maneira ampla, a proposta deixa muitos graus de liberdade para o analista modelar o sistema. A liberdade de modelagem é boa para o propósito da UML, mas não é adequada para a TDRRC, pois deseja-se que, idealmente, exista uma única maneira de se modelar e fazer a correspondência entre elementos do modelo e elementos do programa, para que não haja dúvidas de como as anotações devem ser feitas. Resumindo, o meta-modelo de casos de uso da forma como proposta pela UML é muito genérico e, devido à sua granularidade, embora possa ser usado, acaba não sendo o mais adequado para a aplicação da TDRRC.

Torna-se necessário tomar como ponto de partida o meta-modelo de casos de uso oferecido pela UML e detalhá-lo, diminuindo sua granularidade para torná-lo mais adequado para a aplicação na TDRRC. Decidiu-se porém, ao invés de seguir com esta abordagem, utilizar o meta-modelo de casos de uso proposto por Rui (2007), que foi apresentado na seção 2.4. Decidiu-se usar este meta-modelo, pois ele foi criado por alguém que não conhecia a TDRRC e, assim, pode-se testar as adaptações necessárias em um modelo que foi concebido sem considerar as limitações de modelagem da TDRRC. O meta-modelo mostrou-se adequado para a TDRRC, pois formaliza com mais detalhes que o meta-modelo dos casos de uso na forma como definida na UML e mantém a flexibilidade necessária para fornecer suporte à evolução de *software.*

Uma vez definido o meta-modelo, o próximo passo é o seu ajuste para permitir a criação de anotações. Como se trata de um exemplo sem um propósito específico de utilização, as decisões para os ajustes tiveram como objetivo deixar o modelo mais simples de usar e mantê-lo flexível. Considerando ainda que a TDRRC tem aplicação em métodos ágeis e na reengenharia de requisitos, onde a prática de documentar requisitos formalmente não estava ocorrendo, preferiu-se manter as entidades mais importantes do meta-modelo e considerar mais perdas semânticas

como aceitáveis do que se o contexto de utilização definido fosse outro. Os ajustes feitos foram os seguintes:

- Foi considerado que o detalhamento do Nivel de Eventos não era necessário, pois não se desejava processar as informações contidas em sua estrutura interna. Mantê-lo no Meta-Modelo de Requisitos significaria tornar o trabalho do desenvolvedor mais demorado, pois ele teria que anotar a estrutura interna dos eventos envolvidos nos episódios de um caso de uso. Desta forma, o Nível de Eventos foi eliminado do meta-modelo através da Agregação de Atributos de todo o nível de eventos e a Transferência de Informação para a única entidade com qual as entidades destes nível se relacionavam, a entidade Episódio. O resultado deste ajuste está expresso na figura 22;

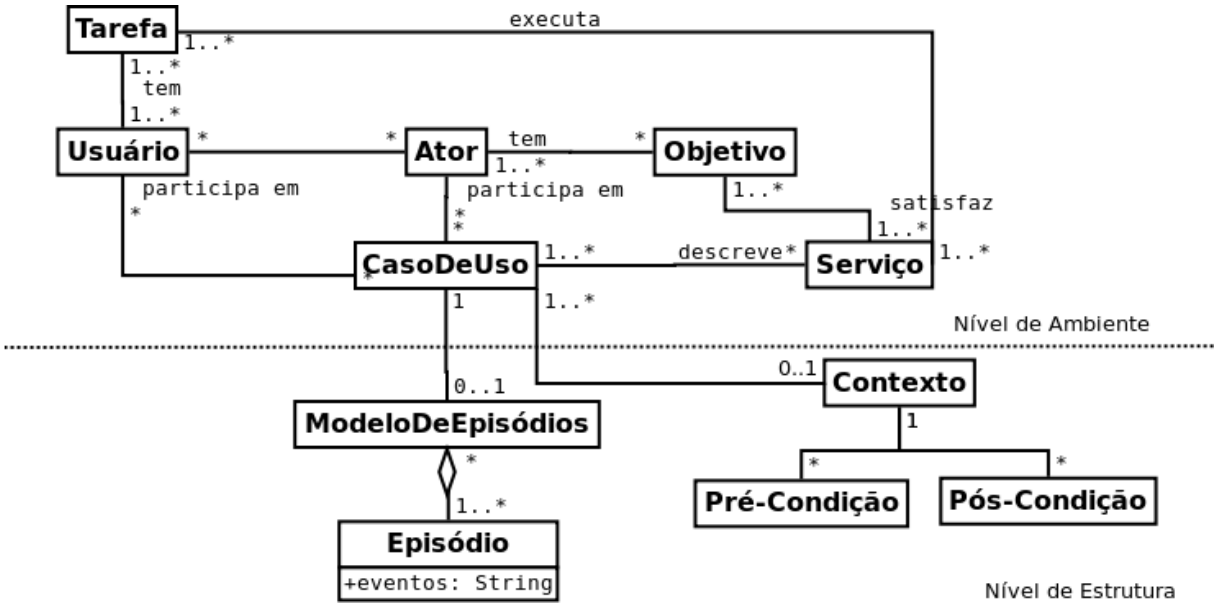

Figura 22 – Meta-Modelo de Casos de Uso após remoção do Nível de Eventos

- As entidades Tarefa e Usuário foram removidas por não terem mapeamento ou significado no código-fonte, pois representam elementos externos ao *software*. A perda da informação sobre tarefas e usuários foi considerada aceitável, pois Ator e Objetivo já registram informação similar de forma mais genérica e, por isso, não houve transferência de informação;

- As entidades Ator e Objetivo não tem um um mapeamento direto para um elemento do código-fonte, por isso decidiu-se removê-las. Antes da remoção foi feita a transferência dos seus atributos para a entidade Serviço. Para determinar os atores de um Caso de Uso, passou a ser necessário verificar os atores do Serviço descrito por este Caso de Uso;

A figura 23 mostra o meta-modelo após os ajustes citados anteriormente.

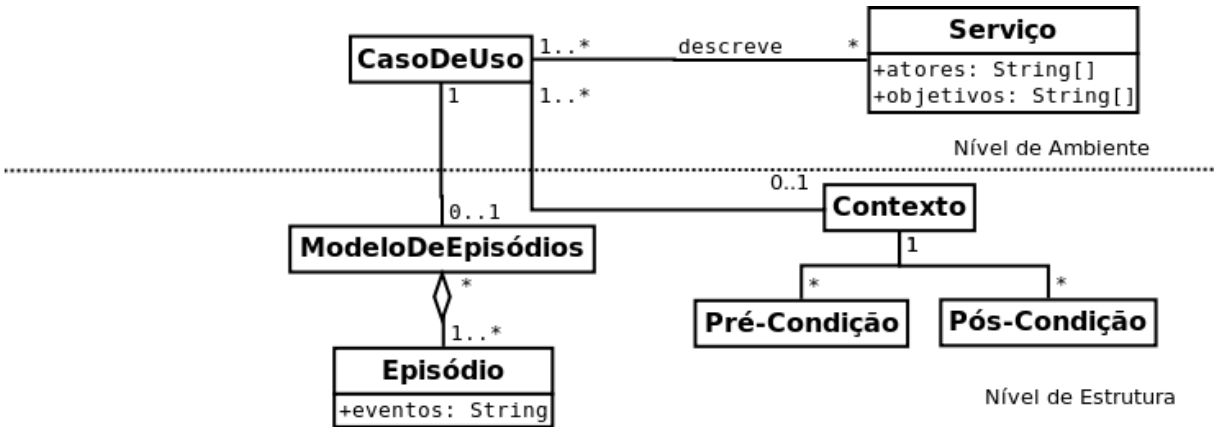

Figura 23 – Meta-Modelo de Casos de Uso após remoção de Ator, Objetivo, Tarefa e Usuário

- A entidade Caso de Uso não tem um mapeamento direto para um elemento do código-fonte, por isso decidiu-se removê-la. Antes da remoção foi feita a transferência dos seus atributos para as entidades com as quais esta se associa. Desta forma, o nome do Caso de Uso passou a ser um atributo das entidades: Serviço, Modelo de Episódios e Contexto;

- Decidiu-se criar uma nova entidade Condição que não existia no meta-modelo, pois foi verificado que a estrutura da Pré-Condição e a da Pós-Condição eram idênticas. Com isso removeu-se estas entidades e a diferença entre elas passou a ser dada pelo nome da associação da Condição com a entidade Contexto;

- A entidade Modelo de Episódios foi removida e sua informação transferida para o Episódio. O Modelo de Episódios, que é o conjunto de Episódios de

um dado Caso de Uso, passou a ser definido a partir da união de todos os Episódios que tenha este Caso de Uso como atributo.

Com estes ajustes, chegou-se ao modelo final que está representado na figura 24.

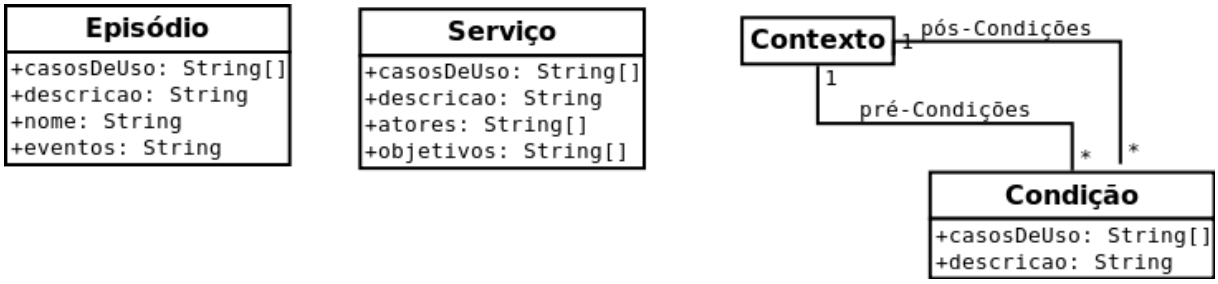

Figura 24 – Meta-Modelo de Casos de Uso Adaptado para a TDRRC

Para se assegurar que o meta-modelo é adequado para o uso na TDRRC, o Projetista da TDRRC deve verificar a qualidade do seu trabalho em cada um dos passos envolvidos na definição do meta-modelo. Ou seja, deve-se verificar se:

- A representação de requisitos é adequada ao domínio do problema, verificando o primeiro passo: “escolha da representação de requisitos”;
- A granularidade do meta-modelo é adequada, verificando o segundo passo: “detalhamento do meta-modelo de acordo com a granularidade desejada”;
- A representação escolhida é mapeável no código-fonte, verificando o terceiro passo: “ajustes no meta-modelo para torná-lo mapeável no código-fonte e compatível com o mecanismo de anotações”.

Como se trata de um exemplo, a verificação do primeiro e segundo passos foi discutida de forma inconclusiva, pois não existe um domínio do problema ou granularidade desejada sem que haja um projeto específico. Conforme já descrito no início do exemplo, procurou-se deixá-lo simples e fácil de usar para praticantes de métodos ágeis e projetos de reengenharia de requisitos. Acredita-se que as perdas semânticas adotadas são aceitáveis para a maioria dos projetos, pois o meta-

modelo de Rui (2007) é mais completo que o original de UML e, portanto, mesmo com as simplificações feitas tem-se um grau de detalhes maior do que se teria ao usar o meta-modelo de casos de uso da UML.

Em relação ao meta-modelo definido na figura 24, deve-se, inicialmente, verificar se ele é adequado ao domínio do problema. O meta-modelo permite definir casos de uso através da definição dos Serviços que são descritos por ele, do seu Contexto de utilização e dos Episódios que o compõem. Ao utilizar o Meta-Modelo de Requisitos deste exemplo, a definição de casos de uso está limitada a esta forma e é necessário verificar se esta limitação permite especificar aquilo que se deseja e que o domínio do problema exige. Caso estas limitações não permitam isso, o meta-modelo não está adequado e deve ser revisto.

Em relação ao meta-modelo definido na figura 24, deve-se, a seguir, verificar se granularidade do meta-modelo da figura 24 é adequada. Nas modificações realizadas, o Nível de Eventos foi removido por uma questão de simplificação. Ao fazer isso, optou-se por deixar a descrição do Nível de Eventos como um atributo dentro da entidade Episódio, aumentando a granularidade do Nível de Eventos. A consequência disso é que a especificação dos eventos de um caso de uso é feita em linguagem natural, na forma de um texto no atributo eventos do Episódio. Caso se deseje listar todos os eventos de um episódio, essa granularidade não é adequada e seria melhor manter a estrutura do Nível de Eventos. Por outro lado, é muito mais simples escrever um texto dizendo quais são os eventos do que identificar trechos no código-fonte responsáveis pelos eventos. A decisão da granularidade adequada é também subjetiva e devem ser pesados na decisão: a importância do detalhamento da informação durante o processamento, o domínio do problema e o esforço requerido para fazer a anotação no código-fonte.

Em relação ao meta-modelo definido na figura 24, deve-se, finalmente, verificar se a representação escolhida é mapeável no código-fonte. Pode-se notar que todas as entidades podem ser associadas à métodos ou funções em um código-fonte, pois

Episódios se referem a um passo, atividade ou tarefa de sistema de um caso de uso a ser executado por um método, Serviço se refere a um serviço oferecido pelo *software* que poderia estar implementado na forma de um método ou função. Contexto pode representar as pré-condições e pós-condições de um função ou método. Pode-se dizer que o meta-modelo é mapeável em um código-fonte.

Uma vez definido o Meta-Modelo de Requisitos, o próximo passo para o Projetista da TDRRC é definir os tipos anotação correspondentes a este meta-modelo.

5.3.2 Definição dos Tipos Anotação

Para definir os Tipos Anotação do Meta-Modelo, o Projetista da TDRRC deve mapear cada entidade do meta-modelo em um Tipo Anotação, de maneira análoga ao trabalho de um desenvolvedor frente a um diagrama de classes. De maneira geral, envolve as seguintes tarefas:

- Para cada entidade do meta-modelo deve criar um tipo anotação, que permitirá a criação de anotações no código-fonte;
- Para cada atributo da entidade deve ser criado um atributo no tipo anotação, que permitirá descrever a estrutura interna da entidade no código-fonte;
- Para as associações, deve-se decidir por mantê-la e ter redundância no código anotado ou transformá-la em uma referência por um identificador e não contar com o compilador para validá-la.

Dos itens listados anteriormente, o único que tem uma complexidade maior é o tratamento das associações, pois os outros são mecânicos e não requerem nenhuma tomada de decisões. Por este motivo, os detalhes deste item são discutidos através de um exemplo tomando, como base, o meta-modelo da figura 25.

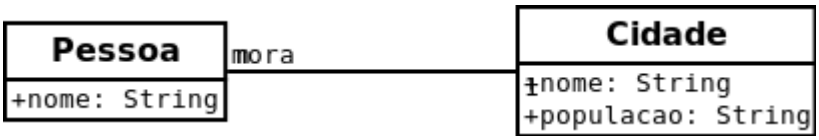

Figura 25 – Exemplo de Meta-modelo com associação

Neste meta-modelo, uma Pessoa tem um nome e mora em uma Cidade. Cidade tem um nome e uma população. Se os tipos anotação forem definidos de forma a manter a associação, eles ficariam conforme a figura 26 e as anotações usando estes tipos conforme a figura 27.

```
Public @interface Cidade {
   public String nome();
   public int populacao();
}
public @interface Pessoa {
   public String nome();
   public Cidade mora();
}
```

Figura 26 – Tipo anotação mantendo associação do meta-modelo

```
@Pessoa( nome = “Jose”, mora=@Cidade(nome=”São
Paulo”, populacao=2000000))
@Pessoa( nome = “Maria”, mora=@Cidade(nome=”São
Paulo”, populacao=2000000))
```

Figura 27 – Anotações com redundância

Como se pode observar, a população da Cidade fica repetida em toda anotação Pessoa, podendo inclusive ter inconsistências caso se coloquem valores diferentes para a população de uma mesma cidade nas diferentes anotações. Para evitar estas inconsistências, deve-se criar um *software* validador que processe as anotações e

indique quando valores distintos forem usados. Este validador, por conveniência, pode ser uma parte do Processador de Anotações da TDRRC.

