UNIVERSIDADE DO VALE DO RIO DOS SINOS – UNISINOS
UNIDADE ACADÊMICA DE PESQUISA E PÓS-GRADUAÇÃO
PROGRAMA INTERDISCIPLINAR DE PÓS-GRADUAÇÃO
EM COMPUTAÇÃO APLICADA
NÍVEL MESTRADO

PABLO DALL'OGLIO

UM SISTEMA MULTI-AGENTE COLABORATIVO PARA GESTÃO DA MUDANÇA DE REQUISITOS DE SOFTWARE

SÃO LEOPOLDO
2010

Pablo Dall'Oglio

UM SISTEMA MULTI-AGENTE COLABORATIVO PARA GESTÃO DA MUDANÇA DE REQUISITOS DE SOFTWARE

Dissertação apresentada como requisito parcial para a obtenção título de Mestre pelo Programa de Pós- Graduação em Computação Aplicada da Universidade do Vale do Rio dos Sinos.

Orientador: Prof. Dr. Sérgio Crespo Coelho da Silva Pinto

SÃO LEOPOLDO
2010

CIP – CATALOGAÇÃO NA PUBLICAÇÃO

Dall'Oglio, Pablo

Um Sistema Multi-Agente Colaborativo para Gestão da Mudança de Requisitos de Software / por Pablo Dall'Oglio – São Leopoldo: Ciências Exatas e Tecnológicas da Unisinos, 2010.

130 f.:il.

Dissertação (mestrado) – Universidade do Vale do Rio dos Sinos. Ciências Exatas e Tecnológicas. Programa Interdisciplinar de Pós-Graduação em Computação Aplicada, São Leopoldo, BR-RS, 2010. Orientador: Crespo, Sérgio.

1. Gerenciamento de Requisitos. 2. Mudança de Requisitos. 3. Rastreabilidade. I. Sérgio Crespo. II Título

UNIVERSIDADE DO VALE DO RIO DOS SINOS
Reitor: Dr. Marcelo Fernandes de Aquino
Diretoria da Unidade de Pós-Graduação e Pesquisa: Profª. Drª. Ione Bentz
Coordenador do PIPCA: Prof. Dr. Arthur Tórgo Gómez

RESUMO

Muitas metodologias surgiram para organizar o desenvolvimento de software, a maioria baseada na previsibilidade dos requisitos. Como nem sempre é possível obter com exatidão todos os requisitos de um projeto antes de sua construção, torna-se fundamental saber gerenciar as mudanças de requisitos. A gestão da mudança compreende a análise de impacto, que visa identificar as entidades afetadas por uma mudança. A análise de impacto apoia-se fundamentalmente na informação de rastreabilidade, que consiste em ligações entre as informações produzidas no desenvolvimento de software. A informação de rastreabilidade tende a deteriorar durante a evolução do projeto mesmo em organizações com alto nível de maturidade no processo de desenvolvimento. Artefatos de software criados em um processo definido muitas vezes acabam desconectados uns dos outros devido à falta de métodos e de ferramentas que apoiem a rastreabilidade.

As ferramentas atuais para gerenciamento de requisitos oferecem um apoio restrito para a manutenção da rastreabilidade. Poucas ferramentas cobrem todo o ciclo de desenvolvimento de software, sendo que a maioria não está preparada para tratar a informação de rastreabilidade com o grau de precisão necessário. Por fim, a maioria das ferramentas é centrada na ação do usuário, criando uma passividade que acaba possibilitando a ocorrência de falhas no processo de comunicação que podem determinar o fracasso de um projeto.

Este trabalho tem como objetivo melhorar a gestão da mudança de requisitos por meio da implementação de uma ferramenta *web* apoiada por agentes de software que controlem com precisão a informação de rastreabilidade e suportem de forma pró-ativa a gestão da mudança de requisitos e a análise de impactos durante todo o ciclo de desenvolvimento de software. Para que isto seja possível, torna-se necessário a criação de um modelo para representar com precisão a informação de rastreabilidade em um projeto, bem como as mudança ocorridas e os impactos gerados.

Palavras-Chave: Engenharia de Software. Engenharia de Requisitos. Mudança de Requisitos. Gerência da Mudança. Rastreabilidade. Análise de Impacto.

ABSTRACT

Many metodologies were created to organize the software development, most of them based upon the requirements previsibility premise. Since getting a precise set of requirements before the software construction is the exception, it's fundamental to manage the requirements changes. The change management comprises the impact analysis, that identifies the entities affected by a change in a system. The impact analysis is fundamentally supported by the traceability information, that represents the links between the information produced during the software development. The traceability information tends to deteriorate during the project evolution, even in organizations with a high level of maturity in the development process. Software artifacts created in a defined process gets disconnected because of the lack of methods and tools that support the traceability effectively.

The actual requirement management tools offer a restricted support to mantain the traceability information. Few of them cover all the software development life cycle, and most of them are not prepared to give the necessary degree of granularity to the traceability information. Finally, most tools are focused on the user action, creating a kind of passivity that makes possible the occurrence of failures in the communication process. These failures may determine the complete project unsucess.

The objective of this work is to improve the requirements change management through the implementation of a web tool. This tool is supported by software agents that control the traceability information with a high degree of granularity and support the change management and the impact analysis pro-actively through all the software development life cycle. To make it possible, it's necessary to create a model that supports the representation of the granularity for traceability information, the occured changes and the generated impacts.

Keywords: Software Engineering. Requirements Engineering. Requirements Change. Change Management. Traceability. Impact Analysis.

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE SIGLAS

ACL	Agent Communication Language
BDI	Belief-Desire-Intention
CASE	Computer-Aided Software Engineering
DAML	DARPA Agent Markup Language
DARPA	Defense Advanced Research Projects Agency
DOORS	Dynamic Object Oriented Requirements System
ESOA	Engenharia de Software Orientada a Agentes
FIPA	Foundation for Intelligent Physical Agents
FOOPS	Functional and Object Oriented Programming System
IEC	IEC - International Electrotechnical Commission
ISO	International Organization for Standardization
KIF	Knowledge Interchange Format
KQML	Knowledge Query and Manipulation Language
MASE	Mase Multiagent Systems Engineering Methodology
ODF	Open Document Format
OIL	Ontology Inference Layer
ORM	Object Relational Mapping
OWL	Web Ontology Language
RDF	Resource Description Framework
TOOR	Traceability of Object Oriented Requirements
UML	Unified Modeling Language
W3C	World Wide Web Consortium
XMI	XML Metadata Interchange
XML	Extensible Markup Language

SUMÁRIO

1 INTRODUÇÃO

Este capítulo realiza a introdução da proposta de dissertação. Neste sentido, o domínio do problema é contextualizado dentro da respectiva área da engenharia de software e as principais motivações que levaram ao desenvolvimento do presente trabalho são apresentadas. Além disto, são apresentados os objetivos gerais e específicos para o desenvolvimento da dissertação, bem como a organização da presente proposta.

1.1 CONTEXTUALIZAÇÃO

Para Leite (2001), a sociedade tem passado a ver cada vez mais *software* como parte de suas vidas e isto tem gerado também uma demanda maior por qualidade nos produtos, que por sua vez possuem um custo, por vezes não compreendido. Um dos maiores desafios da engenharia de software está em prover tal qualidade com os recursos limitados disponíveis.

Muitas metodologias surgiram ao longo do tempo para organizar o desenvolvimento de projetos de software, provendo maior qualidade com menor custo. A maioria delas inspiradas em metodologias tradicionais da engenharia, cujos princípios baseiam-se na previsibilidade dos requisitos. Ou seja, todos os requisitos devem ser levantados na etapa de projeto (FOWLER, 2002).

Em se tratando de software, nem sempre é possível obter com exatidão todos os requisitos de um projeto antes de sua construção. Muitos fatores subjetivos conspiram neste sentido. Em geral, os requisitos mudam, uma vez que o cliente só consegue validar um processo após vê-lo implementado. Neste caso, é necessário saber o que fazer quando ocorre a mudança (BECK; ANDRES, 2004).

Para Leite (2001), muitas das razões que levam à produção de *softwares* que não satisfazem as necessidades dos clientes são derivadas da falta de atenção para com a tarefa de definir e acompanhar a evolução dos requisitos de software durante a execução de um projeto. É de senso comum que a qualidade final de um software é diretamente afetada pela qualidade de seus requisitos (ZONG et al., 2007). Neste sentido, gerenciar as mudanças de requisitos, tais como a adição ou remoção de requisitos é fundamental.

Neste contexto, ganha importância a engenharia de requisitos, mais especificamente a gestão da mudança de requisitos, uma vez que o sucesso do software vai depender de quanto

ele se adequar às mudanças de ambiente (NUSEIBEH; EASTERBROOK, 2000). Para Nurmuliani, Zowghi e Fowell (2004), as mudanças de requisitos durante o processo de desenvolvimento de software são inevitáveis. Assim, é preciso gerenciar os impactos gerados por estas mudanças.

Dentro da gestão da mudança de requisitos, a análise de impacto é a atividade que identifica as entidades que podem ser afetadas por uma mudança proposta no sistema. O objetivo principal da análise de impacto é minimizar os efeitos colaterais de uma mudança (QUEILLE et al., 1994). Dentre algumas formas de se realizar a análise de impacto, a rastreabilidade é a principal delas.

A rastreabilidade de requisitos consiste em ligações entre as informações produzidas no desenvolvimento de software (ARNOLD; BOHNER, 1993) e representa os relacionamentos existentes entre requisitos, *design* e implementação do sistema. A rastreabilidade se estabelece não somente entre componentes de software, mas entre quaisquer artefatos produzidos durante os estágios de desenvolvimento do sistema, incluindo documentação e procedimentos (RAMESH et al., 1995).

1.2 MOTIVAÇÃO

A informação de rastreabilidade em um projeto é essencial para que se possa gerenciar com eficiência as mudanças ocorridas, bem como analisar os impactos gerados por estas mudanças. Entretanto, a informação de rastreabilidade tende a deteriorar durante a evolução do projeto, devido à fatores como a pressão de tempo sobre os colaboradores, que acabam falhando em manter atualizada tal informação (CLELAND-HUANG et al., 2007).

Para Cleland-Huang, Hayes e Domel (2009), muitas organizações investem alto na construção de matrizes de rastreabilidade para acompanhar o processo de gerência de requisitos. Entretanto, as matrizes tornam-se frequentemente desatualizadas e inúteis para que os colaboradores do projeto possam realizar suas atividades relativas à gerência da mudança e análise de impacto. Para Zhang et al. (2008), é sabido que, mesmo em organizações com alto nível de maturidade no processo de desenvolvimento de software, artefatos de software criados em um processo definido muitas vezes acabam desconectados uns dos outros. O principal motivo para isto está na falta de métodos e de ferramentas que apoiem a

rastreabilidade de maneira efetiva (CLELAND-HUANG; HAYES; DOMEL, 2009).

Conforme Cleland-Huang et al. (2007), as ferramentas atuais para gerenciamento de requisitos oferecem um apoio restrito para a criação e manutenção da informação de rastreabilidade. Já para Ghazi (2008), as ferramentas existentes se especializam em armazenar registros e manter a informação de rasteabilidade para apenas alguns tipos de informação (artefatos), ou seja, não cobrem todo o ciclo de desenvolvimento de software. Além disto, nem todas ferramentas estão preparadas para tratar a informação de rastreabilidade com o grau de precisão (granularidade) adequado, o que dificulta uma maior adoção da técnica.

Para Mäder, Gotel e Philippow (2009), uma das grandes restrições à maior adoção da rastreabilidade é o grande esforço relacionado com a criação e manutenção da informação de rastreabilidade em um projeto. Para Cleland-Huang et al. (2007), fatores como a distância física entre os colaboradores tendem a aumentar o problema . A distância física é característica da prática de desenvolvimento distribuído de software, cuja adoção tem se tornado comum pela indústria de software (Bruegge et al., 2006). Esta distância reforça a necessidade da existência de ferramentas que ofereçam suporte para que tal processo seja conduzido de maneira colaborativa (LANG; DUGGAN, 2001).

As ferramentas existentes geralmente focam no trabalho do indivíduo, relegando à este muitas vezes a tarefa de requisitar à ferramenta uma análise de impacto quando da ocorrência de uma mudança de requisitos. Esta passividade, que se traduz pela dependência da ação pessoal acaba possibilitando a ocorrência de falhas no processo de comunicação em uma etapa crucial para o desenvolvimento do projeto. Existe a possibilidade de que estas falhas de processo nunca venham a ser detectadas, determinando em alguns casos o fracasso de um projeto. Desta forma, percebe-se a necessidade da existência de ambientes de desenvolvimento que não dependam da ação direta dos colaboradores, e que ofereçam recursos de proatividade em relação ao processo.

Para que se possa efetivamente gerenciar as mudanças de software é necessário um método que permita controlar com precisão a rastreabilidade entre requisitos, e gerenciar as mudanças ocorridas e os impactos gerados de forma pró-ativa em um ambiente possivelmente distribuído. Apesar de vários trabalhos terem sido propostos com este objetivo, são percebidas deficiências nestes, sendo que questões como proatividade, granularidade e manutenção dos links de rastreabilidade não estão resolvidas por completo nas abordagens existentes.

Para Mäder, Gotel e Philippow (2009), a essencia de um método está na definição e

utilização de um modelo que suporte a informação de rastreabilidade. Estes modelos fornecem suporte para decidir quais são os artefatos de software a serem armazenados, bem como quais relacionamentos devam ser definidos. Para Cimitile (1999), para que o processo de análise de impacto possa identificar os artefatos afetados por mudanças de sistema é necessário um modelo de representação do software que possa formalizar o conhecimento sobre as dependências entre artefatos.

Para Mäder, Gotel e Philippow (2009), apesar de diversos estudos terem propostos modelos de rastreabilidade ao longo do tempo, definindo tipos de artefatos e de relacionamentos, tais modelos de rastreabilidade ainda são raramente utilizados em virtude de limitações frente às ferramentas de desenvolvimento atuais. Tendo em vista os pontos levantados, o desenvolvimento de um método deve estar apoiado pela criação de um modelo para representar com precisão a informação de rastreabilidade em um projeto, bem como as mudança ocorridas e os impactos gerados de forma que se integre facilmente aos ambientes de desenvolvimento atuais.

1.3 QUESTÃO DE PESQUISA

A questão central deste trabalho é: como controlar com precisão a rastreabilidade em um projeto de software para que se possa gerenciar efetivamente as mudanças ocorridas e os impactos gerados durante todo o ciclo de desenvolvimento de forma pró-ativa em um ambiente possivelmente distribuído ?

1.4 OBJETIVOS

Este trabalho tem como objetivo melhorar a gestão da mudança de requisitos por meio da implementação de uma ferramenta *web* apoiada por agentes de software que controlem com precisão a informação de rastreabilidade e suportem de forma pró-ativa a gestão da mudança de requisitos e a análise de impactos durante todo o ciclo de desenvolvimento de software. Para que isto seja possível, torna-se necessário a criação de um modelo para representar com precisão a informação de rastreabilidade em um projeto, bem como as

mudança ocorridas e os impactos gerados. Os objetivos específicos deste trabalho são:

- Criar um modelo para representação de requisitos com suporte à rastreabilidade com precisão (granularidade), representação das mudanças e dos impactos gerados;
- Definir um método para gerenciar a mudança de requisitos em projetos de software de forma pró-ativa durante todo o ciclo de desenvolvimento de software;
- Implementar a ferramenta, baseada no modelo e método propostos;
- Avaliar a ferramenta criada com base em cenários pré-definidos;

1.5 METODOLOGIA

A metodologia desta proposta inicia pela definição do problema a ser resolvido para que, a partir disto, possa se realizar uma revisão bibliográfica sobre gerência da mudança de requisitos, análise de impacto e rastreabilidade. Desta forma, procura-se efetuar um levantamento das suas principais características com o objetivo de elaborar um modelo mais aderente aos conceitos estabelecidos.

Juntamente com a revisão bibliográfica, será realizado um estudo sobre os principais modelos já propostos para o gerenciamento da mudança de requisitos e de rastreabilidade. Com isso, espera-se realizar o mapeamento de suas principais características, a fim de se especificar um modelo único, contendo as características mais importantes e indispensáveis neste contexto. Além de um estudo de modelos relacionados, também será efetuado um levantamento bibliográfico das principais propostas de métodos que tratam da mudança de requisitos e de rastreabilidade a fim de se realizar um mapeamento de suas características relevantes para compor um novo método aderente ao modelo proposto.

A partir dos modelos e métodos estudados, será elaborado um modelo de representação de requisitos que suporte as características essenciais identificadas e procure suprir os requisitos para os quais não foi encontrado suporte nos modelos avaliados. Além do modelo de representação, será elaborado um método para gerência da mudança de requisitos e um modelo de agentes que irão automatizar tarefas relativas à gerência da mudança de requisitos e rastreabilidade.

A partir do modelo e do método propostos, será criada uma ferramenta que implemente o modelo construído em uma arquitetura constituída por agentes com o objetivo

de suportar a gestão da mudança de requisitos conforme os objetivos do trabalho.

A partir da ferramenta criada, serão elaborados cenários reais de uso em um ambiente com mudança de requisitos de software. Para tal, serão mapeadas algumas situações especiais bem como a forma pela qual a ferramenta deve se comportar em cada situação. A partir dos cenários criados, a ferramenta poderá ser avaliada por meio de suas respostas à situações previamente identificadas.

1.6 ORGANIZAÇÃO DO TRABALHO

O presente trabalho está organizado em sete capítulos organizados da seguinte forma:

- O capítulo um introduz a proposta de dissertação, contextualiza o domínio do problema e apresenta as principais motivações que levaram ao desenvolvimento do presente trabalho. Além disto, são apresentados os objetivos gerais e específicos para o desenvolvimento da dissertação, bem como sua organização;
- O capítulo dois trata da revisão bibliográfica básica dos conceitos envolvidos no trabalho, como engenharia de software, gerência de requisitos, rastreabilidade, análise de impacto e modelagem orientada a agentes;
- O capítulo três aborda os principais trabalhos relacionados. Para tal, apresentará trabalhos que apresentam catálogo de requisitos para o desenvolvimento de aplicações deste tipo, trabalhos que envolvem modelos de representação de requisitos, trabalhos que envolvem métodos para gestão de requisitos e trabalhos que envolvem métricas relacionadas à área pesquisada;
- O capítulo quatro apresenta a proposta do presente trabalho, envolvendo um modelo e um método para gestão de requisitos com suporte à rastreabilidade construído com base nos requisitos identificados nos trabalhos relacionados de forma que contemple os objetivos da dissertação;
- O capítulo cinco realiza uma análise comparativa entre os trabalhos pesquisados frente ao trabalho proposto conforme critérios pesquisados. Serão comparados os modelos de representação de requisitos e os métodos para gestão de requisitos;
- O capítulo seis descreve as considerações finais, os resultados esperados, as limitações do modelo e indicações para trabalhos futuros.

2 ARCABOUÇO TEÓRICO

Este capítulo aborda os conceitos que formam a base do presente trabalho. Desta forma, após uma breve apresentação da engenharia de software, será abordada a engenharia de requisitos e suas principais atribuições, como: elicitação, análise, documentação, validação e gerência. Uma das principais funções da gerência de requisitos é a gerência da mudança. Neste sentido, serão abordadas as principais fases que integram o processo de gerência da mudança de requisitos. Uma das atividades da gerência da mudança de requisitos é a análise de impacto, que irá determinar o que deverá ser alterado no sistema para que se implemente uma mudança proposta. Desta forma, a análise de impacto será conceituada e serão apresentadas suas principais fases. Ainda neste sentido, para uma efetiva gerência da mudança de requisitos e análise de impacto, é primordial a utilização de alguma técnica de rastreabilidade. Neste sentido, a rastreabilidade será conceituada e classificada de acordo com diferentes taxonomias.

Uma das grandes restrições à maior adoção da rastreabilidade é o grande esforço relacionado com a sua manutenção. Fatores como a distância física, característica do desenvolvimento distribuído de software, reforçam a necessidade da existência de ferramentas que ofereçam suporte para que tal processo seja conduzido de maneira colaborativa. Neste sentido será abordado o tema “desenvolvimento distribuído de software”, suas principais características e desafios.

A arquitetura da aplicação será fortemente dependente do trabalho de agentes de software, que irão automatizar diversas tarefas representadas por papeis que em muitas vezes são desempenhados por pessoas. Tais agentes, que serão implementados na forma de *Web Services*, irão monitorar a mudança de requisitos, notificar membros da equipe de desenvolvimento, realizar questionamentos e classificações, gerenciar versões de requisitos, gerenciar mudanças de requisitos e realizar análises de impacto. Desta forma, serão abordados os assuntos *Web Services* e Engenharia de Software Orientada a Agentes (ESOA), bem como suas principais metodologias para que se dê prosseguimento à modelagem da aplicação.

2.1 ENGENHARIA DE SOFTWARE

Durante muito tempo, o centro das atenções da engenharia de software não foi o

processo. Quando a produção era focada em fases sequenciais com produtos bem definidos, a qualidade deveria estar no produto e nas suas formas de representação (linguagens), e não no processo de concepção do mesmo (LEITE, 2001).

Atualmente, sabe-se que o processo de produção é fundamental para que tenhamos produtos de qualidade, neste sentido, procura-se cada vez mais a obtenção e transformação de dados sobre o processo produtivo em conhecimento. Para Leite (2001), a escolha dos métodos que serão utilizados no processo produtivo de um software tem relação direta com a qualidade resultante.

Para Leite (2001), muitas das razões que levam à produção de softwares que não satisfazem as necessidades dos clientes são derivadas da falta de atenção para com a tarefa de definir e acompanhar a evolução dos requisitos de software durante a execução de um projeto. Aspectos de qualidade são tratados no próprio processo de definição do software, através de processos como o da gerência de requisitos.

A qualidade de um software é diretamente influenciada pela maturidade do seu processo de desenvolvimento na organização. Processos como o cascata, espiral ou prototipação incluem atividades de análise de requisitos e é de senso comum que a qualidade final da solução depende dos requisitos sobre os quais o sistema foi construído (LLOYD; ROSSON; ARTHUR, 2002).

2.2 ENGENHARIA DE REQUISITOS

Para De Bortoli e Price (2000), é papel da engenharia de requisitos a proposição de métodos, técnicas e ferramentas que venham a facilitar as atividades da definição de requisitos de software. Para Paetsch, Eberlein e Maurer (2003), o papel da engenharia de requisitos é o de descobrir o propósito de um sistema por meio da identificação e documentação das necessidades dos clientes. A engenharia de requisitos conduz à um processo que leva desde o reconhecimento de um problema a ser resolvido até sua especificação detalhada. Várias técnicas foram desenvolvidas para a elicitação de requisitos, como: entrevistas, casos de uso, cenários, observação, *brainstorming* e prototipação.

Basicamente suas técnicas visam assegurar um conjunto consistente e significativo de requisitos antes da construção do sistema, para se evitar mudanças posteriores. Estas idéias

baseiam-se no tradicional pressuposto de que quanto mais tarde os erros forem identificados, mais dispendiosa será a sua correção (PAETSCH; EBERLEIN; MAURER, 2003). Muitos estudos já foram elaborados para tentar calcular o custo de uma correção de um software em diferentes fases. Algumas pesquisas indicam um custo em torno de 100 à 200 vezes maior para se corrigir um problema na fase de manutenção do que na fase de especificação de requisitos (LLOYD; ROSSON; ARTHUR, 2002).

Para Batista e Carvalho (2003), descobrir as necessidades do usuário é uma das tarefas mais árduas do processo de desenvolvimento de software. Além disto, se mal realizada, pode comprometer toda fase de engenharia de requisitos e por consequência, todo o projeto do sistema.

Para Espindola et al. (2005), a engenharia de requisitos pode ser considerada uma disciplina voltada para a análise das necessidades e documentação dos requisitos. Para tal, ela fornece mecanismos para facilitar as atividades de análise, documentação e verificação. A engenharia de requisitos pode ser representada por fases como: elicitação, análise, documentação, validação e gerenciamento.

2.2.1 FASES

A seguir, cada uma das fases da engenharia de requisitos será detalhada.

2.2.1.1 ELICITAÇÃO

A fase de elicitação corresponde ao momento no qual os requisitos são descobertos a partir das necessidades dos usuários por meio de técnicas como entrevistas ou *brainstorming*, além de outras como pesquisas e análises de documentos e formulários (DURÁN et al., 1999). O resultado desta etapa é um documento contendo os requisitos.

Para Kotonya e Sommerville (1998), a elicitação envolve a descoberta de requisitos funcionais e não-funcionais que o software deve possuir para satisfazer as diversas necessidades do cliente. Para Horrian, Mahmud e Karthikeyan (2003), o objetivo da elicitação é descobrir os requisitos de um sistema no domínio da aplicação e dos negócios, bem como o

escopo do mesmo, suas necessidades e restrições através de seus usuários.

Para Nuseibeh e Easterbrook (2000), a elicitação de requisitos também consiste em identificar os usuários do sistema, bem como os desenvolvedores envolvidos no processo, as tarefas que seus usuários desempenham e como eles gostariam de desempenhar estas mesmas tarefas.

2.2.1.2 ANÁLISE

Na fase de análise, os requisitos são analisados para detectar inconsistências, bem como identificar requisitos faltantes. A análise geralmente é conduzida por meio da utilização de um modelo orientado a objetos (DURÁN et al., 1999). Nesta etapa, os usuários podem participar desde que estes tenham sido treinados em técnicas de modelagem. Os requisitos dos usuários são então transformados em requisitos de software. Nesta etapa muitas vezes são desenvolvidos os primeiros protótipos do sistema.

Para Kotonya e Sommerville (1998), durante a análise, os requisitos são analisados em busca de problemas e conflitos para que o analista, juntamente com o cliente possa acordar sobre as mudanças. Para Horrian, Mahmud e Karthikeyan (2003), a análise de requisitos verifica a consistência, completeza e aplicabilidade dos mesmos, observando se são contraditórios, apresentam informações necessárias para sua implementação e estão dentro das restrições impostas pelo cliente. Esta fase envolve resolução de conflitos através de técnicas como negociação de priorização e levantamento de pré-requisitos técnicos.

2.2.1.3 DOCUMENTAÇÃO

Na fase de documentação, é realizada a documentação dos requisitos, que é utilizada para comunicar os requisitos do sistema entre cliente e desenvolvedor, através da descrição do domínio da aplicação e do sistema a ser desenvolvido (HORRIAN; MAHMUD; KARTHIKEYAN, 2003). Para Nuseibeh e Easterbrook (2000), a forma pela qual os requisitos de um sistema são documentados é de extrema importância, para garantir que os mesmos possam ser lidos, analisados e validados.

2.2.1.4 VALIDAÇÃO

Na fase de validação, analistas e usuários devem validar os requisitos e avaliar o protótipo, o que muitas vezes leva à descoberta de novos requisitos. Este é um processo iterativo que se repete até que todos os requisitos sejam validados ou não existam mais requisitos a descobrir (DURÁN et al., 1999).

Para Seybold, Meier e Glinz (2004), a validação de requisitos e a consequente detecção de falhas é um importante fator para aumentar a qualidade e reduzir o custo do desenvolvimento de software. Para Kotonya e Sommerville (1998), nesta fase é verificada a consistência, completude e precisão dos requisitos, para garantir que não exista ambiguidade, inconsistências, erros e omissões nos requisitos.

O processo de validação objetiva garantir que os requisitos reflitam as necessidades do cliente em relação ao sistema (completeza), através da inspeção por padrões organizacionais, conhecimento sobre a organização e do sistema a ser desenvolvido, garantindo que os requisitos estejam livres de inconsistências (HORRIAN; MAHMUD; KARTHIKEYAN, 2003). Para Nuseibeh e Easterbrook (2000), se torna necessário não apenas validar os requisitos, mas também resolver possíveis conflitos entre membros da equipe (*stakeholders*) com idéias divergentes.

2.2.1.5 GERENCIAMENTO

Por último, a fase de gerenciamento consiste em gerenciar as mudanças nos requisitos e garantir que elas ocorram de uma maneira controlável (KOTONYA; SOMMERVILLE, 1998). Durante o ciclo de vida de um projeto, é necessário gerenciar as mudanças de requisitos. Neste sentido, dependências e relações entre os requisitos são controlados via matriz de rastreabilidade, que permite ao gerente conhecer quais outros requisitos serão afetados por uma mudança. Para tal, utiliza-se versionamento de requisitos, para que se mantenha o registro de quem solicitou a alteração do requisito, quais foram as alterações e qual foi o impacto decorrente (HORRIAN; MAHMUD; KARTHIKEYAN, 2003).

Para Nuseibeh e Easterbrook (2000), o sucesso de um software depende do quanto ele irá se adequar as mudanças do ambiente. Neste sentido, gerenciar as mudanças de requisitos,

tais como a adição ou remoção de requisitos é fundamental, uma vez que focar a gerência da mudança apenas no código-fonte, através de controles de versões, por exemplo, leva à dificuldades na manutenção. A figura a seguir procura representar as etapas da engenharia de requisitos.

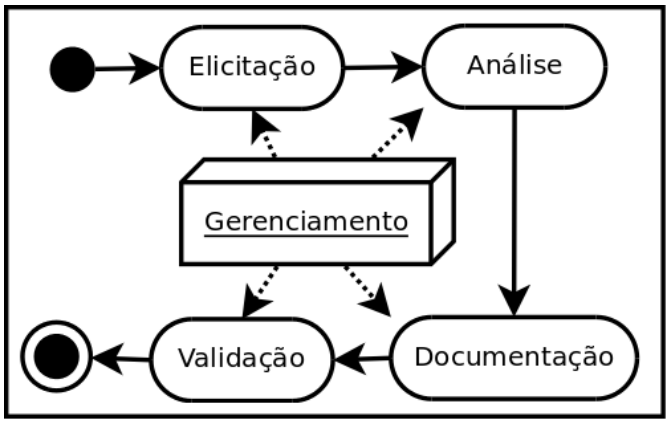


Figura 1. Etapas da engenharia de requisitos

2.2.2 TÉCNICAS DE ELICITAÇÃO

O processo de elicitação de requisitos pode ser compreendido como um conjunto de atividades voltadas para a descoberta e entendimento dos requisitos de um software. Os problemas advindos destas atividades podem ser classificados em problemas de escopo (decisão dos limites do sistema), entendimento (comunicação entre os envolvidos) e volatilidade (mudanças nos requisitos ao longo do tempo) (DURÁN et al., 1999).

Um dos principais problemas de entendimento está em expressar e registrar os requisitos de forma que estes possam ser entendidos por analistas bem como usuários. Técnicas atuais para elicitação de requisitos como entrevistas ou *brainstorming* tratam da descoberta dos requisitos mas não em como estes serão expressos (DURÁN et al., 1999).

Para Martins e Daltrini (2001) apesar das várias técnicas existentes para auxiliar a elicitação de requisitos como casos de uso e *brainstorming*, existem problemas essenciais que ainda constituem um desafio, como a dificuldade do usuário saber o que precisa e dificuldade na comunicação entre membros da equipe.

Para Martins e Daltrini (2001), para se realizar o efetivo levantamento das necessidades do usuário, é necessário entender as atividades por ele desempenhadas inseridas

no contexto de seu trabalho. Para tal, é fundamental realizar uma análise de suas ações e seus objetivos.

Para Martins e Daltrini (1999), os problemas inerentes à especificação de requisitos podem ser classificados em dois grupos: essenciais e acidentais. Os acidentais são causados pela falta de controle sobre as atividades de engenharia de requisitos, ou seja, falta de documentação e revisão dos requisitos. Já os essenciais são intrínsecos e inerentes ao processo, como: o usuário não saber exatamente o que quer, dificuldade de comunicação entre o usuário e o analista e a mudança natural dos requisitos.

Os problemas acidentais podem ser minimizados por meio da adoção de um processo sistemático que oriente a elicitação, análise, especificação, validação e gerenciamento dos requisitos. Entretanto, os problemas essenciais são consequência do processo de compreensão e comunicação do ser humano. Desta forma, mais do que instituir um processo, é necessário compreender o contexto e o ambiente social e psicológico nos quais as pessoas desenvolvem suas atividades, conhecer o histórico de evolução destas atividades e as ferramentas necessárias para desenvolvê-las (MARTINS; DALTRINI, 1999). Para Espindola et al. (2005), problemas na elicitação de requisitos podem tornar necessária a realização de um novo ciclo de especificação, projeto, codificação e teste, impactando diretamente os custos e prazos do projeto. A seguir, serão detalhadas as técnicas de elicitação de requisios mais utilizadas.

2.2.2.1 ENTREVISTAS

Entrevistas são utilizadas para obter o conhecimento sobre o domínio por meio de perguntas realizadas aos usuários especialistas. Permite aos analistas entenderem os processos da organização, bem como perceber as expectativas dos usuários. As entrevistas podem ser não-estruturadas, semi-estruturadas ou estruturadas (BATISTA; CARVALHO, 2003).

Entrevistas constituem um método que pode ser aplicado informalmente ou por meio de questionários com perguntas pré-definidas. Sua utilização traz à tona fatos e opiniões dos usuários do sistema. Por um lado as entrevistas resultam em um conjunto rico em informações, por outro, demandam grandes esforços de análise, tendo em vista que podem trazer visões conflitantes (PAETSCH; EBERLEIN; MAURER, 2003).

2.2.2.2 BRAINSTORMING

Brainstorming é uma técnica que envolve reunião em grupo, e é baseada na geração de idéias. Envolve a reunião de especialistas de negócio e de sistema tendo em vista a criação de idéias para resolução de problemas. Envolve alguns princípios como a não-crítica e o não-julgamento das idéias geradas.

A técnica de *Brainstorming* pode ser aplicada no início da etapa de elicitação de requisitos quando o universo de requisitos conhecidos é pequeno, sendo necessário gerar novas idéias (BATISTA; CARVALHO, 2003). As idéias são geradas de forma rápida, coletadas e então discutidas e avaliadas pelo grande grupo (PAETSCH; EBERLEIN; MAURER, 2003).

2.2.2.3 CASOS DE USO E CENÁRIOS

Casos de Uso descrevem as interações entre o sistema e seus atores, trazendo à tona suas relações e os diferentes papéis desempenhados pelos usuários no sistema. Casos de Uso são usados nos primeiros estágios de análise para demonstrar os requisitos funcionais do sistema.

Cenários provêm descrição de um determinado conjunto de atividades entre o sistema e o usuário, simulando o fluxo de interações entre os mesmos após um estado inicial. (PAETSCH; EBERLEIN; MAURER, 2003).

2.2.2.4 OBSERVAÇÃO

A observação consiste na observação do usuário do sistema em suas atividades cotidianas, visando descrever suas tarefas por meios diretos, como a anotação, ou indiretos como a gravação (PAETSCH; EBERLEIN; MAURER, 2003).

2.2.2.5 PROTOTIPAÇÃO

Um protótipo é uma versão inicial de um sistema com vistas à validação de requisitos ainda em estágios iniciais de desenvolvimento. O protótipo de um sistema pode ser descartado ou evoluir para uma versão final do sistema (PAETSCH; EBERLEIN; MAURER, 2003). Para Mannio e Nikula (2001), os protótipos minimizam a distância na comunicação entre cliente e analista e constituem uma forma concreta de mostrar para o cliente os conceitos absorvidos.

2.2.2.6 JAD

A técnica JAD consiste em sessões de trabalho baseadas em dinâmica de grupo e utilização de recursos visuais, onde a equipe envolvida na elicitação, sendo usuários e analistas trabalham para projetar o sistema, desde seus requisitos até projeto de interfaces. Sessões JAD permitem a coleta de uma grande quantidade de requisitos junto aos usuários-chave, considerando suas necessidades específicas. É combinada especialmente bem com a prototipação, que pode ser utilizada para demonstrar os resultados parciais de sessões JAD (BATISTA; CARVALHO, 2003).

2.2.3 DOCUMENTAÇÃO

A forma mais utilizada para expressar requisitos de software é a linguagem natural, já que na maioria das vezes esta é a única linguagem compreendida pelos usuários e analistas. Existem alguns problemas conhecidos e inerentes à utilização da linguagem natural para expressar requisitos de software. Apesar disto, a utilização de notações formais já nos estágios iniciais de levantamento de requisitos não é aconselhada, uma vez que pode impossibilitar posteriormente o entendimento dos requisitos por parte dos envolvidos (DURÁN et al., 1999).

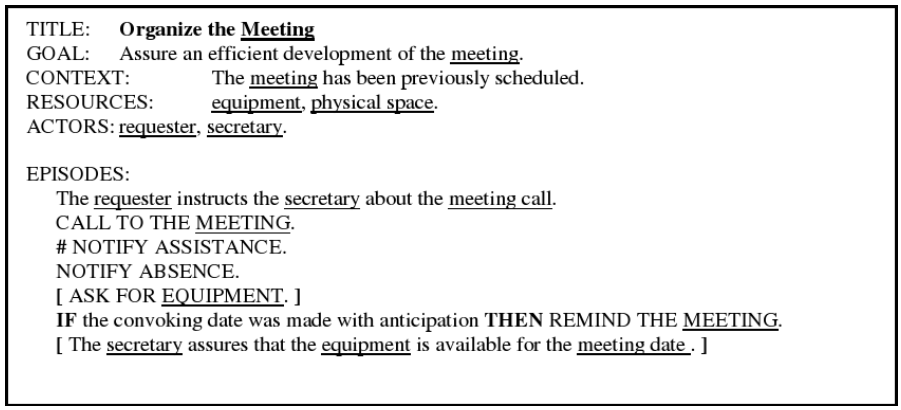
Para Fantechi e Spinicci (2005), a utilização da linguagem natural é a forma mais comum de especificação de requisitos de software, devido a sua fácil compreensão e comunicação entre as pessoas envolvidas no processo de software. Entretanto, a ambiguidade da linguagem natural pode levar à definições incorretas ou inconsistentes, com contradições

ou mesmo a falta de informações.

A extração de informações a partir de documentos de especificação de requisitos é tarefa complexa, uma vez que a utilização de modelos de documento e regras de escrita não é uma premissa neste tipo de documentação técnica. Empresas e organizações utilizam seus próprios padrões para escrever seus documentos de requisitos. (FANTECHI; SPINICCI, 2005).

Leite (2001) propõe a utilização de cenários, uma técnica que visa facilitar o entendimento e a comunicação entre os membros de uma equipe, devido à sua facilidade de simular o pensamento. Para Leite et al. (2004), um cenário constitui uma descrição parcial do comportamento da aplicação em dado momento e contexto, geralmente feita em linguagem natural. A palavra "cenário" tem significado diferente de acordo com a disciplina aplicada. Dentro da engenharia de software, é considerada uma técnica descritiva focada no processo e centrada no usuário. Apesar das diferentes interpretações do conceito de cenário, é de senso comum que um cenário se trata de uma descrição narrativa de exemplos de uso. No mundo da orientação a objetos, o termo caso de uso é mais utilizando, sendo que um cenário é considerado o detalhamento deste.

Cenários descrevem situações observando aspectos de uso e permitindo o analista definir o problema, unificar critérios e ganhar o comprometimento do usuário, além de formar conceitos que funcionarão como âncoras para a rastreabilidade. A utilização de cenários demanda a aplicação de uma linguagem descritiva de fácil aprendizado. Note na figura a seguir, que termos em maiúsculo indicam a existência de um sub-cenário, uma forma simples de se manter o elo entre os artefatos (LEITE et al., 2004).


TITLE: **Organize the Meeting**
GOAL: Assure an efficient development of the meeting.
CONTEXT: The meeting has been previously scheduled.
RESOURCES: equipment, physical space.
ACTORS: requester, secretary.

EPISODES:
The requester instructs the secretary about the meeting call.
CALL TO THE MEETING.
NOTIFY ASSISTANCE.
NOTIFY ABSENCE.
[ASK FOR EQUIPMENT.]
IF the convoking date was made with anticipation **THEN** REMIND THE MEETING.
[The secretary assures that the equipment is available for the meeting date .]

Figura 2. Roteiro (LEITE, 2001)

Para Breitman, Leite e Berry (2005) a especificação de um cenário é composta de:

- Título: identifica o cenário;
- Contexto: localiza o cenário nas dimensões tempo e espaço;
- Recursos: objetos ou informações disponibilizadas;
- Atores: pessoas, organizações, sistemas com papéis no cenário;
- Objetivos: o que deve ser realizado pelos episódios;
- Episódios: representa o curso da ação do cenário;
- Exceções: indica possíveis problemas para atingir os objetivos;

Leite et al. (2004) propõe construir as descrições dos cenários sobre um vocabulário do domínio da aplicação. Para tal, deve-se iniciar a descrição de cenários com um dicionário léxico pré-existente. O dicionário léxico descreve o vocabulário da aplicação e o conjunto de cenários descreve a aplicação.

O objetivo do dicionário léxico é registrar palavras ou frases de um determinado domínio. Cada entrada do dicionário é descrita e pode ser relacionada com sinônimos. Cada entrada é classificada em sujeito, objeto, verbo e estado. Para utilizar o vocabulário, dois princípios devem ser seguidos: o princípio da circularidade, ou seja, maximizar o uso de alguns termos para descrever outros, e o princípio do vocabulário mínimo, ou seja, minimizar o uso de símbolos externos ao vocabulário.

2.3 GERÊNCIA DE REQUISITOS

A construção de um software é um processo evolutivo formado por várias etapas. Uma delas é a gerência de requisitos, que tem por papel gerenciar e controlar os requisitos de um software. Os requisitos evoluem ao longo do tempo e sua própria definição gera um *feedback* contínuo, gerando novos requisitos e modificando os existentes. Desta forma, faz-se necessário que a gerência de requisitos esteja cada vez mais associada ao conceito de *baseline*, ou seja, que se crie uma linha evolutiva que demonstre a evolução dos requisitos ao longo do tempo.

Para De Grande e Martins (2006), os requisitos evoluem com o tempo, seja por erros detectados, seja pela evolução do conhecimento do cliente. O gerenciamento dos requisitos é necessário para que se possa controlar as mudanças e refletir estas mudanças no sistema e nos

objetivos da organização, dando subsídios para as análises de custos e impactos.

Para Kotonya e Sommerville (1998), "As principais atribuições da gerência de requisitos são: Gerenciar mudanças nos requisitos definidos; Gerenciar as relações entre os requisitos; Gerenciar as dependências entre o documento de requisitos e outros documentos produzidos ao longo do processo".

2.3.1 MUDANÇA DE REQUISITOS

Em se tratando de software, nem sempre é possível obter com exatidão todos os requisitos de um projeto antes de sua construção. Muitos fatores subjetivos conspiram neste sentido. Em geral, os requisitos mudam, uma vez que o cliente só consegue validar um processo após vê-lo implementado. Neste caso, é necessário saber o que fazer quando ocorre a mudança (BECK; ANDRES, 2004). Para Nurmuliani, Zowghi e Fowell (2004), é necessário adotar uma abordagem que possa gerenciar os impactos das mudanças. Para melhor gerenciar a mudança de requisitos, antes é necessário identificar suas principais causas.

As mudanças em requisitos de software durante o desenvolvimento de projetos complexos são inevitáveis e causadas por diversos fatores como: mudanças nos objetivos de negócio, nos requisitos de sistema, no ambiente organizacional, na legislação, dentre outros. Além disto, o próprio processo de software é dinâmico, trazendo mudanças nos requisitos durante o desenvolvimento do projeto. A mudança de requisitos tem sido apontada como sendo um dos principais desafios nos projetos de desenvolvimento de software, o que reforça que a gestão da mudança de requisitos continua sendo um problema a ser resolvido (NURMULIANI; ZOWGHI; FOWELL, 2004).

Conforme Kotonya e Sommerville (1998), os requisitos mudam por diversos motivos. Na medida em que os mesmos vão sendo desenvolvidos, o cliente obtém um melhor entendimento do que deseja no sistema, erros e inconsistências surgem, as prioridades do cliente mudam devido à alterações no ambiente de negócios, dentre outros.

Um dos objetivos da engenharia de requisitos tradicional é gerenciar as mudanças, registrando-as para permitir posterior rastreabilidade. Para Nuseibeh e Easterbrook (2000), é de importância crescente a necessidade de que os requisitos possam não somente ser escritos, mas também rastreados, para que se possa gerenciar sua evolução no tempo. Vários padrões

de documentação de requisitos já foram desenvolvidos, entretanto, sua estrutura necessita ser construída para cada contexto em particular. De acordo com Pinheiro (2004), o uso de modelos formais auxilia o processo de automatizar a rastreabilidade, através de sua geração automática, permitindo inclusive a criação de procedimentos que venham a permitir a verificação de sua consistência.

Nurmuliani, Zowghi e Fowell (2004) identificam quatro fases que integram o processo de gerência da mudança de requisitos:

- **Requisição inicial**: um membro da equipe submete uma proposta de mudança. O problema é identificado, analisado e descrito em termos da mudança proposta e impactos;
- **Validação e Avaliação**: a mudança é validada em termos de impactos em cronograma. A mudança é avaliada e revisada por aprovadores supervisionados pelo gerente de projeto;
- **Implementação**: a mudança aceita e aprovada é implementada e passa a ser parte integrante do sistema. O gerente de projeto define os responsáveis pela implementação;
- **Verificação**: é verificado se a mudança foi implementada corretamente. Quem verifica é o gerente de projeto ou equipe de qualidade (quando esta existir). Se a mudança foi implementada com sucesso, a requisição inicial é fechada, caso contrário, as mudanças precisarão de maior análise.

Este diagrama a seguir procura demontrar as quatro fases de Nurmuliani, Zowghi e Fowell (2004).

Figura 3. Fases da gerência de requisitos (NURMULIANI; ZOWGHI; FOWELL, 2004)

Segundo Kotonya e Sommerville (1998), o gerenciamento da mudança de requisitos compreende três etapas:

- **Análise do problema e especificação das mudanças**: o problema é identificado por meio de análise em documentos e de novas necessidades do cliente, dentre outros;

- **Análise das mudanças e custo**: as mudanças são analisadas. Nesta etapa são verificados componentes bem como outros requisitos impactados pelas mudanças para o cálculo de custo;
- **Implementação das mudanças**: as mudanças são implementadas e uma nova versão do documento de requisitos do sistema é gerada.

O diagrama a seguir procura demontrar as três fases de Kotonya e Sommerville (1998).

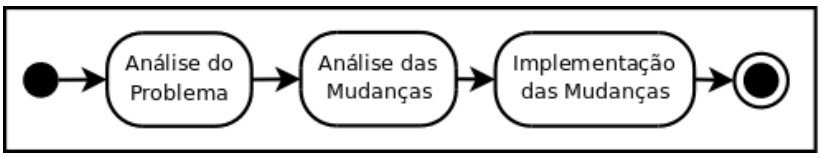


Figura 4. Etapas da gerência de requisitos (KOTONYA; SOMMERVILLE, 1998)

Nurmuliani, Zowghi e Fowell (2004) desenvolveram uma taxonomia para classificar as origens das mudanças de requisitos para um melhor entendimento de suas causas e melhorar o processo de gerência da mudança. A taxonomia foi desenvolvida por meio da análise e agrupamento e foram derivados a partir de formulários reais de solicitação de mudança de requisitos em uma grande companhia de desenvolvimento. A taxonomia é dividida em três dimensões (tipos de mudanças, motivos de mudanças e origens de mudanças).

Conforme a primeira dimensão da taxonomia proposta pela classificação de Nurmuliani, Zowghi e Fowell (2004), são considerados tipos de mudanças:

- Adição: um requisito é adicionado;
- Exlusão: um requisito é excluído;
- Mudança: um requisito é alterado;

Conforme a segunda dimensão da taxonomia proposta pela classificação de Nurmuliani, Zowghi e Fowell (2004), são considerados motivos de mudanças:

- Correção de defeito: correção de um defeito já existente;
- Requisito faltante: requisito não foi identificado inicialmente;
- Melhoria funcional: funcionalidade melhorada;
- Estratégia de produto: mudança originada por demandas de marketing;
- Melhoria de projeto: mudança originada por melhor entendimento da equipe sobre o

produto;

- Redução de escopo: funcionalidade removida devido à falta de recursos;
- Funcionalidade redundante: funcionalidade que já existe ou pode ser substituída por outra;
- Funcionalidade obsoleta: funcionalidade que não mais é necessária, não possui valor;
- Requisito errado: requisito identificado de forma errônea;
- Requisito redefinido: requisito reescrito para melhor entendimento;

Conforme a terceira dimensão da taxonomia proposta pela classificação de Nurmuliani, Zowghi e Fowell (2004), são considerados origens de mudanças:

- Relatórios de erros;
- Observações da gerência do projeto;
- Observações do grupo de marketing;
- Análise do desenvolvedor;
- *Feedback* do projeto;
- Discussão técnica;
- Revisão funcional;
- Proposta de revisão de funcionalidade;
- Discussões entre o cliente e a equipe;

2.3.2 ANÁLISE DE IMPACTO

De acordo com Arnold e Bohner (1993), a análise de impacto pode ser definida como a atividade que identifica o que deve ser alterado para que se implemente determinada mudança no sistema, identificando suas potenciais consequências. Dentre algumas formas de realizar a análise de impacto, pode-se citar: a utilização de listagens cruzadas, relacionamentos de rastreabilidade, consulta à especificação, dentre outros, para descobrir quais outras partes do sistema são afetadas pela mudança.

Para Knethen (2002), a análise de impacto identifica as entidades que são possivelmente afetadas por uma mudança proposta no sistema. Como um sistema consiste não

apenas de código, mas também de documentos contendo especificações de requisitos cujo nível de abstração é maior, esta análise em muitos casos é realizada de forma intuitiva pelos analistas, o que pode ser suficiente para pequenos sistemas, mas não para sistemas maiores.

Para Požgaj, Sertić e Boban (2003), os requisitos devem ser continuamente monitorados e supervisionados a fim de se descobrir a mudança de requisitos o mais cedo possível. Quando uma mudança ocorrer, esta deve ser analisada e seu impacto definido. A partir desta análise e das consequências das mudanças, define-se as ações a serem tomadas e se a mudança será implementada ou não.

A análise de impacto é tida como de sucesso quando consegue identificar corretamente o conjunto das partes afetadas por uma mudança no sistema e é tida como falha se os testes revelarem que outras partes do sistema também necessitam ser alteradas. O objetivo principal da análise de impacto é minimizar os efeitos colaterais inesperados de uma mudança (QUEILLE et al., 1994).

A análise de impacto determina o que deve ser alterado em duas direções, a primeira é o código-fonte e determina quais elementos de um programa podem ser afetados por uma mudança e utiliza relações de dependência de dados e de controle. A segunda utiliza os relacionamentos de rastreabilidade entre os artefatos criados durante o desenvolvimento.

De acordo com Kotonya e Sommerville (1998), são seis as atividades decorrentes da etapa "Análise das Mudanças e custo", de seu modelo, que corresponde à análise de impacto:

1. Verificar se a solicitação de mudança é válida;
2. Identificar os requisitos afetados diretamente;
3. Descobrir os requisitos dependentes;
4. Propor mudanças nos requisitos;
5. Estimar os custos das mudanças;
6. Avaliar se os custos são aceitáveis;

As seis atividades propostas por Kotonya e Sommerville (1998) são vistas a seguir.

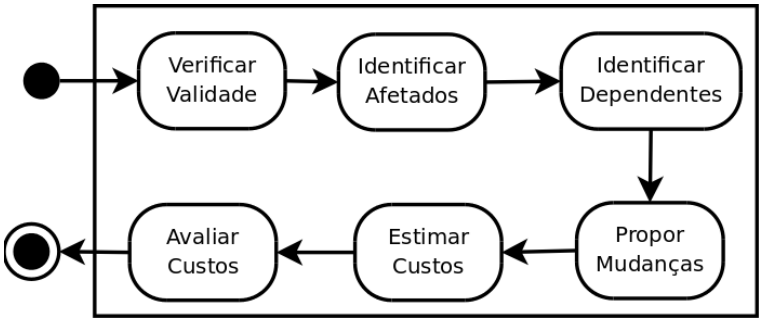


Figura 5. Etapas de uma análise de impacto (KOTONYA; SOMMERVILLE, 1998).

Para De Lucia, Fasano e Oliveto (2008), o processo de análise de impacto (conforme visto na figura a seguir) inicia analisando as requisições de mudança e a documentação para identificar o grupo inicial de artefatos (*starting impact set*) que podem ser afetados pela mudança. Após formar o grupo inicial, estes artefatos são então analisados para identificar outros artefatos candidatos que podem ser afetados pela mudança, formando um grupo de candidatos de impacto (*candidate impact set*). A partir da implementação da mudança, o conjunto de artefatos modificados é chamado de grupo atual de impacto (*actual impact set*).

A análise de impacto é um processo iterativo no qual novos artefatos afetados podem ser descobertos durante a implementação de uma mudança, formando um grupo de impacto descoberto (*discovered impact set*). Ao mesmo tempo, artefatos que não são afetados pelas mudanças podem ser relacionados de forma errada, constituindo um grupo de falso-positivos (*false positive impact set*).

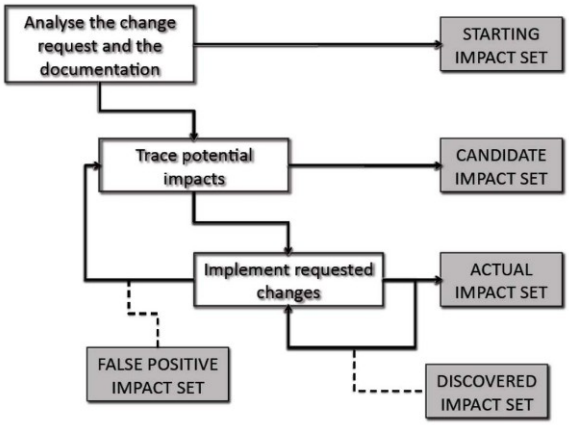


Figura 6. Processo da análise de impacto (DE LUCIA; FASANO; OLIVETO, 2008)

Para Huang e Song (2006), as técnicas de análise de impacto podem ser divididas em estáticas e dinâmicas. As técnicas estáticas podem ser divididas em duas categorias:

- **Baseadas em rastreabilidade**: determinam o impacto por meio dos *links* de rastreabilidade definidos desde os requisitos até o código-fonte, passando pelo projeto;
- **Baseadas em análise de dependência**: analisam a sintaxe do programa procurando por dependências semânticas entre as entidades do programa e calculam o impacto com base nestas dependências;

Diferentemente das técnicas de análise de impacto estáticas, as técnicas dinâmicas são baseadas no comportamento do sistema, coletado durante sua execução. Na prática, as

técnicas dinâmicas inserem pedaços de código nas aplicações para coletar informações de interesse durante a sua execução. Além disto, existem técnicas para análise de impacto específicas para sistemas orientados a objeto, cuja descoberta do impacto é baseada nas características das linguagens orientadas a objeto, tais como: herança, agregação, composição, dentre outras. Estas técnicas podem se tornar extremamente complexas devido às propriedades envolvidas em alguns relacionamentos, principalmente a transitividade. Já as técnicas para análise dinâmica de impacto determinam o impacto da mudança com base nos dados coletados durante a execução do software (HUANG; SONG, 2006).

Para Kung et al. (1994), algumas características da orientação a objetos como encapsulamento, agregação, herança, polimorfismo, dentre outros, tornam mais complexa a identificação dos partes afetadas de um software (classes, métodos), quando da ocorrência de mudanças de requisitos. Existem dependências de dados, de controle e de comportamento entre classes. Além disto, alguns relacionamentos como herança e agregação são transitivos, o que geram dependências também transitivas. Portanto se torna complexo compreender e entender o efeito combinado que estas estruturas exercem.

2.4 RASTREABILIDADE

Para acompanhar o ritmo do ambientes de negócios, projetos de software devem lidar com a constante mudança de objetivos e necessidades do cliente. A maioria dos sistemas muda durante seu ciclo de vida. Quando esta mudança ocorre é necessário estimar o custo e o escopo da mudança. Tradicionalmente, esta análise de impacto tem sido conduzida de forma intuitiva pelos profissionais de software, por meio da análise de código-fonte e da documentação. Esta abordagem pode ser suficiente para projetos pequenos, mas não para projetos complexos. Além disto, investigações empíricas demonstram que mesmo profissionais experientes realizam estimativas erradas, apontando um conjunto incompleto de mudanças (KNETHEN, 2002).

Portanto, é necessário ouvir o cliente durante todas as fases do desenvolvimento, tornando as decisões de design, bem como a especificação de requisitos rastreáveis. A rastreabilidade é fator chave para proporcionar o rápido acesso à informação, diminuindo o impacto da mudança (LEE; GUADAGNO; JIA, 2003). A rastreabilidade de requisitos

consiste em ligações entre as informações produzidas no desenvolvimento de software. Para Pinheiro (2004), tais ligações são essenciais no desenvolvimento de sistemas, uma vez que é grande o volume de informações produzidas e estas devem estar relacionadas, ainda mais em ambientes distribuídos com várias equipes. Neste contexto, a rastreabilidade é inevitável, pois precisamos dela para procurar informações que nos ajudem a tomar decisões e a descobrir fatos como investigar as causas de um erro em testes de aceitação, estudar o impacto da mudança de um requisito, dentre outros.

De acordo com Pinheiro (2004), a rastreabilidade é uma tarefa auxiliar que serve tanto para propósitos técnicos quanto gerenciais. Rastrear significa procurar, seguir um caminho, manter o histórico como em um mapa, onde podemos ir e voltar de coordenada em coordenada, desde suas origens até o seu desenvolvimento. A rastreabilidade deve ser produzida de maneira natural, dentro do processo e deve permitir ao interessado visualizar somente os rastros de seu interesse.

Para Pinheiro (2004), três aspectos devem ser cobertos pela rastreabilidade: a definição (o que será rastreável), a produção (registro dos objetos e seus relacionamentos) e a extração (processo de seleção e visualização do que foi rastreado).

Sayão e Leite (2006) destaca como sendo vantagens advindas do uso da rastreabilidade:

- Verificação de alocação entre requisitos e sua implementação;
- Identificação das origens de requisitos conflitantes;
- Verificação de requisitos para os quais não foram previstos testes;
- Validação do sistema, verificando se o mesmo atende ao conjunto de requisitos proposto;
- Estimativas de custos de prazos quando da inserção de uma nova funcionalidade;
- Identificação de riscos que possam impactar os requisitos;
- Visualização de relacionamentos entre hardware e software que auxiliam em mudanças de ambiente operacional;
- Identificação de ligações entre código e documentos de análise que possam proporcionar futuro reaproveitamento de código;

Para Hayes, Dekhtyar e Sundaram (2006), o processo de rastreabilidade compreende as atividades a seguir. Toma-se por base que a rastreabilidade está sendo elaborada de forma

póstuma, ou seja, os requisitos já existem e está sendo realizada a partir dos requisitos de alto nível para os de baixo nível.

1. Identificar cada requisito;
2. Atribuir um identificador único para cada requisito;
3. Para cada requisito de alto nível, localizar os de baixo nível;
4. Para cada requisito de baixo nível, localizar os requisitos-pai;
5. Determinar se cada requisito de alto nível foi satisfeito;
6. Elaborar um relatório para exibir a matriz de rastreabilidade;
7. Preparar um relatório resumo que apresenta o nível de rastreabilidade;

Para Pinheiro (2000), as principais técnicas de rastreabilidade são:

- **Matriz de rastreabilidade**: Utilizada para relacionar requisitos com outros artefatos. Geralmente os requisitos são listados nas linhas e os artefatos (especificação, programas) nas colunas. Para cada interseção (encontro) é realizada uma marca identificando o relacionamento. Existem matrizes mais sofisticadas que utilizam diferentes marcas ou pesos para os relacionamentos.
- **Referências cruzadas ou índices**: São referências construídas ao longo dos artefatos para indicar relacionamentos entre os mesmos.

2.4.1 DIREÇÕES DA RASTREABILIDADE

Os requisitos são vinculados aos documentos que os geraram, pois tais documentos podem conter a descrição exata dos motivos que levaram à criação de tal requisito, bem como podem conter informações valiosas acerca deste requisito. Nestes casos, ambos devem ser relacionados para que um leve ao outro, e para que possamos seguir os rastros deixados pelos requisitos em outros documentos (pós-rastreabilidade), bem como os rastros deixados por outros documentos nos requisitos (pré-rastreabilidade). O mesmo acontece entre os requisitos e as decisões de *design* e os testes de aceitação, os quais só existem devido aos requisitos que os geraram (PINHEIRO, 2004).

A pré-rastreabilidade ou rastreabilidade para trás (*backward traceability*), permite que se chegue até as origens dos requisitos (pessoas, processos, documentos), a partir de artefatos

de *design*. É útil quando os requisitos mudam e não se entende os motivos que levaram à isto. Desta forma, podemos investigar os documentos e decisões que levaram à especificação deste requisito (PINHEIRO, 2004).

A pós-rastreabilidade ou rastreabilidade para frente (*forward traceability*) permite que se chegue até a implementação e componentes de *design*, a partir dos requisitos. É útil quando mudam os requisitos e desejamos visualizar qual será o impacto da mudança no *design* e nos testes para que possamos antever possíveis mudanças nestas rotinas (PINHEIRO, 2004).

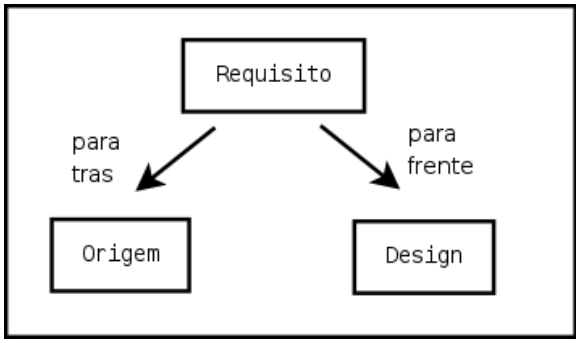


Figura 7. Direções da rastreabilidade (PINHEIRO, 2004)

2.4.2 RASTREABILIDADE VERTICAL E HORIZONTAL

A rastreabilidade pode ser horizontal e vertical. A rastreabilidade vertical está relacionada com a capacidade de relacionar artefatos dependentes dentro de um modelo, enquanto que a rastreabilidade horizontal está relacionada à habilidade de relacionar artefatos entre diferentes modelos. Entre os artefatos que podem estar relacionados estão requisitos, artefatos de análise, de *design*, código-fonte, casos de teste, dentre outros.

Para De Lucia, Fasano e Oliveto (2008), a rastreabilidade vertical fornece apenas uma visão limitada sobre os artefatos afetados por uma mudança. Sistemas complexos necessitam de modelos com vários níveis de abstração, como por exemplo, o código-fonte. Desta forma, muitas abordagens acabam estendendo a rastreabilidade vertical com a horizontal, permitindo gerenciar dependências entre requisitos e artefatos de design, entre requisitos e o código-fonte, entre requisitos e casos de teste, e entre artefatos de design e o código-fonte. Na figura a seguir, tem-se uma representação da rastreabilidade horizontal.

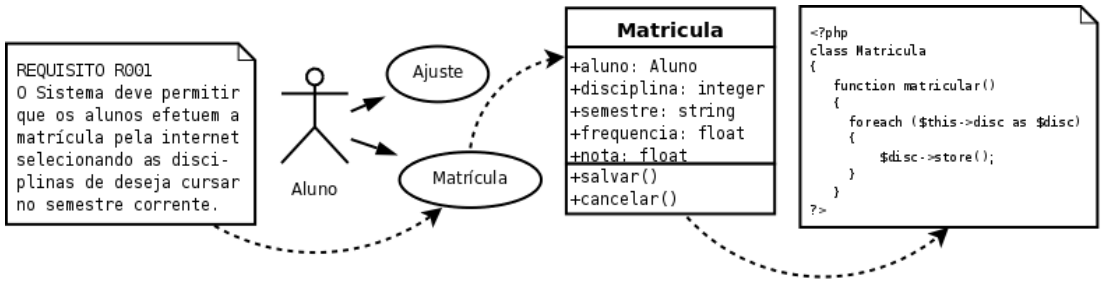


Figura 8. Representação da rastreabilidade horizontal

Para De Lucia, Fasano e Oliveto (2008), alguns relacionamentos podem ser derivados por meio da análise de artefatos, observando sua sintaxe e sua semântica. Relacionamentos do tipo de herança, composição, agregação entre classes de um sistema orientado a objetos podem ser descobertas por meio de uma leitura (*parsing*) do código-fonte de acordo com a gramática da linguagem.

Para De Lucia, Fasano e Oliveto (2008), a informação de rastreabilidade pode se tornar rapidamente desatualizada devido à natureza evolucionária do software e a matriz de rastreabilidade pode se tornar incompleta devido a relacionamentos novos ainda não registrados e também podem conter relacionamentos que não existem mais.

2.4.3 RASTREABILIDADE FUNCIONAL E NÃO-FUNCIONAL

A rastreabilidade funcional é relacionada aos aspectos do desenvolvimento de software que podem ser descritos em termos de estados, testes, modelos, diagramas, artefatos e notações. A rastreabilidade é derivada das interligações entre estes modelos. Dentre estes, podemos citar os modelos de análise, que referenciam requisitos como entrevistas e documentos (memorandos, manuais), e os modelos de *design* que referenciam classes, atributos, métodos e sequências (PINHEIRO, 2004).

A rastreabilidade não-funcional está relacionada à aspectos de qualidade e à conceitos intangíveis, tais como objetivos, metas, decisões e intenções. Geralmente são vinculados à requisitos não-funcionais. Dentre os aspectos não-funcionais que merecem destaque, estão a definição de metas, a interpretação e justificativa de artefatos, descrição do ambiente de desenvolvimento, bem como do ambiente de negócios, incluindo aspectos sociais. Para Pinheiro (2004), estes precisam ser traduzidos para uma forma funcional para que possam ser

verificados, através de modelos organizacionais que reflitam aspectos políticos, metas e papéis.

2.4.4 ASPECTOS DA RASTREABILIDADE

Para Pinheiro (2004), três aspectos devem ser cobertos pela rastreabilidade: a definição (o que será rastreável), a produção (registro dos objetos e seus relacionamentos) e a extração (processo de seleção e visualização do que foi rastreado).

Na definição devemos escolher quais tipos de caminhos devem existir entre os objetos a serem rastreados, qual o significado de suas ligações e sob quais condições eles serão rastreados. O significado das ligações é importante para evitar má interpretação, que pode causar problemas maiores do que a falta de rastreabilidade. A produção ocorre quando da existência de um evento. Tão cedo isto acontecer, ele deve ser registrado.

A extração consiste em meios de se obter a informação registrada. Para Pinheiro (2004), a extração pode ser:

- **Seletiva**: restringe a rastreabilidade à certos padrões de objetos e seus relacionamentos, contexto ou fase do desenvolvimento.
- **Interativa**: permite navegar entre os objetos relacionados, guiado pelos seus relacionamentos. Neste caso, pode-se utilizar marcações do tipo <*traces-from*> e <*traces-to*> na especificação dos requisitos.
- **Não-guiada**: permite ao usuário ir de um objeto ao outro, investigando seu conteúdo, quando se tem pouca informação sobre como prover rastreabilidade.

2.4.5 DESAFIOS

A rastreabilidade de requisitos é reconhecida como um fator importante no desenvolvimento e manutenção de software. Apesar disto, a falta de um suporte automatizado pode fazer com que a tarefa de manter a informação de rastreabilidade tome tempo em demasia. Desta forma, a informação de rastreabilidade acaba por se tornar desatualizada ou mesmo ausente durante o processo de desenvolvimento de software (OLIVETO et al., 2007).

A rastreabilidade entre o código-fonte e documentos de alto nível como a especificação de requisitos pode auxiliar na compreensão do sistema, permitindo mapear conceitos definidos em termos abstratos em artefatos de nível mais detalhado. Desta forma, a rastreabilidade colabora também com a manutenibilidade do software, permitindo localizar pedaços de software relacionados com documentos de alto nível, auxiliando também na análise de impacto (OLIVETO et al., 2007).

Para Oliveto et al. (2007), ainda é considerado um desafio para a rastreabilidade a definição do nível apropriado de granularidade a ser utilizada entre os *links* de relacionamento do projeto e também quais são os fatores que afetam a granularidade destes *links*.

Conforme Oliveto et al. (2007), outro fator considerado um desafio para a rastreabilidade é a evolução dos artefatos de um projeto. Durante o processo de desenvolvimento, *links* se tornam inválidos e novos *links* surgem em consequência de mudanças realizadas no projeto. A maioria das ferramentas de gerenciamento de configuração trata o problema de versionamento dos artefatos sem considerar a evolução dos *links* de rastreabilidade entre eles. Desta forma, ainda é um tópico em aberto o desenvolvimento de sistemas de gerenciamento da mudança que suportem de forma efetiva a evolução dos *links* de rastreabilidade entre os artefatos de software.

Muitos foram os métodos e ferramentas propostos para suportar a recuperação automática de *links* de rastreabilidade. Entretanto tais métodos e ferramentas falharam na automatização completa de tal processo, uma vez que estes podem detectar *links* falsos, bem como deixar de detectar *links* verdadeiros (OLIVETO et al., 2007).

2.5 DESENVOLVIMENTO DISTRIBUÍDO DE SOFTWARE

Para Lemos e De Souza (2008), a indústria de software atualmente recebe uma crescente demanda por soluções, ao mesmo tempo em que sofre com a falta de mão-de-obra qualificada. Estes fatores, aliados com a forte globalização, tem levado as empresas a buscarem soluções inclusive em outros países. Este novo cenário tem motivado o desenvolvimento de uma nova área da Engenharia de Software preocupada com o desenvolvimento distribuído de software, que se caracteriza pela distância física ou temporal entre os envolvidos no processo, sejam clientes, analistas ou desenvolvedores.

Para Lloyd, Rosson e Arthur (2002), o desenvolvimento de projetos de forma distribuída tem se tornado cada vez mais factível graças a melhorias nas estruturas de comunicação. Diversos fatores colaboram com o desenvolvimento distribuído, como custos de deslocamento e falta de profissional qualificado localmente. Para Espindola et al. (2005), o crescente desenvolvimento desta área, tem levado à inúmeras pesquisas com o objetivo de minimizar os problemas relativos ao desenvolvimento de software, como a distância geográfica.

2.5.1 DESAFIOS

A disciplina de engenharia de requisitos merece uma atenção especial no contexto do desenvolvimento distribuído de software, uma vez que se trata de uma atividade complexa e que envolve interação entre pessoas com culturas e conhecimentos diferentes. Para Brito e Vasconcelos (2006), apesar da engenharia de requisitos ser um processo associado com aquisição e representação de informações, as ferramentas que a apóiam oferecem pouco suporte à colaboração.

Em se tratando de engenharia de requisitos, um dos problemas mais complexos para resolver é a distância na comunicação entre os membros da equipe. Os analistas utilizam termos técnicos, ao passo que o cliente utiliza uma linguagem natural comum ao ambiente de negócios. Estas diferenças podem fazer com que os analistas não tenham um claro entendimento sobre o que o sistema que está sendo projetado deve realizar. Este entendimento errado muitas vezes é percebido somente durante a implantação do produto final (MANNIO; NIKULA, 2001).

Para Lemos e De Souza (2008), a engenharia de requisitos já é suficientemente complexa quando realizada localmente. No cenário distribuído, é necessário lidar com diferentes aspectos culturais, temporais e geográficos. A distância, por exemplo, aumenta a lacuna existente entre os analistas de requisitos e os desenvolvedores, e também entre os analistas e os clientes, uma vez que a comunicação diminui devido à distância. Neste caso, aumentam as possibilidades de mal-interpretação dos requisitos, fazendo com que estes sejam de pouca valia para o cliente final. Além disto, existe uma forte necessidade de ferramentas, técnicas e metodologias que apóiem as demais etapas do desenvolvimento distribuído, como

elicitação, negociação, e modelagem dos requisitos.

Alguns dos problemas inerentes ao desenvolvimento distribuído são: a comunicação inadequada, a gerência ineficiente do conhecimento, a diversidade cultural e a diferença de fuso-horário. Estes problemas trazem uma série de dificuldades para a engenharia de requisitos que podem ser minimizados quando adotada a metodologia adequada (LEMOS; DE SOUZA, 2008). Para Lloyd, Rosson e Arthur (2002), das diversas técnicas existentes para elicitação de requisitos existentes, algumas funcionam muito bem em um ambiente distribuído, enquanto outras falham em virtude de limitações tecnológicas ou de interação. Técnicas que se utilizam de grande interação social, como a negociação, apresentariam problemas quando da utilização de meios de comunicação mais simples, como o textual.

2.6 MODELAGEM ORIENTADA A AGENTES

A utilização de sistemas multi-agentes tem crescido e vem sendo considerado um novo paradigma de computação (DÁRIO, 2005). Um agente representa uma abstração de software que não pode ser definido somente em termos de atributos ou métodos, como um objeto. Para Falbo, Pezzin e Schambach (2004), um agente é uma entidade de software autônoma com um comportamento definido por metas, capaz de perceber o ambiente que o cerca e responder a mudanças, usando sua própria experiência ou conhecimento sobre este ambiente. O crescimento deste tipo de sistema contribuiu para o surgimento da Engenharia de Software Orientada a Agentes (ESOA) e uma série de metodologias para desenvolvimento de sistemas.

Sistemas multi-agentes são sistemas compostos por múltiplos elementos de software chamados de agentes. Um agente é um sistema com pelo menos duas capacidades: a capacidade de autonomia, podendo decidir o que fazer para atingir algum objetivo pré-determinado, e a capacidade de interagir socialmente com outros agentes, não somente trocando dados, mas também executando atividades de cooperação, coordenação e negociação (WOOLDRIDGE, 2002).

Para Wooldridge (2002), qualquer sistema de controle, pode ser visto como um agente. Um termostato, por exemplo, monitora o ambiente e apresenta a temperatura. A partir deste dado, ações podem ser tomadas no sentido de aquecer ou esfriar o ambiente. A maioria dos serviços de software que funcionam em *background* como processos, monitorando o ambiente

e realizando ações de notificação, podem também ser vistos como agentes. De forma geral, agentes podem ser vistos simplesmente como programas de computador capazes de uma ação autônoma em um dado ambiente para satisfazer os objetivos definidos. Apesar de estes exemplos serem considerados agentes, não são agentes inteligentes, são considerados agentes reativos. Agentes reativos baseiam suas decisões inteiramente no presente (estado atual), sem fazer referência à algum estado passado. Nestes casos, estes agentes simplesmente respondem ao ambiente.

Um agente inteligente precisa apresentar algumas capacidades básicas, entre elas:

- **Reatividade**: Capacidade de perceber o ambiente e responder a mudanças para satisfazer objetivos pré-definidos;
- **Pró-Atividade**: Capacidade de tomar a iniciativa para satisfazer aos objetivos pré-definidos;
- **Habilidade Social**: Capacidade de interagir com outros agentes para satisfazer os objetivos pré-definidos;

2.6.1 METODOLOGIAS

Algumas metodologias voltadas ao projeto de sistemas orientados a agentes surgiram ao longo do tempo, sendo que algumas merecem destaque: GAIA, Message/UML, Tropos, Prometheus e MaSE. Tais metodologias consistem em um conjunto de modelos e uma seqüência de atividades. A seguir, serão expostas algumas das principais características de cada metodologia.

2.6.1.1 GAIA

GAIA é uma metodologia de uso geral, aplicável à uma gama de domínios de aplicação. Para GAIA, um sistema multi-agente é visto como uma organização composta de vários papéis. A metodologia provê um *framework* conceitual que permite ao analista sistematicamente refinar requisitos de maneira tal que estes possam ser diretamente implementados, detalhando papéis, interações, responsabilidades, permissões e propriedades

na fase de análise (WOOLDRIDGE, 2002).

Apesar de ser uma metodologia de uso geral, GAIA é voltada aos aspectos organizacionais e fornece um formalismo que dá uma ênfase muito maior à etapa de análise, sendo que a etapa de projeto provê apenas um detalhamento mais refinado, produto do trabalho já realizado. Para Caire (2001), a etapa de projeto não torna claro como métodos da orientação a objetos poderão ser utilizados para implementar a solução proposta.

2.6.1.2 MESSAGEUML

MessageUML é uma metodologia construída sobre práticas existentes da engenharia de software. Cobre análise e projeto de sistemas multi-agentes e define conceitos e uma notação baseada em UML. A metodologia procura estender a UML, adicionando uma "camada de conhecimento" de conceitos para agentes.

Os principais conceitos da UML utilizados para definir uma mensagem são: ação, evento e estado. Em MessageUML, o mundo é visto como um conjunto de máquinas de estado (CAIRE, 2001). Em um nível mais conceitual, a metodologia procura classificar as entidades como: organizações, papéis, recursos, tarefas, interações e objetivos.

2.6.1.3 TROPOS

TROPOS é uma metodologia que cobre as primeiras fases da análise de requisitos, permitindo um profundo entendimento do ambiente no qual o software irá interagir. Provê uma linguagem formal e modelos que representam as intenções sociais utilizadas para dar suporte ao desenvolvimento do sistema. Adota o modelo *i**, por meio do qual permite representar atores, que podem ser agentes, papéis ou posições, objetivos e dependência entre atores. Para a fase de design, a metodologia sugere a utilização de outras abordagens como a MessageUML (GIORGINI et al., 2003).

2.6.1.4 PROMETHEUS

Prometheus é uma metodologia que procura cobrir o ciclo completo de desenvolvimento. No lugar de tratar agentes como processos de software, suporta o desenvolvimento de sistemas inteligentes com objetivos, crenças, planos e eventos, ou seja, possui um foco no desenvolvimento de agentes BDI, sendo de menor utilidade para modelar outros tipos sistemas multi-agentes. Provê um processo detalhado no qual o produto de cada fase pode ser derivado da fase anterior (PADGHAM; WIKIKOFF, 2002).

2.6.1.5 MASE

MaSE é uma metodologia que se caracteriza por ser independente de arquitetura e linguagem de programação. Na metodologia MaSE, os agentes são vistos como processos de software que interagem uns com os outros para atingir os objetivos do sistema, definição abrangente e de acordo com o tipo de agente proposto no presente trabalho. Além disto, a metodologia apresenta um conjunto de diagramas que permitem retratar com um grande grau de fidelidade os agentes reativos.

Para De Loach (2000), ao longo dos últimos anos, houve muitas tentativas de criar ferramentas e metodologias para o desenvolvimento de sistemas multi-agentes. Entretanto, a maioria delas focou muito mais em descrições abstratas e conceitos sobre arquiteturas específicas de agentes do que em uma metodologia de projeto. MaSE e Prometheus, por outro lado, procuram cobrir todo o ciclo de vida do desenvolvimento de um sistema multi-agente, desde a análise, o projeto e até o desenvolvimento. Como o sistema modelado no presente trabalho não envolve agentes BDI, a metodologia MaSE torna-se mais adequada para os propósitos do presente trabalho, sendo assim, a metodologia escolhida.

2.7 WEB SERVICES

Web Services são serviços disponibilizados pela internet, por meio de um conjunto de tecnologias independentes de plataforma, que permite interoperabilidade através da entrega de

serviços (VAUGHAN, 2002) e a comunicação entre aplicações modularizadas, que podem ser descritas, publicadas e invocadas pela internet para utilização imediata, ou mesmo composição de novos serviços (HANSEN et al., 2002) intregrando tudo através de padrões abertos e conhecidos como XML, SOAP, WSDL e UDDI (CHUNG; LIN; MATHIEU, 2003).