A outra opção é transformar a associação em uma referência a um identificador, ou seja, ao invés de Pessoa ter uma referência para uma Cidade, Pessoa tem um atributo que serve para identificar qual Cidade ela referencia, no caso, o nome da cidade. Dessa maneira- obtém-se os tipos anotação conforme a figura 28 e anotações conforme a figura 29.

```
Public @interface Cidade {
   public String nome();
   public int populacao();
}
public @interface Pessoa {
   public String nome();
   public String mora();
}
```

Figura 28 – Tipo anotação sem associação

```
@Cidade(nome=”São Paulo”, populacao=2000000)
@Pessoa( nome = “Jose”, mora=”São Paulo”)
@Pessoa( nome = “Maria”, mora=”São Paulo”)
```

Figura 29 – Anotações sem redundância

A anotação de cidade fica separada da anotação de pessoa. Com esta opção, as inconsistências podem ocorrer, se uma Pessoa referencia uma cidade que não foi anotada. Neste caso, pode-se, de maneira análoga à opção de manter a associação, criar um validador que identifica estas inconsistências. Uma outra questão é onde anotar a Cidade; se a entidade não tiver um mapeamento claro para o código-fonte, pode-se inadvertidamente anotar duas vezes a mesma cidade em trechos de

código-fonte diferentes. Neste caso, a melhor opção é manter a associação e conviver com a redundância.

O meta-modelo de casos de uso, definido como exemplo na seção 5.3.1 (figura 24), foi usado como exemplo da Definição de Tipos Anotação. Desta forma, cada uma das suas entidades: Serviço, Contexto, Episódio e Condição; foi transformada em um tipo anotação.

O tipo anotação Serviço é utilizado para anotar os métodos que implementam um serviço e está representado na figura 30.

```
public @interface Servico {
   public String[] casosDeUso();
   public String descricao;
   public String[] atores;
   public String[] objetivos;
}
```

Figura 30 – Tipo anotação Serviço

Os atributos do tipo anotação Serviço são representados através de uma lista de *Strings* com os nomes dos casos de uso que descrevem este serviço, uma *String* com a descrição do serviço, uma lista de *Strings* com os nomes dos atores destes casos de uso e uma lista de *Strings* com a descrição dos objetivos do serviço.

O tipo anotação Condição e o tipo anotação Contexto com suas respectivas pré-condições e pós-condições foram representados na figura 31.

```
public @interface Condicao {
   public String[] casosDeUso();
   public String descricao();
}
public @interface Contexto {
   public Condicao[] preCondicoes();
   public Condicao[] posCondicoes();
}
```

Figura 31 – Tipos anotação Condição e Contexto

Os atributos do tipo anotação Condição são: uma lista de *Strings* com os nomes dos casos de uso ao qual esta condição se refere e uma *String* com a descrição da condição. O Contexto é representado por duas listas de Condição, uma se referindo às pré-condições e outra às pós-condições.

O tipo anotação Episódio foi representado na figura 32.

```
public @interface Episodio {
   public String[] casosDeUso();
   public String descricao();
   public String nome();
   public String eventos();
}
```

Figura 32 – Tipo anotação Episódio

Os atributos do tipo anotação Episódio são: uma lista de *Strings* com os nomes dos Casos de Uso ao qual este episódio se relaciona, uma *String* com a descrição do episódio, uma *String* com o nome do Episódio e uma *String* chamada eventos para descrever todo o modelo de eventos do episódio.

Estes tipos anotação permitem anotar um *software,* associando elementos do programa aos elementos do meta-modelo de um caso de uso.

Uma vez definidos os Tipos Anotação do Meta-Modelo, o próximo passo para o Projetista da TDRRC é desenvolver o Processador de Anotações, capaz de extrair o Modelo de Requisitos de um Código-Fonte Anotado.

5.3.3 Desenvolvimento do Processador de Anotações

Para o desenvolvimento do Processador de Anotações, o Projetista da TDRRC deve criar um *software* capaz de receber, como entrada de dados, o Código-Fonte Anotado e tem, como saída, o Modelo de Requisitos que está representado no código. Para desenvolver esse *software* deve-se utilizar a ferramenta APT *(Annotation Processing Tool),* uma ferramenta da linguagem JAVA para o processamento de anotações, que foi detalhada na seção 4.4. O Processador de Anotações deve varrer todas as anotações, extrair as informações da anotação e de onde ela está localizada no código-fonte através da APT e construir um Modelo de Requisitos baseado no Meta-Modelo de Requisitos definido no projeto da TDRRC. Com o Modelo de Requisitos, o Processador de Anotações pode gerar uma saída no formato mais adequado aos seus usuários: um documento, diagrama, arquivo XML ou qualquer formato que se julgar apropriado.

Os Tipos Anotações do Meta-Modelo definidos como exemplo na seção 5.3.2 e o Meta-Modelo de Requisitos definido na seção 5.3.1 são usados para ilustrar o desenvolvimento do Processador de Anotações.

O Processador de Anotações deve: levar em consideração as limitações impostas na modelagem pelo mecanismo de anotações e ser capaz de fazer validações sintáticas que não são feitas pelo compilador JAVA e de recuperar o meta-modelo mais próximo daquele anterior aos ajustes feitos para torná-lo compatível com o mecanismo de anotações.

As validações são necessárias, pois a associação entre os tipos anotação pode deixar inconsistências que não são validadas pelo compilador JAVA. Mais especificamente, quando a associação é feita pelo valor de um identificador em um atributo, como no exemplo das figuras 28 e 29, o compilador JAVA não consegue identificar se a anotação está referenciando uma anotação que não existe. Também quando se decide manter a associação, como no exemplo das figuras 26 e 27 , pode haver redundância e potencialmente inconsistências não detectáveis pelo compilador JAVA.

Quanto à recuperação do meta-modelo mais próximo daquele anterior aos ajustes, vale observar que os ajustes no meta-modelo foram necessários para permitir o uso do mecanismo de anotações e o mapeamento do modelo de requisitos no código-fonte. Durante estes ajustes a semântica original foi transformada, ou perdida, dependendo das decisões tomadas na definição do meta-modelo. O Processador de Anotações deve reverter os ajustes, recuperando a semântica mais próxima da original possível frente aos ajustes feitos. A semântica original só pode ser totalmente recuperada quando nenhum dos ajustes trouxer perdas semânticas.

No meta-modelo de casos de uso proposto na figura 24, é necessário validar as referências aos nomes dos casos de uso. As anotações do tipo anotação Condição definidas dentro de uma anotação do tipo anotação Contexto referenciam uma lista de nomes de casos de uso (figura 31). O mesmo ocorre nos tipos anotação Serviço (figura 30) e Episódio (figura 32). Neste caso, o Processador de Anotações deve verificar se os casos de uso anotados em um contexto foram anotados em pelo menos um serviço ou episódio.

Uma vez feita a validação, deve-se desenvolver a recuperação do Modelo de Requisitos. O pseudo código do Processador de Anotações para realizar esta tarefa baseado no Meta-Modelo de Requisitos definido na seção 5.3.1 está representado na figura 33.

```
Lista epLista = Todas anotações do Tipo Anotação Episódio;
Lista srvLista = Todas anotações do Tipo Anotação Serviço;
Lista ctxLista = Todas anotações do Tipo Anotação Contexto;
Lista casosLista = Nova lista;
Para cada Episódio ep em epLista:
      Lista casosTemp = ep.casosDeUso;
      Para cada String caso em casosTemp:
            Se caso não existe em casosLista:
                  Cria objeto casoDeUso com nome caso
                  Adiciona caso à casosLista
                  Associa ep ao objeto casoDeUso
            Senão:
                  Recupera o objeto casoDeUso de casosLista
                  Associa ep ao objeto casoDeUso
Para cada Serviço srv em srvLista:
      Lista casosTemp = srv.casosDeUso;
      Para cada String caso em casosTemp
            Se caso não existe em casosLista:
                  Cria objeto casoDeUso com nome caso
                  Adiciona caso à casosLista
            Senão:
                  Recupera o objeto casoDeUso de casosLista
            Associa srv ao objeto casoDeUso
            Lista atoresTemp = srv.atores;
            Para cada String ator em atoresTemp
                  Se ator não existe na lista de atores de um caso de uso:
                        Adiciona ator à lista de atores do caso de uso
                  Associa os objetivos do Serviço ao ator
Para cada Contexto ctx em ctxLista:
      Para cada Condicao pre em ctx.preCondicoes;
            Lista casosTemp = pre.casosDeUso;
            Para cada String caso em casosTemp:
                  Se caso não existe em casosLista:
                        Cria objeto casoDeUso com nome caso
                        Adiciona caso à casosLista
                        Associa pos ao contexto do objeto casoDeUso
                  Senão:
                        Recupera o objeto casoDeUso de casosLista
                        Associa pos ao contexto do objeto casoDeUso
      Para cada Condicao pos em ctx.posCondicoes;
            Lista casosTemp = pos.casosDeUso;
            Para cada String caso em casosTemp:
                  Se caso não existe em casosLista:
                        Cria objeto casoDeUso com nome caso
                        Adiciona caso à casosLista
                        Associa pos ao contexto do objeto casoDeUso
                  Senão:
                        Recupera o objeto casoDeUso de casosLista
                        Associa pos ao contexto do  objeto casoDeUso
```

Figura 33 – Pseudo código da recuperação do Modelo de Requisitos

De maneira geral, o pseudo-código da recuperação do Modelo de Requisitos tem como objetivo a criar um modelo de requisitos que consiste de:

- Uma lista de casos de uso, obtida a partir da recuperação dos seus episódios;
- Serviços que satisfazem estes casos de uso;
- Contexto de cada um dos casos de uso.

O pseudo-código da figura 33 realiza os seguintes passos:

- Identifica os casos de uso anotados e cria objetos do tipo CasoDeUso. Para isso, cria uma lista sem repetições com todos os casos de uso anotados no código, a partir da busca de todos os casos de uso encontrados em todas a anotações Episódio. Com esta lista, cria os objetos CasoDeUso e associa-os aos respectivos Episódios;
- Completa a lista de casos de uso e associa os serviços que satisfazem estes casos de uso. Para isso, busca todos os casos de uso encontrados em todas a anotações Serviço. Caso algum dos casos de uso encontrado não esteja na lista iniciada no passo anterior, cria o objeto CasoDeUso e acrescenta-o à lista. Para cada a anotação Serviço encontrada: busca o CasoDeUso correspondente na lista e associa-os, pega os nomes dos atores anotados no Serviço, cria objetos Ator e acrescenta-os à lista de atores do CasoDeUso. Pega a lista de objetivos anotadas no serviço e cria objetos Objetivo, associando-os a todos atores do Serviço e ao Serviço;
- Busca os casos de uso anotados dentro das pré e pós-condições pertencentes às anotações Contexto, acrescentando-os, se necessário, à lista de casos de uso, conforme feito nos passos anteriores, e associando-os ao Contexto correspondente. Com isso a lista de Casos de Uso fica completa e cada caso de uso tem pré e pós-condições associadas ao seu contexto.

Com o modelo de requisitos recuperado, o Processador de Anotações pode armazená-lo em um banco de dados, sincronizá-lo com um repositório de requisitos, gerar documentos, diagramas ou qualquer outro tipo de formato desejado pelo usuário.

O desenvolvimento de um Processador de Anotações completa o projeto da TDRRC, a próxima atividade é a Programação e Anotação do Código-Fonte que é feita pelo Desenvolvedor.

5.3.4 Programação e Anotação do Código-fonte

A Programação e Anotação do Código-Fonte é realizada pelo Desenvolvedor e pode ocorrer de duas maneiras:

- Na implementação de um novo requisito que foi claramente definido durante o processo de desenvolvimento;
- Na manutenção de um código-fonte legado em que se deseja documentar os requisitos já implementados.

A diferença entre as duas maneiras está no processo mental envolvido na realização desta atividade. Quando se está implementando um novo requisito que foi claramente definido, o Desenvolvedor já tem o requisito estruturado em uma representação mapeável para as anotações e, ao programar, ele anota o código-fonte, documentando aquilo que foi implementado. Se a representação não é mapeável para as anotações, a representação ou as anotações precisam ser revistas.

Como os requisitos estão organizados em uma representação que tem uma certa estrutura e, pelo fato do Desenvolvedor estar ciente de que tem que fazer as anotações, a implementação tende a ser organizada em função desta estrutura. A representação dos requisitos não diz como implementar, mas lista, em passos ou

itens, o que deve ser feito e esta lista acaba direcionando a organização de uma nova implementação.

Já na manutenção de um código-fonte legado, a implementação já foi feita e o Desenvolvedor está alterando o código-fonte que foi organizado sem levar em conta a estrutura do Meta-Modelo de Requisitos. Além de programar as alterações do código-fonte, ele anota o código-fonte com aquilo que ele descobre sobre os requisitos implementados. Dependendo do seu grau de conhecimento do *software*, os requisitos podem estar mais ou menos claros em sua mente e, à medida que o seu conhecimento amplia, o Desenvolvedor consegue fazer mais anotações no código-fonte. Uma das dificuldades, neste cenário, deve-se ao fato de que o código-fonte foi implementado sem levar em consideração a estrutura do Meta-Modelo de Requisitos. Caso não estejam compatíveis pode ser necessário refatorar o código-fonte antes de anotá-lo.

Uma outra dificuldade está relacionada com a capacidade do Desenvolvedor de abstrair e documentar o requisito lendo o código-fonte. Geralmente, os desenvolvedores tendem a visualizar a solução ao invés do problema. É bastante difícil para muitos deles separar a descrição dos requisitos da sua implementação. Para melhorar a documentação, neste caso, os requisitos documentados podem ser revisados por outros membros da equipe através da Geração do Modelo de Requisitos.

5.3.5 Geração do Modelo de Requisitos

A Geração do Modelo de Requisitos é uma atividade executada pelo *stakeholder* quando necessita obter os requisitos que estão documentados no código-fonte. Para isto, esta pessoa precisará ter acesso ao Código-Fonte Anotado e ao Processador de Anotações. Ao executar o Processador de Anotações com o Código-Fonte Anotado como entrada, obtém-se o Modelo de Requisitos. Os formatos disponíveis

para a visualização do Modelo de Requisitos dependem do Meta-Modelo de Requisitos e da sofisticação da implementação do Processador de Anotações.

5.4 Aplicação da TDRRC

Na motivação desta tese foram destacadas as seguintes situações dentro do contexto do desenvolvimento e manutenção de *software* que levaram a este trabalho:

- Um processo de desenvolvimento ágil, em que a documentação dos requisitos é vista como um processo lento e de pouco valor e que a documentação formal tira agilidade do desenvolvimento (CAO; RAMESH, 2008);
- Manutenção de *software*, em que os documentos de requisitos não existem ou estão desatualizados, fato que ocorre na maioria dos projetos de desenvolvimento de *software* depois de um certo tempo de uso (FAHMI et al., 2007);
- Manutenção de *software,* em que técnicas de gestão de requisitos foram aplicadas e um repositório de requisitos foi criado.

Destas situações, com o desenvolvimento do trabalho, foram escolhidos três contextos para aplicação da TDRRC:

- Reengenharia de requisitos: pois, quando os documentos de requisitos não existem ou estão desatualizados, a reengenharia de requisitos é a forma de recuperá-los;
- Integrada à gestão de requisitos: pois, durante o desenvolvimento do trabalho, descobriu-se que a TDRRC contribui para uma parte não resolvida do trabalho de Fahmi (2007) que propõe a integração da engenharia reversa de requisitos na gestão de requisitos;

- Métodos ágeis: pois permite apresenta como a TDRRC permite que a documentação não seja tão lenta quanto nos processos de desenvolvimento tradicionais e permite manter os príncipios ágeis.

A aplicação da TDRRC é descrita para cada um dos contextos, assim como os seus potenciais benefícios.

5.4.1 Aplicação da TDRRC na reengenharia de requisitos

A reengenharia de requisitos, conforme visto na seção 3.3, é a aplicação das técnicas de reengenharia de *software* com o objetivo de recuperar, melhorar a qualidade ou reorganizar a documentação sobre os requisitos de um sistema (SNEED, 2008). As técnicas propostas pela literatura para a reengenharia de requisitos envolvem a análise do código-fonte, entrevista com usuário e utilização do *software* e podem ser usadas independente da existência de documentos de requisitos (YU et al., 2005; YANG et al., 2006).

A reengenharia de requisitos é realizada através de dois processos:

- Recuperação dos requisitos: consiste da aplicação de técnicas para determinar os requisitos que foram implementados em um *software*;

- Documentação dos requisitos: consiste do registro e do armazenamento das informações coletadas no processo de recuperação dos requisitos.

Estes dois processos são realizados de maneira iterativa e cíclica, de forma que os documentos de requisitos servem para auxiliar a recuperação dos requisitos ainda não documentados, e o resultado da recuperação de requisitos permite gerar novos documentos de requisitos mais completos que os anteriores.

A TDRRC é capaz de oferecer uma alternativa para a documentação de requisitos e servir como apoio à recuperação de requisitos. O uso de técnicas de recuperação de requisitos continua sendo necessário mesmo com a aplicação da TDRRC.