A figura a seguir procura transmitir a idéia geral da arquitetura de um *Web Service*, formada por um conjunto de tecnologias de padrão aberto, que interagem sob uma plataforma de internet. No meio de todo o processo, provendo a comunicação entre as aplicações, está o protocolo SOAP (*Simple Object Access Protocol*). SOAP é um protocolo herdeiro do padrão XML que encapsula um conjunto de regras para descrição de dados e processos, através de um mecanismo simples para definir a semântica de uma aplicação através de um modelo de empacotamento e um mecanismo de codificação (Chavda, 2004). É projetado para a troca de informações em um ambiente descentralizado (HANSEN et al., 2002) através do protocolo de comunicação HTTP e do formato XML (VAUGHAN, 2002). Dessa forma, para uma aplicação trabalhar com WS, basta a compatibilidade com SOAP, tanto no lado do cliente (criando o documento XML com a informação necessária para invocar o serviço) quanto no lado do servidor (responsável por executar a mensagem como um interpretador) (HANSEN et al., 2002).

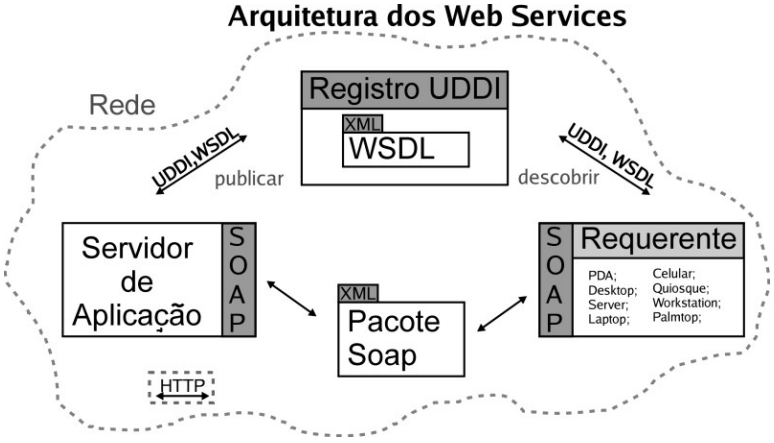


Figura 9. Arquitetura dos Web Services

3 TRABALHOS RELACIONADOS

3.1 REQUISITOS

Neste capítulo serão apresentados alguns requisitos para o desenvolvimento da ferramenta proposta pelo presente trabalho. Estes requisitos foram coletados a partir de revisão bibliográfica e constituem uma base sólida de três partes que procuram demonstrar cada um dos aspectos presentes na aplicação proposta: gerenciamento de requisitos, gerenciamento de colaboração e gerenciamento de rastreabilidade.

3.1.1 REQUISITOS DE GERENCIAMENTO

Para Hoffmann et al. (2004), a utilização de sistemas para gerenciamento de requisitos auxiliam em manter a especificação consistente, atualizada e acessível. Para o autor, existem muitos requisitos que devem ser considerados no desenvolvimento deste tipo de sistema. Desta forma, apresenta um catálogo de requisitos para o desenvolvimento de ferramentas voltadas para o gerenciamento de requisitos.

O trabalho de Hoffmann et al. (2004) procura elencar as características essenciais que devem estar presentes em ferramentas para gerenciamento de requisitos. De maneira geral, tais ferramentas devem suportar a especificação e agrupamento de requisitos, suportar derivação para níveis mais detalhados, permitir relacionar requisitos com artefatos de design e permitir o desenvolvimento distribuído na organização. Os requisitos apresentados variam conforme fatores como o domínio do produto, o tipo de projeto e o tamanho da organização.

Para Hoffmann et al. (2004), há uma necessidade de ferramentas para gerenciamento de requisitos, principalmente em sistemas complexos que trabalham com um conjunto grande de requisitos. Para estes, há uma gama de ferramentas já disponíveis no mercado, cada qual com suas características, capacidades, conceitos e grau de maturidade. Para facilitar a escolha de ferramentas deste gênero, Hoffmann et al. (2004) elaborou um catálogo de características funcionais para ferramentas de gestão de requisitos. Dentre alguns requisitos funcionais identificados, destacam-se:

- Suportar diferentes modelos de requisitos;
- Suportar diferentes visões sobre a informação de rastreabilidade;
- Suportar requisições de mudanças (*workflow*);
- Suportar versões dos artefatos (*baseline)*;
- Suportar rastreabilidade entre requisitos;
- Suportar interfaces com outras ferramentas;
- Permitir trabalho colaborativo;
- Prover funções de análise de progresso do projeto;

Além destes, Hoffmann et al. (2004) destaca requisitos não-funcionais:

- Permitir administração centralizada;
- Permitir criar usuários, papéis e permissões;
- Permitir ilimitado número de requisitos e usuários;
- Permitir adaptabilidade e extensibilidade;
- Utilizar tecnologia escalável de banco de dados;
- Suportar uso *offline*;
- Suportar interface web;
- Suportar transações;
- Suportar criptografia;

Para melhorar o suporte ao gerenciamento de informações em ambientes de desenvolvimento distribuído de software, (BRITO; VASCONCELOS, 2006) propõe a integração de ferramentas de groupware com ferramentas de gerenciamento de requisitos. Para tal, uma aplicação foi desenvolvida e incorporada à ferramenta eGroupware (2009), de código-livre. Antes de realizar tal integração, o autor procurou avaliar ferramentas de gerenciamento de requisitos para realizar um levantamento prévio do conjunto de características necessárias para o desenvolvimento da ferramenta proposta. Neste estudo foram identificadas as seguintes características:

- Permitir a manutenção dos requisitos por meio da web, facilitando o acesso distribuído;
- Permitir a autoria cooperativa, possibilitando o acesso de múltiplos usuários na edição de requisitos;

- Permitir a troca de mensagens entre os participantes por meio de fóruns e mensagens instantâneas;
- Permitir registro de comentários, dúvidas e anotações referentes a um determinado requisito;
- Permitir coordenação e acompanhamento da execução das tarefas;
- Permitir suporte à execução do processo, como o registro de listas de itens a fazer e de reuniões;
- Permitir controle de acesso e permissões para manipulação dos requisitos;
- Permitir *baseline* e controle de versões;
- Permitir base de conhecimento, disponibilizando informações sobre o domínio do problema;
- Permitir manutenção do raciocínio, por meio da vinculação com arquivos e discussões de origem;
- Permitir histórico de alteração, registrando quem, por que, quando e como a operação foi realizada;
- Permitir consultar o estado do requisito e seu progresso (proposto, aceito, etc.);
- Permitir consultar requisitos de acordo com critérios (data, responsável, etc.);

3.1.2 REQUISITOS DE COLABORAÇÃO

O processo de especificação de requisitos de software envolve uma gama de colaboradores com os mais diversos papéis. Em decorrência da complexidade envolvida na comunicação, é fundamental que o seu gerenciamento seja um processo colaborativo suportado por uma ferramenta informatizada (LANG; DUGGAN, 2001).

Apesar da importância do trabalho cooperativo ser largamente reconhecida, poucas ferramentas para gestão de requisitos oferecem suporte à esta abordagem, sendo que a maioria é focada no trabalho individual. Mesmo entre aquelas ferramentas que oferecem suporte ao trabalho colaborativo, poucas são adequada para o uso em equipes multidisciplinares e distribuídas, onde os envolvidos possuem diferentes necessidades. Além disto, pesquisas revelam que há pouca adoção em nível industrial de ferramentas automatizadas que suportem o desenvolvimento colaborativo de software (LANG; DUGGAN, 2001).

Para Togneri, Falbo e Menezes (2002), a engenharia de requisitos é uma atividade que envolve intensa comunicação e interação entre os membros da equipe. Apesar da complexidade inerente ao processo da engenharia de requisitos em si, parte das falhas ocorridas nestas atividades é decorrente da dificuldade da equipe trabalhar de forma cooperativa, tendo em vista que o profissional envolvido neste processo não tem um perfil que privilegie o compartilhamento de informações e a interação. Portanto estes aspectos podem ficar relegados à um segundo plano, o que tende a piorar a medida em que a equipe cresce. Tendo em vista este cenário, uma ferramenta computacional adequada ao trabalho cooperativo poderia minimizar tais problemas.

Para Lang e Duggan (2001), há muitas ferramentas no mercado que dizem suportar o processo de gerenciamento de requisitos. Porém, para que seja possível conduzir uma avaliação, antes é necessário identificar quais são os requisitos de uma ferramenta para gerenciamento de requisitos colaborativa para então avaliar se estas características estão presentes em ferramentas comerciais. Para tal, o autor destaca as seguintes características:

- Manter descrições únicas de todos os requisitos;
- Classificar requisitos em grupos lógicos definidos pelo usuário;
- Especificar requisitos de forma textual e gráfica;
- Definir relacionamentos rastreáveis entre os requisitos;
- Verificar atribuições de requisitos à especificações de *design* feitas pelo usuário;
- Manter arquivo das versões de mudanças;
- Permitir mecanismo para autenticação e aprovação de mudanças;
- Suportar segurança e concorrência;
- Suportar trabalho cooperativo multidisciplinar;
- Suportar trabalho distribuído;
- Suportar sistemas padrão de modelagem;
- Manter repositório compartilhado de componentes e requisitos;
- Gerar relatórios;
- Gerar documentos padrão para apresentação;
- Suportar protocolos de comunicação com outras ferramentas;

Para Togneri, Falbo e Menezes (2002), a maioria dos softwares suportam a interação entre humano-sistema, sendo que poucos oferecem suporte adequado para a interação

humano-humano. Para suportar atividades em grupo, o autor destaca como sendo necessário.

- **Comunicação mediada por computador**: permite a troca de informações entre pessoas de forma síncrona ou assíncrona;
- **Coordenação mediada por computador**: permite o gerenciamento das atividades da equipe, garantindo a execução das atividades e o alcance dos objetivos;
- **Cooperação mediada por computador**: consiste na cooperação entre os membros na execução de uma atividade para resolução de um problema por meio da comunicação;
- **Ciência**: permite que cada membro da equipe esteja ciente das atividades desempenhadas pelos outros, para que possa ajustar a execução das suas;
- **Gerenciamento do conhecimento**: permite a gerência do conhecimento com vistas à seu reuso, minimizando a perda da informação e permitindo identificar cada histórico de decisões por traz de uma atividade;

3.1.3 REQUISITOS DE RASTREABILIDADE

Muitas organizações falham na tentativa de implementar práticas de rastreabilidade devido à dificuldades de manter os relacionamentos entre os requisitos ou por que percebem pouco retorno na sua adoção. Geralmente os relacionamentos entre os requisitos são gerenciados por planilhas, bancos de dados ou ferramentas comerciais. Estes relacionamentos tendem a se deteriorar a medida que o projeto evolui e a equipe falha em mantê-los atualizados devido à pressões de tempo. Relacionamentos mantidos manualmente tendem a se tornarem inconsistentes e incompletos.

As ferramentas atuais oferecem poucos recursos em relação à criação e manutenção dos relacionamentos. Utilizam-se técnicas de *drag-and-drop* para criação e marcação de links para os quais os requisitos associados foram modificados, sugestão de relacionamentos desatualizados, dentre outros. Mesmo assim, este tipo de suporte não é suficiente e os usuários acabam tendo relatórios contendo uma grande quantidade de relacionamentos desatualizados. Além disto, há uma demanda em estabelecer relacionamentos entre requisitos e documentos complementares contendo descrições mais flexíveis (textuais, gráficas), ainda não resolvida na maioria das ferramentas (CLELAND-HUANG et al., 2007).

Cleland-Huang et al. (2007) propõe a utilização de ferramentas automatizadas para detectar rastreabilidade, utilizando técnicas como a proximidade das palavras, removendo artigos como “o,a,os,as” e levando as demais palavras à seus radicais, fazendo com que “aprovar” e “aprovado” sejam consideradas equivalentes. Além disto, nomes de métodos como getLastName() são separados para equivalerem à sentenças como “get last name”. Além disto, o autor sugere um conjunto de “boas práticas” para ferramentas automatizadas de rastreabilidade, dentre elas:

- Para cada *link*, o usuário deve poder localizar a origem, o destino e o tipo do requisito, onde e em qual ferramenta está armazenado;
- Os colaboradores devem poder decidir qual o nível de granularidade da rastreabilidade é mais apropriado para cada tipo de artefato;
- Prover rastreabilidade para artefatos construídos e armazenados em seus ambientes nativos (*case tools*).

Além disto, uma série de boas práticas relacionadas com o gerenciamento de requisitos pode melhorar consideravalmente o desempenho de ferramentas de rastreabilidade, tais como:

- Uso de um glossário para o projeto desde o início do projeto, para aumentar a consistência dos termos;
- Requisitos devem estar corretos, não-ambíguos, completos, concisos, consistentes, priorizados, verificáveis, legíveis e identificáveis;
- Construção de uma hierarquia de informação, categorizando os documentos de requisitos, incluindo informações semânticas;

3.2 MODELOS

Neste capítulo serão apresentados os principais trabalhos relacionados que propõem modelos de representação de requisitos. A partir destes trabalhos, será proposto um modelo único, com o objetivo de atender aos principais os requisitos identificados relevantes à proposta do presente trabalho.

3.2.1 MODELO DE RAMESH

Um dos pioneiros a abordar modelos de rastreabilidade de requisitos foi Ramesh et al. (1995b). Apesar de já terem decorridos muitos anos desde seu estudo, é relevante ressaltar a importânicia histórica de seu trabalho. O trabalho de Ramesh et al. (1995b) introduziu questionamentos relevantes sobre a rastreabilidade e propuseram um dos primeiros modelos desta área.

Ramesh et al. (1995) conduziu um estudo de caso sobre o uso de rastreabilidade de requisitos para uma divisão de tecnologia em armamento da força aérea americana, com o objetivo de determinar o impacto do uso da rastreabilidade em seus projetos e analisar os benefícios de sua adoção. Apesar da organização não ter desenvolvido uma metodologia formal para a implantação da rastreabilidade, este trabalho permitiu Ramesh desenvolver um modelo de informação que pode ser utilizado para representar a semântica dos vários tipos de informações de rastreabilidade sendo capturados e utilizados pela organização.

Conforme Ramesh et al. (1995), na época em que o estudo fora realizado não haviam indícios na literatura sobre um modelo que representasse explicitamente o tipo de informação a ser rastreada. Desta forma, o autor apresenta um estudo de caso sobre a aplicação de rastreabilidade e desenvolve um modelo que descreve a sua prática em uma organização. A figura a seguir procura demonstrar o modelo criado por Ramesh. Este modelo identifica a informação de rastreabilidade, os requisitos, raciocínios (*rationale*), alocação de requisitos e de recursos.

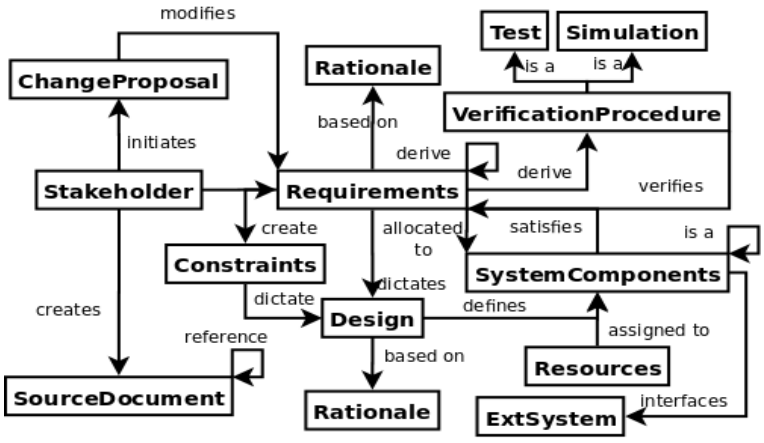


Figura 10. Modelo de Ramesh (RAMESH et al., 1995)

Para Ramesh et al. (1995), o modelo permite que o gerente do projeto consiga rastrear os requisitos e suas derivações, a fim de garantir ao cliente que todos os requisitos sejam compreendidos e permite também validar os requisitos por meio de testes de aceitação vinculados. Como características relevantes deste modelo destacam-se: a representação de requisitos textuais, o relacionamento entre requisitos (*requirements*) e suas derivações, entre estes com os documentos de origem (*rationale*), entre requisitos e documentos de testes (*test*) entre requisitos e componentes (*System Componentes*), entre requisitos e *stakeholders*, dentre outros. O relacionamento entre artefatos de um mesmo modelo configura ao modelo de Ramesh o suporte à rastreabilidade vertical. O modelo permite ainda representar as solicitações de mudanças propostas (*Change Proposal*). Entretanto, o modelo não suporta a representação de outros tipos de modelo como UML ou código-fonte, não permitindo a rastreabilidade horizontal. O modelo também não suporta a representação de diferentes projetos, não comporta a representação dos impactos gerados por uma mudança e também não possibilita representar versionamento e configuração de granularidade.

3.2.2 MODELO DE LANG E DUGGAN

Lang e Duggan (2001) apresentam um protótipo chamado *RM-Tool*, uma ferramenta que gerencia e controla requisitos dentro de um processo multidisciplinar. Este protótipo não ambiciona cobrir todos tipos de requisitos que fazem parte do desenvolvimento de um sistema. Basicamente, a ferramenta proposta, bem como o modelo representado no diagrama a seguir, focam em características não cobertas satisfatoriamente por ferramentas comerciais.

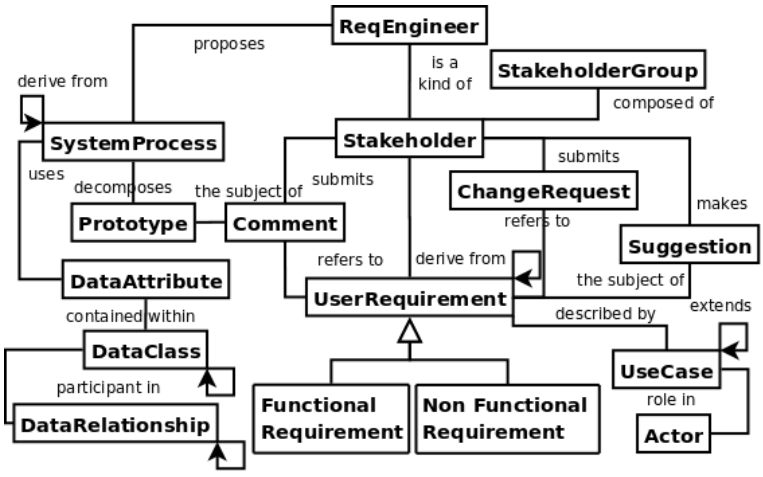


Figura 11. Modelo de Lang e Duggan (2001)

Da mesma forma que o modelo de Ramesh et al. (1995), o modelo de Lang e Duggan (2001) permite a representação de requisitos textuais (*UserRequirement*) e seus relacionamentos de derifação e referência, configurando ao modelo o suporte à rastreabilidade vertical. O modelo permite ainda a separação entre requisitos funcionais e não-funcionais e o relacionamento entre requisitos e *Stakeholders*. O modelo oferece um suporte limitado à modelos de *design*, restringindo-se à representação de *use cases* e atores (*Actor*) e seu relacionamento com os requisitos, configurando ao modelo o suporte à rastreabilidade horizontal. O modelo permite também representar explicitamente as mudanças ocorridas (*Change Request*) e o relacionamento entre as mudanças com os S*takeholders*.

O modelo de Lang e Duggan (2001) não permite representar o relacionamento entre requisitos e documentos externos, tais como documentos texto ou planilhas. Também não suporta a representação de artefatos de implementação (código-fonte). Da mesma maneira que o modelo de Ramesh et al. (1995), o modelo de Lang (2001) não permite a representação de diferentes projetos, a representação dos impactos gerados por uma mudança e também não possibilita representar versionamento e configuração de granularidade.

3.2.3 MODELO DE LETELIER

Para Letelier (2002), apesar do amplo reconhecimento da importância do papel da rastreabilidade, não existe consenso sobre suas práticas, quais informações devem ser coletadas, como tal informação deve ser usada e qual a semântica dos relacionamentos existentes entre os requisitos. Desta forma, as práticas variam muito de equipe para equipe.

Requisitos de software tem sido tradicionalmente especificados por meio de descrições textuais em linguagem natural. Assim, as ferramentas existentes para gerenciamento de requisitos focam seus esforços na manipulação de requisitos textuais relacionados, formando um grafo de rastreabilidade. Para Letelier (2002), graças a sua extensibilidade, a UML surge como um meio de estabelecer um modelo comum para representar a rastreabilidade entre requisitos. Desta forma, o autor apresenta um modelo para rastreabilidade de requisitos baseado em UML, que possibilita integrar desde especificações textuais até elementos de modelos de *design*. Desta forma, o autor espera obter uma representação comum que possa ser utilizada para identificar quaisquer artefatos de software, bem como as relações entre eles.

Na figura a seguir, Letelier (2002) apresenta um modelo para representar a rastreabilidade entre requisitos por meio de um diagrama de classes, onde as classes representam os tipos de entidades e as associações representam os tipos de relacionamentos de rastreabilidade. Além disto, o autor utiliza papéis em algumas associações para tornar os relacionamentos mais legíveis. Neste modelo, as classes filhas de *TraceableSpecification* representam os tipos de artefatos que encontram representação no modelo, e as associações (*modifies*, *responsibleOf*, *rationaleOf*, *validatedBy*, *verifiedBy* e *assignedTo*) representam os tipos de rastreabilidade suportados.

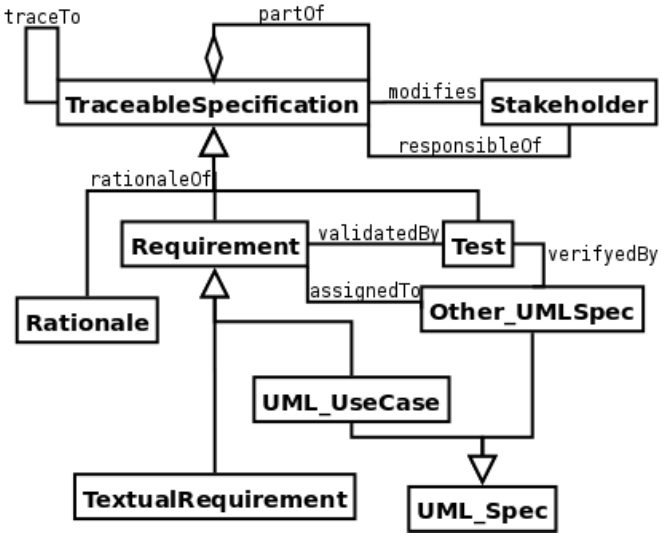


Figura 12. Modelo de Letelier (2002)

O modelo proposto por Letelier (2002) possui como características positivas a riqueza dos detalhes na representação dos links de rastreabilidade, provendo uma semântica com um nível de detalhamento grande entre os elementos que formam a especificação de software. O modelo permite a representação de requisitos textuais (*TextualRequirement*), bem como elementos de *design* UML (*UML_Spec*), além de permitir relacionar os artefatos com os *Stakeholders*. O modelo suporta rastreabilidade vertical, bem como a rastreabilidade horizontal, permitindo relacionar diferentes elementos do tipo *TraceableSpecification* por meio do relacionamento de rastreabilidade (*traceTo*). Como a classe *TraceableSpecification* é a super-classe da hierarquia, o relacionamento de rastreabilidade pode ser estabelecido entre elementos de um mesmo modelo (rastreabilidade vertical), bem como elementos de modelos diferentes (rastreabilidade horizontal). Além disto, o modelo permite configurar o nível de rastreabilidade, uma vez que a classe *TraceableSpecification* permite compor uma estrutura de

agregação com a própria classe.

O modelo de Letelier (2002) não permite o relacionamento entre artefatos e documentos externos, não permite a representação de artefatos de código-fonte, não permite a representação de artefatos de diferentes projetos, não permite representar as mudanças ocorridas e impactos provocados e também não suporta o versionamento de artefatos.

3.2.4 MODELO DE BRIAND, LABICHE E O'SULLIVAN

Para Briand, Labiche e O'Sullivan (2003) o uso de diagramas UML em projetos complexos leva invariavelmente à uma grande quantidade de diagramas interdependentes. Ao longo do desenvolvimento do sistema, estes diagramas sofrem mudanças que podem provocar alterações em outros elementos ou mesmo alterações em outros diagramas. Assim, é importante que as mudanças possam ser automaticamente identificadas a fim de manter tais diagramas atualizados, o que pode auxiliar também na previsão de custo e da complexidade das mudanças.

Para atingir os objetivos citados, permitindo melhorar o processo decisório e o planejamento de mudanças, Briand, Labiche e O'Sullivan (2003) propõem um modelo UML que suporte o gerenciamento de mudanças e a análise de impacto, como pode ser visto na figura a seguir. Dentre os objetivos do modelo, destacam-se: verificação de consistência, representação de mudanças, análise e priorização de impactos.

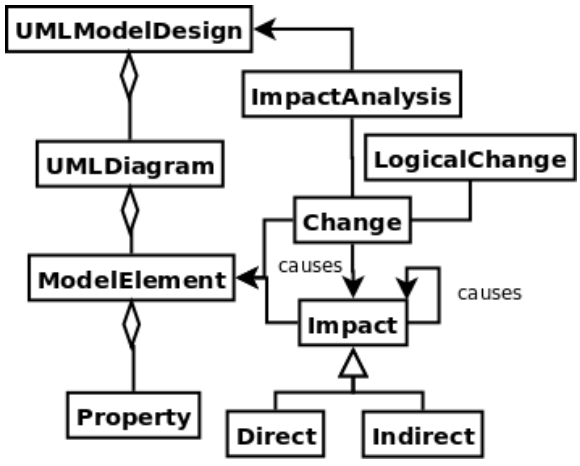


Figura 13. Modelo de Briand, Labiche e O'Sullivan (2003)

O modelo de Briand, Labiche e O'Sullivan (2003) permite a representação de

elementos de design UML (*UMLModelDesign*, *UMLDiagram*, *ModelElement*). O modelo permite representar as mudanças ocorridas (*Change*) e impactos gerados (*Impact*) sobre elementos de um modelo (*Model Element*) em UML, que pode ser uma classe, mas não uma de suas partes (*Property*). Neste modelo, uma propriedade (*Property*) pode representar um atributo ou um relacionamento entre dois elementos do modelo, configurando ao modelo o suporte à rastreabilidade vertical. Além disto, o modelo suporta versionamento de artefatos e configuração de granularidade que se dá pelos relacionamentos de agregação entre *UMLDiagram*, *ModelElement* e *Property*.

O foco prioritário do trabalho de Briand, Labiche e O'Sullivan (2003) são modelos UML. O autor cita em seu trabalho o relacionamento de modelos UML com código-fonte, porém este relacionamento não encontra correspondente no modelo proposto, uma vez que todas as classes giram em torno do conceito "*UML Diagram*". Como o modelo não permite representar requisitos textuais ou código-fonte, consequentemente não suporta a rastreabilidade horizontal. O modelo também não suporta o relacionamento entre artefatos e documentos externos. Pode-se notar que o modelo não permite representar artefatos de diferentes projetos e também não permite relacionar artefatos com *Stakeholders*.

3.2.5 MODELO DE DE GRANDE E MARTINS

De Grande e Martins (2006) apresentam uma ferramenta automatizada para gerenciar requisitos, chamada SIGERAR. Dentre os objetivos da ferramenta, podem ser citados o gerenciamento das mudanças de requisitos durante todo o ciclo de vida do software e a rastreabilidade entre os requisitos e documentos de especificação. A ferramenta permite coletar, armazenar e manter os requisitos, gerenciar suas mudanças e rastrear as relações entre requisitos e entre requisitos e outros documentos.

A modelagem da ferramenta se deu por meio do uso da UML (*Unified Modeling Language*), como pode ser visto na figura a seguir. O modelo permite representar os diferentes projetos gerenciados pela organização (*Projeto*), controla glossários, termos, usuários e seus vínculos com a organização (*Departamento*), permite representar os requisitos de cada projeto e sua dependência (*RequisitoDependente*), configurando ao modelo o suporte à rastreabilidade vertical. Permite ainda controle de mudanças sobre uma versão de um

requisito (*VersaoRequisito*), que contém a descrição da alteração de mudança, seus motivos, solicitante, importância, o custo e o estado da solicitação.

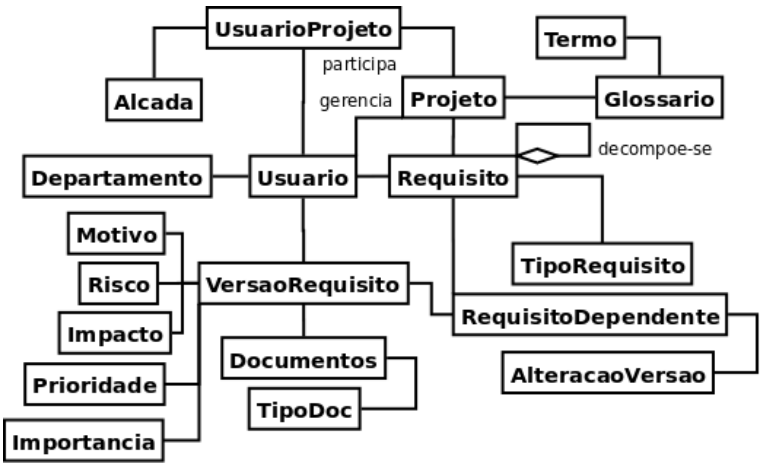


Figura 14. Modelo de De Grande e Martins (2006)

O modelo possui versionamento (*VersaoRequisito*), que possui dados de controle de versão e relação com documentos envolvidos no processo (*Documentos*), permitindo rastreabilidade. Permite representar também impactos (*Impacto*) e relacionar requisitos com *Stakeholders* (*Usuario*).

Salienta-se que o modelo de De Grande e Martins (2006) foca somente em especificações textuais. Como o modelo não suporta modelos de *design* UML ou código-fonte, também não permite representar a rastreabilidade horizontal. Além disto, não permite a configuração de granularidade nos relacionamentos entre os requisitos.

3.2.6 MODELO DE KASSAB, ORMANDJIEVA E DANEVA

Para Kassab, Ormandjieva e Daneva (2008), muitas organizações concentram seus esforços na implementação da rastreabilidade para requisitos funcionais, entretanto a rastreabilidade de requisitos não-funcionais tem sido esquecida por completo. Para o autor, um dos motivos que leva à este esquecimento é a dificuldade de sua implementação, uma vez que os requisitos não-funcionais tendem a se distribuir ao longo de diversos módulos quando estes são mapeados no sistema.

Kassab, Ormandjieva e Daneva (2008) propõem um modelo em linguagem UML para

representar os conceitos relacionados aos requisitos não-funcionais, bem como requisitos funcionais e seus relacionamentos. Neste modelo, um requisito funcional está relacionado a um modelo, que está relacionado a uma fase do ciclo de vida do software. Como exemplo, pode-se citar uma classe, que está relacionada ao modelo de classes e este, à fase de *design*. Cada modelo pode ser composto de artefatos (*use case*, diagrama de classes) e o artefato em si pode ser composto de elementos como associações e heranças. Neste modelo, que pode ser visto na figura a seguir, um requisito não-funcional como "*a tela do cadastro deve ser web*" pode estar associado à um ou diversos requisitos funcionais.

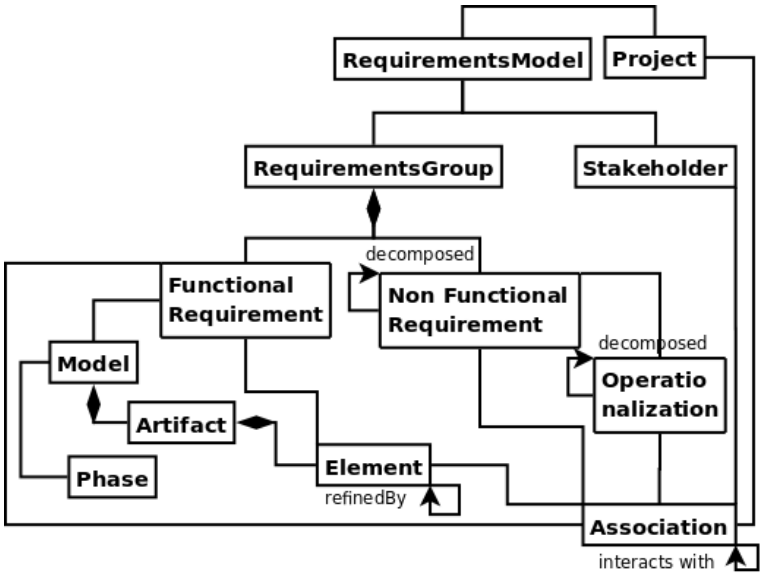


Figura 15. Modelo de Kassab, Ormandjieva e Daneva (2008)

O modelo de Kassab, Ormandjieva e Daneva (2008) suporta a representação de artefatos de diferentes fases do ciclo de desenvolvimento de um *software*, como requisitos textuais ou modelos de design, por meio da composição entre as classes *Model*, *Artifact* e *Element*. Esta estrutura em composição permite a representação da granularidade e também permite o relacionamento (*Association*) entre elementos (*Element*) de um mesmo modelo (*Model*), configurando a rastreabilidade vertical, ou entre elementos de modelos diferentes, configurando a rastreabilidade horizontal. O modelo permite ainda a representação de diferentes projetos (*Project*), o relacionamento entre artefatos e *Stakeholders* e entre artefatos e documentos externos, que podem ser representados por requisitos não funcionais.

O modelo não permite representar mudanças e impactos gerados. Além disto, o autor não comenta em momento algum a utilização de código-fonte como um artefato do sistema, bem como se trata ou não do versionamento de artefatos. Como o modelo não exibe tais

conceitos, pressupõe-se sua ausência.

3.2.7 MODELO DE GHAZI

Para Ghazi (2008), as técnicas atuais de rastreabilidade não estão adaptadas ao contexto global de desenvolvimento, tendo em vista suas diversas perspectivas. A rastreabilidade requer a captura de diversas categorias de informação e também está relacionada com diversas disciplinas da engenharia de software como: engenharia de requisitos, design, gerência de projeto, desenvolvimento, testes, dentre outras.

Para Ghazi (2008), é necessário construir um modelo que represente a informação de rastreabilidade, bem como os demais produtos derivados deste processo. Para tal, o autor propõe um método de gerenciamento de rastreabilidade com múltiplas perspectivas (*MV - TMM: A Multi View Traceability Management Method*).

O modelo proposto, que pode ser visto na figura a seguir (construído em UML), permite gerenciar diversos tipos de rastreabilidade. O modelo permite representar atores e papéis envolvidos no projeto (*Agent*), bem como seu envolvimento com os artefatos (*TraceElement*), que podem ser requisitos textuais, modelos de *design* ou código-fonte. O modelo permite relacionar artefatos (*TraceElement*) por meio da classe *TraceabilityLink.* Os relacionamentos podem ser estabelecidos entre elementos de um mesmo modelo (rastreabilidade vertical) ou entre elementos de diferentes modelos (rastreabilidade horizontal). Também permite classificar estes relacionamentos conforme uma taxonomia própria, que indica o tipo de link (contribuição, dependência, evolução, dentre outros), permite representar diferentes projetos (*ProjTraceModel*) e relacionar artefatos com documentos externos.

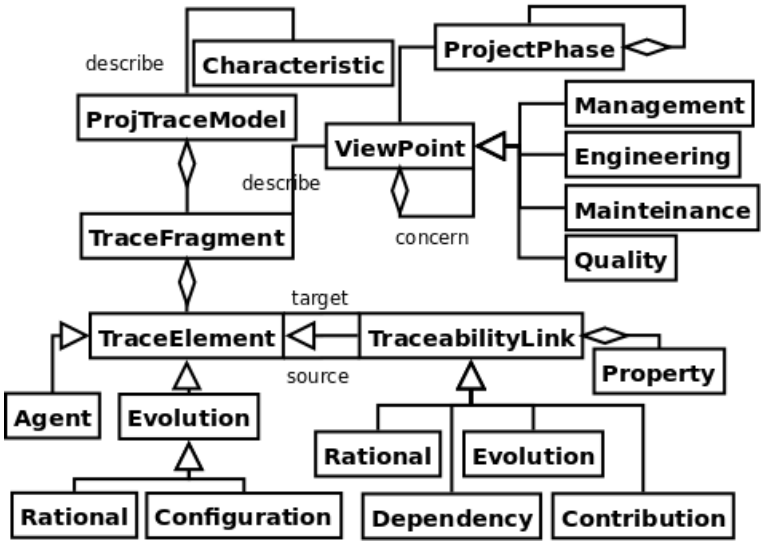


Figura 16. Modelo de Ghazi (2008)

Além disto, o modelo possibilita representar a evolução dos links de rastreabilidade de um elemento (*Evolution*) e também relacionar elementos com visões, que representam um determinado aspecto da rastreabilidade a ser capturado, como por exemplo: gerenciamento, engenharia, qualidade e manutenção.

Nota-se que o modelo de Ghazi (2008) não comporta a configuração de granularidade, sendo que os relacionamentos são estabelecidos em nível de artefato (*TraceElement*). Também não permite a representação mudanças ocorridas, de impactos gerados e do versionamento de artefatos.

3.2.8 MODELO DE LEAL, FIGUEIREDO E DE SOUZA

Leal, Figueiredo e De Souza (2008) descrevem uma abordagem semi-automática para criação e manutenção da informação de rastreabilidade a partir dos artefatos de software produzidos, a fim de maximizar a qualidade dos relacionamentos. Para tal, combina metodologias como UML, template de artefatos e análise de documentos textuais. Na fase de aquisição dos dados nesta metodologia, os dados são descobertos por um parser e armazenados em um banco de dados, cuja representação se dá pelo modelo representado pela figura a seguir. A partir de então, a análise propriamente dita é iniciada.

O modelo de Leal, Figueiredo e De Souza (2008), apresentado na figura a seguir, gira em torno do conceito *item*, que representa diferentes artefatos suportados, como requisitos

textuais (*requirement*), modelos de *design* UML (*useCase*, *actor*), e também relacionamentos existentes entre os artefatos (*link*). Os conceitos *useCase*, *actor*, *requirement* e *link* são tipos de item. Portanto tanto um artefato em si, quanto um *link* de relacionamento são do tipo *item*. Cada *item* possui um tipo de item (*typeItem*), e cada *link* possui um tipo de link (*typeLink*) e cada item possui um *status*. Como o modelo permite o estabelecimento de *links* entre elementos do tipo *item*, pode-se dizer que o mesmo suporta a rastreabilidade vertical quando se dá o estabelecimento de *links* entre elementos de um mesmo modelo (como entre dois *requirement*) e a rastreabilidade horizontal quando se dá o estabelecimento de links entre elementos de modelos diferentes (como entre um *requirement* e um *useCase*). Além disto, o modelo permite representar a versão de cada artefato (atributo *version*).

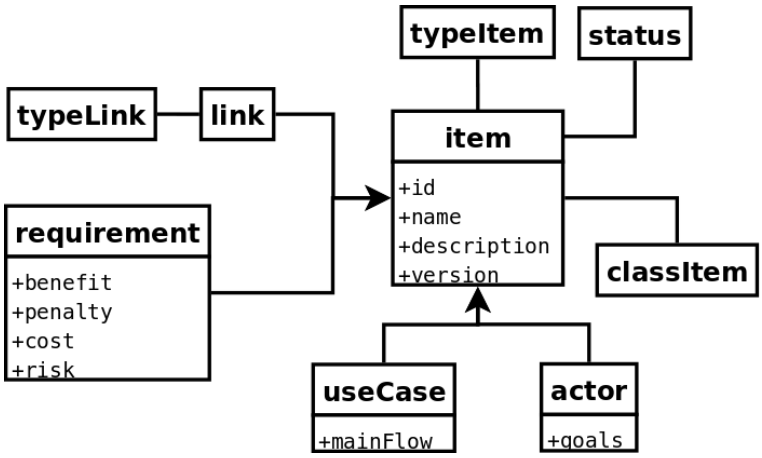


Figura 17. Modelo de Leal, Figueiredo e De Souza (2008)

Pode-se notar no modelo apresentado por Leal, Figueiredo e De Souza (2008), a ausência do relacionamento entre artefatos e documentos externos, a ausência de estruturas de informação para armazenar elementos de código-fonte, ausência da configuração de granularidade, tendo em vista que a granularidade máxima se dá em nível de artefato (*item*). O modelo também não permite a representação de diferentes projetos, das mudança ocorridas, dos impactos gerados, e do relacionamento entre artefatos e *stakeholders*.