No processo de recuperação de requisitos, deve-se usar as técnicas propostas pela literatura para recuperação dos requisitos e identificar, um a um, quais são os requisitos do *software*. À medida que vão sendo descobertos, os requisitos vão sendo anotados no código-fonte. Como, muitas vezes, a recuperação dos requisitos ocorre através da inspeção do código-fonte, é adequado anotar no próprio local sendo analisado o que aquele trecho de código-fonte faz. As anotação feitas nos trechos de código-fonte já analisados facilita o trabalho de análise do código-fonte, pois evita releituras e reanálise dos mesmos trechos. Isto é especialmente importante em *software* de grande porte e quando a equipe responsável pela reengenharia de requisitos tem mais do que uma pessoa.

Quando existem documentos de requisitos, ao aplicar a TDRRC, deve-se não só analisar o código-fonte como também copiar as informações da documentação existente para ele através de anotações. Para determinar que requisitos devem ser anotados em que trecho do código-fonte, a recuperação das ligações de rastreabilidade é importante. Neste caso, é possível utilizar técnicas para recuperação automática da rastreabilidade entre o código-fonte e os requisitos, através de um *software* que faz a análise semântica de ambos (MARCUS; MALETIC, 2003; JERMAKOVICS et al., 2008; DE LUCIA et al., 2008; LORMANS; DEURSEN, 2009; OLIVETO et al., 2010).

Com a aplicação da TDRRC, o processo de documentação dos requisitos fica bem mais simples, pois os requisitos já estão armazenados junto com o código-fonte e para obter os documentos de requisitos, executa-se o Processador de Anotações. Caso se deseje armazenar os requisitos em um repositório de requisitos, surge um desafio de sincronizar os requisitos que estão armazenados no repositório com os

que estão anotados no código-fonte. É importante lembrar que, a partir do código-fonte anotado é possível gerar os documentos de requisitos. No entanto, para gerar e atualizar anotações em um código-fonte a partir dos documentos de requisitos, é necessário manter ligações de rastreabilidade entre os documentos e o código-fonte e criar um programa bem mais sofisticado do que o Processador de Anotações apresentado, pois deve-se atualizar anotações sem alterar o funcionamento do código-fonte. Em função disso, quando o uso do repositório de requisitos é mandatório, recomenda-se aplicar a TDRRC e gerar os documentos a cada atualização, armazenando-os no repositório.

A outra solução seria colocar nos documentos, ligações de rastreabilidade para as anotações no código-fonte e desenvolver uma ferramenta para sincronizar ambas. Para esta solução, é questionável se haverá vantagens em aplicar a TDRRC, já que a informação de requisitos está duplicada e poderá ser editada nos dois lugares, abrindo-se a possibilidade de conflitos e inconsistências entre os documentos de requisitos e os requisitos anotados no código-fonte. Só é viável adotar este caminho com uma boa ferramenta para a sincronização dos requisitos no repositório e no código-fonte. Em função dessas dificuldades, recomenda-se utilizar ao aplicar a TDRRC utilizar o código-fonte anotado como o único ponto para documentação e armazenamento de requisitos, evitando-se o trabalho de sincronização com o repositório de requisitos.

A aplicação da TDRRC neste contexto:

- Facilita o processo de documentação de requisitos, ao oferecer uma ferramenta que gera automaticamente os documentos de requisitos;
- Elimina a necessidade de um repositório separado para requisitos, pois os requisitos passam a ser armazenados junto com o código-fonte;

- Facilita o processo de recuperação de requisitos, ao oferecer uma maneira sistemática de documentar os requisitos à medida que forem sendo descobertos diretamente no código-fonte.

5.4.2 Aplicação da TDRRC integrada à Gestão de Requisitos

A TDRRC torna viável implementação da gestão de requisitos baseada em engenharia reversa conforme proposto por Fahmi et al. (2007) e apresentado na seção 3.4. Dentro da proposta de Fahmi et al. (2007), a gestão de requisitos deveria ser um processo iterativo e cíclico onde requisitos implementados são usados na análise de novos requisitos no *software.* Embora o modelo de Fahmi et al. (2007) esteja bem estruturado e apresente diversas vantagens que foram bem embasadas, ele depende do desenvolvimento de um mecanismo automático de engenharia reversa de requisitos que permita gerar os requisitos implementados a partir do código-fonte que foi produzido no ciclo anterior. Na pesquisa do estado da arte, verificou-se que a tecnologia para a construção deste mecanismo ainda não está disponível.

Ao aplicar a TDRRC, pode-se substituir a engenharia reversa pela geração de documentos de requisitos e, com isso, viabilizar a implantação do modelo de Fahmi et al. (2007). Para isso, são necessárias modificações no modelo proposto por FAHMI et al. (2007) e mostrado na figura 6 da seção 3.4. O modelo modificado é apresentado na figura 34.

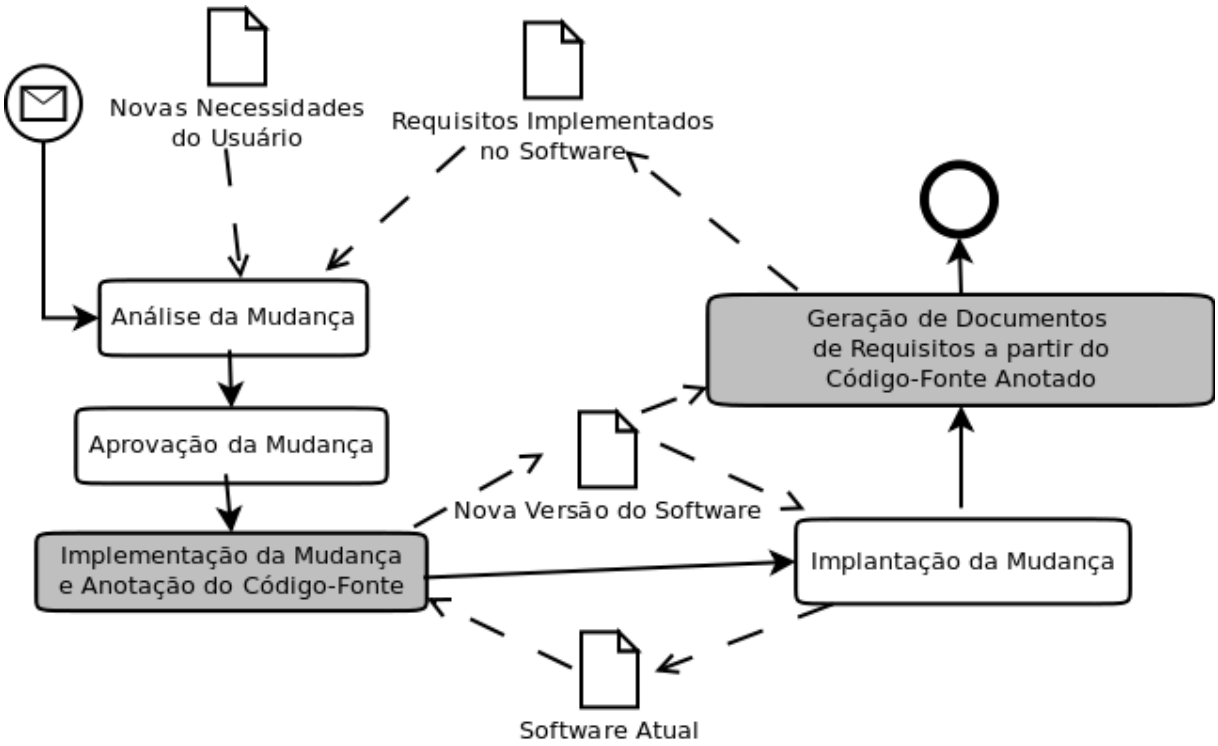

Figura 34 – Modelo de Fahmi et al. (2007) adaptado para a TDRRC

As diferenças entre o modelo de Fahmi et al. (2007) e o modelo adaptado para a TDRRC estão nas duas atividades que tem fundo branco:

- Implementação da mudança e Anotação do Código-fonte: na proposta de Fahmi et al. (2007), esta atividade consistia apenas da implementação da mudança no *software*. Ao aplicar a TDRRC, esta atividade passa a envolver, além da programação, a anotação dos requisitos no código-fonte a ser modificado ou a atualização das anotações já existentes;

- Geração de Documentos de Requisitos a partir do Código Anotado: na proposta de Fahmi et al. (2007) previa-se atualizar os documentos de requisitos a partir de um processo automático de engenharia reversa de requisitos. Ao aplicar a TDRRC, esta atividade é viável e feita através da geração dos documentos de requisitos com um Processador de Anotações.

Através da anotação dos requisitos no código-fonte, torna-se possível obter os Requisitos Implementados no *Software* necessários para a proposta de Fahmi et al. (2007). Em seu trabalho, este passo seria feito pela Engenharia Reversa de Requisitos, um processo que, na sua proposta, seria automático, rápido e que no futuro sua implementação seria viável, mas que até o momento não possuía uma solução.

O modelo modificado viabiliza a proposta de Fahmi et al. (2007) através da aplicação da TDRRC, porém tem as seguintes restrições:

- Os requisitos recuperados podem não ser os requisitos implementados. Diferente do modelo de Fahmi et al. (2007), os requisitos são recuperados a partir das anotações no código-fonte e não do trecho do código-fonte que implementa o requisito. Por este motivo, para que a aplicação seja bem sucedida, é necessário que os desenvolvedores tenham disciplina e cuidado para anotar corretamente o que foi implementado;

- O modelo de Fahmi et al. (2007) tem como pré-condição para a Análise da Mudança a existência de um documento com todos os requisitos implementados no *software* a ser alterado. Isso quer dizer que a TDRRC só viabiliza o modelo de Fahmi et al. (2007) após a anotação de todos os requisitos implementados no código-fonte. Essa anotação pode ser demorada em *software* de médio e grande porte já existente e desnecessária em novos desenvolvimentos, pois não há requisito implementado.

Com a TDRRC, apesar de não ter viabilizado a Engenharia Reversa de Requisitos conforme concebeu Fahmi et al. (2007), oferece-se uma solução viável para que a gestão de requisitos ocorra integrada com um ciclo de geração de requisitos implementados trazendo os mesmos benefícios pretendidos por Fahmi et al. (2007) em sua proposta.

5.4.3 Aplicação da TDRRC em Métodos Ágeis

A TDRRC pode ser uma solução para ter documentos de requisitos em um ambiente onde métodos ágeis são aplicados. A documentação dos requisitos é vista como um processo demorado e de baixa importância pelos métodos ágeis. Prefere-se conversar sobre os requisitos com as partes interessadas e implementá-los diretamente, ao invés de formalizar os requisitos e depois implementá-los.

Ao aplicar a TDRRC neste contexto, realiza-se a documentação dentro da própria atividade de programação, ao invés de realizar um processo de documentação com ciclo de vida próprio. A anotação do código-fonte é feita durante a programação logo após a conversa sobre os requisitos com as partes interessadas.

Quando comparada com o esforço de documentar requisitos dentro de um processo de desenvolvimento tradicional, a atividade está integrada ao ambiente do desenvolvedor. Diferentemente da elaboração de um documento, a documentação durante a programação proposta pela TDRRC consiste da anotação do código-fonte e da descrição dos atributos do requisito definidos no Tipo Anotação.

Desta maneira, a documentação torna-se menos demorada. No entanto, isso não quer dizer que não exija um esforço maior e que ainda não possa ser vista como um processo demorado e de baixa importância por praticantes dos métodos ágeis. Acredita-se que muitos dos praticantes dos métodos ágeis podem considerar o uso da TDRRC benéfico para a produtividade e agilidade, quando verificados os benefícios de se ter a documentação dos requisitos implementados, especialmente em projetos grandes, onde já tenha havido uma certa rotatividade da equipe.

Existe uma variedade de métodos ágeis, mas todos se baseiam em uma série de princípios conhecido como Manifesto Ágil (BECK et al., 2001). Para a aplicação da TDRRC dentro de um método ágil, deve-se considerar estes princípios de forma a não criar um processo muito complicado, o que iria contra a agilidade. A aplicação

da TDRRC, neste contexto, exige uma alteração do método aplicado nos seguintes pontos:

- Captura de requisitos – a captura de requisitos nos métodos ágeis ocorre de maneira bastante informal e geralmente utiliza esquemas feitos de maneira informal em uma lousa ou papel e o usuário é constantemente envolvido em todo o processo. A partir do momento que já exista *software* desenvolvido com anotações, recomenda-se o uso dos requisitos gerados pelo processamento das anotações no código-fonte como base para a discussão dos novos requisitos;

- Documentação de requisitos – a documentação dos requisitos não é feita nos métodos ágeis, pois acredita-se que através da comunicação entre o desenvolvedor e o usuário que demandou o requisito, os requisitos podem ser implementados, sem a necessidade de registrá-los formalmente. Ao aplicar a TDRRC, esta documentação passaria a ser necessária e ficaria a cargo do desenvolvedor que está implementando os requisitos;

- Testes de unidade – juntamente com os testes de unidade é importante validar se o requisito foi corretamente documentado no código-fonte, garantindo assim a qualidade dos documentos de requisitos.

Das alterações propostas nos métodos ágeis, a única a enfrentar mais críticas é a documentação dos requisitos durante a programação. Esse passo realmente aumenta a quantidade de tarefas que o desenvolvedor passa a ter que fazer e pode ser considerada demorada. Entende-se que esta atividade é vantajosa e não muito demorada pelos seguintes motivos:

- O próprio desenvolvedor captura os requisitos da mesma forma que faria seguindo um método ágil;

- Diferentemente da documentação de requisitos tradicional, o desenvolvedor não tem que se preocupar com formatação de texto, cores, fontes, apenas com o conteúdo dos requisitos;

- Através do uso do Processador de Anotações, o desenvolvedor pode usar o Modelo de Requisitos gerado para validar com o cliente o entendimento dos requisitos que estão sendo implementados;

- A documentação de requisitos facilita a compreensão do código-fonte e do propósito do *software*, acelerando a curva de aprendizado caso novos membros entrem na equipe.

Em função destas razões, acredita-se que a TDRRC possa ser aplicada com sucesso em métodos ágeis sem comprometer os seus princípios, trazendo os benefícios de permitir a documentação de requisitos.

5.5 Considerações Finais

O capítulo apresentou o processo de definição e utilização da TDRRC, seus artefatos e suas atividades. Para as atividades relativas ao projeto da TDRRC foi selecionado o meta-modelo de casos de uso de Rui (2007), o qual permitiu ilustrar a definição da TDRRC.

O meta-modelo de Rui (2007) foi escolhido como um exemplo na definição da TDRRC e, portanto, não deve ser considerado como parte desta. A escolha do meta-modelo e os ajustes feitos nele para a utilização na TDRRC, trouxeram contribuições por mostrarem passos necessários para tornar um meta-modelo de requisitos compatível com o mecanismo de anotações e por apresentar um meta-modelo que permite a documentação de casos de uso, uma representação de requisitos bastante utilizada e que, portanto, tem maior chance de ser reutilizada por muitos daqueles que decidirem aplicar a TDRRC após os ajustes mais específicos.

Foi criado também um pseudo código para extração e validação das anotações feitas no programa. Além das validações propostas, outras validações são possíveis, ao se associar o conhecimento da arquitetura do sistema ao Meta-Modelo de Requisitos. Por exemplo, um certo elemento da arquitetura é utilizado para implementar a interface de serviços, tornando a anotação Serviço mandatória neste elemento. Uma validação deste tipo torna-se muito mais complexa, pois exige dar ao Processador de Anotações inteligência para reconhecer elementos da arquitetura.

A TDRRC tem como base o registro dos requisitos no código-fonte. Uma outra opção para registro dos requisitos, seria a criação de um documento de requisitos com ligações de rastreabilidade que poderia oferecer benefício similar. A aplicação da TDRRC tem como vantagem não exigir que a criação de um documento. Com isso, o desenvolvedor não necessita trabalhar em dois lugares (IDE para código-fonte e Processador de Texto para o documento), o que simplifica o seu trabalho. Além disso, o Modelo de Requisitos pode ser desenvolvido de maneira gradual, pois mesmo que o Desenvolvedor não saiba com exatidão como um trecho de código-fonte se encaixa dentro do Meta-Modelo de Requisitos, ele pode anotá-lo com o seu entendimento naquele momento e, posteriormente, com o maior conhecimento, alterar estas anotações, refinando o Modelo de Requisitos.

A documentação de requisitos no código-fonte mostra-se mais vantajosa para ambientes em que os documentos de requisitos praticamente inexistem. A reflexão realizada sobre as possíveis aplicações da TDRRC trouxe à tona possíveis problemas, caso se deseje manter os requisitos em um repositório, principalmente quando se deseje atualizar os requisitos diretamente nele. Neste caso, passa a ser necessária uma ferramenta para sincronizar os requisitos do repositório com os do código-fonte e o custo de implementar esta ferramenta pode não compensar os benefícios trazidos pela TDRRC.

6 ESTUDO DE CASO

A avaliação da viabilidade prática da TDRRC foi realizada através de um estudo de caso. O estudo de caso consiste da aplicação da TDRRC na manutenção e na evolução do SGVD (S*oftware* de Gestão e Vendas de Domínios) de um grande provedor de *Internet*. O desenvolvimento deste *software* foi feito através de métodos ágeis e, para a sua manutenção e evolução, necessitava-se realizar uma reengenharia dos requisitos. Estas condições permitiram avaliar simultaneamente a aplicação da TDRRC em métodos ágeis e na reengenharia de requisitos.

Este capítulo apresenta o estudo de caso e está organizado da seguinte forma:

- Descrição do contexto, que fornece um panorama sobre o estado do SGVD e o processo utilizado para o seu desenvolvimento;
- Aplicação da TDRRC, que descreve o processo adotado para a evolução do SGVD e a aplicação da TDRRC;
- Situações típicas, que mostram benefícios da aplicação da TDRRC na evolução de um *software* com exemplos práticos;
- Análise dos resultados, que consolida os principais resultados obtidos com o estudo de caso.

6.1 DESCRIÇÃO DO CONTEXTO

Antes de iniciar a descrição do estudo de caso, são apresentados os principais conceitos necessários para o seu melhor entendimento, iniciando com provedor de *Internet.*

Um provedor de *Internet* é uma empresa que presta serviços relacionados à *Internet*, entre os quais se incluem: acesso à Internet, contas de e-mail, hospedagem de páginas de *Internet* e venda de domínios.

Um domínio é um nome que serve para localizar e identificar um ou mais servidores na *Internet.* Um servidor é um computador que está conectado à *Internet* e oferece um conjunto de serviços aos usuários que o conectam. Para navegar na *Internet*, o usuário digita um endereço, que pode conter o número IP do servidor ou um domínio. Caso digite o número do IP, a conexão entre o computador do usuário e o servidor é feita diretamente. Caso digite um domínio, a conexão só é feita após a tradução do nome do domínio para um número IP. Esta tradução é feita por um servidor chamado DNS *(Domain Name Server*).

Um DNS funciona de forma hierárquica e distribuída. Ele possui um banco de dados próprio onde é mantida uma tabela de tradução entre nomes de domínio e números de IP. Esta tabela é atualizada diariamente através da consulta de outros DNS. Um DNS só não consulta outro DNS quando ele tem autoridade sobre o domínio.

A autoridade sobre os domínios segue uma hierarquia determinada pelo nome do domínio. Um nome de domínio é formado por um conjunto de caracteres alfanúmericos separados por pontos. A forma como estes nomes são organizados é governada pelo ICANN na hierarquia mais alta.

O ICANN (acrônimo em inglês para Corporação da Internet para Atribuição de Nomes e Números) é uma entidade internacional sem fins lucrativos, responsável pela alocação do espaço de números IP e pela administração do sistema de nomes de domínio de primeiro nível que são delegados a empresas registradoras de primeiro nível.

Uma empresa registradora é aquela que realiza venda e registro de domínios. Caso ela seja uma empresa registradora de primeiro nível, a responsabilidade sobre o registro é dela mesma, caso não seja, ela faz o relacionamento com o cliente e

repassa a solicitação de registro para uma empresa registradora de primeiro nível com qual tenha se credenciado.

Os nomes de domínio de primeiro nível correspondem à última sequência de caracteres de um nome de domínio depois do último ponto e se dividem em nomes genéricos e nomes de país. Os nomes genéricos são: aero, arpa, biz, com, coop, edu, gov, info, int, jus, mil, museum, name, net, org e pro. Os nomes de país são formados por dois caracteres e sua administração é delegada para a empresa registradora de primeiro nível do país em questão. O nome de domínio de primeiro nível para o Brasil é br e o registro.br é a empresa registradora de primeiro nível responsável por ele. Com isso, todos que desejarem ter um domínio terminado com .br; por exemplo, usp.br; devem registrá-lo junto ao registro.br ou à uma empresa registradora que tenha sido credenciada pelo registro.br. Cada empresa registradora de primeiro nível pode criar suas regras e requisitos para o registro dos domínios terminados com os nomes de primeiro nível sobre o qual tem autoridade.

Ao registrar um domínio, o usuário passa a ter autoridade sobre ele e deve manter um DNS com a tabela de tradução entre o nome do domínio e o número de IP. Pode também criar sub-domínios deste domínio e mantê-los neste DNS ou delegá-los a um DNS de nível hierárquico mais baixo. Um sub-domínio é formado por um prefixo seguido de um ponto e do nome do domínio. Por exemplo, poli.usp.br é um sub-domínio de usp.br.

O dono de um domínio é chamado de entidade e pode ser uma pessoa física ou jurídica. Para registrar um domínio no registro.br, é necessário ser uma entidade legalmente representada ou estabelecida no Brasil e deve possuir CNPJ ou CPF.

Os dados necessários para registrar um domínio são:

- Nome da entidade: nome da pessoa física ou jurídica que será proprietária do domínio. O registro.br exige além do nome, o CPF ou CNPJ, o telefone e um endereço de contato no Brasil;

- Contato administrativo: pessoa física ou jurídica (nome, endereço, e-mail e telefone) que será responsável pela administração dos dados do domínio;
- Contato técnico: pessoa física ou jurídica (nome, endereço, e-mail e telefone) que será responsável pela manutenção do DNS do domínio e da conexão de rede;
- Contato financeiro: pessoa física ou jurídica (nome, endereço, e-mail e telefone) que será responsável pelo pagamento do registro e renovação do domínio;
- DNS Primário: Número IP do principal servidor de DNS que terá autoridade sobre o domínio;
- DNS Secundário: Número IP do servidor de DNS que terá autoridade sobre o domínio e será acionado caso o DNS Primário esteja indisponível;
- Quaisquer outras informações que empresa registradora de primeiro nível em questão assim o exigir.

Para que o registro de um domínio seja efetivado é necessário que o domínio não esteja registrado. Deve-se solicitar o registro do mesmo a uma empresa registradora que esteja credenciada para o registro do domínio de primeiro nível desejado, prover os dados mencionados anteriormente e efetuar o pagamento pelo período desejado. Se todas as informações estiverem corretas o domínio é registrado e se torna ativo.

Após o fim do período de registro, o registro pode ser renovado. Caso seja renovado ele permanece ativo. Caso não seja renovado ele é congelado. Um domínio congelado não funciona, pois a empresa registradora revogou a autoridade do DNS do domínio. Porém, não pode ser registrado por uma outra entidade. Caso a atual entidade do domínio efetue o pagamento ele se torna ativo novamente e a autoridade do DNS do domínio é restabelecida.

Após um certo prazo congelado, que varia de 15 a 90 dias, o domínio é cancelado. O prazo para cancelamento depende das regras da entidade registradora. Uma vez cancelado, o domínio se torna disponível para registro por qualquer entidade.

Uma vez apresentados os conceitos envolvidos no registro de domínios, o contexto do estudo de caso é apresentado.

O provedor de *Internet*, procurando diversificar seus negócios, decidiu vender domínios. Visando uma entrada rápida neste mercado, adquiriu várias empresas registradoras de domínios e deparou-se com o desafio de integrar as plataformas tecnológicas destas empresas. Devido à diversidade dos sistemas das empresas adquiridas, não foi possível aproveitar o *software* de nenhuma delas para a operação integrada. Para a acelerar o desenvolvimento de um novo *software* que incorporaria a operação de todas as empresas adquiridas, a empresa definiu o sistema SGVD, constituído pelos módulos conforme a figura 35 e terceirizou o desenvolvimento de cada módulo para uma *software house* diferente.

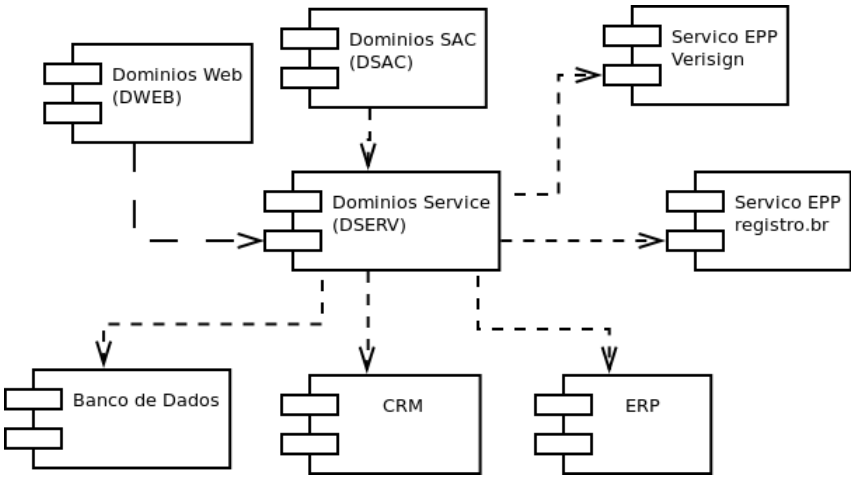

Figura 35 – Visão Geral dos Módulos do SGVD

Os módulos do SGVD são os seguintes:

- CRM *(Customer Relationship Management*): sistema interno para administração do relacionamento com o cliente;

- ERP (*Enterprise Resource Planner*): sistema administrativo responsável pela administração financeira da empresa;
- Serviço EPP *Verisign*: serviço para registro e administração de domínios da empresa *Verisign,* uma empresa registradora de primeiro nível com autoridade para o registro de todos os domínios de primeiro nível genéricos e todos domínios internacionais comercializados pela empresa;
- Serviço EPP registro.br: serviço para registro e administração de domínios da empresa registro.br, uma empresa registradora de primeiro nível com autoridade para o registro de domínios de primeiro nível do Brasil, ou seja, aqueles terminados em .br;
- Banco de Dados: banco de dados *Oracle* responsável por armazenar os dados dos clientes e seus domínios;
- Domínios Service (DSERV): aplicação JEE responsável pela camada de negócios relativas a venda e administração de domínios;
- Domínios Web (DWEB): aplicação web responsável pela interface entre o clientes e o provedor;
- Domínios SAC (DSAC): aplicação web com funcionalidade similar ao DWEB com funções administrativas extras e utilizada pela central de atendimento telefônico para atender solicitações dos clientes.

O foco deste estudo de caso foram os módulos DWEB, DSAC e DSERV e sobre eles foi aplicada a TDRRC. Os outros módulos foram mantidos por outras equipes e não fizeram parte deste estudo de caso.

Os principais casos de uso destes módulos são os seguintes:

- Registrar domínio: permite ao usuário tornar um domínio ativo, fornecendo os dados necessários para o seu registro e efetuando o pagamento;
- Consultar domínios: permite listar todos os domínios registrados por um usuário;
- Renovar domínio: permite a extensão do registro do domínio por um período fixo, mantendo-o ativo;
- Atualizar dados domínio: permite a atualização dos dados de um domínio: endereço, telefone e e-mail da entidade do domínio, contatos administrativo, técnico e financeiro, DNS primário e DNS secundário;
- Cancelar domínio por fraude: faz a solicitação de cancelamento de um domínio por suspeita de fraude junto à empresa registradora. Este caso de uso é exclusivo do DSAC;
- Congelar domínio: permite tornar o domínio congelado, revogando a autoridade do DNS do domínio sobre o mesmo. Esta é um caso de uso exclusivo do DSAC e utilizada em geral para domínios com pagamentos em atraso;
- Descongelar domínio: permite tornar ativo um domínio que estava congelado restabelecendo a autoridade sobre o domínio ao DNS do domínio. É um caso de uso exclusivo do DSAC e utilizado em geral quando um usuário paga a renovação de um domínio que já havia sido congelado.

A estratégia de compra das empresas registradoras foi um sucesso do ponto de vista de negócios, mas gerou um problema de manutenção. Como a prioridade foi o *time to market*, a integração dos módulos, desenvolvidos por empresas diversas, foi

feita sem a preocupação com a arquitetura. Além disso, as equipes responsáveis pelo desenvolvimento original nas empresas adquiridas foram desfeitas e as empresas terceiras contratadas para integração foram dispensadas. O resultado final foi um *software* sem uma estrutura bem definida, mantido por uma equipe que não participou da sua concepção.

O SGVD foi desenvolvido utilizando a linguagem JAVA e ferramentas de código aberto, dentre as quais *Hibernate, Spring, Velocity, Maven* e *JUnit.* Muitos dos recursos da tecnologia JEE foram utilizados, entre eles: EJB, JMS, *Servlets,* JSP, RMI e *WebServices.* O uso destas tecnologias foi feita de maneira inadequada, pois não se seguiu nenhuma padronização e, em muitos casos, estes recursos eram desnecessários, pois não havia nenhum requisito do *software* que exigisse a necessidade de várias das tecnologias em questão. Desta forma, a profusão de bibliotecas, ferramentas e tecnologias tornou a manutenção do sistema bastante difícil. Devido à falta de conhecimentos básicos de orientação a objetos e às dificuldades na integração dos diferentes módulos, os desenvolvedores iniciais do *software* criaram classes com milhares de linhas de código que não representavam nenhuma entidade ou abstração. O sistema possuia 1.525 classes e 332.436 linhas de código no momento da avaliação inicial.

O processo de desenvolvimento foi baseado no método ágil *Scrum* e dividido em iterações chamadas de *sprints.* No início de cada *sprint,* era feita uma reunião de planejamento na qual era decidido o que seria implementado e estimado o esforço para esta implementação. O cálculo do esforço era feito a partir de estimativas da complexidade de cada uma das tarefas e medido em pontos de *Scrum*[1]. No final de cada *sprint* era feita a retrospectiva, uma reunião para avaliação dos resultados do

[1] A medida em pontos baseia-se em um escala de dificuldade definida pelos participantes da reunião de planejamento. Serve como apoio a discussão sobre a dificuldade de uma tarefa sem estimá-la em horas de desenvolvimento.

sprint passado. Durante a retrospectiva, era comparado o que foi planejado e o que foi executado e analisadas as causas do sucesso ou das falhas.

A captura dos requisitos foi completamente informal e o seu registro não foi mantido. Os testes de unidade, apesar de preconizados pelos métodos ágeis não foram a prática durante o desenvolvimento. À medida que o SGVD foi evoluindo, a arquitetura que já não era boa no início deteriorou-se rapidamente, tornando a menor das modificações uma tarefa enorme de programação.

Do ponto de vista de ciclo de vida do sistema, embora existissem vários argumentos para descontinuá-lo, não existia, naquele momento, um substituto viável. O SGVD era crítico para a empresa, pois através dele eram realizadas as vendas de domínio, responsáveis por uma fatia importante do faturamento da empresa. A falta de um substituto se devia à dificuldade em determinar o que deveria ser implementado, pois existiam muitas regras de negócio implementadas e não documentadas no SGVD. A descoberta dos requisitos implementados pelo SGVD era fundamental para que se pudesse desenvolver um substituto com uma arquitetura melhor e, assim, poder descontinuá-lo. Ao mesmo tempo, em função do seu uso como único canal de vendas de domínios, novos requisitos e regras de negócio eram constantemente demandados para mantê-lo bem posicionado frente às necessidades do mercado e à concorrência. Enquanto o SGVD não era descontinuado, a empresa tinha que arcar com altos custos de manutenção. Devido às contínuas demandas da área de negócios, o SGVD era constantemente modificado, tornando-se cada vez mais difícil de manter e com uma quantidade de requisitos implementados e não documentados cada vez maior.

Em função deste cenário pode-se dizer que o SGVD apresentava todas as características desejáveis para se aplicar a TDRRC:

- Apresentava alto custo de manutenção;

- Não se podia substituí-lo por causa do não conhecimento dos requisitos nele implementados;
- A única fonte confiável para se determinar os requisitos era o código-fonte;
- O código-fonte era complexo e difícil de ser entendido;
- O sistema foi implementado em JAVA que possui suporte para anotações.

6.2 APLICAÇÃO DA TDRRC

A equipe de desenvolvimento que aplicou a TDRRC era responsável pela manutenção e evolução dos seguintes módulos do SGVD:

- Domínios Service (DSERV): aplicação JEE responsável pela camada de negócios relativas a venda e administração de domínios;
- Domínios Web (DWEB): aplicação web responsável pela interface entre o cliente e o provedor;
- Domínios SAC (DSAC): aplicação web com funcionalidade similar ao DWEB com funções administrativas extras e utilizada pela central de atendimento telefônico para atender solicitações do clientes.

A aplicação da TDRRC ocorreu depois de três meses do início do trabalho. Neste período já haviam ocorrido quatro *sprints* onde foram entregues 35, 37, 40 e 38 pontos de *Scrum* em cada um deles.