3.3 MÉTODOS

Neste capítulo serão apresentados os principais trabalhos relacionados que propõem

métodos para gestão de requisitos. Estes trabalhos envolvem o estudo de técnicas para melhorar a gestão da mudança de requisitos e na maioria das vezes estão vinculados com o desenvolvimento de uma ferramenta como forma de provar as idéias propostas. A partir do estudo destes trabalhos, foi desenvolvido um novo método e consequentemente uma nova ferramenta, com o objetivo de atender a um maior número de requisitos funcionais identificados na literatura a partir destes trabalhos relacionados.

3.3.1 MÉTODO ADAMS

ADAMS (*ADvanced Artefact Management System*) é uma ferramenta *web* que integra características de gerenciamento de projetos, alocação de recursos, gerenciamento de artefatos, coordenação e cooperação dos colaboradores, versionamento de artefatos e rastreabilidade. ADAMS é uma ferramenta de apoio à um processo baseado em artefatos, onde um artefato pode ser um simples arquivo ou também uma composição de artefatos, onde tal composição pode se dar em diversos níveis de granularidade. Desta forma, pode-se adicionar sub-artefatos em um artefato. Para Bruegge et al. (2006), ADAMS trata diferentes tipos de documentos submetidos à ele (requisitos texutais, modelos de *design* UML, código-fonte, etc). Para tal, o usuário pode criar um artefato composto, onde cada parte deste artefato pode ser um outro artefato, permitindo assim uma rastreabilidade *fine-grained*. Desta forma, pode-se facilmente criar requisitos compostos de outros requisitos. ADAMS também permite que se crie um artefato do tipo UML contendo nodos, onde cada nodo pode ser um outro diagrama. O relacionamento entre os diagramas se dá por meio de um tipo especializado de link de rastreabilidade. Além disto, ADAMS permite que seu usuário registre a rastreabilidade entre os artefatos do sistema, sejam artefatos da mesma hierarquia (rastreabilidade vertical) ou entre artefatos de hierarquias diferentes (rastreabilidade horizontal) (DE LUCIA et al., 2005).

Na figura a seguir, tem-se uma visão do formulário de definição dos *links* de rastreabilidade da ferramenta ADAMS. Por meio deste formulário, a ferramenta permite a criação de *links* de rastreabilidade entre artefatos de um projeto. É possível especificar três tipos de rastreabilidade: dependência, ligação indireta e composição.

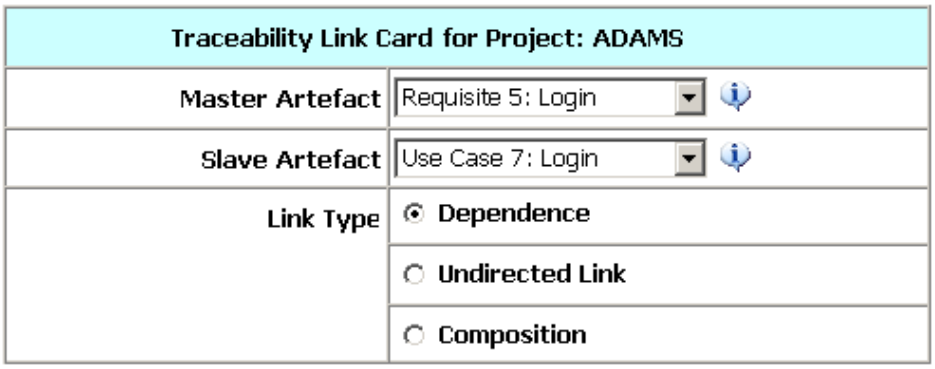


Figura 18. Registro de rastreabilidade – ADAMS (DE LUCIA et al., 2005)

A ferramenta ADAMS conta ainda com um módulo de recuperação de rastreabilidade, implementada por meio de uma técnica de recuperação de informação (IR) chamada de *Latent Semantic Indexing* (LSI), que permite identificar *links* de rastreabilidade horizontal por meio da similiaridade entre termos. Já existem plugins para a ferramenta ADAMS que recuperam a informação a partir de requisitos textuais, diagramas UML, e arquivos contendo código-fonte (este último dentro do ambiente *eclipse*). A abordagem adotada pela ferramenta ADAMS compara um artefato com os demais, observando sua similaridade. Para o autor, a intervenção humana continua sendo essencial neste processo, uma vez que este tipo de abordagem poderá falhar em identificar relações verdadeiras, ao passo que também poderá apresentar falsos positivos. Assim, o colaborador poderá utilizar esta abordagem automática, como também poderá definir os relacionamentos de forma manual (DE LUCIA; FASANO; OLIVETO, 2008). A figura a seguir demonstra o resultado de uma recuperação de informação realizada pelo ADAMS.

List of Lost Links Discovered

ID	Master Artefact	ID	Slave Artefact	Similarity measure	Action		
					Discard	Remove	None
8	Inserimento anagrafica laboratorio (Use Case)	7	Modifica anagrafica laboratorio (Use Case)	100.0	○	○	◉
8	Inserimento anagrafica laboratorio (Use Case)	9	GUI Anagrafica Laboratorio (Class Specification)	100.0	○	○	◉
8	Inserimento anagrafica laboratorio (Use Case)	6	Inserimento anagrafica paziente (Use Case)	83.24	○	○	◉

List of Warning Links Discovered

ID	Master Artefact	ID	Slave Artefact	Similarity measure	Action		
					Discard	Remove	None
8	Inserimento anagrafica laboratorio (Use Case)	4	GUI Vista (Class Specification)	47.68	○	○	◉

Figura 19. Recuperação de rastreabilidade – ADAMS (DE LUCIA et al., 2005)

A arquitetura do ADAMS é dividida em sub-sistemas independentes distribuídos em

uma estrutura em camadas, que pode ser vista na figura a seguir. A camada de apresentação (*Presentation Layer*) é desenvolvida em HTML e JSP. *Artefact Management Subsystem* gerencia os artefatos, seu ciclo de vida e a rastreabilidade. *Project Management Subsystem* gerencia o projeto, alocação de recursos e agendamentos.Há ainda uma camada para administração (*Administration Layer*) e para comunicação (*Communication Subsystem*) (De Lucia et al., 2004).

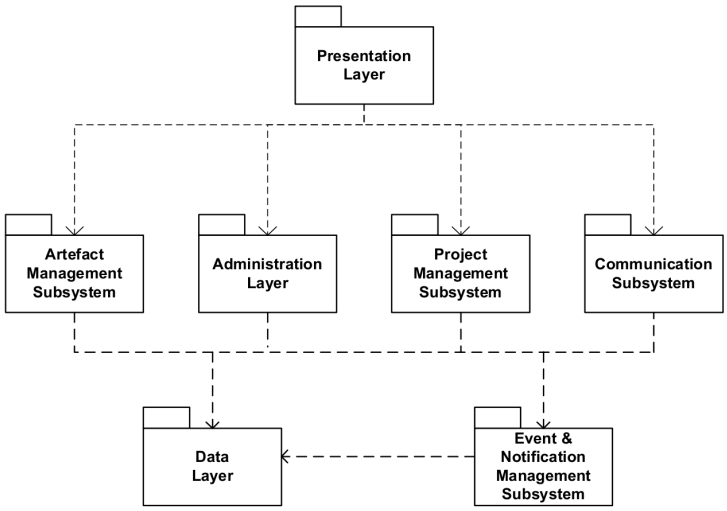


Figura 20. Arquitetura da ferramenta ADAMS (DE LUCIA et al., 2005)

ADAMS permite que os colaboradores possam ser notificados de eventos que ocorrem sobre os artefatos. Desta forma, um colaborador é notificado quando outro colaborador está editando o mesmo artefato. Além disto, os *links* de rastreabilidade da ferramenta ADAMS são utilizados para a realização de análise de impacto e notificação dos colaboradores envolvidos quando ocorre uma mudança em um artefato, criando um efeito de notificação recursiva (propagação). Os colaboradores também podem inscrever-se para serem notificados quando determinados eventos ocorrerem aos artefatos.

ADAMS não oferece um gerenciamente pró-ativo da mudança de requisitos, ou seja, a ação sempre deve partir do usuário, que irá acionar a ferramenta quando for alterar um artefato, uma vez que a ferramenta funciona como um repositório central de artefatos. ADAMS também não possui meios de tomada de decisão que possam mudar os rumos do processo (*workflow*), não oferece gráficos gerenciais com indicadores estatísticos, e sua arquitetura não possui mecanismos facilitadores para prover extensibilidade e agregação de novos recursos sem ter amplos conhecimentos sobre a tecnologia atual na qual a ferramenta foi desenvolvida.

3.3.2 MÉTODO EVENT BASED TRACEABILITY

Para Cleland-Huang, Chang e Christensen (2003), a utilização de *links* de rastreabilidade entre os requisitos e outros artefatos do projeto permitem uma série de atividades de manutenção, incluindo validação, análise de impacto, dentre outros. Entretanto, um dos grandes desafios da rastreabilidade reside em manter sua integridade durante a evolução do software, ou seja, após sucessivas alterações. Fatores como a falta de tempo, profissionais qualificados e cooperação entre os envolvidos estão entre os fatores que contribuem negativamente para isto. Este cenário torna-se ainda mais complexo em um ambiente de desenvolvimento distribuído.

A rastreabilidade inadequada é um dos principais fatores que contribuem para falhas em projeto. Como forma de solução para esta questão, Cleland-Huang, Chang e Christensen (2003) propõem um novo método de rastreabilidade baseado em notificação de eventos. Neste método, chamado *Event-Based Traceability* (EBT), os artefatos de software são relacionados por meio de uma estrutura de publicação/assinatura. Assim, os artefatos alterados assumem o papel de publicadores (*publishers*) e os artefatos dependentes assumem o papel de assinantes (*subscribers*). Desta forma, sempre que uma mudança ocorre, eventos são publicados em um "servidor de eventos" e notificações são enviadas aos responsáveis pelos artefatos dependentes (*subscribers*). O principal ojetivo é notificar os artefatos e desenvolvedores dos artefatos dependentes sobre as mudanças ocorridas.

Para Cleland-Huang, Chang e Christensen (2003), a principal vantagem na implementação da abordagem baseada em eventos (EBT) está em melhorar a coordenação entre os membros da equipe responsáveis pelas alterações nos artefatos. Para Cleland-Huang, Chang e Christensen (2003), em um ambiente tradicional, onde utiliza-se técnicas como a matriz de rastreabilidade ou grafos para manter a rastreabilidade, os *links* de rastreabilidade devem ser mantidos com muito cuidado entre as diferentes atualizações do sistema. Entretanto, a técnica EBT reduz os esforços necessários para manter a rastreabilidade quando ocorre uma mudança.

A figura a seguir exibe a arquitetura do EBT, formada por *Requirements Manager*, *Event Server* e *Subscriber Manager*, que estão conectados por meio de um protocolo de comunicação. *Requirements Manager* é responsável pelo gerenciamento dos requisitos e por disparar eventos de mudança. *Subscriber Manager* registra subscrições (assinaturas) de

artefatos gerenciados, recebe notificações e as encaminha ao responsável pelo artefato. *Event Server* gerencia as subscrições, recebe mensagens e realiza notificações.

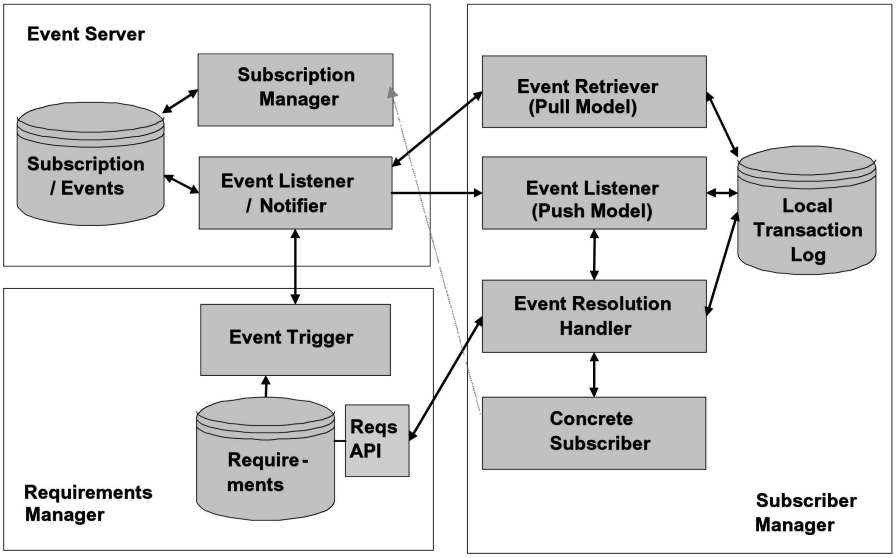


Figura 21. Arquitetura do método EBT (CLELAND-HUANG; CHANG; CHRISTENSEN, 2003)

Na figura a seguir podem ser vistos: *Requirement*, que é o artefato monitorado, Dependent artifact, que é o artefato dependente e Subscription, que trata da subscrição, ou seja, trata da observação do artefato principal por meio do link de rastreabilidade.

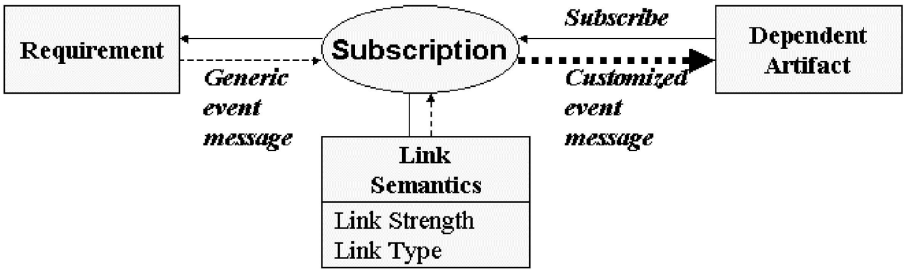


Figura 22. Sistemática de eventos do método EBT (CLELAND-HUANG; CHANG; CHRISTENSEN, 2003)

Cabe salientar que o método EBT foi projetado para integrar-se apenas com ferramentas de gestão de requisitos textuais como DOORS ou RequisitePro, não suportando modelos de *design* UML ou código-fonte. Verificou-se também a ausência de configuração de projeto, da configuração de granularidade da rastreabilidade (rastreabilidade *fine-grained*), versionamento dos requisitos e de mecanismos para detecção da rastreabilidade. Não possui estatísticas gerenciais e também não oferece mecanismos de extensibilidade e adaptabilidade.

3.3.3 MÉTODO SIGERAR

De Grande e Martins (2006) apresentam uma ferramenta automatizada para gerenciar requisitos, chamada SIGERAR. Dentre os objetivos da ferramenta, podem ser citados o gerenciamento das mudanças de requisitos durante todo o ciclo de vida do software e a rastreabilidade entre os requisitos e documentos de especificação. A ferramenta permite gerenciar a configuração dos projetos, a alocação de seus *stakeholders*, manter os requisitos textuais, modelos de *design* e as relações entre eles, bem como entre requisitos e outros documentos, gerenciar a rastreabilidade, gerenciar e classificar as mudanças ocorridas, analisar os impactos conforme a informação de rastreabilidade, controlar a execução do processo (*workflow*) e notificar os *stakeholders* sobre as mudanças e os impactos.

Para De Grande e Martins (2006), a ferramenta permite que os responsáveis pelos requisitos analisem e atribuam valores aos requisitos alterados, quanto à risco, importância, impacto, prioridade e custo, além de demonstrar os requisitos dependentes da modificação. Na figura a seguir tem-se uma das telas da ferramenta SIGERAR, onde pode-se visualizar os atributos de uma versão de um requisito.

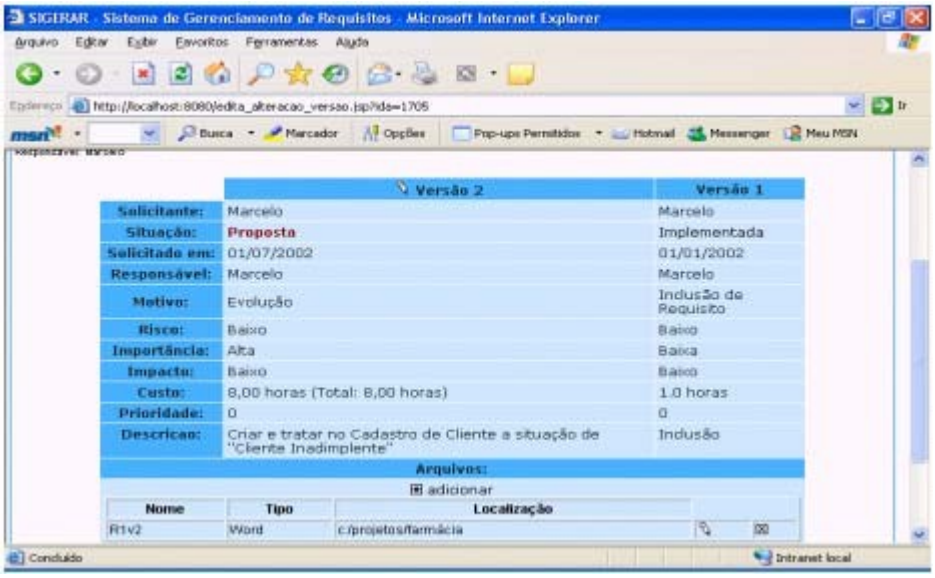


Figura 23. Visualização de requisito – SIGERAR (DE GRANDE; MARTINS, 2006)

Além das características analisadas até então, deve-se citar a ausência de mecanismo de *parsing* das estruturas dos artefatos, sendo que a ferramenta somente analisa se houveram alterações em determinado elemento, não observando a granularidade do artefato, ou seja, a ferramenta não suporta a rastreabilidade *fine-grained*. A ferramenta também não possui

suporte à estruturas de código fonte e não permite integração automática com outras ferramentas de autoria de requisitos como editores de texto, nem ferramentas CASE, o que é citado nas conclusões como possíveis trabalhos futuros. Além disto, não permite uma gestão pró-ativa, uma vez que cabe ao usuário acessar a ferramenta para obter a análise de impacto correspondente à mudança e também não possui mecanismo de recuperação da rastreabilidade, seja vertical ou horizontal. Não possui gráficos contendo indicadores gerenciais e também não apresenta mecanismos de extensibilidade para agregação de novos recursos.

3.3.4 MÉTODO IACMTOOL

Para Briand, Labiche e O'Sullivan (2003), a gestão da mudança em software torna-se ainda mais complexa em ambientes de desenvolvimento com grandes equipes, onde vários colaboradores trabalham de maneira concorrente em diferentes partes do projeto. Nestes casos, o suporte automatizado é necessário para permitir que a equipe possa gerenciar a complexidade das mudanças, identificar seus impactos e comunicar corretamente as mudanças ocorridas entre os colaboradores impactados pelas alterações.

Para resolver os problemas citados anteriormente, Briand, Labiche e O'Sullivan (2003) propõem uma ferramenta de análise de impacto chamada iACMTool. Esta ferramenta realiza a leitura de dois modelos UML compostos por diagramas e realiza um relatório de análise de impacto baseado nas diferenças entre estes modelos.

A ferramenta permite a leitura do modelo XMI (*XML Metadata Interchange*) e de expressões OCL presentes nos modelos. A informação resultante é então persistida para posterior recuperação. A ferramenta permite a detecção de mudanças por meio da comparação de duas versões do modelo UML. As mudanças são detectadas e classificadas conforme taxonomia criada pelo autor, conforme a operação (adição, alteração, exclusão). A ferramenta verifica também a consistência de uma determinada versão do modelo, utilizando para isto uma série de regras. Por último, realiza a análise de impacto para as mudanças ocorridas e gera relatórios de consistência, verificação e de análise de impacto.

Esta ferramenta é composta por sete pacotes, como pode ser visto na figura a seguir. O subsistema *parser* tem a função de realizar a leitura de arquivos XML, bem como expressões

OCL que descrevem um modelo UML. A informação obtida é armazenada no pacote *model*, responsável pela persistência. O pacote *modelChanges* é responsável pela detecção das mudanças por meio da análise do modelo e o pacote *consistencyVerification* verifica a consistência de cada versão do modelo. Já o pacote *control* é responsável por controlar o fluxo da aplicação, enquanto que o pacote *impactAnalysis* realiza a análise de impacto e *reportGeneration* gera os relatórios necessários para o sistema.

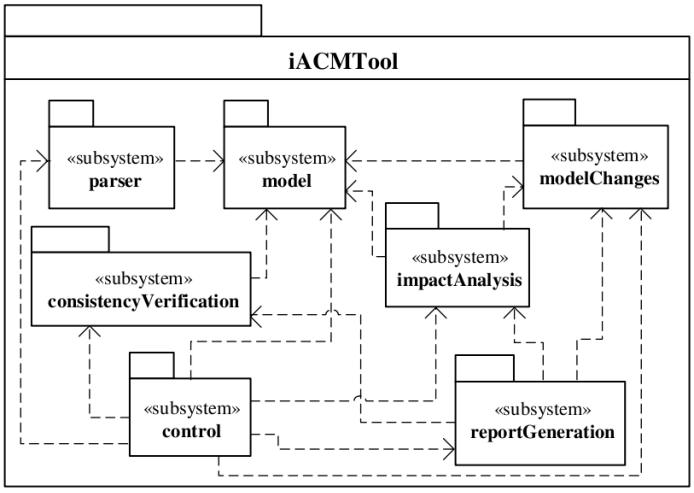


Figura 24. Pacotes da ferramenta iACMTool (BRIAND; LABICHE; O'SULLIVAN, 2003)

Dentre as principais qualidades da ferramenta iACMTool, destacam-se a compreensão do modelo UML, tendo em vista que a mesma conta com um *parser* de documentos XMI e portanto, permite a obtenção de informações de artefatos e relacionamentos com maior grau de fidelidade. Além disto, suporta a rastreabilidade *coarse-grained*, provê gerenciamento das mudanças e análise de impacto, melhorando a comunicação e coordenação da equipe. Como a ferramenta é focada em documentos UML, ela não possui suporte de integração com requisitos textuais, tampouco com artefatos de código-fonte, não oferecendo recursos de rastreabilidade horizontal. Também não permite a configuração de projetos, utilização on-line, extensibilidade e gerenciamento pró-ativo, uma vez que toda ação deve partir do usuário.

Para Briand, Labiche e O'Sullivan (2003), a análise de impacto de um modelo UML é conduzida de maneira que um elemento impactado é identificado quando um de seus atributos ou métodos requerem modificação. Para o autor, tais modificações decorrem de alterações de lógica, correções de erros, melhorias de design e alterações de requisitos.

3.3.5 MÉTODO CHIANTI

Ren et al. (2004) propõe um método de análise de impacto baseado em testes de unidade, ou seja, é um método focado em análise do código-fonte da aplicação, não oferecendo suporte à requisitos textuais ou modelos de *design*. Seu método inicia com a análise do código-fonte, para obter as mudanças ocorridas a partir da comparação entre duas versões do sistema. Cada mudança é classificada conforme taxonomia própria. A partir de então, constrói um grafo semântico de execução para cada *unit test*, que demonstra as chamadas para cada teste de unidade. Este grafo contém um nodo para cada método e uma aresta para cada relacionamento de chamada. Neste grafo, pode-se perceber quais testes de unidade dependem de quais classes ou métodos, baseado em suas chamadas de execução. A partir deste grafo, o método detecta quais foram os testes afetados por uma mudança. Percebe-se o suporte à rastreabilidade *fine-grained*, uma vez que as relações estabelecem-se em nível de métodos e atributos, e não somente em nível de classe.

Inicialmente, o método determina as mudanças que podem impactar testes a partir de duas versões do código-fonte. Na figura a seguir, é exibida uma versão de código-fonte contendo as modificações em destaque entre retângulos, bem como testes relacionados à este trecho de código.

```
class A {
  public A(){ }
  public void foo(){ }
  public int x;
}
class B extends A {
  public B(){ }
  public void foo(){ B.bar(); }
  public static void bar(){ y = 17; }
  public static int y;
}
class C extends A {
  public C(){ }
  public void foo(){ x = 18; }
  public void baz(){ z = 19; }
  public int z;
}
```

```
class Tests {
  public static void test1(){
    A a = new A();
    a.foo();
  }
  public static void test2(){
    A a = new B();
    a.foo();
  }
  public static void test3(){
    A a = new C();
    a.foo();
  }
}
```

Figura 25. Código-fonte analisado pela ferramenta Chianti (REN et al., 2004)

A partir das mudanças realizadas, a ferramenta Chianti obtém todas modificações ocorridas e, a partir destas, gera um modelo no qual as mudanças são classificadas em: AC:

Added classes, DC: *Deleted classes*, AM: *Added methods*, DM: *Deleted methods*, CM: *Changed methods*, AF: *Added fields*, DF: *Deleted Fields* e LC: *Lookup changes*, ou seja, mudanças decorrentes da execução. Na figura a seguir, é possível visualizar o grafo de dependências sintáticas entre as mudanças, detectado pela ferramenta. A adição da chamada B.bar() em B.foo(), representada pela caixa número 6, do tipo *Change method*, resultaria em um programa inválido, a menos que o método B.bar() fosse adicionado ao sistema, o que é representado pela caixa número 5, do tipo *Added method*. Portanto a mudança 6 é dependente da mudança 5. O método de Ren considera qualquer adição de método como sendo inicialmente uma adição de assinatura e posteriormente uma alteração para adição do corpo do método. Por este motivo existe a caixa número 8, exibindo a modificação do método recém adicionado.

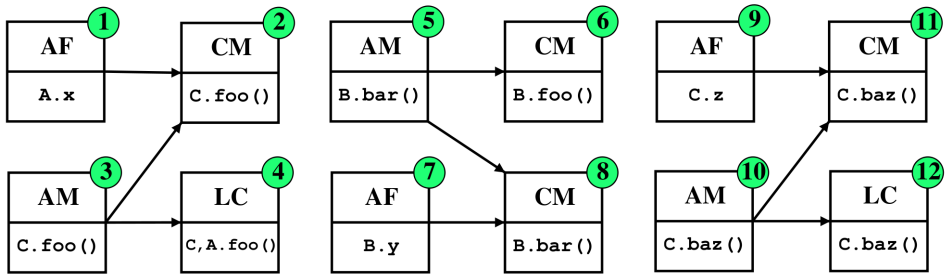


Figura 26. Grafo de dependências entre as mudanças da ferramenta Chianti (REN et al., 2004)

O método de Ren difere dos demais trabalhos relacionados à análise de impacto já estudados. Enquanto que a grande maioria dos trabalhos tem o objetivo de determinar um conjunto de entidades (métodos e classes) afetados por uma determinada mudança, o método de Ren determina quais testes de unidade foram afetados pelas mudanças.

A partir da identificação das mudanças, são identificados os testes afetados por estas. Para tal, é construído um grafo que demonstra as chamadas para cada teste de unidade. Este grafo contém um nodo para cada método e uma aresta para cada relacionamento de chamada. A figura a seguir procura demonstrar este grafo. Neste grafo, pode-se perceber que a execução do teste 2 depende do método construtor da classe B, que por sua vez depende do construtor da classe A, além do método B.foo(), que depende do método B.bar(). Esta última dependência foi adicionada pelas mudanças. Logo, este teste foi afetado pelas mudanças 6 e 8. Este recurso configura um tipo de recuperação de rastreabilidade vertical, uma vez que detecta o relacionamento entre artefatos de um mesmo modelo.

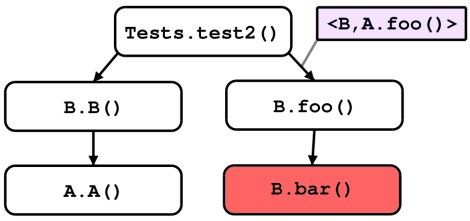


Figura 27. Grafo de dependências entre chamadas da ferramenta Chianti (REN et al., 2004)

A ferramenta Chianti, proposta por Ren et al. (2004), trata-se de um protótipo de ferramenta para análise de impacto, cujo funcionamento basicamente se dá por meio da leitura do repositório de um projeto de desenvolvimento (CSV) em intervalos regulares a partir das edições por parte dos desenvolvedores e armazena as informações das mudanças e das dependências em arquivos XML para posterior análise. A ferramenta também possui gráficos gerenciais que indicam as mudanças ocorridas em um determinado período.

A figura a seguir procura demonstrar a ferramenta Chianti em funcionamento. Nela, pode-se ver na direita uma estrutura hierárquica contendo as mudanças ocorridas e suas relações e no centro, o trecho de código-fonte impactado pela mudança selecionada.

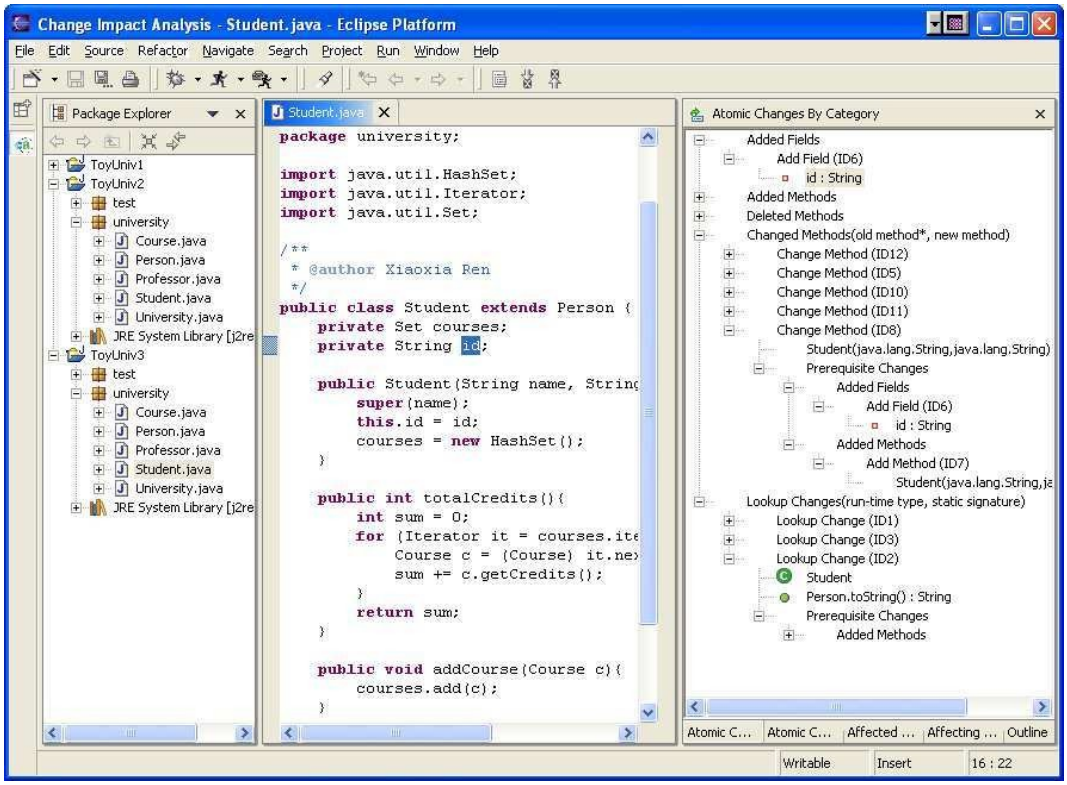


Figura 28. Ferramenta Chianti em ação (REN et al., 2004)

Dentre algumas características ausentes estão a configuração de projetos, o suporte à

requisitos textuais, modelos de *design* e a consequente rastreabilidade horizontal, o versionamento de artefatos, o gerenciamento pró-ativo, uma vez que as análises dependem de ação do usuário, mecanismos de controle de processos e tomada de decisão (*workflow*) e também suporte à extensibilidade da ferramenta.

3.3.6 MÉTODO DE LEAL, FIGUEIREDO E DE SOUZA

Para Leal, Figueiredo e De Souza (2008), a rastreabilidade apoia todas as fases do desenvolvimento de software, contribuindo para sua qualidade final. Entretanto, apesar dos inúmeros benefícios advindos de sua utilização, a rastreabilidade ainda é uma atividade cara e seu custo está diretamente ligado com o nível de detalhamento que se deseja em sua representação. Este detalhamento advém do obtenção dos *links* de rastreabilidade. Um dos motivos que impede uma adoção maior da rastreabilidade em ambientes industriais é a dificuldade em automatizar a geração dos *links* de rastreabilidade entre os artefatos.

Leal, Figueiredo e De Souza (2008) descrevem uma abordagem semi-automática para criação e manutenção da informação de rastreabilidade a partir dos artefatos de software produzidos, a fim de maximizar a qualidade dos relacionamentos. Para tal, combina metodologias como UML, templates de artefatos e análise de documentos textuais.

Para Leal, Figueiredo e De Souza (2008), em um processo de desenvolvimento de software existem artefatos que possuem riqueza em sua sintaxe, como o código-fonte e diagramas UML (representados em XMI) e existem artefatos descritos por textos livres como a especificação de requisitos. A descoberta de conhecimento em artefatos com riqueza de sintaxe é simples e direta. Entretanto, a descoberta de conhecimento em textos livres, por exemplo, é uma tarefa complexa e de qualidade variável. Neste segundo caso, algumas abordagens utilizam técnicas de Processamento de Linguagem Natural (PLN) para extração de conhecimento.

Leal, Figueiredo e De Souza (2008) sugerem como técnica para otimizar a qualidade das informações extraídas de artefatos textuais, a utilização de templates, com o objetivo de fornecer uma estrutura organizada e padronizada aos documentos de artefatos textuais, a utilização de técnicas de PLN e também a utilização de padrões como a UML e XMI, utilizados largamente na indústria de software.

Um template consiste em uma estrutura de organização para determinado artefato e possui o objetivo de padronizar a sua construção. Na figura a seguir, tem-se um exemplo de template (LEAL; FIGUEIREDO; DE SOUZA, 2008).

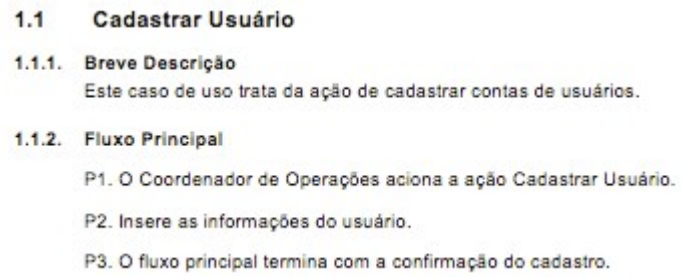


Figura 29. Exemplo de template (LEAL; FIGUEIREDO; DE SOUZA, 2008)

Na maioria dos processos de software, existe uma grande quantidade de artefatos baseados em texto. Nestes casos, os templates guiam a edição destes artefatos, indicando quais campos devem ser preenchidos e validando o mesmo por meio de regras de obrigatoriedade. O autor cita o exemplo da metodologia RUP, na qual existem diversos artefatos baseados em informações não-estruturadas, como especificações de casos de texto, documentos de visão, dentre outros (LEAL; FIGUEIREDO; DE SOUZA, 2008).

Para Leal, Figueiredo e De Souza (2008), uma das técnicas mais utilizadas para análise de documentos não-estruturados é a PLN, que possui vários níveis de execução, desde o nível morfológico, pragmático, semântico, estatístico e sintático. O autor propõe uma abordagem que utiliza uma combinação de análise sintática, pragmática e semântica para extrair as informações de artefatos, bem como seus relacionamentos a partir de documentos textuais, bem como diagramas UML em XMI. Com isto, consegue reconhecer a rastreabilidade explícita, que é a rastreabilidade pré-definida no modelo. Já a rastreabilidade implícita é obtida por meio da detecção de nomes de diferentes artefatos (*Name Tracing*). Atualmente, a ferramenta gera os seguintes tipos de matrizes de rastreabilidade: necessidade x funcionalidade; funcionalidade x ator ; ator x caso de uso; caso de uso x caso de uso; caso de uso x caso de teste ; caso de uso x diagrama de classes; classe x classe.

Na fase de aquisição dos dados nesta metodologia, os dados são descobertos por um *parser* e armazenados em um banco de dados. A partir de então, a análise propriamente dista é iniciada. Após obter a informação, ocorre a disponibilização dos resultados para o usuário final por meio de visualizações. A figura a seguir apresenta o resultado em um documento de

visão. No lado esquerdo é exibido o conteúdo do documento e a direita as informações extraídas hierarquicamente. A partir as informações armazenadas, é possível criar visualizações que mostrem a rastreabilidade em forma de matrizes, como pode ser visto a seguir.

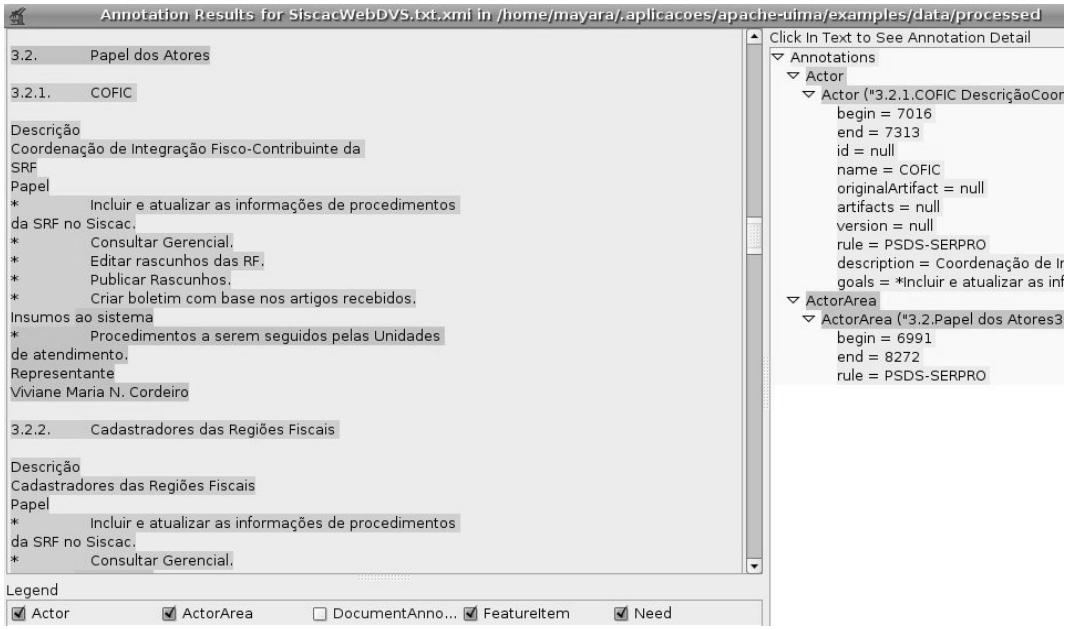


Figura 30. Ferramenta proposta por Leal, Figueiredo e De Souza (2008)

Dentre as características da ferramenta proposta por Leal, Figueiredo e De Souza (2008), estão o suporte à requisitos textuais, bem como modelos de *design* UML, a detecção automática da rastreabilidade entre elementos de um mesmo modelo (vertical) e entre elementos de modelos diferentes (horizontal) por meio da técnica de *Name Tracing*. A ferramenta proposta realiza *parsing* das estruturas complexas dos requisitos e dos modelos a fim de se obter as informações com maior grau de fidelidade, suporta utilização on-line, rastreabilidade, versionamento, integração com outras ferramentas e suporta a relatórios de análise de impactos.

Dentre as características ausentes na ferramenta proposta, destacam-se o suporte à configuração de projetos, suporte ao processo de gerenciamento das mudanças, suporte à notificação de eventos, representação de estruturas de código-fonte, estatísticas gerenciais, gerenciamento do processo (*workflow*), e gerenciamento pró-ativo, ou seja, a possibilidade de obter resultados sem a ação direta do usuário. Também nota-se a ausência da rastreabilidade *fine-grained* e de mecanismos de extensibilidade.

3.3.7 MÉTODO RM-TOOL

Para Lang e Duggan (2001), a especificação de requisitos consiste em documentos que evoluem e crescem rapidamente durante o ciclo de desenvolvimento de um software. Além disto, a especificação de requisitos envolve colaboradores com diferentes papéis e visões, fazendo com que o processo de comunicação ganhe proporções maiores. Desta forma, é importante que a especificação possa ser gerenciada e de preferência conduzida por uma ferramenta automatizada que observe os aspectos colaborativos.

Lang e Duggan (2001) apresentam um protótipo chamado RM-Tool, uma ferramenta *web* que busca gerenciar e controlar requisitos dentro de um processo multidisciplinar com equipes distribuídas. Este protótipo possui um escopo limitado e não ambiciona cobrir todos tipos de requisitos encontrados nas demais ferramentas. A ferramenta, que pode ser visualizada na figura a seguir, bem como o modelo proposto, concentram características que não são cobertas satisfatoriamente por ferramentas comerciais. Dentre suas características, estão a de facilitar o controle de mudanças e a colaboração entre os desenvolvedores por meio de notificação de eventos aos mesmos quando da alteração dos requisitos.

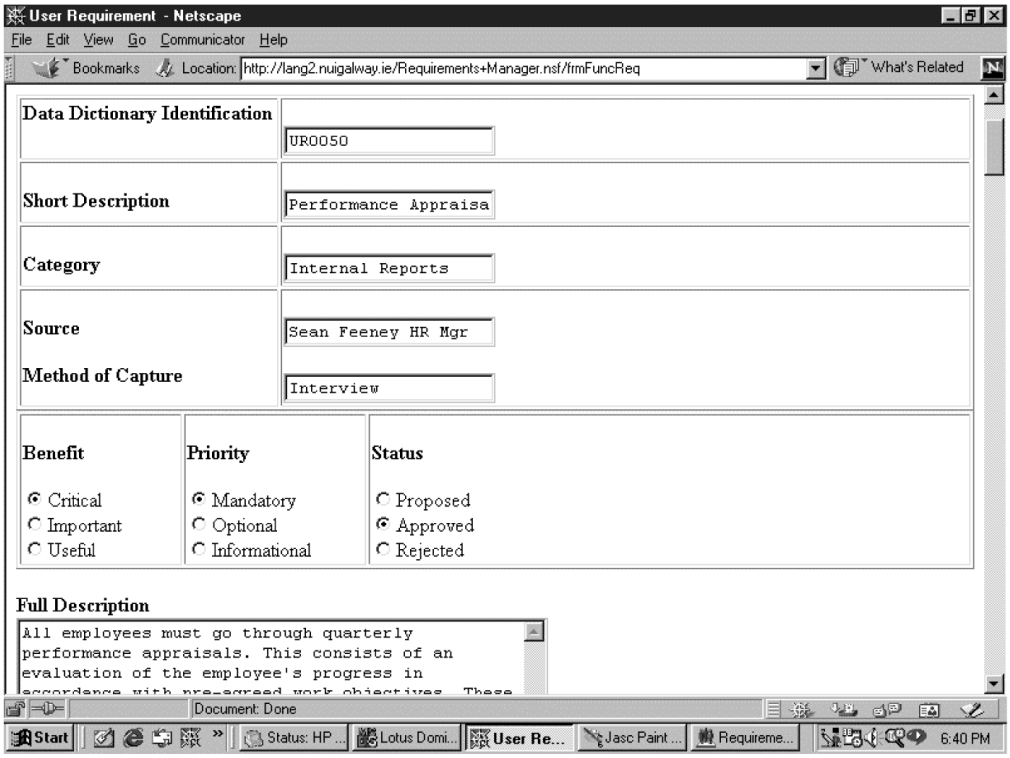


Figura 31. Ferramenta RM-Tool (LANG; DUGGAN, 2001)

A ferramenta, possui um dicionário de dados compartilhado entre os usuários, contendo descrições dos requisitos com *hyperlinks* entre as mesmas. Além disto, suporta documentos multimídia, comunicação entre colaboradores, rastreabilidade de requistios desde a sua origem e até sua implementação, gerência de projeto, relatórios gerenciais, dentre outros. A interligação dos requisitos por meio de *hyperlinks* permite que se obtenha o impacto de uma mudança, embora a ferramenta não ofereça este recurso atualmente.

A ferramenta é orientada para a concepção de requisitos textuais em linguagem natural, oferecendo diversos recursos para tal, como a importação de ferramentas externas como planilhas e editores de texto. Devido ao escopo reduzido, a ferramenta não suporta modelos de *design* UML ou código-fonte.

Dentre os recursos oferecidos pela mesma, estão a possibilidade de derivar requisitos a partir de outros, definir relacionamentos de dependência e evolução entre requisitos. A ferramenta permite ainda categorizar os requisitos em grupos lógicos, definir esforços, utilização de recursos, viabilidade técnica, priorização, além de armazenar para cada requisito o tempo e custo envolvido para alterações.

A ferramenta melhora a comunicação e colaboração entre os colaboradores por oferecer recursos de editoração online e controle de versões. Após o registro dos requisitos na ferramenta, os desenvolvedores realizam estimativas sobre a quantidade de esforço e recursos necessários para implementá-los e, a partir disto, os usuários são solicitados à classificarem os requisitos, aprovando-os ou rejeitando-os.

Dentre as caracterísitcas ausentes na ferramenta, destacam-se a análise de impacto, a recuperação de *links* de rastreabilidade, a rastreabilidade *fine-grained*, a análise dos artefatos (*parsing* da estrutura), a gestão pró-ativa (a ferramenta centraliza todas ações, cabendo ao usuário a iniciativa) e a possibilidade de extender suas funcionalidades.

3.3.8 MÉTODO ONTRACE

Para Noll e Ribeiro (2007), durante a modelagem de um sistema, é desenvolvido o modelo de domínio, que descreve os conceitos pertinentes ao sistema e relacionamentos entre estes, utilizando diagramas de classes UML. Para o autor, existem similaridades entre um modelo de domínio e uma ontologia, uma vez que ambos especificam conceitos e seus

relacionamentos. O modelo de domínio representa a aplicação sob o ponto de vista do negócio, com o objetivo de entender o sistema e compartilhar uma visão de conceito entre os colaboradores, sendo que uma ontologia também pode ser utilizada para representar uma conceitualização entre os colaboradores de um projeto.

Noll e Ribeiro (2007) propõem o uso de ontologias para suportar o processo de desenvolvimento de sistemas, principalmente o Processo Unificado (UP). Para Noll e Ribeiro (2007), ontologias podem ser utilizadas para especificar modelos conceituais que representem os artefatos de diferentes modelos do sistema, desde requisitos até o design. Para tal, é apresentada uma ferramenta chamada ONTrace, desenvolvida na forma de um plugin do software ArgoUML e que faz uso de ontologias para apoiar o processo de desenvolvimento de software baseado no processo unificado (UP), com o objetivo de integrar diferentes modelos de sistema.

A ferramenta ONTrace produz um modelo lógico que busca representar o domínio da aplicação por meio da ligação de elementos do software com conceitos de uma ontologia. A ferramenta ONTrace visa a integração de uma ontologia dentro do processo unificado, objetivando gerar *links* de rastreabilidade entre o modelo de negócios e o modelo de análise e *design*. O principal objetivo em desenvolver uma ontologia para o processo de desenvolvimento de software é a possibilidade de relacionar todos os artefatos e seus elementos, a fim de recuperar a rastreabilidade implícita, utilizando um mecanismo de inferência.

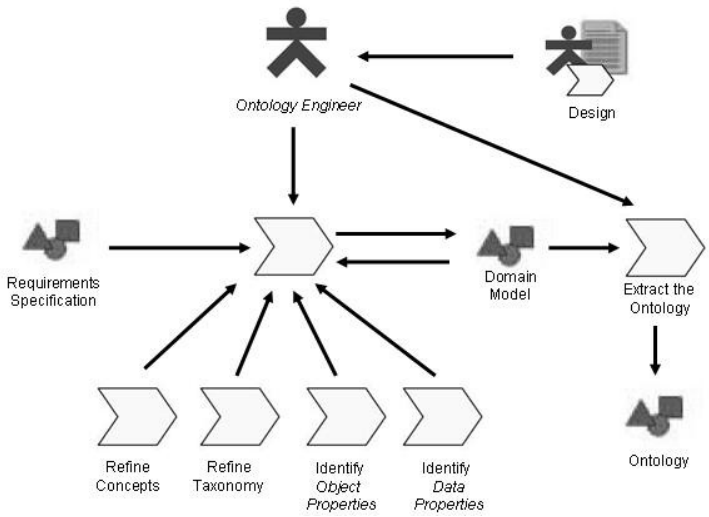


Figura 32. Atividade de *design* na ferramenta ONTrace (NOLL; RIBEIRO, 2007)

A metodologia construída sobre a ferramenta engloba três atividades: *design*, manutenção e validação. A atividade de *design* consiste na definição da ontologia e seus conceitos. A figura anterior procura demonstrar esta atividade. O papel de *Ontology Engineer* é responsável por extrair uma versão preliminar da ontologia com base na especificação de requisitos e no modelo de domínio, com base em determinadas regras de mapeamento.

Como um completo mapeamento entre um modelo orientado a objetos e um modelo lógico não é possível, devido à limitações no modelo de objetos em representar determinadas regras, a atividade de manutenção será responsável por identificar os conceitos da especificação relacionados ao domínio e refinar a ontologia de forma manual, observando aspectos de taxonomia organizacional. Como principal resultado, tem-se a geração de *links* de rastreabilidade entre conceitos de diferentes modelos, desde requisitos até o *design* (rastreabilidade horizontal).

A figura a seguir representa esta atividade. Nesta etapa, o papel *Ontology Engineer* irá atualizar a ontologia a partir da especificação de requisitos, do modelo de análise e *design* durante todo o ciclo de desenvolvimento, adicionando relacionamentos entre os elementos do modelo e refinando conceitos, propriedades e restrições.

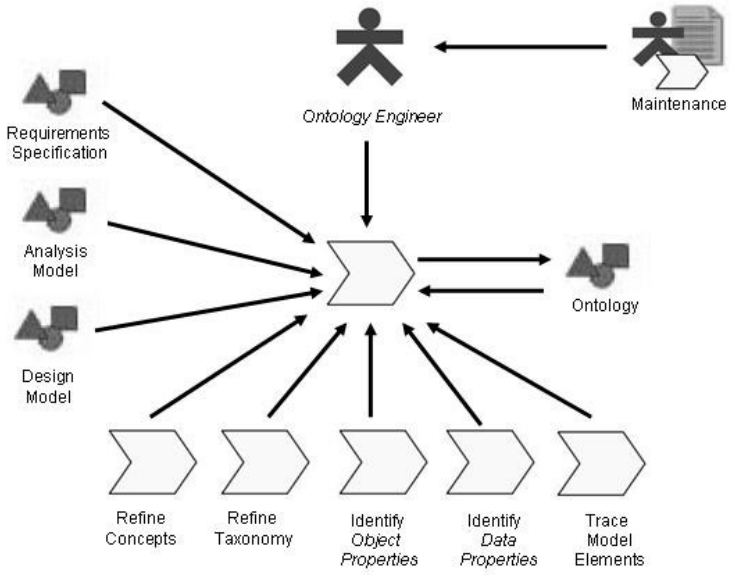


Figura 33. Atividade de manutenção na ferramenta ONTrace (NOLL; RIBEIRO, 2007)

Além das atividades já citadas, existe a atividade de validação, que engloba a avaliação de integridade da ontologia. Desta forma, para cada iteração do desenvolvimento, ocorre uma validação para verificar se o modelo lógico ainda é consistente. A figura a seguir procura demonstrar o mapeamento entre entidades de um modelo de *design* e a ontologia.

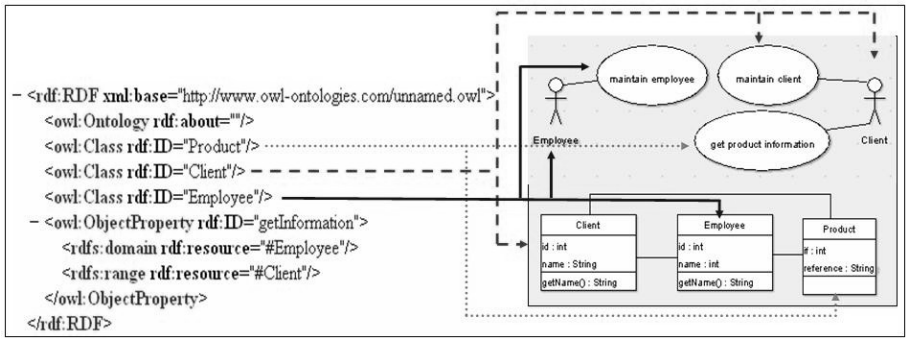


Figura 34. Validação do modelo na ferramenta ONTrace (NOLL; RIBEIRO, 2007)

A ferramenta não suporta análise de impacto diretamente, ou seja, não possui esta opção de forma explícita, embora possua todos os meios para se realizar análise de impacto por meio de consultas à ontologia. Em síntese, a ferramenta ONTrace permite a geração da ontologia a partir do modelo de classes do projeto construído pela ferramenta ArgoUML. Esta ontologia ainda pode ser importada e exportada a partir da ferramenta ArgoUML+ONTrace, uma vez que a ferramenta ONTrace é construída sobre o ArtoUML. Desta forma, é possível editar a ontologia, adicionando restrições de consistência por meio de ferramentas como o Protègè. Na figura a seguir é possível ter uma noção do funcionamento da ferramenta ONTrace.

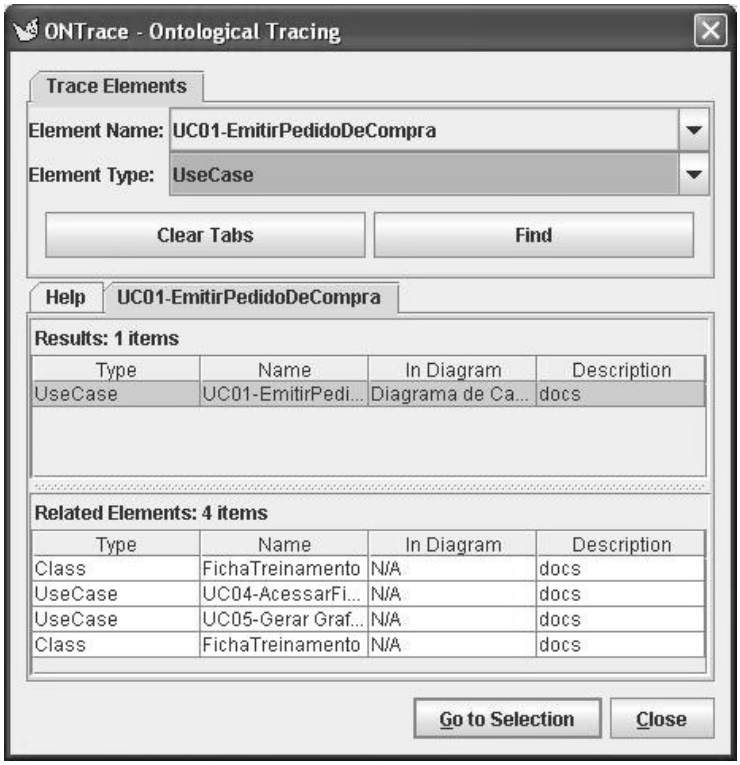


Figura 35. Interface principal da ferramenta ONTrace (NOLL; RIBEIRO, 2007)

A partir da ontologia, a ferramenta exibe os "recursos" encontrados, contendo classes, propriedades, dentre outros. Para cada elemento do modelo de *design*, é permitido relacionar recursos por meio da interface do ONTrace, que atualizará a ontologia automaticamente. A ferramenta permite também recuperar os *links* de rastreabilidade por meio de pesquisas à ontologia utilizando um mecanismo de inferência. Desta forma, o usuário poderá buscar os elementos relacionados na própria interface.

Dentre os principais recursos da ferramenta ONTrace destacam-se o suporte refinado à modelos de *design* UML, permitindo *parsing* de suas estruturas internas por meio dos arquivos XMI, integração com uma ferramenta CASE, o suporte a rastreabilidade fine-grained e a possibilidade de recuperar a informação de rastreabilidade por meio de consultas à ontologia.

Dentre as características ausentes merecem citação: a ausência de mecanismos para o gerenciamento de mudanças e análise de impacto, mecanismos de notificação aos colaboradores sobre mudanças no modelo, ausência de interface *web*, de suporte à outros artefatos de software como o código-fonte, gestão pró-ativa, mecanismos de *workflow*, informações gerenciais e mecanismos de extensibilidade.

3.4 MÉTRICAS

As métricas tem evoluído com o tempo dentro da Engenharia de Software e, em algumas organizações, já são implementadas de forma pró-ativa (HAZAN; LEITE, 2003). Para garantir a qualidade de um processo, é fundamental ter acesso à informações qualitativas precisas, tais como as fornecidas por indicadores. Para tal, Hazan e Leite (2003) propõe indicadores que possam ser utilizados dentro do processo da gerência de requisitos. Estes indicadores foram divididos em dois grupos: Indicadores de Estabilidade e Indicadores de Rastreabilidade. Enquanto o primeiro grupo está mais voltado para medir o grau de mudanças nos requisitos e o impacto destas mudanças, o segundo mede a aderência dos artefatos à seus requisitos associados.

3.4.1 INDICADORES DE ESTABILIDADE

Conforme Hazan e Leite (2003), estes indicadores atendem a área de processo "Medição e Análise" do nível 2 do CMMI, que objetiva "desenvolver e sustentar uma capacidade de medição usada para suportar gerencialmente as necessidades de informação". A seguir, um quadro com os indicadores de estabilidade propostos por Hazan e Leite (2003).

Tabela 1 -Quadro de indicadores de estabilidade de requisitos (HAZAN; LEITE, 2003)

Requisitos novos no período	
Unidade	**Descrição**
Quantidade	Número de requisitos novos criados no período
Percentual	Número de requisitos novos / total requisitos alocados
Requisitos alterados no período	
Unidade	**Descrição**
Quantidade	Número de requisitos alterados no período
Percentual	Número de requisitos alterados / total requisitos alocados
Requisitos excluídos no período	
Unidade	**Descrição**
Quantidade	Número de requisitos excluídos no período
Percentual	Número de requisitos excluídos / total requisitos alocados

Nurmuliani, Zowghi e Fowell (2004) adicionam ainda à este grupo, o seguinte indicador de volatilidade dos requisitos: (Número de requisitos incluídos + Número de requisitos alterados + Número de requisitos excluídos) / total de requisitos.

Para melhor demonstrar os indicadores, Hazan e Leite (2003) sugerem a utilização de gráficos para análise. Dentre os gráficos sugeridos, destaca-se o gráfico quantitativo de mudanças, estratificado por tipo de mudança e por responsável, como pode ser visto a seguir.

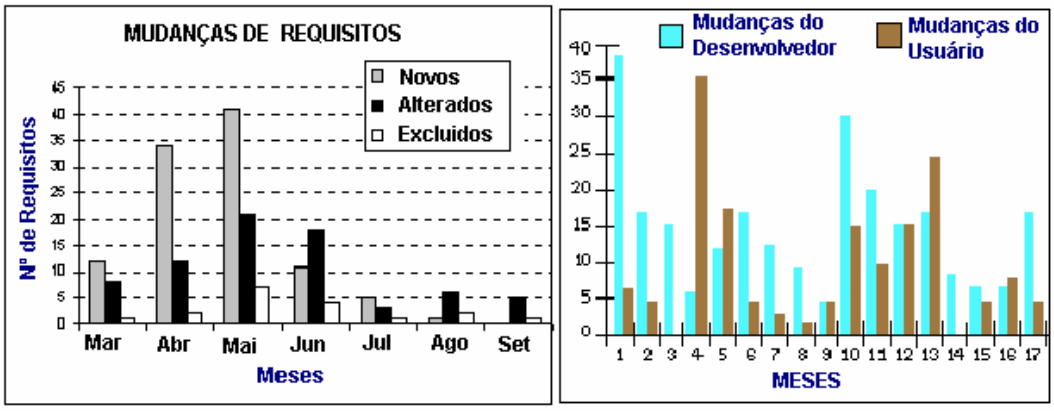


Figura 36. Gráfico com indicadores de estabilidade (HAZAN; LEITE, 2003)

3.4.2 INDICADORES DE RASTREABILIDADE

Para Hazan e Leite (2003), a rastreabilidade consiste em uma informação valiosa para que se possa obter melhor entendimento sobre os relacionamentos entre os artefatos de um projeto. Em sistemas complexos, esta informação pode ser utilizada para garantir que determinado conjunto de artefatos satisfaça determinado conjunto de requisitos, auxiliando na verificação e validação. Para tal, Hazan e Leite (2003) propõem um grupo de indicadores de rastreabilidade, conforme podem ser vistos na tabela a seguir.

Tabela 2 -Indicadores de rastreabilidade de requisitos (HAZAN; LEITE, 2003)

Requisitos rastreáveis até a origem (pré-rastreabilidade)	
Unidade	**Descrição**
Quantidade	Número de requisitos rastreáveis até a origem
Percentual	Número de requisitos rastreáveis para a sua origem/total
Requisitos rastreáveis até a próxima etapa (pós-rastreabilidade)	
Unidade	**Descrição**
Quantidade	Número de requisitos rastreáveis para a próxima atividade.
Percentual	Número de requisitos rastreáveis a próxima atividade/total
Impacto dos requisitos modificados (efeitos)	
Unidade	**Descrição**
Percentual	Número de requisitos impactados/ total requisitos alocados

A figura a seguir procura exibir os indicadores de rastreabilidade. Para Hazan e Leite (2003), estes indicadores permitem apresentar o progresso da rastreabilidade do software ao longo de seu ciclo de desenvolvimento.

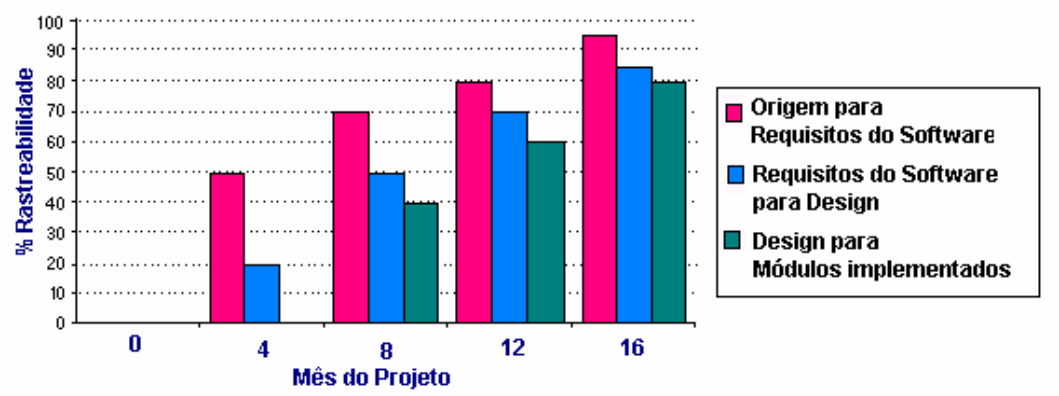


Figura 37. Indicadores de rastreabilidade (HAZAN; LEITE, 2003)

3.4.3 INDICADORES DE PROGRESSO

Além dos indicadores já vistos, Hazan e Leite (2003) sugerem o indicador de progresso - Percentual de Abrangência de Requisitos, como pode ser visto na figura a seguir. Este indicador procura exibir o percentual de artefatos implementados em cada iteração em relação aos requisitos estabelecidos para o projeto.

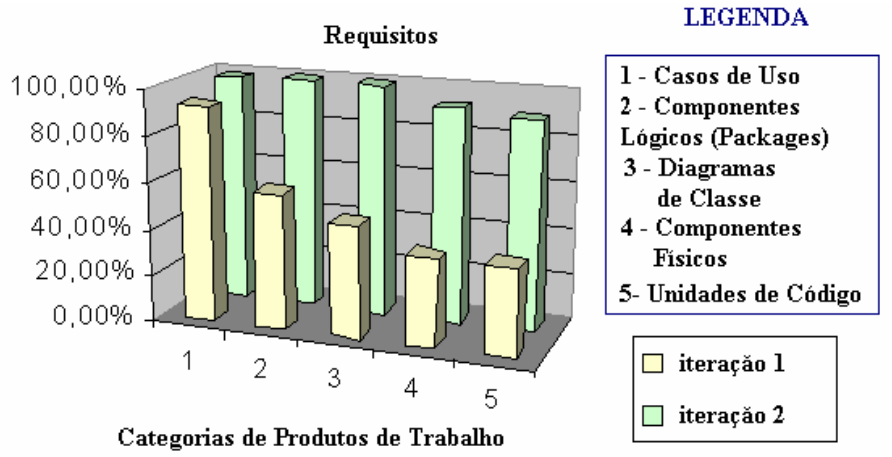


Figura 38. Indicador de Progresso (HAZAN; LEITE, 2003)

4 TRABALHO PROPOSTO

Neste capítulo será apresentado o trabalho proposto pela presente dissertação, que consiste na implementação de uma ferramenta *web* apoiada por agentes de software que controlem com precisão a informação de rastreabilidade e suportem de forma pró-ativa a gestão da mudança de requisitos e a análise de impactos durante todo o ciclo de desenvolvimento de software. Para que isto seja possível, torna-se necessário a criação de um modelo para representar com precisão a informação de rastreabilidade em um projeto, bem como as mudança ocorridas e os impactos gerados. Desta forma, este capítulo apresentará a modelagem da ferramenta, sua arquitetura, o método utilizado pela mesma para atingir os objetivos e o modelo de representação criado para suportar as funcionalidades desenvolvidas.

4.1 VISÃO GERAL

Com base na proposta de trabalho e nos requisitos descobertos, é proposto um método para gerência da mudança requisitos suportado por agentes de software. Para detalhar melhor o conceito deste método, foi elaborada uma figura, que pode ser vista logo a seguir da seguinte lista, formada por três dimensões (ambiente, agentes e modelo), que representam os três aspectos diferentes gerenciados pela ferramenta proposta. As dimensões são:

- **Dimensão Ambiente**: Esta dimensão representa os artefatos do sistema, desde requisitos textuais, modelos de *design* UML, até o código-fonte, em seu ambiente de desenvolvimento. A idéia principal desta dimensão é que os artefatos estejam representados no formato original no qual foram criados, seja por editores de texto, ferramenta *CASE* ou *IDE* e estejam armazenados em um repositório central gerenciado por uma ferramenta de controle de versões (*C.V.S.*, *subversion, git*);
- **Dimensão Agentes**: Esta dimensão representa os agentes de software que compõem a aplicação. Estes agentes de software são responsáveis por tarefas que vão desde a leitura e interpretação dos artefatos de software (dimensão "ambiente") a partir de seus formatos nativos, entendimento e estabelecimento da rastreabilidade, verificação de inconsistências do modelo, comparação entre diferentes versões de um modelo, detecção de mudanças em um modelo, notificação dos *stakeholders* envolvidos em um projeto, realização de análises de impacto e estabelecimento de métricas;

- **Dimensão Modelo**: Esta dimensão representa os fatos conhecidos sobre a dimensão "ambiente" registrados pelos agentes. Estes fatos serão descobertos pelos agentes de software e serão armazenados pela base de dados da aplicação. Esta base de dados irá armazenar todas informações relativas aos artefatos do sistema, bem como os relacionamentos entre eles, seja o relacionamento entre artefatos de um mesmo modelo (rastreabilidade vertical), quanto o relacionamento entre artefatos de modelos diferentes (rastreabilidade horizontal). As informações armazenadas nesta base de dados darão suporte ao trabalho dos agentes de software, permitindo a realização de análises de impacto, além do estabelecimento de métricas.

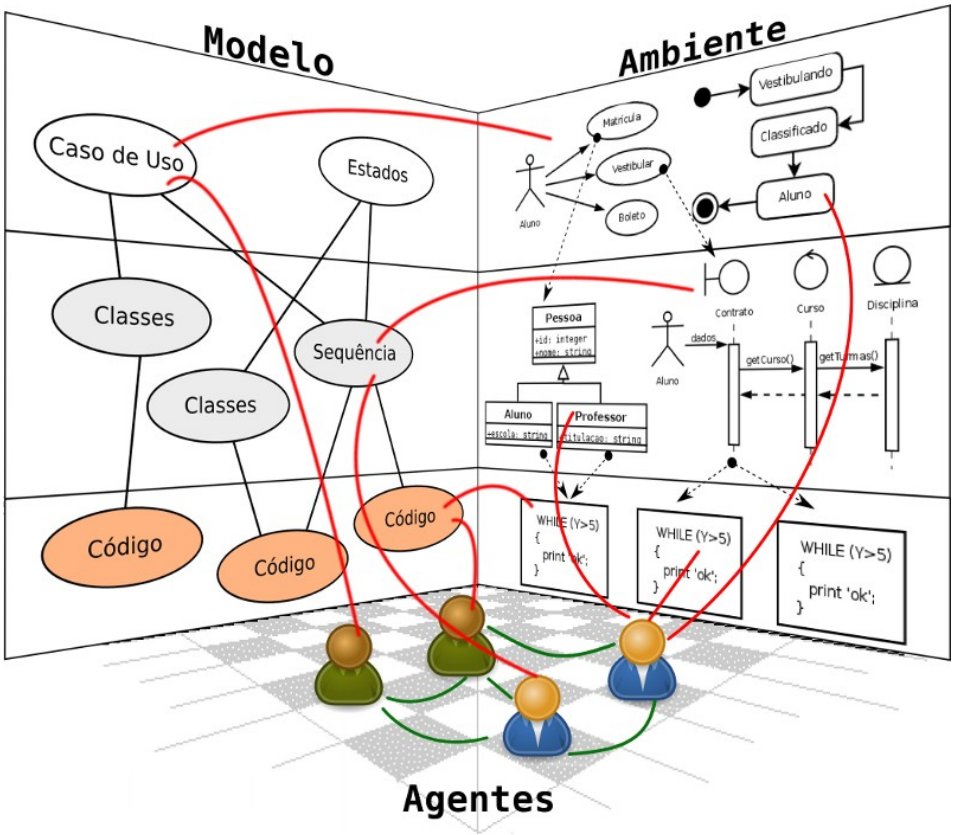


Figura 39. Visão geral do trabalho proposto

4.2 OBJETIVOS DO SISTEMA

Para modelar o comportamento dos agentes (representados pela dimensão "agentes"), será utilizada a metodologia MaSE. A primeira fase da metodologia trata da descoberta dos objetivos, baseada na especificação inicial do sistema, que pode incluir documentos técnicos,

user stories ou especificações formais. O resultado é um conjunto de metas estruturadas de forma hierárquica, como na figura a seguir. Uma vez definidos os objetivos do sistema, a probabilidade de mudança neles é menor nas atividades subsequentes.

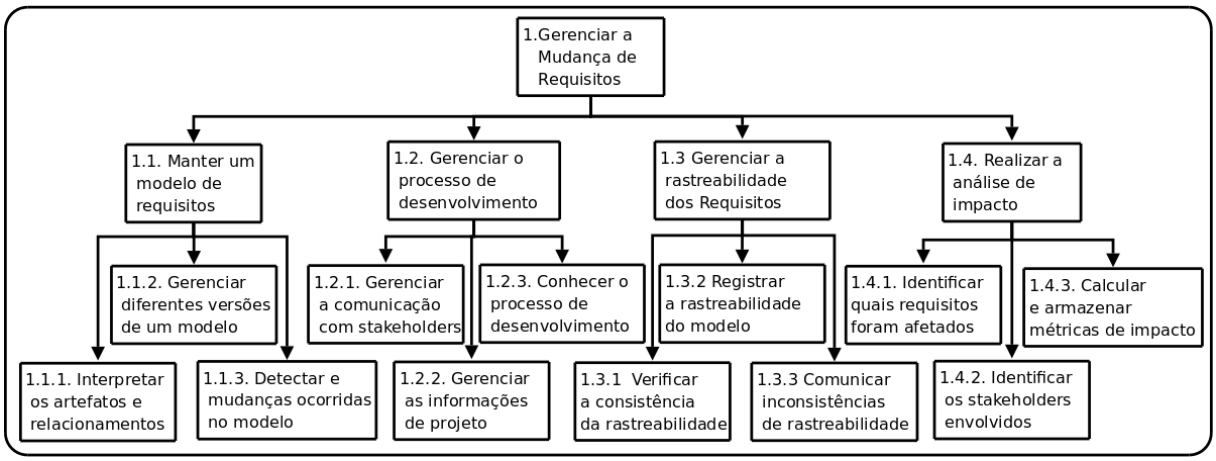


Figura 40. Objetivos conforme metodologia MaSE

O objetivo principal da aplicação é gerenciar a mudança de requisitos. Para tal, este objetivo deve ser desmembrado em quatro responsabilidades distintas e necessárias para o sucesso da aplicação da metodologia proposta, que são descritos a seguir:

- **Manter um modelo de requisitos**: Representa as atividades necessárias ao gerenciamento dos artefatos do software, sejam requisitos textuais ou modelos de *design* ou código-fonte, representados pela dimensão "modelo". Para manter um modelo, é necessário interpretar os artefatos e seus relacionamentos, ou seja, conseguir ler e analisar as informações de como os artefatos estão estruturados e relacionados uns com os outros, quer sejam requisitos textuais, artefatos de *design* como um diagrama de classes, diagrama de casos de uso ou um arquivo contendo o código-fonte da aplicação. Também será necessário detectar e registrar as mudanças que ocorrerem no modelo, tais como a inclusão, alteração e exclusão de artefatos. Quando tais mudanças ocorrerem, será necessário manter uma *baseline* dos requisitos, permitindo gerenciar diferentes versões do modelo e realizar comparações entre versões;
- **Gerenciar o processo de desenvolvimento**: Compreende as atividades necessárias para o gerenciamento do processo de desenvolvimento. Esta etapa compreende o conhecimento sobre o processo de desenvolvimento de software, suas etapas, seus papéis e seus produtos de trabalho, compreende a comunicação com os *stakeholders*, como a notificação sobre mudanças ocorridas no modelo e sobre inconsistências de

rastreabilidade geradas por uma mudança. Compreende a solicitação de informações aos *stakeholders*, como a classificação das mudanças ocorridas e o envio de relatórios de impacto. Por fim, compreende o gerenciamento de informações de projeto, tais como alocação de pessoal e prioridades;

- **Gerenciar a rastreabilidade dos requisitos**: Como visto anteriormente, para realizar uma correta gestão da mudança de requisitos, uma das formas compreende apoiar-se nas informações de rastreabilidade. Desta forma, torna-se importante que tais informações estejam não somente disponíveis, mas também manipuláveis no sistema, para que se possa efetuar uma correta análise de impacto. Conforme os objetivos listados a seguir, esta atividade compreende a verificação da consistência da rastreabilidade de um modelo, procurando por falta de integridade nos relacionamentos, compreende o registro das informações de rastreabilidade do modelo e também a comunicação das inconsistências encontradas;
- **Realizar a análise de impacto**: Compreende as atividades necessárias para a identificação do conjunto de requisitos impactados por uma determinada mudança, a classificação de mudanças e a identificação dos *stakeholders* envolvidos, e também o cálculo de métricas para avaliar o impacto de cada mudança;

4.3 ARQUITETURA DO SISTEMA

A arquitetura do sistema, representada pela próxima figura, será modular e composta por quatro “fronteiras”. No interior da aplicação, encontram-se os agentes responsáveis por realizar os objetivos traçados anteriormente, sendo que estes agentes serão detalhados posteriormente. A seguir, constam os componentes da arquitetura:

- **Agentes**: O sistema é composto por quatro agentes: *TraceabilityManager*, *ImpactAnalyser*, *RequirementEngineer* e *ProcessManager*, que são responsáveis pelos objetivos e são representados por *Web Services* detalhados em seguida;
- **Integração**: Representa a camada lógica responsável pela interpretação do modelo utilizado pela aplicação a partir do repositório de artefatos, que irá conter requisitos textuais, modelos de *design* e código-fonte;

- **Repositório de Artefatos**: Representa um repositório centralizado (*subverison*) onde os artefatos do sistema (requisitos textuais, modelos de *design* e código-fonte) estarão armazenados;
- **Persistência**: Representa a camada lógica responsável por armazenar e recuperar o "conhecimento" obtido sobre os artefatos de software (dimensão "modelo"), bem como seus relacionamentos no banco de dados;
- **Banco de dados**: Representa a base de dados da aplicação, que irá armazenar o modelo de artefatos descoberto pelos agentes de software.
- **Inferência**: Representa a camada lógica responsável pela comunicação com a máquina de inferência externa que irá executar consultas em lógica descritiva baseada no "conhecimento" sobre o modelo a fim de realizar a análise de impacto;
- **Máquina de Inferência**: Representa uma máquina de inferência Prolog responsável por executar consultas em lógica descritiva e retornar resultados para a aplicação;
- **Interface**: Representa a interface dos agentes, que serão disponibilizadas por *Web Services* para outras aplicações que poderão consumir estes serviços. O próprio protótipo apresentado neste trabalho é um exemplo de aplicação que consome estes *Web Services;*
- **Outras APPS**: Representa outras aplicações que podem ser escritas para utilizar o serviços dos agentes, que são implementados na forma de *Web Services*.

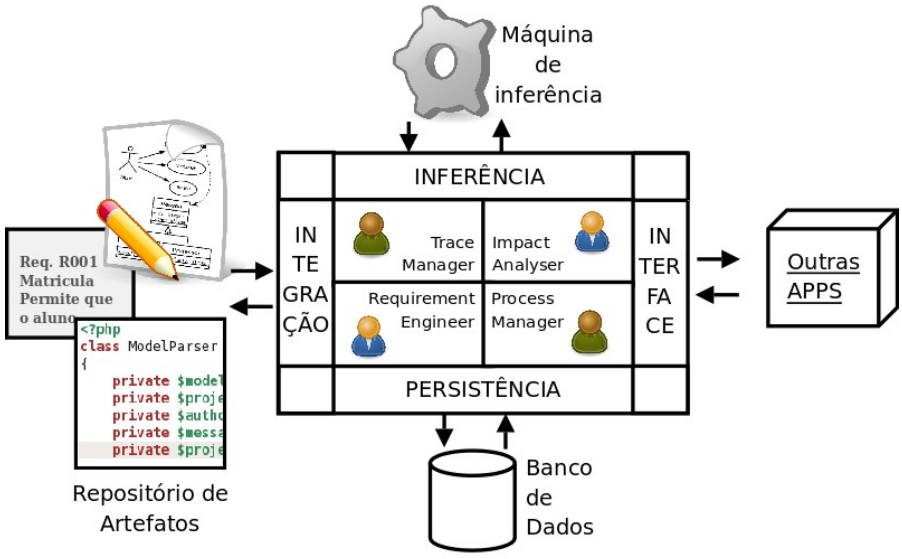


Figura 41. Arquitetura proposta

4.4 TECNOLOGIAS ADOTADAS

Para o desenvolvimento da ferramenta proposta, serão utilizadas as tecnologias listadas a seguir. Deve-se citar que todas são tecnologias livres e independentes de plataforma:

- **Apache 2:** O Apache (www.apache.org) é um servidor de páginas web mundialmente reconhecido por sua qualidade e segurança.
- **PHP 5 :** O PHP (www.php.net) é uma linguagem de programação de última geração voltada originalmente para a web com grandes recursos de orientação a objetos e flexibilidade devido à sua tipagem dinâmica.
- **PostgreSQL 8:** O PostgreSQL (www.postgresql.org) é um banco de dados relacional robusto, conta com um grande poder de armazenamento de informações e é compatível com os padrões SQL.