Para se aplicar a TDRRC, inicialmente foi realizada uma mudança no processo de desenvolvimento. A equipe de desenvolvedores tinha apenas duas pessoas, sendo um deles o autor desta tese, o que tornou o treinamento mais fácil.

O processo de desenvolvimento baseado em *Scrum* mudou nos seguintes aspectos:

- No planejamento do s*print*, passou a haver uma maior preocupação em formalizar os requisitos, seguindo o meta-modelo de casos de uso definido pela TDRRC e apresentado na seção 5.3.1;
- Os desenvolvedores passaram ser responsáveis por anotar, da forma mais correta e completa possível, todo o código-fonte que fosse alterado por eles;
- Junto com o processo de compilação, o processador de anotações era executado e indicava os erros encontrados nas validações das anotações, os quais eram corrigidos pelos desenvolvedores;
- Uma revisão manual dos casos de uso gerados era feita no final do *sprint.*

A figura 36 mostra o processo de desenvolvimento baseado em *Scrum* modificado para a aplicação da TDRRC.

A Criação das Demandas é feita pelo Membro da Área de Negócios que registra as necessidades de alterações no SGVD em uma Lista de Demandas. Com esta lista, é feita a Priorização das Demandas pelo *Scrum Master* (Gerente de Projeto) através de conversas com os Membros da Área de Negócios.

O Planejamento do *Sprint* é feito pela Equipe (Desenvolvedores e *Scrum Master*) em uma reunião de planejamento. Durante esta reunião,.as demandas são detalhadas e o esforço para o seu desenvolvimento estimado em pontos de *Scrum*. A Equipe estima quantos pontos de *Scrum* deverá conseguir entregar durante o *sprint* e, assim, decide-se quais demandas serão implementadas. Antes da aplicação da TDRRC, a única informação utilizada para a estimativa do esforço era o conhecimento do membros da equipe e a lista de demandas. Com a aplicação da TDRRC, a Equipe passou a utilizar também o Modelo de Requisitos gerado pelo processador de anotações. Isto ocorreu depois do segundo *sprint* em que a TDRRC

foi aplicada, pois no primeiro sprint, o código-fonte ainda não estava anotado e não se podia gerar um Modelo de Requisitos. O Modelo de Requisitos se mostrou uma importante fonte de informação para entender melhor o que deveria ser feito e o seu impacto.

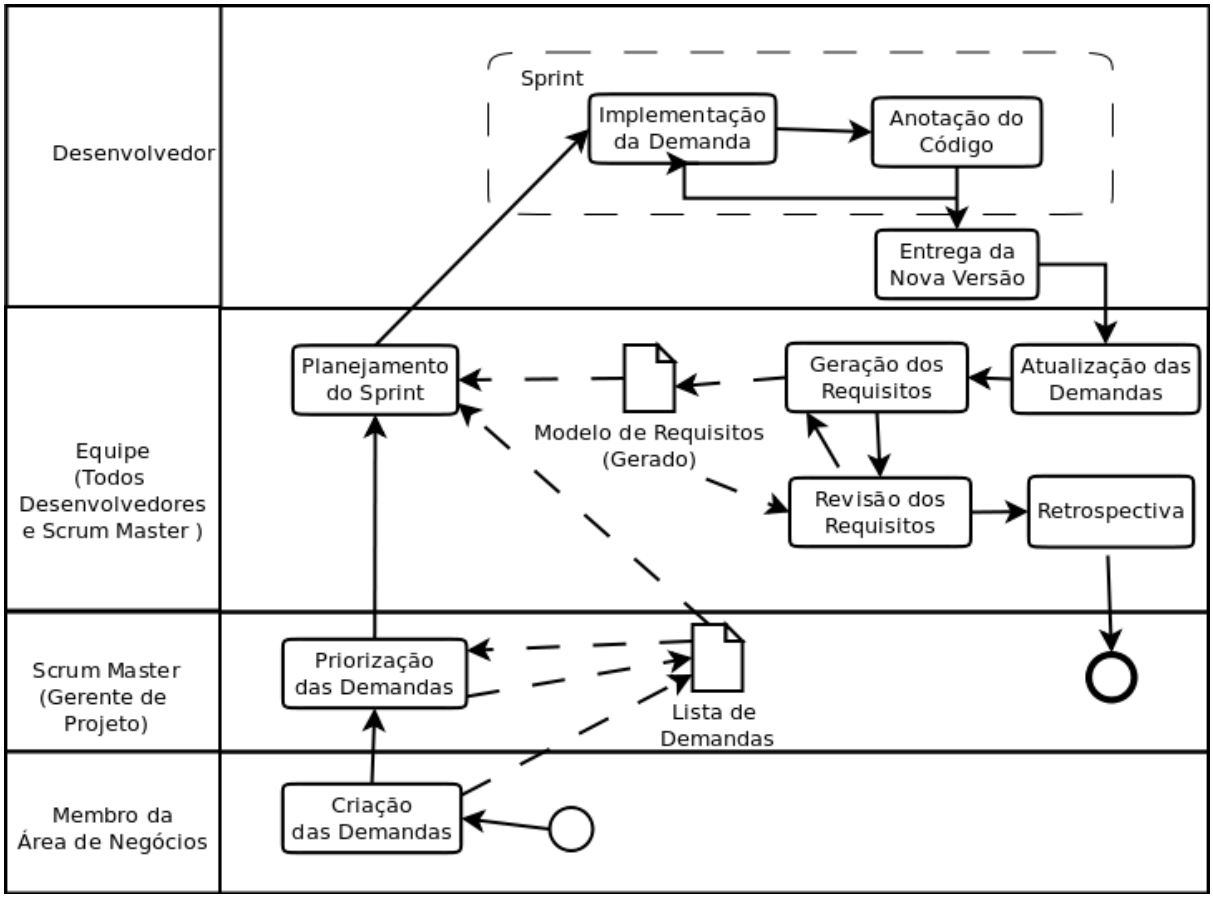

Figura 36 – Processo de desenvolvimento com aplicação da TDRRC

Uma vez terminado o planejamento, inicia-se o *sprint*. Durante o *sprint*, ocorre a Implementação da Demanda, onde os Desenvolvedores pesquisam o código-fonte, esclarecem as dúvidas com a equipe de negócios, programam as alterações para atender a demanda e fazem testes para garantir o seu funcionamento. Em seguida, ou, às vezes, simultaneamente, em função da preferência pessoal, ocorre a Anotação do Código, em que os Desenvolvedores anotam o código-fonte com os requisitos que foram implementados. Durante a Implementação da Demanda, o

modelo de requisitos gerado pelo processador de anotações a passou dar apoio na compreensão do código-fonte, auxiliando no entendimento da demanda e na localização do trecho de código-fonte que seria modificado. Este ciclo se repete até que todas as demandas sejam implementadas ou o prazo de duração do *sprint* termine.

Com o fim do *sprint,* ocorre a Entrega da Nova Versão, em que os Desenvolvedores entregam uma nova versão dos módulos alterados, que são instalados no servidor e colocados em operação. A equipe se reune para realizar a Atualização das Demandas, de acordo com o que foi implementado. É feita uma Geração dos Requisitos a partir do código-fonte anotado e com este documento (Modelo de Requisitos), é feita uma Revisão dos Requisitos. Os problemas encontrados nesta revisão são corrigidos e uma nova Geração de Requisitos é realizada, seguida de uma nova Revisão dos Requisitos e assim, sucessivamente até que o Modelo de Requisitos reflita de maneira correta os requisitos implementados no SGVD.

Em seguida, ocorre a Retrospectiva, em que a Equipe se reúne para comparar o que foi planejado e o que foi executado e analisar as causas do sucesso ou das falhas.

6.3 Situações Típicas

A aplicação da TDRRC foi de grande valia em situações que ocorrem com frequência durante a manutenção e evolução de um *software*. As situações descritas a seguir não esgotam todas as possibilidades, mas representam situações comuns onde o benefício da TDRRC é significativo:

- Descoberta de requisitos implementados em um *software*;
- Refatoração;
- Identificação de clone;
- Detecção de requisitos inconsistentes;

- Análise de impacto de uma modificação.

Para facilitar o entendimento, para cada uma das situações citadas são apresentados exemplos que ocorreram durante o estudo de caso.

6.3.1 Descoberta de requisitos

A TDRRC foi concebida também para ser uma ferramenta de apoio à engenharia reversa de requisitos. Dessa forma, ao aplicar a TDRRC em conjunto com técnicas de engenharia reversa de requisitos baseadas na inspeção de código-fonte, torna-se possível documentar no código-fonte os requisitos que vão sendo descobertos de maneira gradativa. Isso acaba sendo vantajoso, pois permite a construção gradual de um mapeamento entre o código-fonte e os requisitos, que é feito através das anotações.

No início do envolvimento de um desenvolvedor com um *software*, por não conhecê-lo bem, ele consome muitas horas com a leitura e compreensão do código-fonte. Para evoluir o *software*, o desenvolvedor precisa entender o que o *software* faz e onde as suas funções estão implementadas. Quanto maior o porte e a complexidade do *software*, maior o número de horas consumidas pelo desenvolvedor. Além disso, o conhecimento adquirido por um desenvolvedor não é reaproveitado por outros.

Ainda, o próprio desenvolvedor que realizou a investigação do código-fonte, com o tempo, pode acabar esquecendo o que descobriu. Com a aplicação da TDRRC, o conhecimento adquirido pelo desenvolvedor é armazenado e compartilhado através das anotações no código-fonte.

Um exemplo desta situação, ocorrida durante o estudo de caso, é descrita a seguir. Este exemplo foi escolhido por ter ocorrido em uma das primeiras demandas em que a TDRRC foi aplicada e por retratar a percepção de aumento gradual do conhecimento dos requisitos.

A demanda recebida da área de negócios se refere ao registro de renovações feitas pela central de atendimento. A descrição da demanda está na figura 37 e escrita exatamente como foi feita pela área de negócios.

Título: Registro de renovações feitas na Central de Atendimento.
Descrição: Ao comprar um novo domínio pela central, os dados da compra aparecem no painel de eventos do usuário, mas o mesmo não ocorre na renovação.

Figura 37 – Demanda de registro de eventos para renovação de domínios

Como pode ser observado neste exemplo, a descrição das demandas era vaga e exigia um grande conhecimento do negócio e do SGVD para implementá-la. A sugestão do gerente do projeto, foi que o desenvolvedor conversasse com uma pessoa que já tinha trabalhado n*o* SGVD. Esta pessoa relatou que os dados da compra citados pela demanda ficavam guardados em uma tabela que tinha um nome estranho, mas que ela não se lembrava.

Através de esclarecimentos com diversas pessoas, o desenvolvedor descobriu que na aplicação DSAC existe uma tela com informações de operações realizadas pela central de atendimento para um dado usuário. Esta tela é denominada painel de eventos e nela deveria constar todas as operações de compras e renovações de domínio que o usuário fez pela central de atendimento e quem foi o atendente responsável pela operação. Ao fazer uma compra de domínios utilizando o DSAC a operação aparecia no painel de eventos, mas ao fazer a renovação isto não ocorria.

Decidiu-se iniciar a investigação do código-fonte pelo DSAC, comparando os trechos de código referentes à compra e à renovação de domínios por operadores da central de atendimento. Acompanhando a execução e seguindo o fluxo de compra e renovação, foi possível identificar métodos responsáveis por diversos passos de um único caso de uso que tinha o nome de registro de domínios. Cada um dos passos

recebeu uma anotação de episódio e foi possível gerar a documentação apresentada na figura 38.

```
Caso de Uso: Registro de Domínios
Atores: Cliente, Atendente
Episódios:
1 - Valida a assinatura do cliente
2 - Remove domínios duplicados
3 - Valida o domínio
4 - Encontra a entidade vinculada ao domínio
5 - Verifica o dono do domínio
6 - Verifica disponibilidade do domínio
7 - Registra o domínio
```

Figura 38 – Caso de uso registro de domínios gerado pelas anotações

Durante esta investigação não foi possível identificar o trecho de código-fonte que era responsável por registrar a operação para que ela ficasse visível no painel de eventos. Resolveu-se então fazer a investigação ao contrário, e, a partir da tela que mostrava os eventos de usuário, determinou-se qual tabela esta tela consultava. Fazendo uma busca textual em todo código-fonte pelo nome da tabela, ou seja, buscando todos os arquivos que contivessem dentro do seu texto o nome da tabela, encontraram-se 34 ocorrências, que representavam potenciais trechos de código-fonte que manipulavam dados nesta tabela. Através da investigação de cada uma destas ocorrências, encontraram-se 2 métodos que salvavam dados nesta tabela. Fazendo uma nova busca textual para cada um destes métodos e investigando cada um dos lugares onde o método é chamado, encontrou-se, finalmente, onde e como este registro era feito na compra de domínios. Durante a investigação, mais conhecimento sobre o *software* foi adquirido e novas anotações foram feitas; algumas que tinha sido feitas anteriormente tiveram seu texto reescrito para refletir melhor este novo conhecimento, pois nem todos os passos que foram anotados como episódio realmente tinha significado do ponto de vista do usuário. As figuras

39 e 40 mostram os casos de uso de compra e renovação após o final da investigação.

```
Caso de Uso: Compra de Domínios
Atores: Cliente, Atendente
Episódios:
1 - Verifica se a assinatura está ativa e com pagamentos em dia
2 - Remove domínios duplicados
3 - Verifica se o nome do domínio é válido
4 - Encontra a entidade vinculada ao domínio, caso não exista cria uma
entidade
5 - Verifica disponibilidade do domínio
6 - Cria o domínio no orgão registrador
7 - Registra a despesa na conta do cliente
8 - Avisa o cliente que o domínio foi criado
9 - Se o ator é um atendente registra os dados e o responsável pela
operação
```

Figura 39 – Casos de uso de compra de domínios gerado pelas anotações

```
Caso de Uso: Renovação de Domínios
Atores: Cliente, Atendente
Episódios:
1 - Verifica se a assinatura está ativa e com pagamentos em dia
2 - Remove domínios duplicados
3 - Verifica se o nome do domínio é válido
4 - Verifica se o cliente é o dono do domínio
5 - Renova o domínio no orgão registrador
6 - Registra a despesa na conta do cliente
7 - Avisa o cliente que o domínio foi criado
```

Figura 40 – Casos de uso de renovação de domínios gerado pelas anotações

As diferenças nestas anotações podem ser vistas comparando a figura 38 com as figuras 39 e 40 e refletem o aumento do conhecimento sobre o *software*:

- Na figura 38, o caso de uso se chama “Registro de Domínio”, pois o método analisado se chama registrarDominio. Com a análise mais detalhada percebeu-se que dois casos de uso são implementados por este método: “Compra de Domínios” e “Renovação de Domínios”;

- Na figura 38, o primeiro episódio diz apenas que há uma validação da assinatura do cliente. Com a análise mais detalhada foi possível identificar quais validações são feitas. Isso está refletido no primeiro episódio das figuras 39 e 40;

- Na figura 38, o terceiro episódio diz apenas que há uma validação do domínio. Com a análise mais detalhada foi possível identificar que a validação é feita apenas no nome do domínio. Isso está refletido no terceiro episódio das figuras 39 e 40;

- Na figura 38, o quarto episódio diz apenas que o *software* encontra a entidade vinculada ao domínio. Com a análise mais detalhada foi possível identificar que, caso a entidade não seja encontrada, o *software* cria uma nova entidade e que esse episódio se aplica apenas à compra de domínios. Isso está refletido no quarto episódio da figura 39;

- Na figura 38, o quinto episódio diz apenas que o *software* verifica o dono do domínio. Com a análise mais detalhada foi possível identificar que a verificação serve para identificar se o cliente é o dono do domínio e, portanto, pode renová-lo. Isso está refletido no quarto episódio da figura 40;

- Na figura 38, o sexto episódio não foi alterado, mas percebeu-se que se aplica apenas à compra de domínios. Ele está representado no quinto episódio da figura 39;

- Na figura 38, o sétimo episódio foi detalhado e se transformou nos episódios seis a nove da figura 39 e nos episódios cinco, seis e sete da figura 40.

Ao comparar os casos de uso “Compra de Domínios” da figura 39 e “Renovação de Domínios” da figura 40 e confrontá-los com a demanda (figura 37), pode-se notar que o episódio nove da compra de domínios não existe na renovação de domínios. A demanda é justamente implementar este episódio também para a renovação de domínios, fazendo com que o registro da operação do atendente também ocorra na renovação de domínios.