4.5 MODELAGEM DOS AGENTES

Como visto anteriormente, o sistema será composto por um conjunto de agentes responsáveis por implementar os objetivos traçados. Para De Loach (2000), o próximo passo na metodologia MaSE após definir os objetivos dos agentes, é transformar esta hierarquia de objetivos em papéis. Os papéis definem as classes de agentes e representam os objetivos do sistema durante a etapa de projeto. Cada objetivo é garantido por um papel desempenhado por um agente. Existem situações onde é possível combinar objetivos similares com um grande grau de coesão em um único papel. Recursos externos como bancos de dados, bem como interfaces com o usuário, necessitam um papel individual (DE LOACH, 2000).

A aplicação proposta terá quatro classes de agentes que serão responsáveis por atender os objetivos principais da aplicação: *RequirementEnginner*, *TraceabilityManager*, *ImpactAnalyser* e *ProcessManager*, além de recursos externos, que são representadas pelas classes *Model* (modelo), que representa o conjunto de artefatos do sistema, *Stakeholder* que representa o usuário da aplicação, BD, que representa o banco de dados da aplicação, *Interface*, que representa o *controller* da interface da aplicação, por meio do qual, o usuário poderá disparar ações e *TimeManager*, um processo de software que controla execução

periódica de ações no sistema, todas destacadas por uma cor diferente. Cada agente é representado pelo seu nome juntamente com os objetivos que ele atende representados pela numeração logo abaixo do nome. A figura a seguir demonstra a interação entre as classes de agentes. Em seguida, os quatro papeis principais serão detalhados.

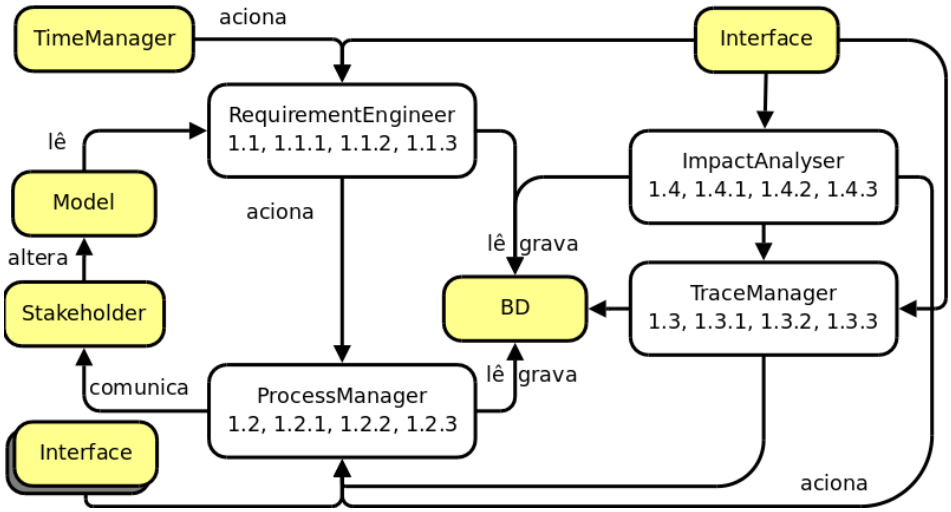


Figura 42. Papéis dos agentes

4.5.1 REQUIREMENTENGINEER

O agente *RequirementEngineer* será responsável pelo objetivo "Manter um modelo de requisitos", que envolve sub-objetivos como interpretar os artefatos, detectar as mudanças e gerenciar as versões do modelo. É acionado periodicamente pelo processo gerenciador de tarefas no tempo *TimeManager* e efetua a leitura de *Model* (modelo), que representa os artefatos do sistema. Aciona o agente *ProcessManager* para notificar os *Stakeholders* sobre as mudanças ocorridas. Pode ser acionado pela interface (recurso *Interface*) do sistema pelo usuário para prover relatórios como listagens de artefatos do sistema.

A principal função do agente *RequirementEngineer* é ler e interpretar o conjunto de artefatos utilizados no desenvolvimento de software. Para tal, este agente é apoiado por uma estrutura de classes responsáveis pela interpretação de diversos tipos de artefatos tais como: documentos de requisitos armazenados em arquivos ODF (*Open Document Format*), padrão internacional homologado pela ISO (*International Organization for Standardization*), modelos de *design* UML armazenados em arquivos XMI (*XML Metadata Interchange*),

padrão de mercado mantido pelo OMG (*Object Management Group*) e também um padrão ISO, e códigos-fonte de aplicações PHP ou Java, atualmente as duas linguagens mais populares do mundo a suportarem a orientação a objetos, conforme o índice Tiobe (2009).

A figura a seguir procura demonstrar o funcionamento deste agente à esquerda e sua interface disponibilizada via *Web Services* à direita. A primeir figura demonstra que o agente realiza a leitura destes artefatos e armazena as informações obtidas, bem como os relacionamentos existentes entre eles no banco de dados da aplicação. A segunda figura demonstra os métodos disponibilizados pelo agente *Requirement Engineer*, que é construído na forma de um *Web Service*. Dentre os principais métodos, podem ser citados os métodos *getters*, como *getArtifact* e *getChange*, que retornam um objeto com base em seu código, os métodos para listagens e relatórios como *listArtifacts* e *listChanges* e o método *parseProjectStrucutre*, que efetua a leitura da estrutura de um projeto para armazenar suas informações na base de dados.

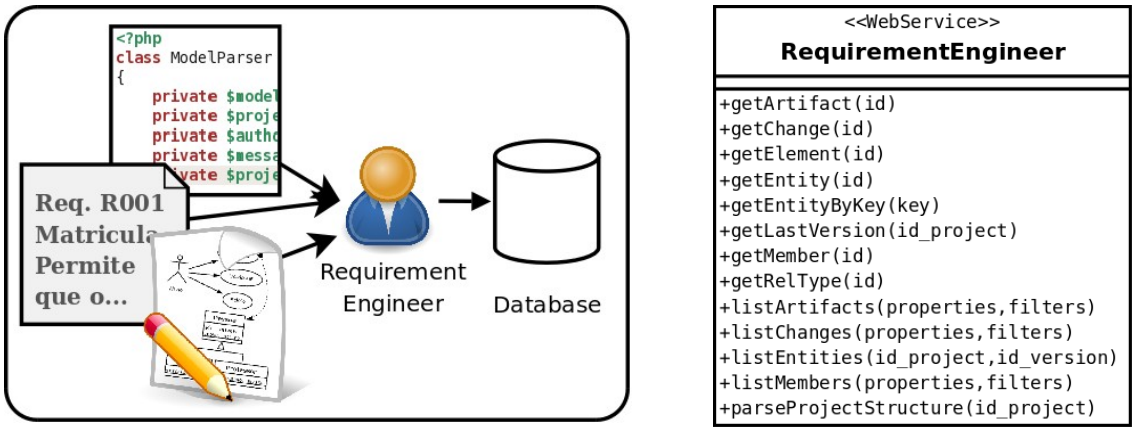


Figura 43. Funcionamento do agente *RequirementEngineer*

4.5.2 TRACEABILITYMANAGER

O agente *TraceabilityManager* será responsável pelo objetivo "Gerenciar a rastreabilidade dos requisitos", que envolve sub-objetivos como registrar a rastreabilidade, verificar e comunicar suas inconsistências. É acionado pelo usuário por meio da interface (*Interface*) do sistema para registrar a rastreabilidade horizontal por via manual ou para descobrir os links de rastreabilidade horizontal entre os artefatos de forma semi-automática

por meio da similaridade entre os termos utilizados. É acionado pelo agente *ImpactAnalyser* para verificar as inconsistências na rastreabilidade e aciona o agente *ProcessManager* para realizar comunicações aos *Stakeholders*. Também pode ser acionado pelo usuário por meio da interface (recurso *Interface*) para prover relatórios de relacionamento entre os artefatos.

A próxima figura procura ilustrar o funcionamento deste agente, bem como demonstrar sua interface. A primeira figura procura transmitir a idéia de que este agente é o responsável em zelar pelo relacionamento entre os artefatos de software que integram o projeto, ao longo do ciclo de vida do mesmo, desde a análise até o projeto. Na segunda figura, tem-se a interface do *Web Service* que implementa este agente. Dentre os principais métodos podem ser destacados: *autoTraceArtifacts*, responsável por identificar a rastreabilidade horizontal de forma semi-automática, *checkTracability*, que irá buscar por inconsistências na rastreabilidade, *listRelations*, que irá listar os relacionamentos entre os artefatos do sistema e *registerTraceability*, que registra uma relação de rastreabilidade horizontal no banco de dados.

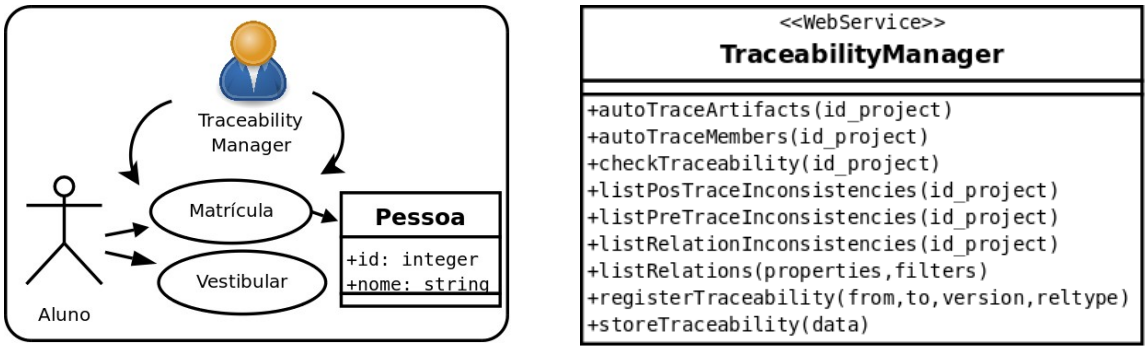


Figura 44. Funcionamento do agente *TraceabilityManager*

4.5.3 IMPACTANALYSER

O agente *ImpactAnalyer* será responsável pelo objetivo "Realizar a análise de impacto", que envolve sub-objetivos como identificar artefatos e *Stakeholders* envolvidos em um impacto, bem como calcular e armazenar métricas de impacto, conforme estudo de Hazan e Leite (2003). É acionado pelo usuário por meio da interface (recurso *Interface*) para desempenhar seu trabalho a partir da classificação das mudanças ocorridas no sistema. Além disto, aciona o agente *ProcessManager* para realizar comunicações aos *Stakeholders*;

A análise de impacto será realizada por meio da exportação da informação que os

agentes possuem sobre os artefatos e seus relacionamentos a partir da base de dados para um conjunto de axiomas em Prolog contendo toda a estrutura de inter-dependência do projeto. A partir de um conjunto de funções em lógica descritiva, este agente será capaz de identificar, a partir de uma mudança, todo o conjunto de artefatos afetados no projeto.

A figura a seguir procura demonstrar a idéia geral da responsabilidade do agente e também a sua interface implementada na forma de um *Web Service*. A primeira figura procura demonstrar a responsabilidade do agente, que é a de identificar possíveis impactos a partir de uma determinada mudança, de forma recursiva. A segunda figura procura demonstrar as funcionalidades deste agente representadas por métodos como: *classifyChange*, responsável por registrar a classificação de uma mudança, *getAffectedRequirements*, que realiza a análise de impacto, descobrindo os artefatos impactados, alguns métodos, alguns métodos que retornam estatísticas de mudanças como *getChangesByOrigin* e *getChangesByReason*, e estatísticas de impactos como *getImpactsByOrigin* e *getImpactsByReason*.

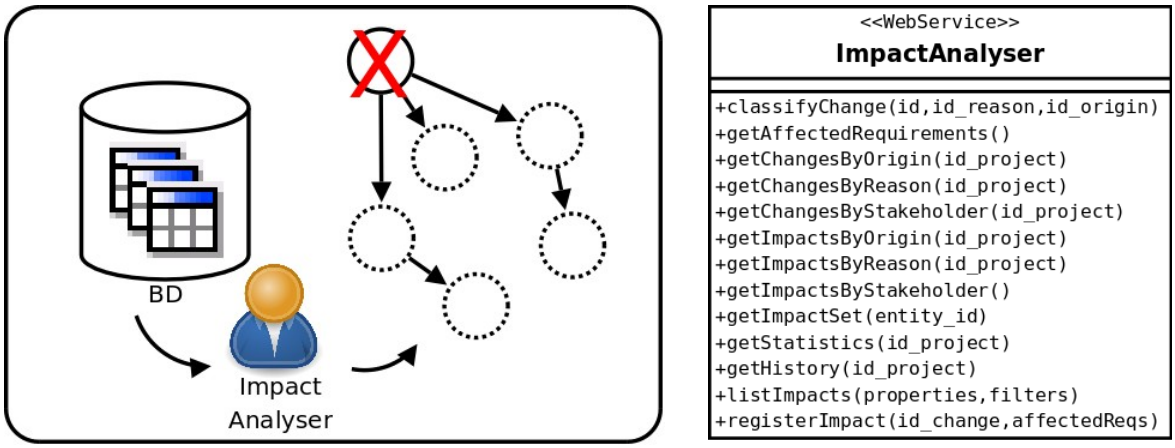


Figura 45. Funcionamento do agente *ImpactAnalyser*

4.5.4 PROCESSMANAGER

O agente *ProcessManager* será responsável pelo objetivo "Gerenciar o processo de desenvolvimento", que envolve sub-objetivos como conhecer o processo de desenvolvimento, gerenciar as informações de projeto e a comunicação com os *Stakeholders*. É acionado pelos agentes *RequirementEnginner*, *TraceabilityManager* e *ImpactAnalyser* para realizar comunicações aos *Stakeholders*;

O agente *ProcessManager* será responsável por conhecer as informações sobre o projeto em desenvolvimento, tais como seus membros, atribuições e alocações, além de gerenciar a comunicação com os *stakeholders*. Este agente é responsável por conhecer as etapas do processo de desenvolvimento, bem como os responsáveis e os produtos de trabalho de cada etapa. A partir o conhecimento que possui sobre o projeto, o agente *ProcessManager* notifica os *stakeholders* sobre mudanças e impactos ocorridos no projeto. Para tal, ele descobre quais os papéis foram envolvidos a partir de informações da análise de impacto e notifica os responsáveis.

A figura a seguir, demonstra o funcionamento do agente *ProcessManager*, bem como sua interface. A figura à esquerda procura demonstrar seu funcionamento, demonstrando que este agente que deverá notificar os stakeholders envolvidos em eventos, como a análise de impactos. Já a segunda figura procura demonstrar as funcionalidades oferecidas por este agente implementado na forma de um *Web Service*. Dentre os métodos oferecidos estão aqueles necessários para a manutenção do projeto como *deleteProject*, *getProject* e *storeProject* respectivamente para excluir, retornar e armazenar um projeto, métodos para listagens como *listProjects* e *listStakeholders*, e métodos para notificação de eventos como *notifyChange* para notificar sobre uma mudança ocorrida, *notifyImpact* para notificar sobre os impactos e *notifyInconsistencies* para notificar sobre inconsistências de rastreabilidade.

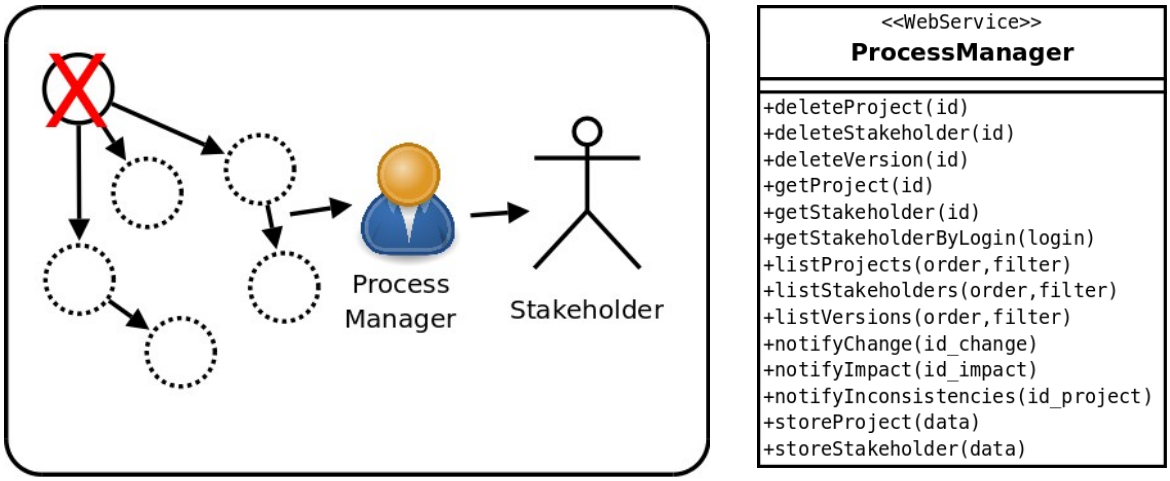


Figura 46. Funcionamento do agente *ProcessManager*

4.6 MODELO

Para Mäder, Gotel e Philippow (2009), a essencia de um método está na definição e utilização de um modelo que suporte a informação de rastreabilidade. Estes modelos fornecem suporte para decidir quais são os artefatos de software a serem armazenados, bem como quais relacionamentos devam ser definidos. Para Cimitile (1999), para que o processo de análise de impacto possa identificar os artefatos afetados por mudanças de sistema é necessário um modelo de representação do software que possa formalizar o conhecimento sobre as dependências entre artefatos.

De acordo com Pinheiro (2000), um modelo deve definir apenas um tipo de relacionamento para indicar a rastreabilidade entre os objetos e deixar a interpretação do mesmo fora do modelo. Apenas em alguns casos muito específicos, é possível determinar um relacionamento de forma totalmente precisa. De maneira geral, a interpretação é deixada fora do modelo (PINHEIRO, 2000).

Na sequência desta proposta, um modelo será apresentado. Este modelo visa representar toda estrutura do sistema que será desenvolvido para suportar o método do presente trabalho. Este modelo será alimentado pelas atividades dos agentes de software, que irão registrar as versões do modelo, seus requisitos textuais, seus artefatos de *design*, as relações entre os requisitos, as mudanças, os impactos, dentre outros.

O desenvolvimento de um novo modelo se faz necessário para contemplar o conjunto de requisitos necessário para o desenvolvimento do método proposto. Esta conclusão poderá ser analisada melhor no capítulo seis, onde o modelo proposto será comparado com os demais modelos estudados. Inicialmente, o modelo será criado por aspectos, sendo que, cada um dos aspectos será abordado separadamente para após serem consolidados em um modelo único.

4.6.1 VERSIONAMENTO

Inicialmente, o modelo deverá ser capaz de armazenar diferentes versões de um modelo, permitindo que se comparem versões diferentes, bem como se tenha o histórico de todas alterações que ocorreram no modelo ao longo do tempo. Estruturas similares com o mesmo objetivo foram encontradas nos trabalhos de Briand, Labiche e O'Sullivan (2003) e De

Grande e Martins (2006).

Briand, Labiche e O'Sullivan (2003) propõem uma abordagem para análise de impactos baseada em um modelo UML. Nesta abordagem, já manifestavam a necessidade de versionamento de um modelo de requisitos, inclusive indicando isto em seu modelo por meio da replicação de estruturas compostas. Entretanto, a preocupação de Briand, Labiche e O'Sullivan (2003) residia principalmente em artefatos de *design*, sendo que requisitos textuais ou código-fonte não são citados em seu modelo.

Este aspecto também é encontrado no trabalho de De Grande e Martins(2006), que considera versionamento de requisitos por meio de uma classe de versionamento. Salienta-se que tal abordagem simplificada funciona devido ao modelo ser focado apenas em requisitos textuais.

Como forma de consolidar as idéias apresentadas em um modelo único, é apresentado o primeiro fragmento do modelo proposto, conforme pode ser visto pela figura a seguir. Cabe ressaltar que uma nova versão do modelo é criada pelo agente *RequirementEngineer* sempre que for detectada uma modificação em algum dos artefatos (*Element*) ou em seus relacionamentos (*Relation*).

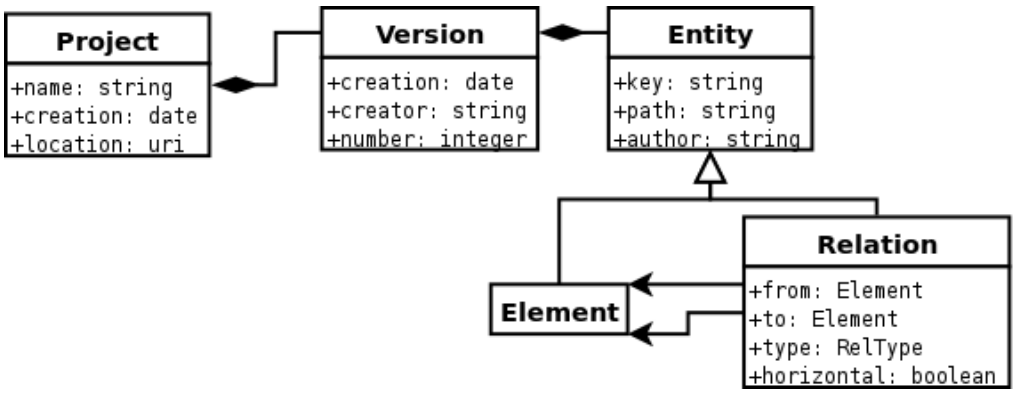


Figura 47. Aspectos de versionamento do modelo

Neste modelo, determinado projeto (*Project*) é composto de versões (*Version*). Uma versão do projeto é composta de entidades (*Entity*), termo genérico que indica um elemento (*Element*) do modelo (diagrama de classe, caso de uso, atributo, método, código-fonte) ou uma relação (*Relation*) entre dois elementos. Esta estrutura permite um fácil mapeamento para uma estrutura Entidade-Relacional e está aderente aos requisitos de versionamento e *baseline* levantados. Sempre que uma nova versão do sistema for detectada, uma nova instância do objeto *Version* será agregada em *Project*, sendo que *Version* por sua vez irá agregar as

entidades (*Entity*) da nova versão, de maneira similar ao trabalho de Briand, Labiche e O'Sullivan (2003). Entretanto, a composição é elaborada de maneira a trabalhar com quaisquer artefatos de software (desde requisitos textuais até código-fonte).

4.6.2 RASTREABILIDADE

A segunda característica que deverá estar presente no modelo, é a capacidade de representar a rastreabilidade. Neste sentido, faz-se necessário um modelo que permita representar não somente a rastreabilidade vertical (entre elementos de um mesmo modelo), mas também a rastreabilidade horizontal (entre elementos de modelos diferentes). Além disto, conforme visto em vários trabalhos, a configuração da granularidade da rastreabilidade ainda constitui um problema a ser resolvido.

Modelos de rastreabilidade com aspectos similares foram identificados nos trabalhos de Letelier (2002), Briand, Labiche e O'Sullivan (2003) e Kassab, Ormandjieva e Daneva (2008). Letelier (2002) propõe um modelo UML que permite registrar requisitos de software, tanto textuais quanto artefatos de *design* em uma estrutura de agregação recursiva. Entretanto, o modelo permite um conjunto finito e pré-definido de relacionamentos entre as entidades do modelo. Já o trabalho de Briand, Labiche e O'Sullivan (2003) é focado exclusivamente em artefatos UML e não suporta rastreabilidade horizontal, enquanto o de Kassab, Ormandjieva e Daneva (2008) não considera código-fonte como artefato.

Neste sentido, procurou-se desenvolver um modelo que permita o registro de relacionamentos entre requisitos de pouca granularidade (*Coarse-Grained)*, como um documento de especificação textual, um caso de uso, uma classe UML ou um arquivo contendo código-fonte e também relacionamentos entre requisitos de muita granularidade (*Fine-Grained*), como uma palavra do documento de especificação, um ator, um método de uma classe UML ou um método de uma classe no código-fonte. Desta forma, será possível indicar precisamente em qual parte do código-fonte, determinado método é implementado. Também poderá se representar em qual caso de uso, está determinado ator citado nos requisitos textuais.

O modelo proposto, conforme pode ser visualizado na figura a seguir, inicia em uma entidade (*Entity*), que pode ser um elemento (*Element*) ou uma relação entre dois elementos

(*Relation*). Uma relação poderá ter um tipo (*TraceTo*, *Implements*, *Dependency*, dentre outros aqui não representados em função do espaço). Já um elemento poderá ser um artefato (requisito textual, uma classe, um caso de uso, dentre outros) ou um membro de um artefato (propriedade, método, estado, atividade, dentre outros). Um artefato poderá ser composto por diversos membros. A configuração da granularidade se dará no relacionamento entre elementos, uma vez que a superclasse *Element* é uma generalização tanto de *Artifact*, quanto de *Member*. Desta forma, poderá existir um relacionamento entre artefatos, entre membros e entre artefatos e membros.

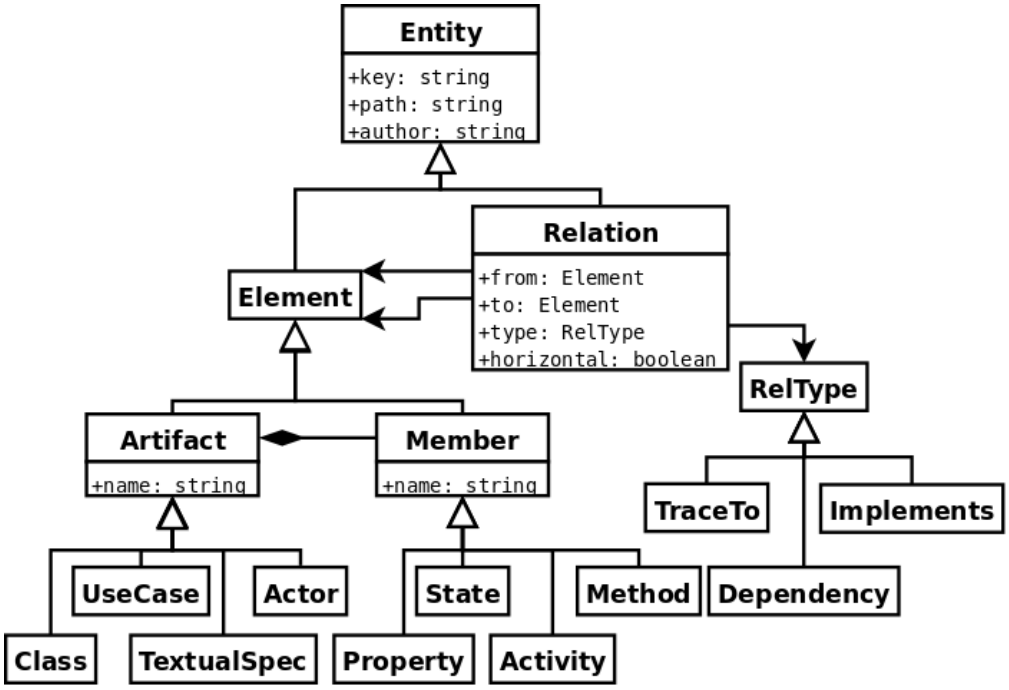


Figura 48. Aspectos de rastreabilidade do modelo.

A figura a seguir procura demonstrar exemplos de elementos considerados artefatos e membros e sua granularidade distinta. A figura contém em sua parte superior exemplos de artefatos e na parte inferior, membros destes artefatos. Os elementos estão divididos na horizontal em relação à fase que pertencem: análise, projeto ou implementação. As setas indicam os relacionamentos entre os artefatos e seus membros correspondentes.

Cabe salientar novamente que o modelo permite criar relacionamento entre artefatos, entre membros e entre membros e artefatos de quaisquer fases, possibilitando a representação de relacionamentos com uma riqueza de detalhes não somente na rastreabilidade vertical, mas também na rastreabilidade horizontal. Neste exemplo, *requirement* representa um requisito textual, sendo que *word* representa um termo relevante dentro deste requisito. Além destes

tem-se *class*, que representa uma classe de *design* (UML), *attribute* e *method*, que representam atributos ou métodos de classes de *design*, *pclass* que representa uma classe de projeto (de implementação/código-fonte), bem como *pattribute* e *pmethod* que representam atributos e métodos desta classe.

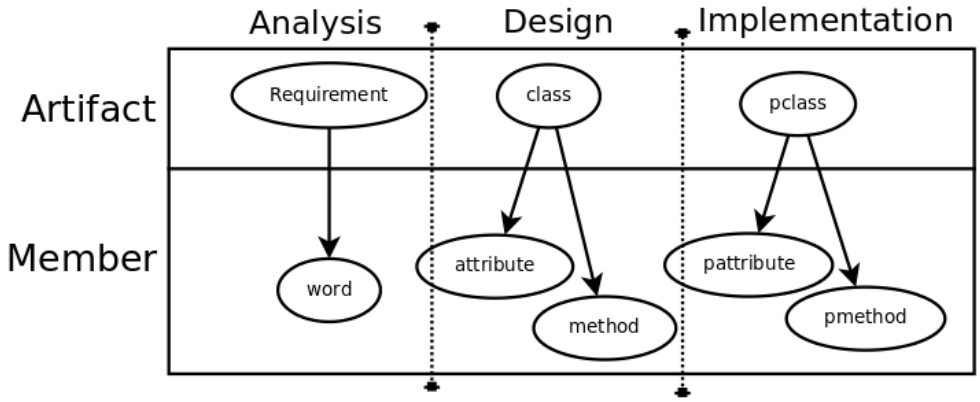


Figura 49. Relacionamentos entre artefatos e membros

4.6.3 GESTÃO DE MUDANÇAS

A terceira característica que deve estar presente no modelo é o registro das mudanças ocorridas, suas origens, seus responsáveis e seus motivos. Modelos similares que apresentaram a mudança como um aspecto relevante, foram encontrados nos trabalhos de Ramesh et al. (1995), Lang e Duggan (2001), Briand, Labiche e O'Sullivan (2003) e De Grande e Martins (2006). Tanto o modelo de Briand, Labiche e O'Sullivan (2003), quanto o modelo de Lang e Duggan (2001) permitem representar somente mudanças ocorridas em elementos de pouca granularidade (*Coarse-Grained*). Já os modelos de Ramesh et al. (1995) e De Grande e Martins (2006) são focados exclusivamente em requisitos textuais. Nenhum dos modelos identifica as origens e razões das mudanças.

Para permitir um controle de mudanças que suporte tanto requisitos de pouca e de muita granularidade, bem como permitir uma classificação das mudanças, é proposto o fragmento de modelo a seguir.

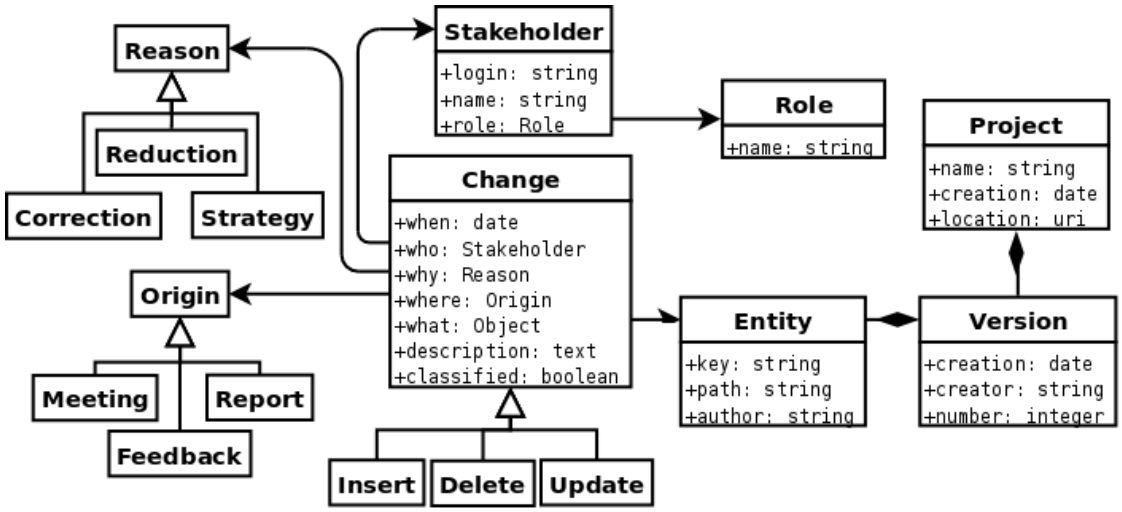


Figura 50. Aspectos de controle da mudança no modelo

Este modelo é centrado no conceito de mudança (*Change*). Uma mudança ocorre em uma entidade (*Entity*). Como visto anteriormente, uma entidade pode ser um elemento do modelo (caso de uso, classe), partes de elementos (membros) ou um relacionamento entre elementos. Desta forma, uma mudança pode ocorrer em um elemento de pouca granularidade como também em um elemento de muita granularidade. Uma mudança é desencadeada por um colaborador (*stakeholder*), que por sua vez possui um papel (*Role*). Cada mudança possui uma razão (*Reason*) e uma origem (*Origin*). Conforme classificação proposta por Nurmuliani, Zowghi e Fowell (2004), são exemplos de razões de mudanças: correção de defeito, requisito faltante, melhoria funcional, estratégia de produto, etc e são exemplos de origens de mudanças: relatórios de erros, observações da gerência do projeto, do grupo de marketing, análise do desenvolvedor, etc. Além disto, cada mudança pode ser do tipo: adição, exclusão ou mudança. No modelo, só foram representados alguns exemplos de razões e origens em função do espaço disponível. Entretanto, as classificações podem ser vistas no capítulo 2.

4.6.4 GESTÃO DE IMPACTOS

O próximo aspecto a ser modelado é a representação de impactos gerados a partir de mudanças. Modelos similares que apresentaram o impacto como aspecto relevante, foram encontrados nos trabalhos de Briand, Labiche e O'Sullivan (2003) e De Grande e Martins (2006).

O modelo de Briand, Labiche e O'Sullivan (2003) permite representar um impacto, relacionar este com uma mudança e indicar para cada impacto, apenas um elemento afetado. Desta forma, percebe-se que o modelo considera cada elemento afetado por uma mudança no sistema como um impacto diferente. O modelo permite ainda representar impactos causados por impactos, como uma recursão da classe *Impact*. O modelo não permite relacionar impactos com *stakeholders* e está focado em artefatos de *design* (UML). Já o trabalho de De Grande e Martins (2006) permite relacionar um impacto à cada versão de requisito encontrada no modelo. Entretanto, o modelo não permite relacionar impactos com mudanças, tampouco com *stakeholders*. Desta forma, é proposto o seguinte fragmento de modelo para representar impactos.

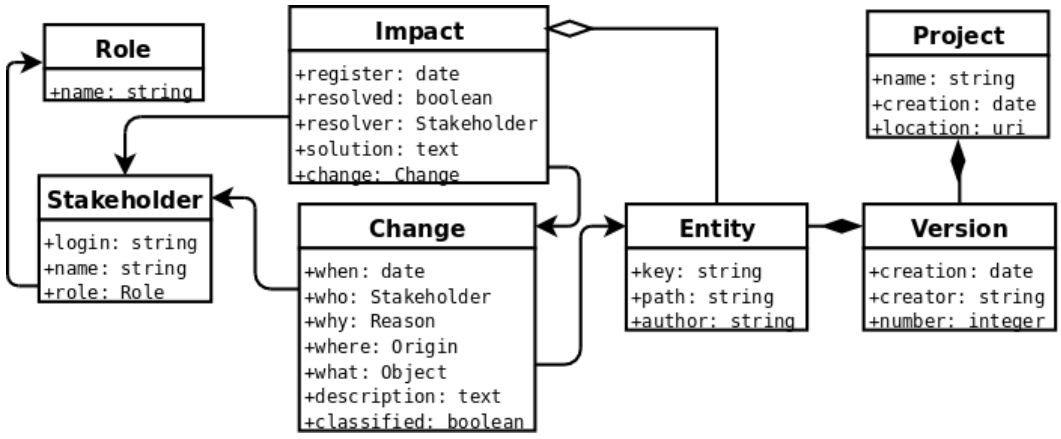


Figura 51. Aspecto de gestão de impactos do modelo

No modelo proposto, um impacto (*Impact*) é uma classe que está relacionada diretamente com uma mudança (*Change*), uma vez que todo impacto será originado a partir de uma mudança. Enquanto a relação da mudança (*Change*) com o *Stakeholder* indica o causador da mudança, a relação de *Impact* com *Stakeholder* indica quem resolveu o impacto. Cada impacto poderá afetar um cojunto de entidades (*Entity*), que poderão ser elementos (artefatos ou membros) ou relacionamentos, representadas por uma estrutura de agregação.

4.6.5 MODELO DE CLASSES CONSOLIDADO

Na figura a seguir, é apresentado o modelo consolidado a partir dos aspectos modelados até o presente momento: versionamento, rastreabilidade, gestão de mudanças e

gestão de impactos. Cada aspecto é representado por uma região de cor diferente no modelo consolidado. A rastreabilidade é representada pela região em amarelo, a gestão da mudança pela região em verde, a gestão de impactos pela região laranja e o versionamento pela região azul. A classe Entidade (*Entity*) que representa qualquer artefato de software, desde um requisito textual, até um diagrama de classes, um método ou um relacionamento entre dois artefatos, é central no modelo pois é compartilhada por todos aspectos.

Acredita-se que este modelo seja completo o suficiente para atender todos os requisitos levantados no presente trabalho. Para verificar esta proposição, na seção 6 o modelo será comparado com os demais modelos estudados com base em certos critérios.

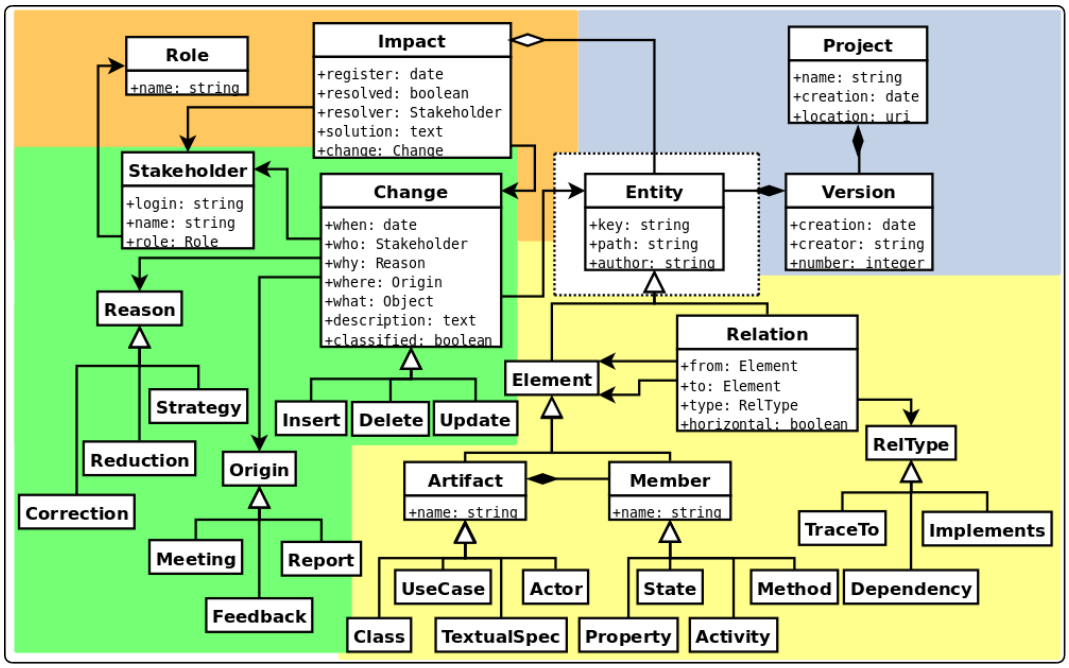


Figura 52. Modelo proposto consolidado

Como visto, o modelo proposto especifica claramente quais os tipos de artefatos e relacionamentos que se fazem necessários para prover a rastreabilidade, ao mesmo tempo em que permite extensibilidade e adaptabilidade por meio da especialização e criação de novas subclasses daquelas já existentes.