Com a aplicação da TDRRC, o conhecimento adquirido como resultado da investigação passou a não ficar apenas na mente do desenvolvedor que acaba esquecendo com o tempo. Ele fica no código-fonte e armazenado no sistema de controle de versões e fica disponível para consulta por todos que tenham acesso ao código-fonte e ao processador de anotações. O uso das anotações durante a investigação aumentou o trabalho de programação, mas não representou um esforço significativo a mais, pois o desenvolvedor levava menos de um minuto para escrever cada anotação e esta acabava auxiliando-o a se lembrar que trecho de código-fonte já tinha lido e o que ele faz, economizando tempo em uma investigação posterior.

6.3.2 Refatoração

Muitas vezes o código-fonte necessita ser refatorado, mas por estar habituado com o mesmo, o desenvolvedor acaba não percebendo a necessidade. A aplicação da TDRRC pode levar o desenvolvedor a fazer uma refatoração do código-fonte, pois durante a investigação do código-fonte, ele verifica que as anotações não podem ser feitas devido à organização do código-fonte. Há duas maneiras de resolver este problema: a primeira é alterar o meta-modelo de requisitos e os tipos anotação e a segunda é refatorar o código-fonte. Assumindo que o meta-modelo foi definido por um especialista e verificado, a incompatibilidade entre o meta-modelo e o código-fonte é um forte indicativo da necessidade de refatoração.

Existem diversas técnicas de refatoração disponíveis e a sua utilização, orientada por um critério de organização, permite que o código-fonte fique mais fácil de ser entendido e mantido. Ao refatorar o código-fonte para torná-lo compatível com o meta-modelo de requisitos, está se usando o meta-modelo de requisitos como um dos critérios para organização do código-fonte.

Durante o estudo de caso, esta situação ocorreu diversas vezes e, em todas as vezes, estava claro que o código-fonte necessitava ser refatorado. A figura 41 mostra a ocorrência mais comum no estudo de caso, em que métodos eram excessivamente grandes e implementavam todo o caso de uso.

```
protected  void freezeDomainImpl(Integer idtDomain, String reason) throws
Exception {
      Transaction transaction = null;
      try {
            transaction = getNewTransaction();
            transaction.begin();
            DomainRegistryEntity domain =
ServiceLocator.daoFactory().getDomainRegistryDAO(getEntityManager()).find(
idtDomain);
            vFrozenLogger.debug("[] Processing idtDomain: " +
domain.getIdt());
            vFrozenLogger.debug("Original object state : " + domain);
            PersonVO personVO =
dasFacade.getPersonVO(domain.getIdtInscription());
            Integer accountId =
dasFacade.getUserAccountId(domain.getIdtInscription());
            domain.setCodDomainRegistryStatus("B");
            transaction.getEntityManager().merge(domain);
            transaction.getEntityManager().flush();
            EppWsFacade eppClient = ServiceLocator.eppWsFacade();
            List<Dns>   dnsListToRemove =
AssembleTools.assembleDnsHostList(Arrays.asList(eppClient.infoDomain(domai
n).getHostList()));
…........... continua com mais código nesse estilo por 600 linhas.
```

Figura 41 – Primeiras linhas do método de congelamento de domínio antes da refatoração

Conforme pode ser visto na figura 41, é bastante difícil entender o que este trecho de código-fonte faz e o método completo tem mais de 600 linhas com esse mesmo

nível de obscuridade. A prática de métodos muito longos é considerada um erro na orientação a objetos e dificulta a manutenção e evolução do *software*.

Como o método continha todo o caso de uso e o meta-modelo diz que um caso de uso é composto de episódios, foi necessário particionar o método em métodos menores que representavam os episódios do caso de uso. O resultado final foi um código-fonte muito mais claro e fácil de manter que está representado na figura 42.

```
protected  void freezeDomainImpl(Integer idtDomain, String reason) throws
Exception {
      vFrozenLogger.debug("[] Freezing idtDomain: " + idtDomain);
      changeDomainStatus(idtDomain, DomainStatus.FROZEN);
      List<Dns> dnsListToRemove = getCurrentDNS(domain);
      saveDNSRecords(idtDomain, dnsListToRemove);
      notifyUserFrozenDomain(idtDomain);
      List<Dns> dnsListToAdd = createFrozenDNSRecords();
      updateDomainDNS(idtDomain, dnsListToRemove, dnsListToAdd);
}

@Episodio(
   casosDeUso = ({“Congelar Dominio”, “Descongelar Dominio”}),
   atores = ({“Atendente”}),
   descricao = “Recupera os registros de dns atuais do domínio”
)
private List<Dns> getCurrentDNS(Integer domain) throws EppDomainException
{
      List<Dns> dnsListToRemove  = null;
      try {
            EppWsFacade eppClient = ServiceLocator.eppWsFacade();
            dnsListToRemove =
eppClient.infoDomain(idtDomain).getHostList()));
      } catch (Exception e) {
            throw new EppDomainException(e.getMessage());
      }
      return dnsListToRemove;
}
```

Figura 42 – Método de congelamento de domínio após refatoração e um episódio anotado

Confrontando a figura 42 com a figura 41, pode-se perceber que o método freezeDomainImpl tinha mais de 600 linhas e passou a ter apenas sete linhas com sete chamadas de métodos que implementam os passos do congelamento de domínios. Cada um destes métodos ainda estava grande (média de 90 linhas), mas

o código ficou bem mais claro. Pelo nome dos métodos chamados em cada linha, uma pessoa que conhece o domínio do problema consegue entender o que o código-fonte faz. Se ainda assim restarem dúvidas, ao navegar nos métodos, o desenvolvedor pode ler as anotações do episódio e entender melhor o que o método faz e representa como se pode ver no método getCurrentDNS que está anotado e presente na figura 42.

6.3.3 Identificação de clone

Clones são trechos de código-fonte que fazem a mesma coisa ou praticamente a mesma coisa e poderiam ser substituídos por um único trecho de código e aumentar, com isso, o reuso dentro de um *software* (ZARZABEK, 2007). A TDRRC não pretende ter, como meta principal, a identificação de clones e existem várias ferramentas e técnicas específicas para este propósito. Entretanto, verificou-se que, ao permitir uma visão de mais alto nível de abstração do código-fonte, a TDRRC pode ser útil na identificação de clones. Durante a leitura da documentação gerada pelo processador de anotações, várias vezes identificaram-se requisitos com partes muito parecidas e que, portanto, poderiam ter sido implementadas reutilizando o mesmo código.

No estudo de caso, esta situação ocorreu com uma razoável frequência, pois ao final de cada s*print,* o modelo de requisitos era gerado e revisado em conjunto. Nota-se que, ao anotar o código-fonte, descreve-se a função dos métodos e estes são classificados como episódios dentro do contexto dos casos de uso. Ao gerar o modelo de requisitos com o processador de anotações, torna-se possível visualizar episódios com descrição muito parecida.

Durante o estudo de caso, ao deparar-se com esta situação, foi realizada uma investigação mais detalhada dos métodos que descreviam episódios semelhantes e descobriu-se que, na maioria das vezes, ambos métodos faziam praticamente a mesma coisa, ou eram muito parecidos, podendo ser adaptados e unidos em um só

método. Ao eliminar esta redundância, centenas de linhas de código foram eliminadas do SGVD, diminuindo sua complexidade, aumentando o reuso e tornando sua manutenção mais fácil.

6.3.4 Detecção de requisitos inconsistentes

A aplicação da TDRRC trouxe benefícios também na detecção de requisitos inconsistentes. Esta situação ocorre com frequência no desenvolvimento de *software* e tende a ocorrer mais em sistemas mais complexos e em estágios avançados do ciclo de vida. Para evitar que isso aconteça, é muito importante conhecer os requisitos que estão implementados (CANFORA, 2007; FAHMI ET AL., 2007; YU, 2005).

No estudo de caso, esta situação ficou evidente com a aplicação da TDRRC. No processo de desenvolvimento original da empresa, quando os membros da área de negócios demandavam alterações no SGVD, as demandas eram priorizadas em uma reunião com o gerente de projeto e um conjunto delas era implementado em um s*print.* O detalhamento e implementação das demandas eram realizados sem uma preocupação com a consistência dos requisitos, pois assumia-se que os membros da área de negócios, por serem responsáveis pelo negócio, eram os responsáveis por isso. Na prática, a equipe de desenvolvimento sabia que isso não era uma verdade, mas somente com a aplicação da TDRRC e a visualização dos requisitos implementados, tornou-se possível detectar as inconsistências e evitar a sua implementação. A equipe passou a fazer a análise dos requisitos novos, confrontando-os com os requisitos implementados durante o planejamento. Se inconsistências fossem encontradas, o problema era discutido com os responsáveis pela demanda, buscando-se uma solução.

A seguir, apresenta-se uma nova demanda (figura 43), inconsistente com um requisito pré existente (figura 44).

Título: Criação de domínios sem servidores de DNS

Descrição: Permitir a criação de domínios sem a informação de servidores de DNS para que o usuário possa comprar o domínio de forma menos burocrática.

Figura 43 – Demanda da promoção novos domínios

```
Caso de Uso: Compra de domínios

Pré-Condições:

1 – O dono do domínio tem um CPF ou CNPJ válido e ativo na SRF

2 – Existem servidores de DNS que respondem por este domínio

3 – A empresa possui saldo suficiente junto à empresas registradora de
primeiro nível
```

Figura 44 – Pré-condições do caso de uso Compra de Domínios

A inconsistência está na segunda pré-condição da compra de domínio. Para comprar um domínio, mais especificamente para registrar um domínio junto à uma empresa registradora, é necessário informar servidores de DNS que respondam por este domínio. A demanda de permitir a criação de domínios sem esta informação é inconsistente com os requisitos da empresa registradora.

O exemplo ajudou a mostrar a importância de se ter o modelo de requisitos, pois, sem ele, este problema só seria descoberto nos testes. A solução encontrada foi a implantação de um servidor de DNS temporário que responderia pelo domínio a ser criado de maneira provisória. Ao solicitar a criação de uma domínio à empresa registradora, o SGVD passou a informar este servidor de DNS temporário caso o usuário não tivesse fornecido esta informação. Desta maneira, foi possível atender a demanda da área de negócios e os requisitos da empresa registradora.

6.3.5 Análise de Impacto

Com a aplicação da TDRRC, passou a ser possível ter nos documentos de requisitos a indicação do local de implementação de requisitos; como os requisitos estão anotados no código-fonte a rastreabilidade é mantida. Entre as informações disponíveis ao processador de anotações estão o nome do arquivo, nome da classe e número da linha de código-fonte onde a anotação foi feita. Com isso, pode-se gerar junto com os requisitos, a informação do trecho do código-fonte em que aquele elemento do requisito está implementado. A figura 45 exemplifica como um caso de uso pode ser apresentado com informação de rastreabilidade para o código-fonte, pois, para cada episódio, o nome do arquivo e número da linha em que o episódio está implementado são apresentados.

A informação de rastreabilidade para o código-fonte pode ser usado em uma situação bastante comum na manutenção de um *software* que é a análise de impacto. Desta maneira pode-se discutir as alterações nos requisitos em um nível de abstração elevado e depois, com a ajuda do mapeamento entre o requisito e o código-fonte, analisar através do código-fonte qual seria o impacto da modificação.

Um exemplo disto foi a demanda de registro de domínios gratuito pelo atendente da central de atendimento. Na reunião de planejamento foi discutido como implementar e tinha sido sugerido aproveitar o fluxo de compra de domínios, permitindo escolher entre pagar pelo domínio e registrar sem pagar no final deste fluxo. Em princípio, parecia muito simples, pois, vendo esta descrição, tratava-se apenas de acrescentar um botão na última tela de venda de domínios e, através deste botão, mandar registrar sem fazer cobrança. Ao validar esta ideia, revendo o caso de uso de venda de domínios, chegou-se à conclusão que embora viável, o impacto desta mudança seria bem maior do que o esperado. Isto não foi possível descobrir apenas lendo o caso de uso que está na figura 45, mas lendo o código-fonte nas linhas indicadas e vendo como ele foi implementado. Verificou-se que, esta solução precisaria muito mais do que adicionar uma condição para verificar se seria uma compra ou registro

gratuito, pois esta informação era utilizada constantemente em diversos pontos do código-fonte e teria que ser passada como argumento em todos os métodos envolvidos neste caso de uso. Além disso, alguns destes métodos eram utilizados em outros casos de uso e suas chamadas deveriam ser alteradas para permitir a inclusão do novo parâmetro. Com isso, no planejamento, alocou-se um número maior de horas para a implementação desta demanda e evitou-se atrasos e problemas na entrega do *sprint*.

Caso de Uso: Compra de Domínios Atores: Cliente, Atendente Episódios:	
1 - Verifica se a assinatura está ativa e com pagamentos em dia	PersistenceService.java linha 123
2 - Remove domínios duplicados	DomainRegisterBusinessImpl.java linha 354
3 - Verifica se o nome do domínio é válido	CustomValidator.java linha 659
4 - Encontra a entidade vinculada ao domínio, caso não exista cria uma entidade	EntityEssential.java linha 43
5 - Verifica disponibilidade do domínio	DomainBusinessTools.java linha 2726
6 - Cria o domínio no órgão registrador	DomainRegistrationProcessImpl.java linha 681
7 - Registra a despesa na conta do usuário	AccountManagerFacade.java linha 115
8 - Avisa o usuário que o domínio foi criado	NotifierFacadeImpl.java linha 61
9 - Se o ator é um atendente registra os dados e o responsável pela operação	SessionLogServiceImpl.java linha 84

Figura 45 – Casos de uso de compra de domínios com informação de rastreabilidade

Como se pode verificar, a TDRRC permite a análise de impacto com uma maior eficiência, pois permite colocar a informação de rastreabilidade no caso uso, ou seja, de onde veio a anotação e, consequentemente, onde foi implementado o requisito.

Com isso, perde-se menos tempo procurando os trechos de código-fonte que implementam o requisito para analisar o impacto de uma mudança.

6.4 Análise dos Resultados Obtidos

A TDRRC foi aplicada durante seis meses na evolução deste sistema. Durante este período, ocorreram oito *sprints* de aproximadamente três semanas cada um. A produtividade foi medida em pontos de *Scrum,* do mesmo jeito que já era medida anteriormente como parte do método *Scrum*. Com isso foi possível comparar a produtividade antes e depois da aplicação da TDRRC.

A tabela 4 apresenta os resultados obtidos nos *sprints* anteriores e durante o estudo de caso. A a equipe que aplicou a TDRRC começou a participar da evolução do SGVD no *sprint* um e a TDRRC começou a ser aplicada no *sprint* cinco. Nestes *sprints,* o número de classes do sistemas não era conhecido, pois não havia a preocupação com esta métrica e por isso foi colocado na tabela a sigla NC - “Não conhecido”, analogamente como a TDRRC ainda não estava sendo aplicada usou-se NA - “Não se aplica” para o número de classes anotadas.

Tabela 4 - Pontos de *Scrum* entregues nos *sprints*

Sprint	1	2	3	4	5	6	7	8	9	10	11	12
Pontos Entregues	35	37	40	38	18	17	27	37	48	55	53	54
Número de Classes Anotadas	NA	NA	NA	NA	5	25	51	83	132	203	276	343
Classes do Sistema	NC	NC	NC	NC	1525	1541	1602	1593	1637	1645	1627	1634

Durante os *sprints* cinco, seis e sete, após a aplicação da TDRRC, as seguintes dificuldades foram enfrentadas:

- Apesar da equipe já ter um entendimento razoável do código-fonte que estava sendo modificado, houve bastante dificuldade em identificar os elementos que deveriam ser anotados;
- A tradução do código de validação para uma mensagem em alto nível, que deveria ser escrita na anotação, muitas vezes não foi uma tarefa trivial. Muitas das anotações feitas apenas traduziam uma lógica de programação para português;
- O texto das anotações trouxe dificuldades, pois dependia de um bom entendimento do negócio e da granularidade desejada.

O impacto destas dificuldades ficou visível na produtividade do *sprint*. Enquanto, nos *sprints* de um à quatro, foram entregues em média 38 pontos, no *sprint* cinco, foram entregues 18 pontos, menos da metade dos pontos que se costumava entregar. Estes problemas foram percebidos durante a revisão dos requisitos feita no final do *sprint* e discutido na reunião de retrospectiva. Os erros detectados nas anotações foram corrigidos e discutidos com toda a equipe para evitar que voltassem a ocorrer.

No *sprint* seis, houve uma melhora significativa na qualidade das anotações, mas a aplicação da TDRRC ainda estava impactando a produtividade e apenas 17 pontos foram entregues.

No *sprint* sete, o efeito das anotações e o modelo de requisitos gerado pelo processador de anotações passaram a ajudar no desenvolvimento. Com a prática, a equipe começou a fazer as anotações sem tomar tanto tempo do desenvolvimento como no início.