4.6.6 AQUISIÇÃO DE INFORMAÇÕES PARA O MODELO

Para Queille et al. (1994), além da definição de um modelo, é necessário indicar como os dados necessários para popular este modelo serão coletados. O modelo proposto será

alimentado pelo agente *RequirementEngineer*, como já explicado anteriormente. Este agente irá realizar a leitura e interpretação de diferentes tipos de artefatos, como estes exibidos na figura a seguir. Nesta figura podemos ver artefatos de três fases distintas do ciclo de desenvolvimento de um software. Em primeiro lugar tem-se um requisito textual com algumas palavras em destaque (aluno e matrícula). Em segundo lugar temos uma classe de *design* chamada *Aluno*, contendo dois atributos e um método chamado *matricular* e por último uma classe de implementação também chamada *Aluno* e com um método *matricular*.

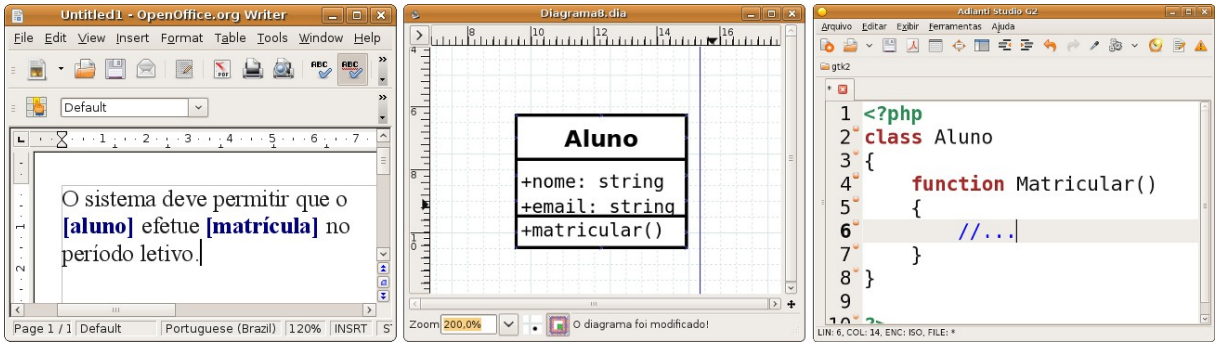


Figura 53. Artefatos de fases distintas do ciclo de desenvolvimento

Quando o agente *RequirementEngineer* realiza a leitura destes artefatos, instancia em memória objetos do modelo para representá-los. A figura a seguir consiste em um diagrama de objetos contendo instâncias para representar partes estes três artefatos exemplificados na figura anterior. Para tal, consideramos que todos os artefatos façam parte da mesma versão (objeto *Version*). A figura é dividida em três partes similares, cada uma mapeando um artefato correspondente. A primeira parte do diagrama possui duas entidades (*Entity*) do tipo elemento (*Element*). O primeiro elemento é o próprio requisito textual *matricular*, considerado um artefato (*Artifact*), que é composto pelo segundo elemento, que é um membro (*Member*) deste artefato, neste caso, a palavra em destaque *aluno*.

Já no segundo grupo de classes, temos também duas entidades, cada uma sendo um elemento (*Element*) distinto, sendo que o primeiro elemento é o artefato (*Artifact*) classe de design *aluno* que é composto de um membro que é o método matricular. É importante salientar que, tanto o artefato (*Artifact*), quanto o membro (*Member*), apesar de serem da mesma hierarquia de classes, constituem objetos diferentes desde suas classes concretas até sua super-classe. Por último, no terceiro grupo de classes temos a classe de programa *aluno*, bem como seu método composto *matricular*. Para simplificar este exemplo, nem todos

atributos foram representados pela imagem.

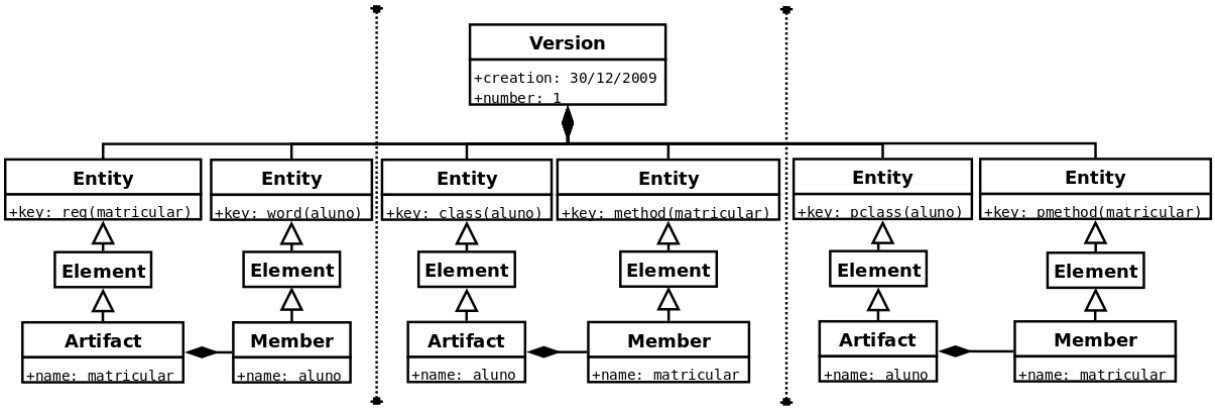


Figura 54. Diagrama de objetos para mapeamento de artefatos e membros

Além de interpretar os artefatos e seus relacionamentos todo/parte como no exemplo anterior no qual métodos foram mapeados como membros (*Member*) de um artefato (*Artifact*), o agente *RequirementEngineer* também interpreta todos os tipos de relações que podem existir entre as classes. Para tal, o agente vasculha artefatos como diagramas de classes buscando por relações como heranças, agregações, composições, associações, dentre outros e também vasculha o código-fonte da aplicação procurando por estes relacionamentos. O método que exemplifica a operação dos agentes será detalhado mais adiante.

A figura a seguir procura demonstrar relações de rastreabilidade vertical dentro dos modelos. Na primeira parte da figura, tem-se um diagrama de classes com um relacionamento de composição entre as classes de *design Aluno* e *Contrato*. Já na segunda parte, tem-se um relacionamento de herança entre as clasess de implementação *Aluno* e *Pessoa*.

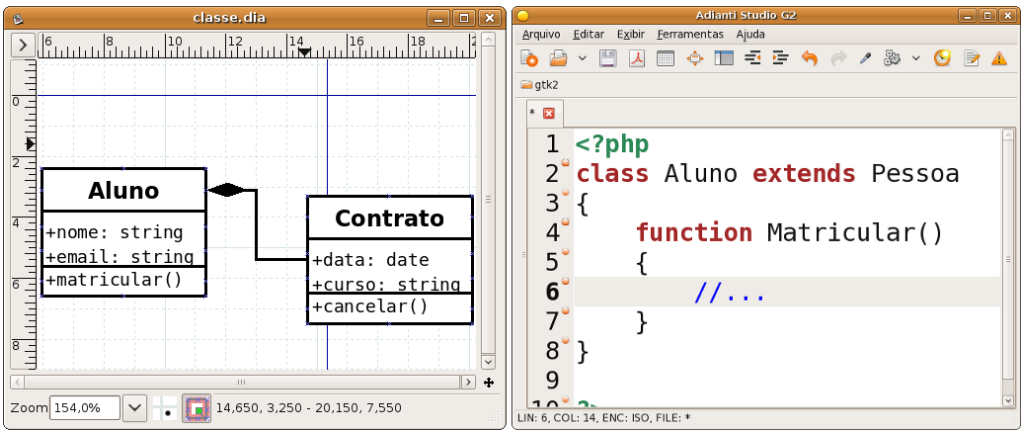


Figura 55. Relações de rastreabilidade vertical

A figura a seguir contém o diagrama de objetos que procura representar como a rastreabilidade vertical destes artefatos é mapeada para o modelo. Neste diagrama, supõe-se

novamente que os artefatos sejam da mesma versão, o que é representada pela relação de composição superior. O modelo é composto de seis entidades: as classes de *design Aluno* e *Contrato*, as classes de implementação *Aluno* e *Pessoa*, bem como o relacionamento entre as classes, representados pela classe *Relation*, destacada por uma cor diferenciada. É importante notar que o relacionamento (*Relation*) também é considerado uma entidade (*Entity*), assim como os demais artefatos que compõem o sistema.

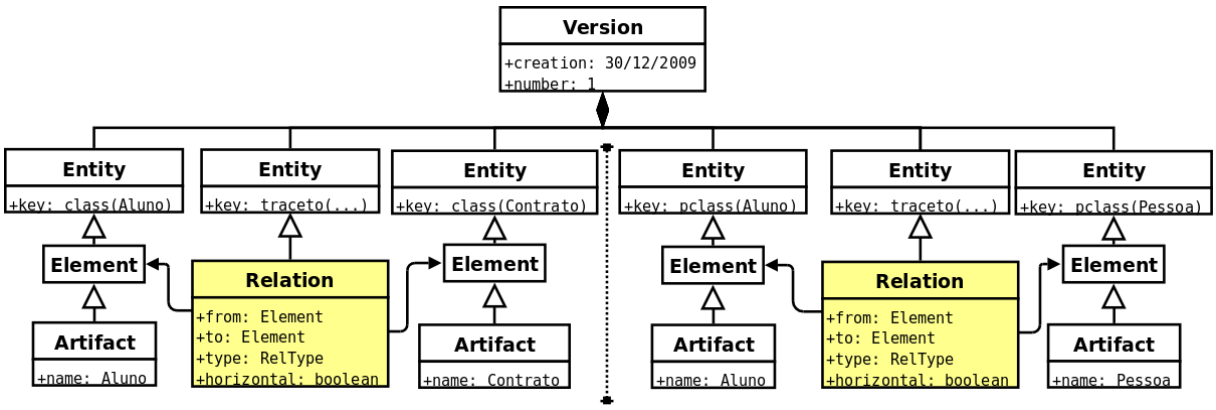


Figura 56. Diagrama de objetos para mapeamento da rastreabilidade vertical

A figura a seguir procura demonstrar uma ligação que representa um tipo de rastreabilidade horizontal. Neste exemplo, a palavra *matrícula* contida pelo requisito textual *matricular* está ligada ao método *matricular* da classe *Aluno*. As relações de rastreabilidade horizontal serão detectadas por um mecanismo de similaridade entre termos e também poderá ser registrada manualmente pelo analista. Em seguida será demonstrado como este relacionamento será representado por objetos em memória.

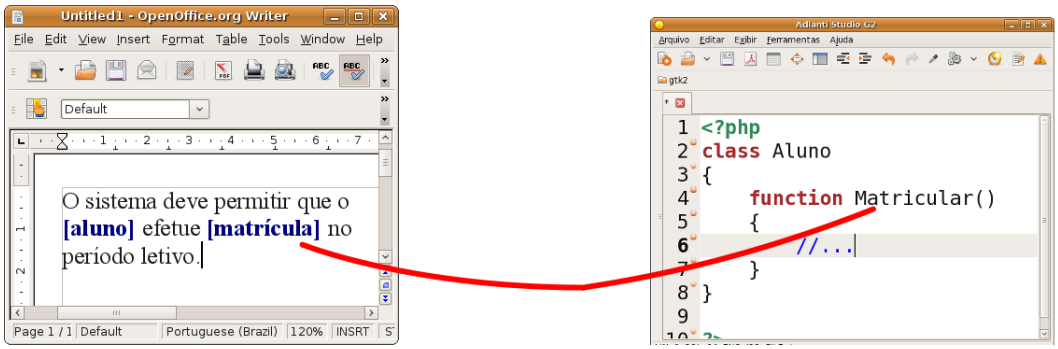


Figura 57. Relação de rastreabilidade horizontal entre um requisito textual e o código-fonte

O diagrama a seguir procura representar as instâncias em memória que identificam os artefatos relacionados do exemplo anterior. Neste caso temos novamente três grupos de objetos. No primeiro grupo, tem-se duas entidades: o requisito textual em si, que é um artefato (*Artifact*) e a palavra *matrícula*, que é um membro (*Member*) deste artefato. No terceiro grupo

de objetos, temos a classe *aluno* que é um artefato e o método *matricular* que é um membro deste artefato. Já no grupo do meio, tem-se uma entidade de relacionamento (*Relation*), que representa o relacionamento entre dois elementos: a palavra *matrícula* (*Member*) do requisito textual e o método *matricular* (*Member*) da classe aluno.

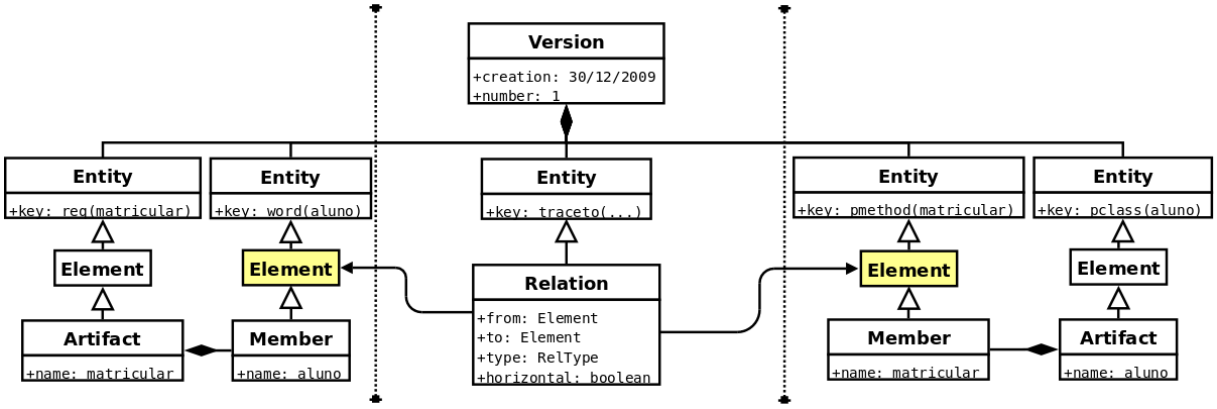


Figura 58. Diagrama de objetos para mapeamento da rastreabilidade horizontal

Sempre que uma entidade (artefato, um membro de artefato ou um relacionamento) foram excluídos entre uma versão e outra do sistema, aquela entidade excluída será mantida na próxima versão para fins de registro, análise de impacto e rastreabilidade. Entretanto, aquele objeto estará vinculado à uma instância *Change*, que representa uma mudança ocorrida. A instância de *Change* poderá ser de três tipos: adição, alteração ou exclusão. Neste caso será do tipo exclusão. A figura a seguir procura demonstrar que a palavra *aluno* contida dentro do requisito textual *matricular* foi apagada. Neste cso é, instanciado um objeto do tipo *Change* que fica associado àquela entidade. Ao objeto do tipo *Change*, é associado o *Stakeholder* causador daquela mudança.

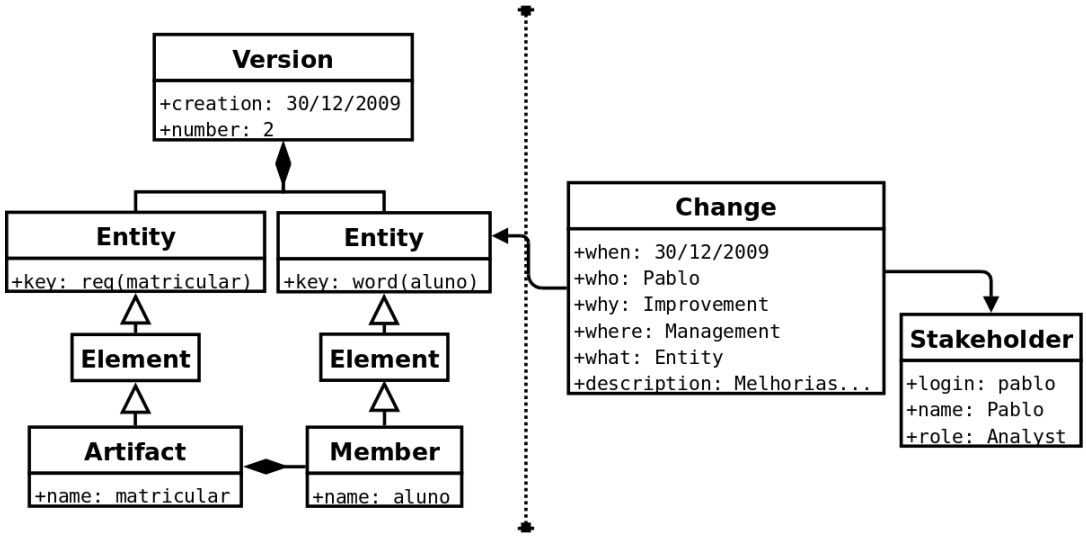


Figura 59. Representação das instâncias perante uma mudança

Após o registro de uma mudança, o sistema dá início à análise de impacto e procura armazenar as informações sobre as entidades afetadas. A figura a seguir procura demonstrar que a partir de uma mudança, representeada pela classe *Change*, o sistema pode ter associado um impacto (classe *Impact*) e este impacto pode consequentemente agregar uma série de entidades, para representar as entidades afetadas pela mudança. Neste caso específico, a exclusão da palavra *aluno* no requisito textual *matricular* provocou impactos em duas entidades de código-fonte: a classe *Aluno* e o método *matricular*.

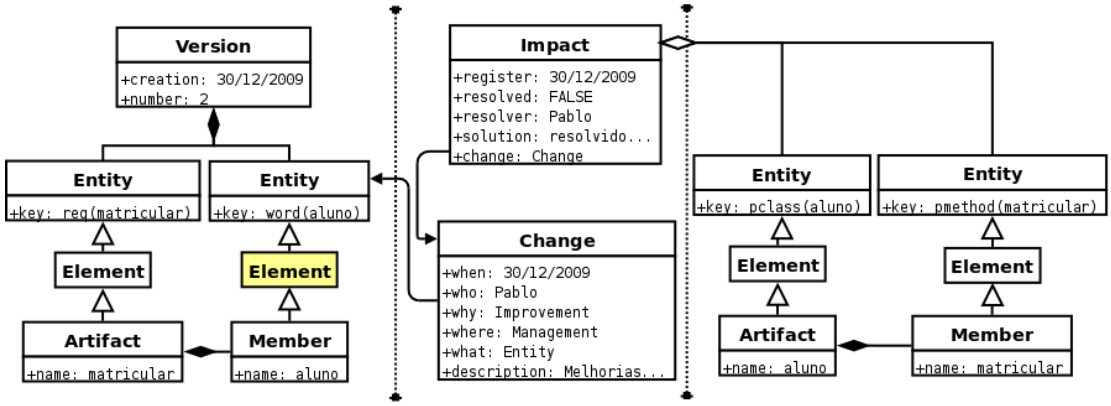


Figura 60. Diagrama de objetos das instâncias perante um impacto

4.6.7 MODELO ENTIDADE-RELACIONAMENTO

A partir do modelo consolidado em UML proposto no item anterior, pode-se realizar seu mapeamento para uma estrutura relacional por meio da aplicação de certos padrões ORM (*Object-Relational Mapping*). Os padrões utilizados neste trabalho para o mapeamento objeto-relacional foram retirados do catálogo de *Design Patters* para uso em aplicações corporativas de Fowler (2002).

Dentre os padrões utilizados neste trabalho para a criação do modelo relacional, pode-se citar o padrão *Class Table Inheritance*, utilizado para mapear os relacionamentos de herança, como os encontrados entre as classes *Entity*, *Element* e *Artifact*, o padrão *Foreign Key Mapping*, para mapear os relacionamentos de associação, como os encontrados entre as classes *Change* e *Stakeholder* e os relacionamentos de composição, como os encontrados entre as classes *Version* e *Entity*. Já para mapear relacionamentos de agregação, como o encontrado entre as classes *Impact* e *Entity*, foi utilizado o padrão *Association Table Mapping*.

Alguns relacionamentos de herança, como os encontrados a partir das classes *Artifact*

e *Member* puderam ser mapeados por meio do padrão *Single Table Inheritance*, tendo uma tabela de apoio para representar o tipo de objeto armazenado na estrutura (*artifact_class* para *artifact* e *member_class* para *member*).

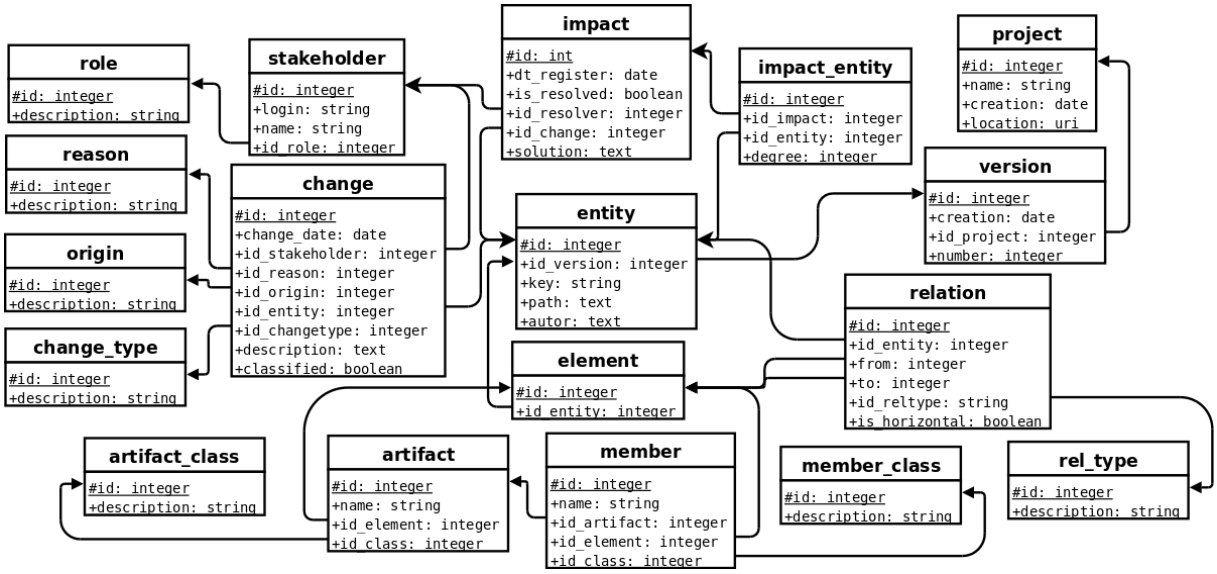


Figura 61. Modelo entidade-relacionamento

4.6.8 EXTRAÇÃO DE INFORMAÇÕES DO MODELO

Para exemplificar a eficiência da versão entidade-relacionamento do modelo proposto, algumas consultas em linguagem SQL serão realizadas. Na figura a seguir, é apresentada uma listagem dos artefatos do sistema, que são os elementos de pequena granularidade (*Coarse-Grained*) do projeto 1, cuja versão também é 1. São listados requisitos, atores e casos de uso. Para obter as informações, é necessário utilizar as tabelas *artifact*, *element*, *entity* que representam a hierarquia de artefatos e *version* que contém as informações de versão e do projeto ao qual os artefatos estão vinculados.

```
SELECT artifact.id,
       artifact_class.description,
       artifact.name
FROM   artifact, element, entity, version,
       artifact_class
WHERE  artifact.id_element = element.id AND
       artifact.id_class   = artifact_class.id AND
       element.id_entity   = entity.id AND
       entity.id_version   = version.id AND
       version.id_project = 1 AND
       version.number = 1
ORDER BY 1 LIMIT 4;

 id | description |    name
----+-------------+------------
  1 | requirement | matricular
  2 | actor       | aluno
  3 | usecase     | matricular
  4 | usecase     | cadastrar
```

Figura 62. Consulta SQL para listar os artefatos

Já na consulta da figura a seguir, é exibida uma listagem dos elementos do projeto de código 1 e versão também 1 de muita granularidade (*Fine-Grained*), que são os membros (*Members*) de artefatos (Artifact). São listados nesta consulta o método *get_nome_curso* da classe *Contrato*, o método *addTurma* da classe *Contrato*, o método *store* da classe *Matricula* e o método *get_nome_disciplina* da classe *Matricula*. Para obter esta informação, é necessário utilizar as tabelas *member*, *element*, *entity,* que fazem parte da hierarquia da classe *Member,* e *artifact,* que contém a informação do artefato ao qual o membro está vinculado. Além destas, *version* possui as informações de versão e do projeto vinculado, enquanto que *member_class* informa qual é o tipo da classe do membro (*method*, *property*, etc...).

```
SELECT member.id as id,
       artifact.name as art_name,
       member_class.description ,
       member.name as member_name
FROM   member, element, entity, artifact,
       version, member_class
WHERE  member.id_element=element.id AND
       member.id_class=member_class.id AND
       member.id_artifact=artifact.id AND
       element.id_entity=entity.id AND
       entity.id_version=version.id AND
       version.id_project = 1 AND
       version.number = 1
ORDER BY 3 LIMIT 4;

 id | art_name  | description | member_name
----+-----------+-------------+--------------
 18 | Contrato  | method      | get_nome_curso
 19 | Contrato  | method      | addTurma
 20 | Matricula | method      | store
 21 | Matricula | method      | get_nome_disciplina
```

Figura 63. Consulta SQL para listar os membros

Na figura a seguir, são exibidos algumas relações entre entidades do sistema. Neste caso, é demonstrado que a classe *Contrato* possui uma associação com a classe *Curso* e a classe *Pessoa* possui uma associação com a classe *Cidade*. Devido ao limitado espaço, relacionamentos *Fine-Grained* foram suprimidos. Para obter a informação, é necessário utilizar algumas tabelas como *entity* até três vezes na consulta, sendo uma vez para representar a entidade origem do relacionamento, uma vez para representar a entidade destino e uma vez para representar o próprio relacionamento. A tabela *rel_type* possui informações sobre o tipo de relacionamento (associação, herança, etc.).

```
SELECT t1.key as from_key,
       rt.description as relation,
       t2.key as to_key
FROM   relation r, element e1, element e2,
       entity t1, entity t2, entity t3, rel_type rt, version
WHERE r.from_element=e1.id and
      r.to_element=e2.id and
      r.id_entity=t3.id and
      t3.id_version=version.id and
      t1.id_version=t2.id_version and
      t2.id_version=t3.id_version and
      e1.id_entity=t1.id and e2.id_entity=t2.id and
      r.id_reltype=rt.id and version.id_project=1 and
      version.number = 1
LIMIT 4;

     from_key      |  relation   |       to_key
-------------------+-------------+-----------------
class('Contrato')  | association |class('Curso')
class('Pessoa')    | association |class('Cidade')
class('Turma')     | association |class('Disciplina')
class('Matricula') | association |class('Turma')
```

Figura 64. Consulta SQL para listar os relacionamentos

A próxima figura procura demonstrar uma heurística para recuperação da rastreabilidade horizontal. Estas heurísticas procuram identificar relacionamentos entre entidades de diferentes modelos. Para isto, a ferramenta utiliza-se da similaridade entre os termos de artefatos e membros de forma *case-insensitive*. Como esta consulta é mais complexa que as demais, ela irá apoiar-se sobre uma *view*. A *view* irá relacionar todas as tabelas da hierarquia de *artifact*, provendo as informações necessárias para a realização da consulta a seguir.

```
CREATE VIEW view_artifact AS
SELECT artifact.name,
       element.id as id_element,
       version.id_project,
       version.number as version_number,
       artifact_class.id as id_class,
       artifact_class.description as class_description
FROM   artifact, element, entity, version, artifact_class
WHERE  artifact.id_element=element.id AND
       artifact.id_class=artifact_class.id AND
       entity.id_version=version.id AND
       element.id_entity=entity.id;
```

Figura 65. Criação de uma view contendo os artefatos do sistema

Após a criação da *view*, pode-se proceder com a consulta inicial, que objetiva identificar a similaridade entre classes de *design* UML e classes de implementação (código-fonte) do sistema. Para tal, a consulta procura relacionar a *view* de artefatos duas vezes, realizando a comparação entre os termos de forma *case-insensitive*. Neste exemplo, a consulta está atuando sobre informações do projeto de número 1, versão 3. A consulta realiza um *join* entre artefatos de classe 3 (classes de *design*) e artefatos de classe 5 (classes de implementação). A recuperação da rastreabilidade horizontal atua sempre sobre a última versão de cada projeto.

Neste exemplo, foram detectadas classes com nomenclatura similar, como é o caso das classes *Pessoa*, *Cidade*, *Aluno*, *Matricula* e *Turma*. Esta é apenas uma das heurísticas suportadas pelo modelo, a mais simples. Ainda são realizadas consultas para descobrir similaridade entre membros de artefatos de diferentes modelos, como entre métodos de classes de implementação e métodos de classe de *design*.

```
select  A.id_element as element,
        A.name,
        A.class_description as class,
        B.id_element as element,
        B.name,
        B.class_description as class
FROM    view_artifact A, view_artifact B
WHERE
    A.id_project=1 AND A.version_number=3 AND
    B.id_project=1 AND B.version_number=3 AND
    A.id_class=5   AND B.id_class=3 AND
    A.name ilike B.name;

 element |   name    | class  | element |   name    | class
---------+-----------+--------+---------+-----------+-------
     204 | Pessoa    | pclass |     187 | Pessoa    | class
     202 | Cidade    | pclass |     188 | Cidade    | class
     203 | Aluno     | pclass |     189 | Aluno     | class
     198 | Matricula | pclass |     191 | Matricula | class
     199 | Turma     | pclass |     195 | Turma     | class
```

Figura 66. Heurística para recuperação da rastreabilidade horizontal

Na próxima figura, é realizada uma consulta para exibir as mudanças ocorridas em membros de artefatos do projeto. Para tal, é utilizada a tabela *changes*, que contém o registro das mudanças e a tabela *entity*, que é vinculada à mudança. Para vincular somente mudanças ocorridas em membros, são relacionadas as tabelas *element* e *member,* para exibir o tipo do membro, é relacionada a tabela *member_class* e para exibir o tipo da mudança, é relacionada a tabela *change_type*.

```
SELECT  change.id as id,
        change_type.description as change_type,
        change.change_date as change_date,
        member_class.description as class,
        member.name as member
FROM    change, entity, element, member,
        member_class, change_type, version
WHERE   change.id_entity=entity.id AND
        entity.id_version=version.id AND
        element.id_entity=entity.id AND
        member.id_element=element.id AND
        member.id_class=member_class.id AND
        change.id_changetype=change_type.id
ORDER BY 1;

 id | change_type  | change_date |  class   |       member
----+--------------+-------------+----------+--------------------
  2 | Exclusion    | 2009-10-28  | method   | store
  3 | Exclusion    | 2009-10-28  | property | carga_horaria
  4 | Exclusion    | 2009-10-28  | property | nome
 10 | Addition     | 2009-10-28  | property | carga_horaria
 12 | Modification | 2009-11-20  | pmethod  | get_nome_professor
 13 | Modification | 2009-11-21  | pmethod  | get_nome_cidade
```

Figura 67. Consulta SQL para listar as mudanças

A consulta a seguir procura demonstrar como é possível descobrir relacionamentos inconsistentes por meio do modelo. Para tal, é realizada um aconsulta sobre a tabela de relacionamentos (*relation*). Todo relacionamento é um elo de ligação entre dois elementos. Desta forma, para descobrir relacionamentos inconsistentes, basta verificar todos os relacionamentos entre elementos em que o elemento de origem ou o elemento de destino também encontram-se presentes em uma mudança (*change*) do tipo exclusão. Assim, considera-se que um elemento excluído, seja de origem ou destino, torna o relacionamento ao qual ele faz parte inconsistente.

```
SELECT relation.id, entity.key, version.number
FROM   relation, entity, version
WHERE  relation.id_entity=entity.id and
       entity.id_version=version.id and
       (  relation.from_element in
              (select E.id from element E, entity T, change C
               where E.id_entity=T.id and T.id=C.id_entity and C.id_changetype=2)
          or
          relation.to_element in
              (select E.id from element E, entity T, change C
               where E.id_entity=T.id and T.id=C.id_entity and C.id_changetype=2)
       );

 id | key                                                                | number
----+--------------------------------------------------------------------+--------
 66 | traceto(class('Turma'),  'association', class('Disciplina'))  |      2
 90 | traceto(pclass('Turma'), 'depends', pclass('Disciplina'))     |      3
 91 | traceto(pclass('Turma'), 'depends', pclass('Disciplina'))     |      3
```

Figura 68. Descoberta de relacionamentos inconsistentes

Nesta próxima consulta, são exibidas estatísticas de mudanças do projeto 1 conforme sua origem. Como exemplo, ocorreram 2 mudanças devido à relatórios de erros e 3 mudanças devido à chamados da engenharia. Para exibir esta estatística, foi necessário vincular a tabelas *change* a fim de contabilizar a quantidade de mudanças e a tabela *origin*, que contém a descrição da origem da mudança.

```
SELECT change.id_origin,
       origin.description,
       count(*)
FROM change, entity, version, origin
WHERE change.id_origin=origin.id and
      change.id_entity=entity.id and
      entity.id_version=version.id and
      change.classified='1' and
      version.id_project=1
GROUP BY 1,2

 id_origin |       description        | count
-----------+--------------------------+-------
         1 | Relatorios de erros      |     2
         2 | Chamado da Engenharia    |     3
         3 | Gerenciamendo do Projeto |     5
         5 | Análise do Desenvolvedor |     1
```

Figura 69. Consulta SQL para listar estatísticas de mudanças

Por último, será demonstrada uma consulta SQL que exibe a quantidade de impactos gerados no projeto conforme a origem da mudança. Para realizar esta contagem, são associadas as tabelas *change* e *impact*, além da tabela origin, que contém a descrição da origem da mudança. Convém ressaltar que nem todas mudanças no sistema geram impactos, por isso a quantidade de impactos provavelmente será menor que a de mudanças.

```
SELECT change.id_origin,
       origin.label,
       count(*)
FROM change, entity, version, origin, impact
WHERE change.id_origin=origin.id and
      impact.id_change=change.id and
      change.id_entity=entity.id and
      entity.id_version=version.id and
      version.id_project=1
GROUP BY 1,2;

 id_origin |    label                  | count
-----------+---------------------------+-------
         1 | Relatório de erros        |     2
         3 | Gerenciamento do Projeto  |     1
```

Figura 70. Consulta SQL para listar estatísticas de impactos

4.7 MÉTODO

Após demonstração do modelo utilizado pela aplicação, será demonstrado o funcionamento do método de trabalho, que trata de como os agentes irão interagir uns com os outros e com os *Stakeholders* durante suas atividades. Para De Loach (2000), estas interações representam as conversações entre os agentes e são descritas na forma de narrativas de uma determinada sequência de eventos construídos com base nos requisitos iniciais.

Para ilustrar o método proposto, foi elaborado um diagrama contendo o fluxo do processo principal da aplicação. Este diagrama apresenta o cenário de funcionamento do método e inicialmente será explicado de forma superficial ser detalhado posteriormente, onde cada uma de suas etapas serão apresentadas individualmente.

Em primeiro lugar, é representada a figura do analista, que inclui, altera ou exclui um artefato de software. Quando este fato acontecer, será percebido pelo agente *RequirementEngineer*, que irá registrar uma nova versão do modelo de artefatos e detectar as mudanças ocorridas. Para cada mudança, ele irá acionar o agente *ProcessManager*, que irá notificar os *stakeholders* envolvidos na modificação e irá solicitar que os mesmos classifiquem as mudanças ocorridas. Após os *stakeholders* classificarem as mudanças, o agente *ImpactAnalyser* poderá registrar estas mudanças na base de dados, bem como suas origens e suas razões.

A partir do momento em que se tem o modelo, bem como as mudanças registradas, o agente *TraceabilityManager* poderá vasculhar o modelo buscando por inconsistências na rastreabilidade. Caso o agente *TraceabilityManager* encontre alguma inconsistência, irá

acionar o agente *ProcessManager*, que irá notificar as inconsistências aos *stakeholders* relacionados. Após esta etapa de verificação da consistência, o agente *ImpactAnalyser* irá identificar os requisitos afetados pelas mudanças de forma recursiva, observando a rastreabilidade. A partir dos requisitos identificados, o agente *ImpactAnalyser* irá notificar os *stakeholders* afetados, emitindo o relatório de impacto, onde irá constar os requisitos afetados. Em seguida, cada uma das etapas será detalhada individualmente.

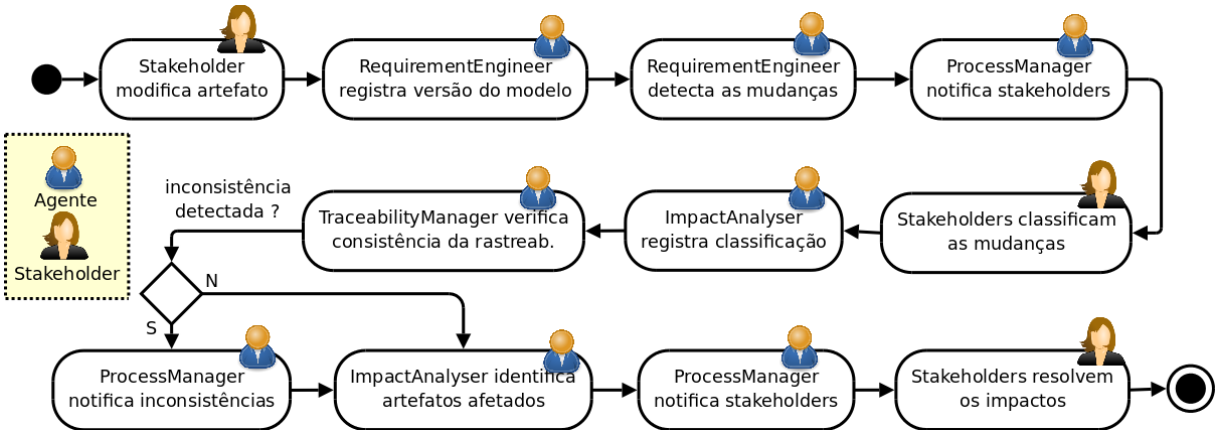


Figura 71. Fluxo do processo principal

4.7.1 AQUISIÇÃO DO MODELO

O funcionamento do sistema tem início quando algum *stakeholder* realiza alguma alteração em algum dos artefatos de software do sistema. Quando isto acontecer, a mudança, bem como os impactos devem ser gerenciados. Para que se possa avaliar quais as consequências que uma modificação teve no sistema, é necessário interpretar os artefatos, bem como seus relacionamentos para que estes possam ser analisados.

O agente *RequirementEngineer* é o responsável pela interpretação dos artefatos de um projeto, desde requisitos textuais, modelos de *design* e arquivos de código-fonte. Para realizar a interpretação dos artefatos, o agente *RequirementEngineer* apoia-se em três classes: *TextualParser*, *ModelParser* e *SourceParser*, que possuem o conhecimento necessário sobre a estrutura dos artefatos que serão lidos. A classe *TextualParser* é especializada na interpretação de requisitos textuais expressos por documentos no padrão ODF. Já a classe *ModelParser* é especializada na interpretação de modelos de *design* expressos por documentos XMI e a classe *SourceParser* é especializada em arquivos contendo código-fonte PHP ou Java. Cada uma destas três classes possui a capacidade de interpretar as estruturas internas dos artefatos e

retornar esta estrutura para a aplicação em um formato orientado a objetos.

Na medida em que as classes *TextualParser*, *ModelParser* e *SourceParser* realizam a interpretação dos artefatos de software, estas criam e mantém em memória um objeto do tipo *ProjectStructure,* que contém toda a estrutura do projeto de forma que ele disponibilize via sua interface, a informação lida. Este objeto é mapeado para o banco de dados da aplicação por meio da classe *ProjectStructureMapper*, que é responsável por registrar uma versão do projeto na base de dados a partir de um *ProjectStructure*. A partir de então as informações ficam disponíveis na aplicação para os outros agentes.

A figura a seguir procura demonstrar a interação que o agente *RequirementEngineer* possui com as demais classes que apoiam seu trabalho. Ele tem contato direto com as classes *SourceParser*, *ModelParser* e *SourceParser*, para as quais ele solicita a leitura dos artefatos. Após estas classes realizarem a interpretação dos artefatos, devolvem para o agente *RequirementEngineer* um objeto *ProjectStructure*, que é então persistido na base de dados por meio da classe *ProjectStructureMapper*, que possui as regras de mapeamento de objetos para bases de dados relacionais.