A produtividade cresceu do *sprint* sete ao dez, refletindo a maior familiaridade com a aplicação da TDRRC e a maior produtividade devido à presença do modelo de requisitos advindo da aplicação da TDRRC, à rastreabilidade crescente dos requisitos e à melhoria na qualidade do código com refatorações e aumento do reuso através da detecção de clones.

A produtividade estabilizou no *sprint* 10 em diante, pois o conhecimento da equipe parou de crescer ou passou a crescer em um ritmo mais lento.

Os resultados obtidos com no estudo de caso mostraram um aumento de produtividade por *sprint*, depois que a equipe adquiriu prática na aplicação da TDRRC. Baseando-se unicamente nos dados não se pode dizer que os benefícios vieram do uso da TDRRC ou que um ganho de produtividade similar será obtido sempre que ela for aplicada. No entanto, a equipe envolvida com o desenvolvimento atribui o aumento de produtividade à aplicação da TDRRC.

Sobre as situações típicas, citadas na seção 6.3, pode-se dizer:

- Descoberta de requisitos (6.3.1) - Durante todo o estudo de caso foram descobertos vários requisitos e regras de negócio que foram anotados no código-fonte. A aplicação da TDRRC permitiu gerar documentos com os requisitos e regras de negócio que estavam implementados no SGVD. Estes documentos permitiram um entendimento maior do SGVD e, com isso, foi possível identificar problemas em requisitos e regras que estavam implementados e que não teriam sido detectados se a TDRRC não tivesse sido aplicada. A TDRRC serviu, portanto, como apoio à reengenharia de requisitos e pode ser usada de maneira gradual, sem interromper a evolução do *software*;

- Refatoração (6.3.2) - A aplicação da TDRRC levou à equipe a realizar diversas refatorações no *software.* As refatorações foram necessárias, principalmente para permitir que os episódios fossem anotados e foi realizada

através da quebra de métodos longos em vários métodos mais curtos. Em alguns casos, a ordem de execução foi alterada para agrupar um trecho de código que representava um episódio em um mesmo método. Como resultado destas refatorações, o código-fonte ficou mais fácil de entender e, portanto, de ser mantido;

- Identificação de Clone (6.3.3) - A aplicação da TDRRC permitiu identificar vários potenciais clones que, em alguns casos, foram refatorados e substituídos por um único trecho de código, aumentando o reuso e facilitando a manutenção. A substituição dos clones não foi feita em todos os casos, pois esta não era a prioridade no *sprint* e o trabalho envolvido para comparar as diferenças entre os clones e gerar o código que serviria para ambos os caso é razoavelmente grande e prejudicaria as entregas planejadas no *sprint.* A aplicação da TDRRC pode ser útil nesta situação para servir como indicador se vale a pena planejar um *sprint* para detecção e remoção de clones. Uma vez que se decida fazê-lo, técnicas específicas para este propósito devem ser utilizadas;

- Detecção de requisitos inconsistentes (6.3.4) - O uso do processador de anotações para gerar o modelo de requisitos foi importante para a detecção de requisitos inconsistentes. O modelo de requisitos foi usado na reunião de planejamento sempre que se discutia um novo requisito relacionado a requisitos que já estavam implementados e anotados. Com isso, foi possível detectar diversos requisitos inconsistentes antes que eles fossem implementados. Estima-se que 20% das novas demandas trouxeram inconsistências que puderam ser corrigidas graças a aplicação da TDRRC;

- Análise de Impacto (6.3.5) - A aplicação da TDRRC para análise de impacto foi importante em vários momentos em que a demanda a ser implementada exigia grandes mudanças. Nesse caso, gerou-se documentos de requisitos com informação de rastreabilidade, conforme já visto na figura 45,

inspecionava-se o código-fonte dos requisitos que seriam impactados para poder estimar o esforço necessário para a mudança. De maneira geral, a presença da rastreabilidade entre o requisito e o código-fonte permitiu um melhor planejamento da mudança e evitou atrasos nas entregas dos *sprints*.

Com a aplicação da TDRRC, pode-se sentir uma maior facilidade em evoluir o SGVD, pois se tornou mais fácil saber onde um requisito estava implementado. Localizar os trechos de código-fonte responsáveis pela implementação de um requisito era uma tarefa bastante demorada antes da aplicação da TDRRC, pois os nomes das classes e a organização dos pacotes não seguiam nenhuma lógica e não havia uniformidade entre os módulos. O modelo de requisitos gerado pelo processador de anotações permitia mapear os casos de uso no código-fonte.

6.5 Considerações Finais

No estudo de caso, a TDRRC foi aplicada em um ambiente que utilizava método ágil na evolução de um *software* complexo que não possuía documentos de requisitos. A ausência destes documentos estava tendo um impacto na produtividade e na qualidade do *software.* Com a aplicação da TDRRC foi possível fazer a reengenharia dos requisitos e, ao mesmo tempo, manter a agilidade esperada na aplicação de um método ágil.

O trabalho de reengenharia de requisitos, permitiu avaliar de forma empírica como o uso da TDRRC pode tornar um *software* mais fácil de se manter e evoluir. As anotações adicionam uma semântica adicional a trechos de código-fonte tornando mais fácil entender o que eles significam e que papel eles desempenham no sistema. Esta semântica adicional, colocada no código-fonte se mostrou uma maneira eficiente de documentar o *software* e com vantagens de ser possível fazer algumas validações sintáticas que não seriam possíveis em um documento eletrônico.

Ainda assim, estas validações não são suficientes para garantir que todos os requisitos estejam corretamente documentados no código-fonte, sendo necessária a intervenção humana para validar e garantir que os desenvolvedores estão documentando o código-fonte como deveriam. O processo aplicado no estudo de caso permitiu que a avaliação por parte dos desenvolvedores fizesse parte do ciclo de desenvolvimento, apoiada no modelo de requisitos gerado pelo processador de anotações.

Os principais benefícios da aplicação da TDRRC foram: um provável aumento de produtividade que pode ser percebido após alguns *sprints,* a recuperação do modelo de requisitos, a melhoria de legibilidade do *software*.

7 CONSIDERAÇÕES FINAIS

Neste capítulo são apresentadas as considerações finais, através da conclusão, das contribuições e dos trabalhos futuros que podem ser desenvolvidos.

7.1 CONCLUSÃO

Este trabalho apresentou a definição e a aplicação da TDRRC, uma técnica para documentação e recuperação de requisitos através de anotações no código-fonte. Durante este trabalho foram identificados três principais contextos para aplicação da TDRRC: reengenharia de requisitos, métodos ágeis e processo de gestão de requisitos proposto por Fahmi et al. (2007).

Na definição da TDRRC, decidiu-se documentar requisitos no código-fonte e esta decisão foi tomada baseada na observação empírica feita pelo autor e também por outros pesquisadores que, em muitos sistemas, com o passar do tempo, o código-fonte passa a ser a única fonte de informação confiável sobre o que foi implementado. Uma vez decidido que se documentaria requisitos no código-fonte, decidiu-se usar o mecanismo de anotações. O mecanismo de anotações foi escolhido porque permite a criação de um meta-modelo de requisitos e conta com o apoio do compilador para verificar se o meta-modelo foi respeitado. A documentação de requisitos no código-fonte através de anotações trouxe as seguintes vantagens:

- As anotações podem ser armazenadas no código binário, permitindo que o modelo de requisitos esteja disponível mesmo que o código-fonte do *software* tenha se perdido;

- Manter os requisitos atualizados fica mais fácil. Ao se remover trechos de código-fonte, suas anotações são removidas. Ao se alterar um trecho de código-fonte, o desenvolvedor vai se deparar com a anotação do requisito e será compelido a modificá-la, pois como a anotação está junto com o código-

fonte que ele está escrevendo, fica mais clara a sua responsabilidade sobre esta;

- A rastreabilidade entre requisitos e código fica mais fácil, pois os requisitos são descritos diretamente neste;
- Torna mais fácil determinar os requisitos que estão implementados, pois as anotações podem ser lidas por ferramentas que são capazes de produzir relatórios contendo esta informação;
- Torna mais fácil determinar qual requisito é implementado por um trecho de código, pois a anotação localizada no próprio código fornece esta informação;
- Torna possível a substituição dos documentos armazenados no repositório de requisitos pelo controle de versão do código-fonte, pois os requisitos passam a ser armazenados no mesmo dentro das anotações;
- Ao colocar requisito e implementação em um mesmo arquivo, o desenvolvedor acaba lendo os dois juntos com mais frequência e, com isso, é capaz de perceber incongruências entre o que está documentado e o que está implementado.

No entanto, ao documentar requisitos, que tem um nível de abstração elevado, no código-fonte, que tem um nível de abstração baixo e uma estrutura distinta dos requisitos, várias dificuldades surgiram, dentre as quais destacam-se:

- Ao documentar requisitos no código-fonte, as entidades do modelo de requisitos precisam ser mapeáveis em um elemento do código-fonte. Devido à diferença de abstração entre eles, nem sempre este mapeamento existe. Sem um mapeamento claro, corre-se o risco de ter uma mesma entidade do modelo de requisitos documentada em mais de um trecho do código-fonte, trazendo ambiguidade e potenciais inconsistências;

- Devido às limitações do mecanismo de anotações, o meta-modelo de requisitos precisa ter as associações entre os seus elementos feitas por *Strings.* Isso torna o processo mais sujeito a erros, pois o compilador não tem como validar seus valores. As validações propostas ajudaram a minimizar este problema, porém a garantia da consistência do modelo só pode ser feita através da intervenção humana.

Com a TDRRC definida, ela foi aplicada em um estudo de caso que permitiu verificar na prática sua efetividade. A partir dos resultados no estudo de caso, foi possível tecer considerações sobre a aplicação da TDRRC nos três contextos previstos para a sua aplicação.

No contexto da reengenharia de requisitos, a TDRRC serve de apoio para recuperação de requisitos ao disponibilizar um meio de registrar o conhecimento adquirido na investigação do código-fonte. O mecanismo de anotações permite que o resultado seja registrado no próprio local investigado. Com isso, mesmo que o código-fonte seja refatorado, as anotações acompanham as mudanças no código-fonte e, se for o caso, são atualizadas pelo desenvolvedor. Esta característica permite que não seja necessário interromper o desenvolvimento, manutenção ou evolução do *software* para realizar a reengenharia de requisitos. Além disso, no processo aplicado no estudo de caso, foi possível realizar a reengenharia de requisitos de forma gradual pelos próprios desenvolvedores que estavam evoluindo o *software.* Durante a programação de novas funções, os desenvolvedores passaram a anotar o conhecimento adquirido com a leitura do código-fonte, que foi necessária para realizar a alteração, ou seja, a reengenharia de requisitos ocorreu como parte do processo de evolução do *software* aplicando a TDRRC.

No contexto dos métodos ágeis, a TDRRC permite que os requisitos sejam documentados sem perder a agilidade. Ao desenvolver um *software* usando um método ágil, os desenvolvedores concentram-se em entender as necessidades do usuário , representá-las de forma informal e descartável, e implementá-las. Esta

forma de trabalho permite que mais requisitos sejam implementados mais rapidamente, porém perdem-se os documentos de requisitos. Ao permitir que a documentação dos requisitos possa ser realizada e validada no próprio código-fonte, a TDRRC torna-se particularmente interessante em ambientes ágeis, pois permite ao desenvolvedor, após sua reunião informal de captura de requisitos, documentar o que foi definido no próprio código-fonte que vai ser implementado. O uso da TDRRC nesta situação permite que os documentos de requisitos passem a existir e se mantenham atualizados sem um grande aumento da burocracia evitada pelos métodos ágeis. O estudo de caso permitiu mostrar que isso é possível.

Finalmente, no contexto do processo de gestão de requisitos proposto por Fahmi et al. (2007), a TDRRC permitiu propor um processo que viabiliza a sua realização. Seguindo a proposta de Fahmi et al. (2007), a gestão de requisitos deveria se apoiar na engenharia reversa de requisitos para determinar os requisitos implementados em um *software.* Apesar das boas ideias, a questão de como realizar a engenharia reversa de requisitos de maneira automática e segura para viabilizar a proposta de Fahmi et al. (2007) estava sem resposta. A aplicação da TDRRC trouxe uma solução para este problema, porém possui as seguintes limitações:

- Maior dependência do fator humano: enquanto na proposta de Fahmi et al. (2007) os requisitos são recuperados da implementação presente no código-fonte, com a aplicação da TDRRC eles são recuperadas das anotações que precisam ter sido feitas pelo desenvolvedor e refletirem o que foi implementado;

- Na aplicação da TDRRC para viabilizar Fahmi et al. (2007), é necessário anotar todos os requisitos antes de iniciar as alterações no *software.*

O estudo de caso mostrou que a TDRRC facilita a evolução de um *software,* pois se torna mais fácil manter os documentos de requisitos atualizados. Mostrou também que é possível aplicá-la na reengenharia de requisitos e em métodos ágeis.

7.2 CONTRIBUIÇÕES

Quando comparada com trabalhos similares, a TDRRC possui as seguintes vantagens:

- A reengenharia de requisitos poder ser implementada de forma gradual sem interromper o desenvolvimento do *software*, pois a TDRRC pode ser usada para documentar os requisitos na medida do necessário no código-fonte, durante o trabalho de programação, diferentemente das outras propostas encontradas na literatura (LIU, 2005; YU et al., 2005; YANG et al., 2006; MARCUS; MALETIC, 2003; JERMAKOVICS et al., 2008; DE LUCIA et al., 2008; LORMANS; DEURSEN, 2009; OLIVETO et al., 2010) que foram concebidas pensando em ter uma atividade própria em um processo próprio de reengenharia de requisitos;

- A TDRRC possui um certo grau de automação que torna o trabalho mais rápido que técnicas totalmente manuais como o trabalho de Liu (2005), pois após a anotação do código-fonte, a geração dos documentos de requisitos é automática através do Processador de Anotações;

- A TDRRC não depende da existência de documentos de requisitos como os trabalhos baseados em *Latent Semantic Indexing* (MARCUS; MALETIC, 2003; JERMAKOVICS et al., 2008; DE LUCIA et al., 2008; LORMANS; DEURSEN, 2009; OLIVETO et al., 2010).

A tese contribuiu da seguinte maneira para a solução dos problemas citados na justificativa desta tese (seção 1.3):

- Ausência de documentos de requisitos atualizados (CANFORA, 2007; FAHMI et al., 2007; YU, 2005): o fator humano é importante para manter os documentos de requisitos atualizados. Com a aplicação da TDRRC, ao se alterar um trecho de código-fonte, o desenvolvedor não necessita acessar

outro arquivo para atualizar a documentação, o que acaba sendo mais fácil e aumenta sua propensão para fazê-lo. Ainda, como a anotação está junto com o código-fonte que ele está escrevendo, fica mais clara a sua responsabilidade sobre a documentação dos requisitos;

- Dificuldade em encontrar o trecho de código que implementa um dado requisito (SCHWARZ et al., 2008): o processador de anotações pode gerar documentos de requisitos que incluem a localização no código-fonte da anotação que o gerou, resolvendo este problema;

- Dificuldade em determinar o requisito que está sendo implementado por um certo trecho de código (SCHWARZ et al., 2008): a aplicação da TDRRC permite a visualização da informação do requisito no código-fonte através da anotação;

- Dificuldade em determinar as mudanças que deverão ser feitas em um *software* para que ele atenda um certo requisito (CIRACI, 2007): com a visualização dos requisitos e dos trechos de código-fonte responsáveis pela sua implementação através do uso da TDRRC, fica mais fácil determinar o impacto da mudança em um requisito no sistema.

- Ausência de formas de verificar se um requisito foi implementado ou não (FAHMI et al., 2007): o processador de anotações permite gerar uma lista com os requisitos implementados e anotados no código-fonte. Se o requisito não estiver listado pode-se dizer que ele ou não foi implementado ou não foi anotado. Aplicando a TDRRC de maneira adequada, pode-se dizer se um requisito foi implementado ou não;

- Dificuldade em verificar se houve mudanças em um requisito (FAHMI et al., 2007): é possível implementar um processador de anotações que indique quando houver uma mudança de requisitos e os requisitos que foram

alterados, comparando o modelo gerado em duas versões do código-fonte anotado.

Uma outra contribuição desta TDRRC é viabilizar a proposta de Fahmi et al. (2007) que sugere um processo de desenvolvimento em que os requisitos implementados no *software* em uma iteração são usados como uma importante fonte de informação para a análise de requisitos da iteração seguinte, dentro de um processo de desenvolvimento iterativo. A ideia de Fahmi et al. (2007) é utilizar a engenharia reversa de requisitos para obter os requisitos implementados, pois estes refletiriam de forma fiel o que foi implementado. No entanto, Fahmi et al. (2007) deixa em aberto como fazer a engenharia reversa de requisitos que seria necessária para que sua proposta seja viável. Fahmi et al. (2007) comenta que, em sua pesquisa do estado da arte, não encontrou trabalhos que resolvessem este problema, algo que foi verificado dentre os trabalhos encontrados e analisados. A TDRRC, porém, viabiliza a proposta de Fahmi et al. (2007) através de um caminho alternativo para obter os requisitos implementados da iteração anterior através da geração do modelo de requisitos a partir do código-fonte anotado. Embora possa não refletir, com a mesma fidelidade, os requisitos implementados como no caso de engenharia reversa, a TDRRC fornece essa mesma informação, dependendo porém da qualidade das anotações feitas pelos desenvolvedores para sua fidelidade.

7.3 Trabalhos Futuros

Esta tese abre caminho para uma discussão mais profunda sobre o papel do código-fonte e o uso dos recursos de meta-modelagem e das anotações dentro da evolução de um *software*. O estudo de caso mostrou que a utilização da TDRRC pode trazer vantagens para a análise de requisitos e manutenção de sistemas, Ainda assim, existe uma série de melhorias que poderia ser feita para que a TDRRC possa trazer maior produtividade e benefícios aos seus utilizadores.

Por isso, propõe-se como trabalhos futuros:

- Melhoria do processador de anotações para permitir uma organização melhor dos episódios;
- Inclusão de mecanismos de sincronização com repositórios de requisitos existentes;
- Aperfeiçoamento do processador de anotações para gerar gráficos e textos melhor formatados para visualização dos requisitos;
- Aumento do número de validações e alertas que permitam garantir uma maior qualidade das anotações realizadas;
- O uso de métodos formais para representar requisitos no conteúdo das anotações;
- Gerar matriz de rastreabilidade a partir das anotações no código-fonte;
- Realizar um experimento para comparar a produtividade com a aplicação e sem a aplicação da TDRRC;
- Desenvolvimento ou aperfeiçoamento de ferramentas para análise da evolução do *software* através da investigação do código-fonte no controle de versão para que incluam, nesta análise, as anotações da TDRRC e permitam relacionar mudanças em requisitos e o seu impacto no código-fonte.

REFERÊNCIAS BIBLIOGRÁFICAS

AMBLER, Scott. **The UML v1.1 and Beyond: The Techniques of Object-Oriented Modeling.** AmbySoft Inc. White Paper, February 11, 2000

BAOJIAN, Z. **Analysis of the Requirements Change for Different Level Users in Software Developing**. In: Advanced Materials Research, v.143, p. 207--210, ISSN:1662-8985, Trans Tech Publ, 2011.

BARGMEYER, B.E. et al. **Metadata Standards and Metadata Registries: An Overview.** Citeseer, 2000.

BECK, K. et al. **Manifesto for agile software development.** In: The Agile Alliance, p. 2002—2004, 2001.

BENNETT , K; RAJLICH, V. **Software maintenance and evolution: a roadmap**. Proceedings of the Conference on The Future of Software Engineering table of contents. Limerick, Ireland. p. 73 – 87. ISBN:1-58113-253-0. Publisher ACM Press New York, NY, USA, 2000.

BERSOFF, E. **Elements of software configuration management.** In: IEEE Transactions on Software Engineering,n.1, p.79-87,ISSN:0098-5589, IEEE , 2009.

BEZERRA, V. e MELNIKOFF, S. **Requirements Oriented Programming in a Web-Service Architecture.** In: Proceedings of IADIS International Conference - www/Internet 2010, Timisoara, Romania , 2010.

BLOCH, J. **Effective Java (2nd Edition).** Prentice Hall; ISBN-13: 978-0321356680; 2a edição, 2008.

BOOCH, Grady et al. **The Unified Modeling Language User Guide**. Reading- MA USA: Adison Wesley, 1999.

BORONAT, A. e MESEGUER, J. **An algebraic semantics for MOF.** In: Fundamental Approaches to Software Engineering. Spring, 2008, p. 377--391

CANFORA, G. e PENTA M. **New frontiers of reverse engineering**. In: Future of Software Engineering, Citeseer, 2007.

CANFORA, G. et al. **Frontiers of Reverse Engineering: a Conceptual Model**. In: Fontiers of Software Maintenance 2008. ISBN: 978-1-4244-2654-6. FoSM, 2008, Beijing, p. 38-47, 2008.

CAO, L. e RAMESH, B. **Agile requirements engineering practices: An empirical study.** In: IEEE Software, v. 25, n. 1, p. 60--67, ISSN: 0740-7459, IEEE, 2008.

CATER-STEEL, A. **Information Technology Governance and Service Management: Frameworks and Adaptations.** Information Science Reference, 1a edição, 2008.

CHAPIN, N. et al. **Types of software evolution and software maintenance**. In: Journal of Software Maintenance and Evolution Research and Practice. Vol. 13, n. 1, p. 3-30. 2001.

CHUNG, L. e LEITE, J. **On non-functional requirements in software engineering.** In: Conceptual Modeling: Foundations and Applications, p. 363--379, Springer, 2009.

CIRACI, S. et al. **A Taxonomy for a Constructive Approach to Software Evolution**. In: Journal of Software. Citeseer, USA, vol. 2, n. 2 p. 84-97, 2007.

CLARK, E. **Formal methods: state of the art and future directions.** ACM Computing Surveys (CSUR) - Special ACM 50th-anniversary issue: strategic directions in computing research. V. 28 N 4. ACM New York, NY, USA. Dez, 1996.

De LUCIA, A. et al. **Adams re-trace: traceability link recovery via latent semantic indexing.** In: Proceedings of the 30th international conference on Software engineering, p. 839-842, ACM, 2008.

EGYED, A. et al. **Value-based requirements traceability: Lessons learned.** In: Design Requirements Engineering: A Ten-Year Perspective, p. 240-257, Springer, 2009.

ENGELS, Gregor et al**. Object-oriented modeling: a roadmap**. In: Proceedings of the conference on The future of Software engineering 2000 , Limerick, Ireland. ACM Press, New York, NY, USA, 2000. p. 103-116.

ERIKSSON, Hans-Erik et al. **UML Toolkit**. New York, NY USA. John Wiley & Sons, 1998.

FAHMI, Syed et al. **Software Reverse Engineering to Requirements**. In: Proceedings of the 2007 International Conference on Conergence Information Technology, 2007. Washington, DC, USA, IEEE Computer Society, p. 2199-2204, 2007.

FLANAGAN, D. **Java In A Nutshell, 5th Edition.** O'Reilly Media;ISBN-13: 978-0596007737; 5a edição, 2005.

GOTEL, O. e FINKELSTEIN, C. **An analysis of the requirements traceability problem.** In: Proceedings of the First International Conference on Requirements Engineering, 1994. p.94—101, ISBN: 0818654805, IEEE, 2002.

HARRY, Andrew. **Formal Methods Fact File: VDM and Z.** ISBN: 978-0471958574. John Wiley & Sons, 1997.

IEEE 610-1991, **IEEE Standard Computer Dictionary. A Compilation of IEEE Standard Computer Glossaries**, 1991.

IEEE-1219. **IEEE Std 1219-1998: Standard for Software Maintenance.** Los Alamitos, CA USA. IEEE Computer Society Press, 1998.

IEEE. CCSE – **Computer Curricula Software Engineering** - 2004. USA, 2004.

JACOBSON, I. et al. **The Unified Software Development Process**. Reading- MA USA: Adison Wesley, 1999.

JERMAKOVICS, A. et al. **Visualizing software evolution with lagrein.** In: Companion to the 23rd ACM SIGPLAN conference on Object-oriented programming systems languages and applications, p. 749—750, ACM, 2008.

KOBRYN, Cris. **UML 2001: a standardization odyssey**. Communications of the ACM, v. 42, n. 10, p. 29-37, Outubro 1999.

KOTONYA, G. e SOMMERVILLE, I. **Requirements engineering**. Wiley, 1998.

KRUCHTEN, Philippe et al. **Tutorial: describing software architecture with UML.** In: Proceedings of the 24th international conference on Software engineering 2002 , Orlando, Florida. ACM Press New York, NY, USA, 2002. p. 693-694.

LAM, W. e SHANKARARAMAN, V. **Requirements change: a dissection of management issues.** In: Proceedings. 25th EUROMICRO Conference, 1999, v. 2, p. 244—251, ISBN: 0769503217, IEEE, 2002.

LAMSWEERDE, A. **Requirements engineering: from craft to discipline.** In: Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of software engineering, p. 238—249, ACM, 2008.

LAUESEN, S. **Software requirements: styles and techniques**. Addison-Wesley Professional, 2002.

LEFFINGWELL, D. e WIDRIG, D. **Managing software requirements: a unified approach.** ISBN:0-321-12247-X, Addison-Wesley Longman Publishing Co., Inc. Boston, MA, USA, 2006.

LIU, K. **Requirements Reengineering from Legacy Information Systems Using Semiotic Techniques .** In: Systems, Signs & Actions . Vol. 1 (2005), No. 1, pp. 38–61 . CITW Press, 2005.

LORMANS, M. e DEURSEN, A. **Reconstructing requirements traceability in design and test using latent semantic indexing.** In: Journal of Software Maintenance and Evolution Research and Practice. Delft University of Technology. Citeseer, 2009.

MARCUS, A. e MALETIC, J.I. **Recovering documentation-to-source-code traceability links using latent semantic indexing.** In: Proceedings of the 25th International Conference on Software Engineering, p.125—135, IEEE Computer Society, 2003.

MEDVIDOVIC, N. et al. **Modeling software architectures in the Unified Modeling Language.** In: ACM Transactions on Software Engineering and Methodology (TOSEM) ACM Press New York, NY, USA, v. 11 , n. 1, January 2002. p. 2-57.

OLIVETO, R. et al. **On the equivalence of information retrieval methods for automated traceability link recovery**. In: Program Comprehension (ICPC), 2010 IEEE 18th International Conference on, p. 68--71, IEEE, 2010.

PAETSCH, F. et al. **Requirements engineering and agile software development**. In: Proceedings of the Twelfth IEEE International Workshops on Enabling Technologies: Infrastructure for Collaborative Enterprises, 2003. WET ICE 2003. p. 308—313, ISBN: 0769519636, ISSN:1080-1383, IEEE, 2003.

POLO, M. **Advances in Software Maintenance Management: Technologies and Solutions.** Idea Group Publishing. Loughborough, UK. 2002.

PRESSMAN, Roger. **Software Engineering – A Practicioner's Approach.** London, England. Mc-Graw Hill Book Company, 2009.

RAMESH, B. et al. **Implementing requirements traceability: a case study**. In: Proceedings of the Second IEEE International Symposium on Requirements Engineering, 1995. p.89--95, ISBN:0818670177, IEEE, 2005.

RAMESH, B. **Factors influencing requirements traceability practice**. In: Communications of the ACM, vol. 41, n.12, p. 37—44, ISSN:0001-0782, ACM, 1998.

RAMOS, M., Neto, J. e Vega, I. **Linguagens Formais: teoria, modelagem e implementação.** Bookman, Porto Alegre, 2009.

ROBERTSON, S. e ROBERTSON, J. **Mastering the requirements process.** ISBN: 0-321-41949-9, Addison-Wesley Longman Publishing Co., Inc. Boston, MA, USA, 2006.

ROYCE, Walker. **Software Project Management – A Unified Framework.** Reading- MA USA: Addison Wesley, 1998.

RUI, K. et al. **Refactoring Use Case Models: The Metamodel**. In: Proceedings of the 26th Australasian computer science conference, Darlinghurst, Australia, 2003. Australian Computer Society Press, p. 301-308, 2003.

RUI, K. **Refactoring use case models.** Tese de Doutorado. Concordia University Montreal, Canada, 2007.

SCHWARZ, H. et al. **Towards querying of traceability information in the context of software evolution.** In:10th Workshop Software Reengineering (WSR 2008. v.126, 2008.

SIERRA, K. **Head First Java, 2nd Edition.** O'Reilly Media; ISBN-13: 978-0596009205, 2a Edição, 2009.

SNEED, H. **20 Years of Software Reengineering--A Resume.** In: Proceedings of 10th GI Workshop on Software Rreengineering, Ed. Gimnich, Kaiser, Quante, Winter, Bad Honeff, 2008.

WEBER, R. et al. **Fit for Change: Steps towards Effective Software Maintenance.** 21st IEEE International Conference on Software Maintenance (ICSM'05). p. 26-33, Budapest, 2005.

YANG, SU et al. **Recovering early-aspects at requirements level from software legacy sytem**. In: Proceedings of the 9th Internactional Conference on Information Technology (CIT2006), Bhubaneswar, India, 2006. IEEE Computer Society Press, 2006.

YOURDON, E. **Modern Structured Analysis.** Prentice-Hall, 1991.

YU, Yijun et al. **RETR: Reverse Engineering To Requirements** In:Proceedings of the 12th Working Conference on Reverse Engineering (WCRE 2005). IEEE Computer Socierty, p. 234. , 2005.

ZARZABEK, S. **Effective Software Maintenance and Evolution: A Reuse-based Approach**. Auerbach Publications, 2007.

ZAVE, P. **Classification of research efforts in requirements engineering.** ACM Computing Surveys (CSUR), vol. 29, n.4, p. 315—321, ACM, 1997.

APÊNDICE I – GRAMÁTICA DAS ANOTAÇÕES EM JAVA

A gramática das anotações em JAVA serve para um melhor entendimento da sintaxe de anotações, mas não está pronta para o desenvolvimento de um *parser,* pois não contém todas as regras da linguagem JAVA, mas apenas aquelas necessárias para o entendimento da sintaxe de anotações. Ela foi escrita usando as convenções na notação E-BNF proposta por Ramos et al. (2009) e baseada na gramática da linguagem JAVA, que poder ser encontrada no site sobre a linguagem JAVA:

http://java.sun.com/docs/books/jls/third_edition/html/syntax.html#18.1.

Para escrita destas expressões foram usadas as seguintes convenções (RAMOS et al., 2009):

- O símbolo | significa que a expressão anterior ou posterior são aceitas. Ex.: Gat(a|o) descreve Gato e Gata;
- O símbolo ? significa que podem existir zero ou uma ocorrência da expressão anterior. Ex.: (ma)?mãe descreve mamãe e mãe;
- O símbolo parenteses é representado por <(> e <)>. Ex.: <(>ma<)>mãe descreve (ma)mãe;
- O símbolo * significa que podem existir zero ou várias ocorrências da expressão. Ex.: go(l)* descreve go, gol, gollllllll;
- O símbolo + significa que podem existir uma ou várias ocorrências da expressão. Ex.: go(l)+ descreve gol, gollllllll;
- As variáveis são descritas por expressões do tipo <nome da variavel> ::= expressão e os nomes das variáveis sempre cercados pelos símbolos menor e maior (< e >);
- 0-9 significa que pode ser qualquer caracter de 0 a 9;

- <%> significa qualquer caracter.

```
<Annotações>::= <Anotação> (<Annotações>)*
<Anotação> ::= @ <NomeTipo> ( <(>
                  <ValorAnotação> |
( ( <Identificador> = <ValorAnotação> (, (Identificador =)? ValorAnotação )
* )
             <)> )*
<NomeTipo> ::= <Identificador>

<ValorAnotação> ::= (<Constante> | <Vetor> | <ValorClasse> |
<ValorEnumerado> | <Anotação> )
<ValorClasse> ::= <Identificador>.class
<ValorEnumerado> ::= <Identificador>.<Identificador>
<Vetor> ::= {  <ValorVetor> (, <ValorVetor>)* }
<Identificador>::= (a-z|A-Z)(a-z|A-Z|0-9)*
<ValorVetor> ::= (<Constante> | ValorClasse> | <ValorEnumerado> |
<Anotação>)
<Constante> ::= ( <ConstanteInteira> | <ConstantePontoFlutuante> |
<ConstanteCaracter> | <ConstanteString> | <ConstanteBooleana> |
<ConstanteNull> )
<ConstanteInteira> ::= (0-9)+
<ConstantePontoFlutuante> ::= (0-9)*.(0-9)+
<ConstanteCaracter> ::= '(<%>)'
<ConstanteBooleana> ::= (true|false)
<ConstanteNull> ::= null
<ConstanteString> ::= “(<%>*)”
<TipoAnotação> ::= @interface <Identificador>  <CorpoTipoAnotação>
<CorpoTipoAnotação> ::=  { <DeclaraçãoAtributosAnotação> }
```

```
<DeclaraçãoAtributosAnotação> ::=  <DeclaraçãoAtributoAnotação>

<DeclaraçãoAtributosAnotação>

<DeclaraçãoAtributoAnotação> ::= <Tipo>  <NomeAtributo>() <ValorPadrão>;

<Tipo> ::= <Identificador>

<NomeAtributo> ::= <Identificador>

<ValorPadrão> ::= default <ValorAnotação>
```