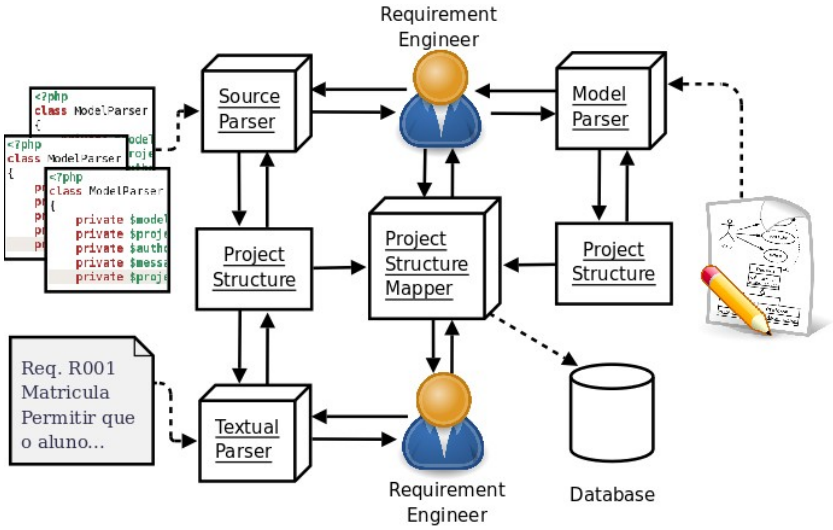


Figura 72. Sistemática da aquisição de informações

O diagrama a seguir procura demonstrar a sequência em que são executadas as atividades do agente *RequirementEnginner* quando este realizar a interpretação dos artefatos de software. O agente *RequirementEngineer* é acionado por um processo de software chamado *TimeManager*, que avalia frequentemente quais são os projetos gerenciados pela aplicação e qual foi a frequência configurada para que o agente *RequirementEngineer* vasculhe o repositório do projeto. A frequência de monitoramento (hora, dia, semana, mês) é

configurável pela aplicação.

Assim que o agente *RequirementEngineer* é acionado, recebe como padrâmetro o projeto que deve ter seus artefatos lidos e atualizados na base de dados. Então é acionado o agente *ProcessManager*, que possui as informações acerca de cada projeto, para que este retorne dados como a localização do repositório de artefatos para *RequirementEngineer.*

Com base nas informações de projeto, o agente *RequirementEngineer* aciona então a classe *TextualParser*, que efetua a leitura de um diretório do repositório contendo os arquivos com requisitos textuais representados por arquivos no padrão ODF, por meio do método *parseDirectory*. Este método retorna ao agente a estrutura lida dos artefatos, representada por um objeto *$ps*, instância de *ProjectStructure*.

A partir da leitura dos requisitos textuais, o agente *RequirementEngineer* aciona a classe *ModelParser* para efetuar a leitura do modelo de *design* da aplicação. Primeiramente, o agente passa para *ModelParser* a estrutura já lida do projeto por meio do método *setProjectStructure*. Então a classe ModelParser efetua a leitura das estruturas internas do modelo de *design* UML representado pelo padrão XMI, contendo atores, casos de uso, diagramas de classe, etc.

Após a interpretação do modelo de *design*, o agente *RequirementEngineer* aciona a classe *SourceParser* para realizar a leitura do código-fonte da aplicação, composto de arquivos PHP ou Java, contendo classes, atributos, métodos e relacionamento entre eles.

A partir da interpretação dos artefatos, o agente *RequirementEngineer* aciona a classe *ProjectStructureMapper*, que recebe a estrutura lida dos artefatos, representada por um objeto *$ps*, do tipo *ProjectStructure* e persiste estes artefatos e relacionamentos na base de dados.

Por fim, para cada mudança detectada no modelo, a partir da comparação com a versão anterior, o agente *RequirementEngineer* solicita ao agente *ProcessManager* a comunicação aos *stakeholders* que causaram estas mudanças no sistema, para que elas possam ser então classificadas. Maiores detalhes sobre esta etapa serão dados em seguida.

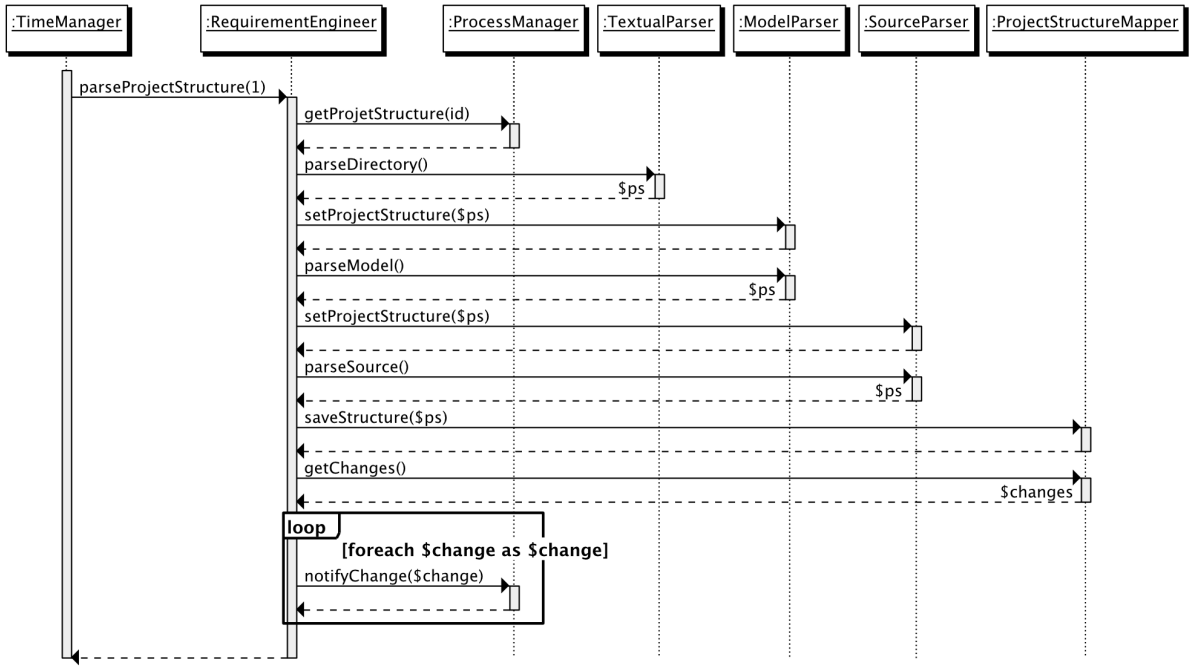


Figura 73. Sequência da atividade de leitura dos artefatos

Como visto, a aplicação proposta suporta documentos contendo requisitos textuais, modelos de *design* UML e código-fonte. A imagem a seguir demonstra um exemplo de arquivo contendo requisitos armazenado no formato ODF (*Open Document Format*), bem como sua estrutura interna. Cabe ressaltar que o formato ODF é o formato adotado pelo presente trabalho para armazenar requisitos textuais. O formato ODF é suportado principalmente pelo aplicativo *OpenOffice* e é padrão internacional ISO.

Basicamente arquivos ODF são arquivos XML compactados no padrão ZIP, que possuem uma estrutura hierárquica pré-determinada. A classe *TextualParser* basicamente descompacta os arquivos ODF e vasculha estes procurando por determinadas *tags* que representam os delimitadores dos campos de formulário que serão preenchidos pelo usuário. Na figura a seguir, pode-se notar que arquivo é composto de um formulário com campos delimitados para o preenchimento do título do requisito, do autor, de sua descrição, dentre outros. Desta forma, a classe *TextualParser* pode efetuar uma leitura precisa. Palavras destacadas entre colchetes são tidas como termos relevantes e armazenadas como membros (*Members*) do artefato principal, que é o próprio requisito. Desta forma, pode-se definir relacionamentos que envolvam não somente o requisito textual em sim, mas termos encontrados nos mesmos.

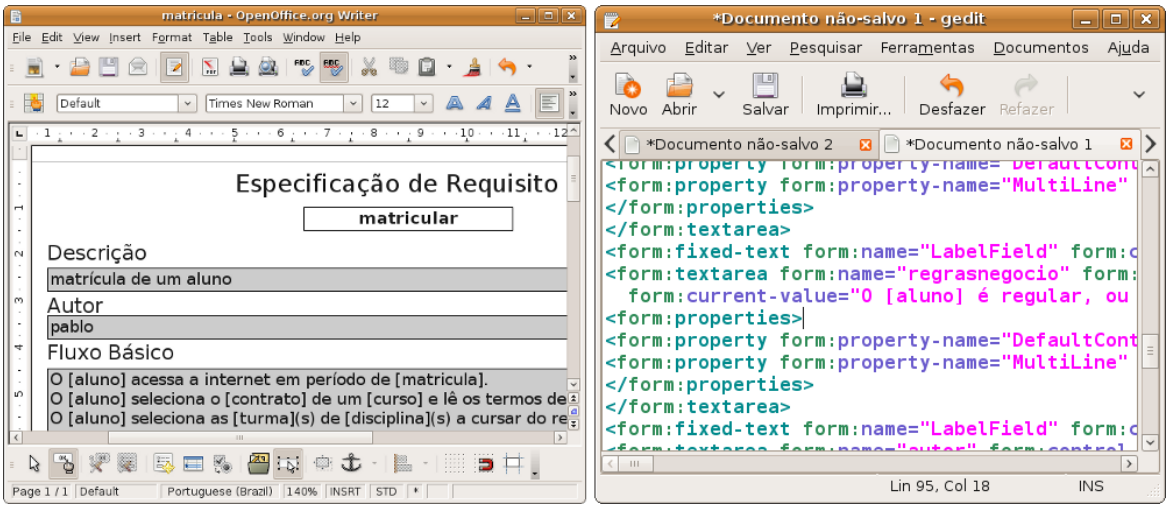


Figura 74. Arquivo ODF contendo requisitos e sua estrutura interna em XML

A figura a seguir procura demonstra um documento contendo o modelo de *design* UML da aplicação, bem como sua estrutura interna. O documento de *design* também é representado no formato XML, mais especificamente, no formato XMI. O formato XMI é um padrão ISO especializado na representação de artefatos de design, contendo atores, casos de uso, atividades, estados, diagrama de classes, dentre outros. Além disto, o padrão XMI é suportado pela grande maioria das ferramentas padrão de mercado voltadas para a modelagem de sistemas. Apesar de ser um padrão, a implementação do XMI difere um pouco de ferramenta para ferramenta. Mesmo assim, o presente trabalho foi testado com sucesso com as ferramentas ArgoUML, Umbrello, Rational Rose e StarUML.

A classe *ModelParser* basicamente efetua uma varredura deste arquivo procurando por determinadas *tags* e armazenando as informações encontradas no objeto *ProjectStructure*. São detectados atores, casos de uso, relações de inclusão, extensão, classes, interfaces, atributos, métodos, relacionamentos entre classes como heranças, associações, agregações, composições, dentre outros.

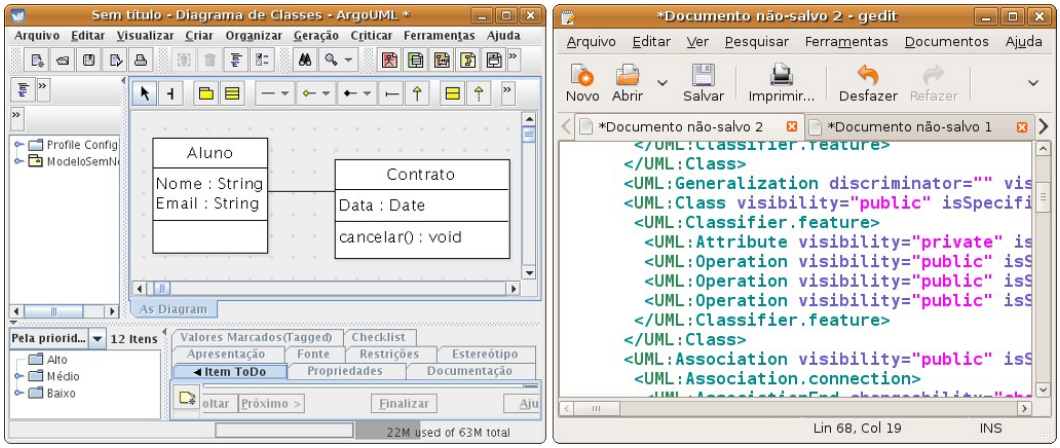


Figura 75. Modelo de classes UML e sua estrutura interna em XMI

A figura a seguir procura demonstrar um código-fonte de um aplicação PHP. A classe *SourceParser* é responsável por efetuar a varredura do código-fonte de uma aplicação, identificando classes, objetos, métodos, atributos e relações de dependência entre as classes, como a relação entre o método *matricular* da classe *Aluno* e a classe *Contrato*, exposto a seguir.

Além de encontrar classes, métodos e relacionamentos, *SourceParser* detecta cabeçalhos de documentação no padrão JavaDoc, contendo informações úteis sobre a descrição das estruturas, bem como os autores responsáveis.

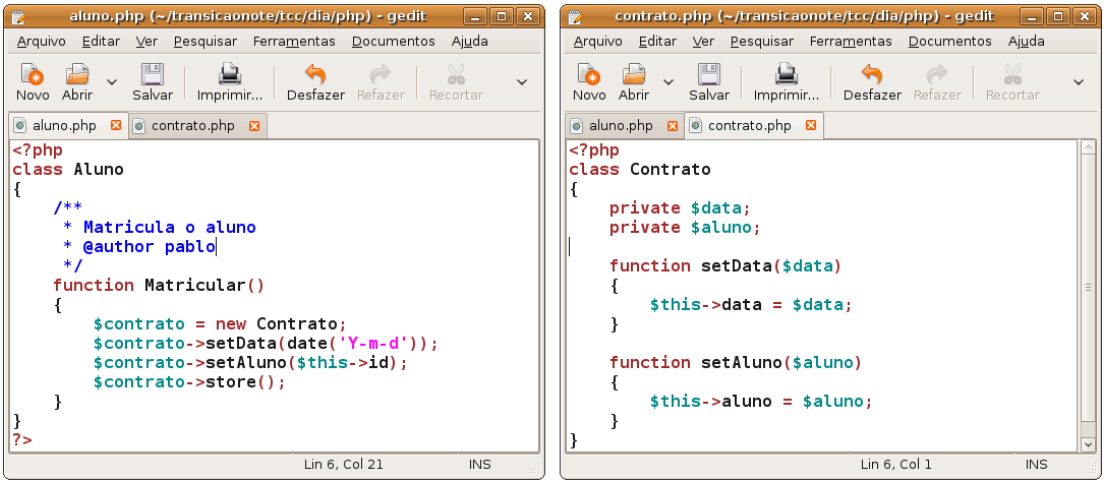


Figura 76. Código-fonte de classes em PHP

Para tornar a classe *SourceParser* genérica, para efetuar a leitura e interpretação dos códigos-fontes, ela baseia-se em um conjunto configurável de expressões regulares que identificam como as construções da linguagem são representadas. Inicialmente, a aplicação

foi testada com sucesso com códigos-fonte PHP e Java, mas pode ser extendida para atender outras linguagens também. A tabela a seguir procura demonstrar as expressões regulares utilizadas nestes trabalho para identificar as estruturas de cada linguagem. Cada linha representa um par de chave e conteúdo que identifica uma determinada construção da linguagem. Dentre as construções presentes, pode-se citar declaração de classes, heranças, declaração de interfaces, definição de métodos, instâncias de classes, chamadas estáticas, dentre outros.

Tabela 3 -Expressões regulares para detecção das estruturas PHP e Java

PHP :: Expressões Regulares
ereg_class = "/\s*class \s*(.*)/" ereg_inherit = "/\s*class \s*(.*)\s* extends\s*(.*)/" ereg_interface= "/\s*class \s*(.*)\s* implements\s*(.*)/" ereg_inheinter= "/\s*class \s*(.*)\s* extends\s*(.*) implements\s*(.*)/" ereg_method = "/(.*)function \s*(.*)\((.*)/" ereg_construct= "/(.*)function \s*__construct\((.*)\)/iU" ereg_instance = "/\$(.*)\s*=\s*new (.*)[;(]/iU" ereg_static = "/([a-zA-Z0-9_]*)::([a-zA-Z0-9_]*)[;(]/iU" ereg_call = "/\s*\$([a-zA-Z0-9_]*)->([a-zA-Z0-9_]*)[(]\s*/iU" ereg_attcall ="/\s*\$([a-zA-Z0-9_]*)->([a-zA-Z0-9_]*)->([a-zA-Z0-9_]*)[(]\s*/iU"
Java :: Expressões Regulares
ereg_class = "/\s*class \s*(.*)/" ereg_inherit = "/\s*class \s*(.*)\s* extends\s*(.*)/" ereg_interface = "/\s*class \s*(.*)\s* implements\s*(.*)/" ereg_inheinter = "/\s*class \s*(.*)\s* extends\s*(.*) implements\s*(.*)/" ereg_method = "/(private static|public static|protected static|private| public|protected) (.*[^Class]) \s*([A-z0-9]*)\((.*)\)/iU" ereg_construct = "/(public|protected|private)\s*([A-z0-9]*)\((.*)\)/iU" ereg_instance = "/\s*([A-z0-9]*)\s*=\s*new ([A-z0-9]*)[;(]/iU" ereg_static = "/\s*([A-z0-9]*).([A-z0-9]*)[(]\s*/iU" ereg_call = "/\s*([A-z0-9]*).([A-z0-9]*)[(]\s*/iU" ereg_attcall = "/\s*([A-z0-9]*).([A-z0-9]*).([A-z0-9]*)[(]\s*/iU"

4.7.2 DETECÇÃO DE MUDANÇAS

Após detectar a uma nova versão dos artefatos de software conforme detalhado no item anterior e registrar estas informações na base de dados, o agente *RequirementEngineer* processa as mudanças ocorridas desde a última versão conforme algoritmo a seguir.

Este algoritmo manipula conjuntos de entidades, sendo que uma entidade é considerada um elemento (artefato ou membro) ou um relacionamento entre elementos. Inicialmente, o algoritmo percorre todas entidades detectadas na nova versão do modelo. Então, para cada entidade detectada, o sistema verifica se ela já existia na versão anterior.

Caso ela já existia, então o sistema verifica se o seu conteúdo mudou desde então. Em caso positivo, o sistema irá criar uma mudança (*Change*) do tipo alteração e associar esta mudança com esta entidade. Caso a entidade não existia na última versão, então o sistema cria uma mudança (*Change*) do tipo criação e associa esta mudança com a nova entidade criada.

Após percorrer as entidades detectadas na versão mais atual do modelo, o sistema irá percorrer as entidades da versão anterior à atual. Para cada entidade pertencente à versão anterior, o sistema irá detectar se ela não existe mais na versão atual. Quando isto acontecer, significa que aquela entidade (artefato, membro ou relacionamento) foi excluída. Neste caso, para manter os registros de rastreabilidade e permitir a análise de impacto, o sistema irá clonar a entidade excluída a partir da última versão e registrar ela na versão mais atual e ao mesmo tempo irá criar uma mudança (*Change*) do tipo exclusão e irá associar esta mudança àquela entidade. Desta forma, o sistema terá o registro de que aquela entidade foi excluída. A figura a seguir procura demonstrar o pseudo-código destas operações.

```
PARA CADA ENTIDADE DETECTADA
        SE ELA JÁ EXISTIA NA VERSÃO ANTERIOR
                SE ELA O CONTEÚDO DA ENTIDADE É DIFERENTE
                        CRIA UMA MUDANÇA (CHANGE) DO TIPO ALTERAÇÃO
                        ASSOCIA A MUDANÇA À NOVA ENTIDADE
        CASO CONTRÁRIO
                CRIA UMA MUDANÇA (CHANGE) DO TIPO CRIAÇÃO
                ASSOCIA A MUDANÇA À NOVA ENTIDADE
        FIMSE
        ARMAZENA A ENTIDADE
FIMPARA

PARA CADA ENTIDADE DA VERSÃO ANTERIOR
        SE ELA NÃO EXISTE MAIS NA VERSÃO ATUAL
                CLONA A ENTIDADE ANTERIOR PARA A NOVA VERSÃO
                CRIA UMA MUDANÇA (CHANGE) DO TIPO EXCLUSÃO
                ASSOCIA A MUDANÇA À ENTIDADE
        FIMSE
FIMPARA
```

Figura 77. Pseudo-Algoritmo que trata a descoberta de uma nova versão dos artefatos

O diagrama a seguir procura demonstrar a organização dos objetos em memória após a detecção de uma nova versão. No lado esquero da figura tem-se a representação do artefato *Aluno*, que trata-se de uma classe, que por sua vez faz parte da versão 1 do sistema (objeto *Version*). Supondo que este artefato tenha sofrido algum tipo de modificação entre a versão 1 e a versão 2, seja uma inclusão, alteração ou exclusão, na versão 2 ele aparece relacionado com um objeto de mudança (*Change*), que ainda precisará ser classificada de acordo com sua razão (*Reason*) e origem (*Origin*), conforme a próxima etapa do método, que será exposta em seguida.

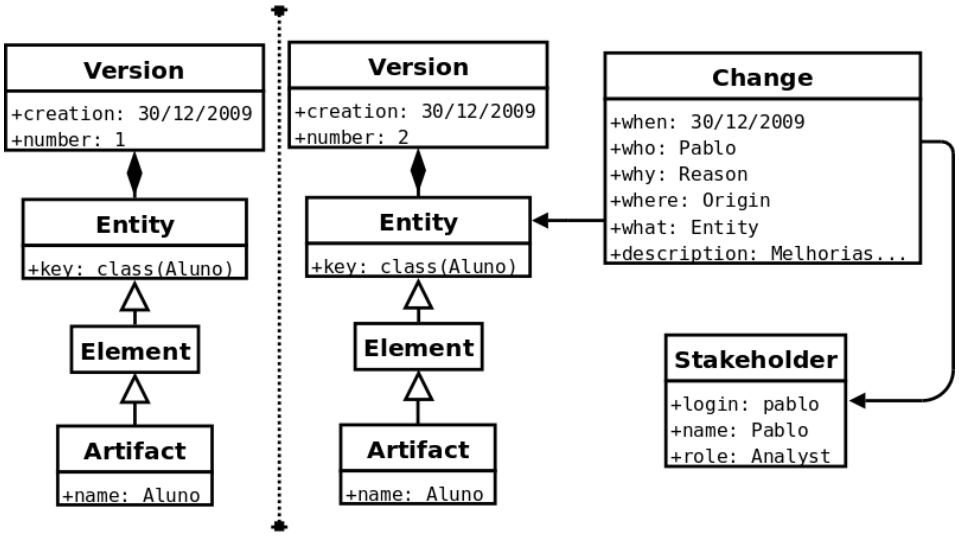


Figura 78. Objetos após registro de uma mudança

Após detectar e registrar as mudanças para inclusões, modificações e exclusões, o sistema irá percorrer todas as mudanças ocorridas e, para cada mudança, irá notificar o *stakeholder* causador daquela mudança. A notificação será realizada pelo agente *ProcessManager* por e-mail, cujo conteúdo pode ser visualizado na figura a seguir. O agente *ProcessManager* irá descobrir o e-mail do causador da modificação a partir das informações do último usuário que realizou um *commit* no repositório de arquivos (*subversion*). A partir do *login* do usuário, o agente *ProcessManager* irá vasculhar a base de *stakeholders* em busca de seu e-mail. A figura a seguir exibe as informações que o stakeholder recebe a partir de uma mudança ocorrida. São informações como a data da ocorrência, a descrição do *commit* efetuado pelo usuário no momento de enviar o arquivo para o repositório (colhida a partir dos *logs* do subversion), o *stakeholder* causador da mudança (o próprio), a entidade alterada e o tipo de mudança. O e-mail de notificação apresentará ao *stakeholder* além dos dados da alteração, um *link* que funciona como um atalho para a aplicação, para que este possa acessar diretamente a funcionalidade que permite classificar a mudança dentro do sistema.

Figura 79. Notificação de mudanças ao stakeholder

4.7.3 CLASSIFICAÇÃO E MÉTRICAS

Após receber notificação sobre uma mudança ocorrida, o *stakeholder* poderá classificar esta mudança diretamente dentro do sistema. Após identificar-se mediante *login*, o stakeholder será direcionado para uma interface que apresenta o conjunto de mudanças provocadas por ele que ainda não foram classificadas. Este recurso permite que o *stakeholder* possa agrupar um conjunto de modificações que tenham relações entre si e aplicar a mesma classificação sobre elas. A figura a seguir demonstra a primeira parte desta interface (aba "mudanças"), onde o *stakeholder* poderá selecionar uma ou muitas mudanças para classificar.

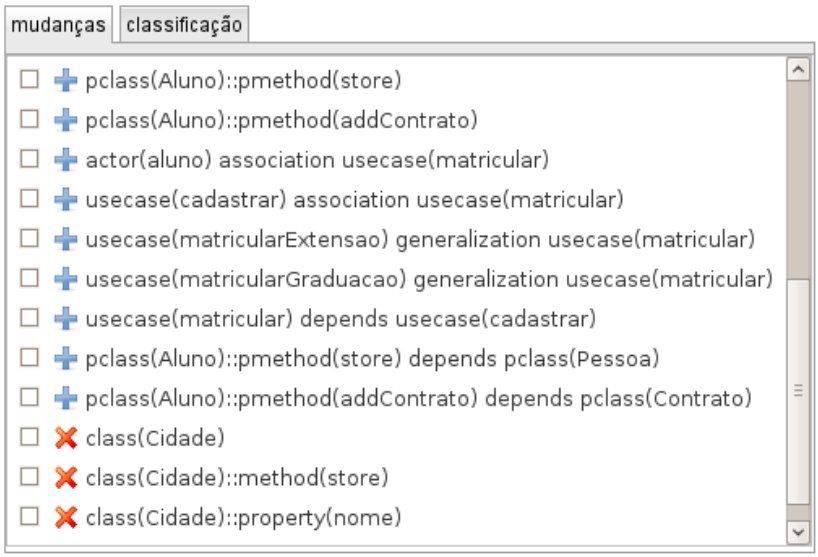


Figura 80. Seleção de mudanças para classificação

Após selecionar as mudanças a serem classificadas, o *stakeholder* poderá partir para a "aba" classificação, onde ele irá definir a razão da mudança ter acontecido e também a origem da mudança, ou seja, o lugar em que ela se originou. Ambas classificações apresentada aqui (razão e origem) seque a taxonomia apresentada por Nurmuliani, Zowghi e Fowell (2004), já detalhada no capítulo 2. As oções selecionáveis pelo usuário são exatamente as mesmas apresentadas por Nurmuliani, Zowghi e Fowell (2004). Após realizar estas duas classificações, o usuário ainda poderá preencher uma descrição textual e armazenar esta classificação na base de dados. A partir da classificação da mudança, o processo terá continuidade com a verificação da consistência da rastreabilidade.

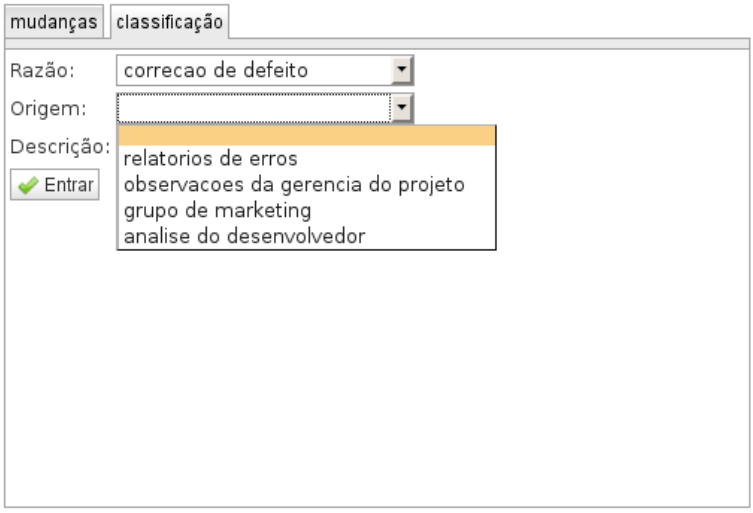


Figura 81. Classificação de mudanças

O diagrama a seguir procura demonstrar a organização lógica dos objetos em memória após a classificação. Neste caso, é atribuído ao objeto de mudança (*Change*), uma origem (*Origin*) e razão (*Reason*).

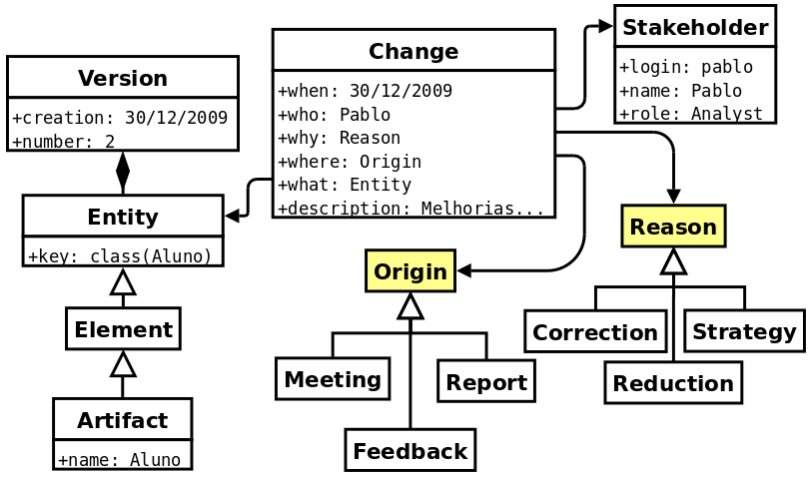


Figura 82. Objetos em memória após classificação das mudanças

4.7.4 VERIFICAÇÃO DE CONSISTÊNCIA

Após a classificação das mudanças ocorridas, o sistema verifica a consistência da rastreabilidade. Basicamente, a verificação de consistência compreende a análise dos relacionamentos entre os elementos do sistema após a mudança, a fim de verificar possíveis causas de impactos.

A verificação de consistência é realizada a partir de consultas ao modelo, onde procura-se detectar três situações: relacionamentos entre elementos excluídos, elementos de análise ou *design* sem pós-rastreabilidade e elementos de *design* e implementação sem pré-rastreabilidade.

A figura a seguir procura demonstrar o e-mail que relata as inconsistências encontradas em uma versão do sistema. Na primeira parte, encontram-se os relacionamentos inconsistentes, na segunda parte as possíveis inconsistências de pós-rastreabilidade e em seguida, as possíveis inconsistências de pré-rastreabilidade.

Figura 83. Notificação de inconsistências de rastreabilidade

Inicialmente, a consulta irá detectar quais são os relacionamentos entre elementos que encontram-se no estado “excluído”. É importante ressaltar que, elementos excluídos

permanecem até até a versão em que a exclusão ocorreu, sendo relacionados com mudanças (*change*) do tipo exclusão. Após descobrir os relacionamentos inconsistentes, o sistema irá vasculhar entre os requisitos textuais e artefatos de *design*, aqueles que não possuem relacionamento de pós-rastreabilidade, ou seja, não possuem ligações com elementos de fases subsequêntes. Por fim, serão descobertos os artefatos de *design* e de implementação que não possuem relacionamento de pré-rastreabilidade, ou seja, não possuem ligações com elementos de fases anteriores do ciclo de desenvolvimento.

4.7.5 ANÁLISE DE IMPACTO

Após o usuário classificar as mudanças e o sistema realizar a verificação da consistência nas relações de rastreabilidade, é procedida a análise de impacto. A análise de impacto é realizada sobre todas aquelas entidades que sofreram algum tipo de mudança, sendo alteradas ou excluídas. Neste caso, é necessário descobrir quais são os outros elementos (artefatos ou membros) que podem ser afetados por esta modificação.

A análise de impacto é realizada sobre o conjunto total de elementos do sistema, não importando sua granularidade. Neste caso, a análise de impacto poderá detectar dependência em nível de requisitos (artefato) ou em nível de palavras (membro), e também em nível de classes (artefato) ou em nível de métodos (membro). A análise de impacto atua sobre relacionamentos de rastreabilidade vertical, como entre duas classes de um modelo de *design* e também sobre relacionamentos de rastreabilidade horizontal, como entre um requisito textual e um caso de uso.

A figura a seguir procura ilustrar as capacidades da análise de impacto. Neste sentido, deve-se visualizar os relacionamentos entre os elementos do sistema como uma rede com diversos nodos interligados, onde cada nodo representa um elemento distinto. No exemplo que segue, tem-se os seguintes elementos: a-requisito textual *matricular*; b-caso de uso *matricular*; c-caso de uso *cancelar*; d-classe de design *Aluno*; e-classe de design *Matricula*; f-classe de design *Contrato*; g-classe de implementação *Aluno*; h-classe de implementação *Matricula*; i-classe de implementação *Contrato*; j-classe de implementação *MatriculaScreen*.

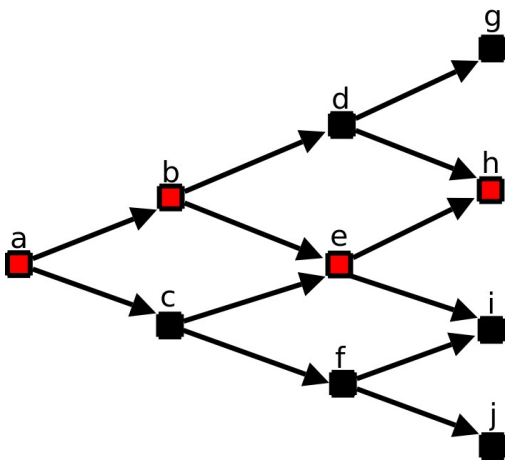


Figura 84. Rede de entidades afetadas por uma mudança

Como exemplo, pode ser citado o caminho a-b-e-h, destacado por uma cor diferenciada na figura anterior e representada de forma visual na próxima figura. Neste caminho, temos o elemento **a**, que representa um requisito textual contendo a descrição de um procedimento de matrícula. Este requisito textual está interligado com o elemento **b**, que representa um caso de uso de um diagrama UML. Este caso de uso, por sua vez, está interligado com o elemento **e**, que representa uma classe chamada *matricula*, dentro do diagrama de classes do sistema. Esta classe, por sua vez, está interligada com o elemento **h**, que representa uma classe de implementação (código-fonte) de mesmo nome. Neste exemplo em particular somente foram exemplificadas relacionamentos *coarse-grained* para simplificar a explicação, embora o sistema suporte o relacionamento não somente entre artefatos, mas também entre membros destes.

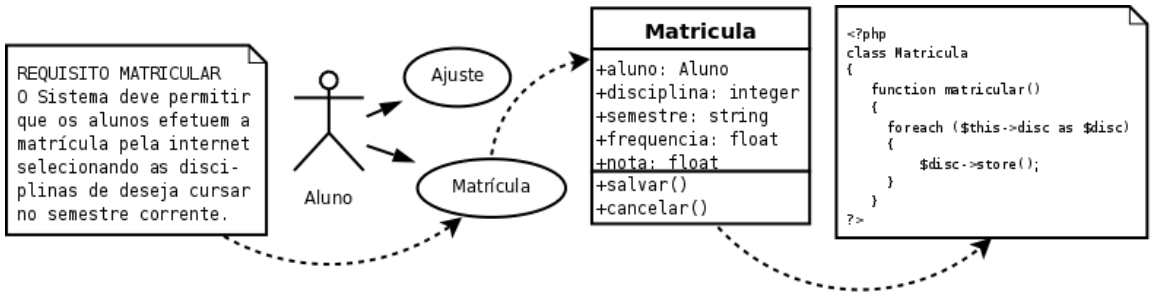


Figura 85. Exemplo de rastreabilidade horizontal

Para realizar a análise de impacto, a aplicação realiza uma conversão entre a informação sobre as entidades (artefatos, membros e relacionamentos) contida em sua base de dados para uma representação de axiomas em Prolog. A tabela a seguir procura demonstrar

quais são as regras de conversão para requisitos textuais. Basicamente tem-se duas definições: *requirement*, que declara um artefato do tipo requisito textual contido em um arquivo e *word*, que é um termo relevante dentro do requisito textual. Este termo é declarado na forma de um membro do requisito (*member*) e a construção *partof* é utilizada para relacionar os dois.

Tabela 4 -Regras de conversão entre requisitos e Prolog

Requisitos textuais	
Requisito	`requirement(<requirement_name>).`

Termo relevante	`member( requirement( <requirement_name> ), word( <word> )).`
	`partof( requirement( <requirement_name> ),`
	`  member( requirement( <requirement_name> ), word( <word> ))).`

A tabela a seguir procura demonstrar as regras de conversão entre elementos de um modelo de *design* UML e Prolog. Neste caso uma classe é representada pela construção *class*, uma interface pela construção *interface*, um ator por *actor* e um caso de uso por *usecase*. Todos estas construções possuem o nome do artefato entre parêntesis. Membros destes artefatos como métodos ou propriedades são representadas pela construção *member* em conjunto com as construções *method* ou *property* para declaração, seguidas pela construção *partof*, que realiza a conexão entre o artefato e seu membro. As construções do tipo *member* sempre são formadas por dois parâmetros: em primeiro lugar é indicado o nome do artefato ao qual o membro pertence e em segundo lugar consta a definição do próprio membro, composta por seu tipo como *method* ou *property* e seu nome. A definição de um membro se dá desta forma por que é comum existirem nomes de membros iguais participando de artefatos diferentes. Um exemplo disto é um nome de método, que pode repetir-se em diversas classes. A construção *partof* existe para tornar explícita a relação entre um artefato (primeiro parâmetro) e seu membro (segundo parâmetro).

Tabela 5 -Regras de conversão entre design e Prolog

Models de design UML	
Classe	`class( <class_name> )`
Interface	`interface( <interface_name> ).`
Ator	`actor( <actor_name> ).`
Use case	`usecase( <usecase_name> ).`

Método	`member( class( <class_name> ), method( <method_name> )).`
	`partof( class( <class_name> ),`
	`  member( class( <class_name> ), method( <method_name> ))).`

Atributo	`member( class( <class_name> ), property( <method_name> )).`
	`partof( class( <class_name> ),`
	`  member( class( <class_name> ), property( <method_name> ))).`

A tabela a seguir procura demonstrar as regras de conversão entre elementos de implementação (código-fonte da aplicação) e Prolog. As declarações seguem basicamente o mesmo padrão que as construções de *design*. Neste caso uma classe é representada pela construção *pclass*, uma interface pela construção *pinterface*. Membros destes artefatos como métodos ou propriedades são representadas pela construção *member* em conjunto com as construções *pmethod* ou *pproperty* para declaração, seguidas pela construção *partof*, que realiza a conexão entre o artefato e seu membro.

Tabela 6 -Regras de conversão entre implementação e Prolog

Código-fonte	
Classe	`pclass( <class_name> ).`
Interface	`pinterface( <interface_name> ).`

Método	`member( pclass( <class_name> ), pmethod( <method_name> )).`
	`partof( pclass( <class_name> ),`
	`  member( pclass( <class_name> ), pmethod( <method_name> ))).`

Atributo	`member(pclass( <class_name> ), pproperty( <method_name> )).`
	`partof( pclass( <class_name> ),`
	`  member( pclass( <class_name> ), pproperty( <method_name> ))).`

Por fim são representadas as regras de conversão para relacionamentos entre os elementos do modelo. Neste caso, a construção adotada chama-se *traceto* e é utilizada para conectar dois elementos, sendo que estes podem ser artefatos (classes, atores, use cases) ou membros (palavras de um requisito, métodos ou propriedades de uma classe). O segundo parâmetro desta declaração correponde ao tipo de relação entre os dois elementos, e podem ser do tipo: associação (*association*), agregação (*aggregation*), composição (*composition*), generaliação (*generalization*), realização (*implements*) e dependência (*depends*).

Tabela 7 -Regras de conversão entre os relacionamentos e Prolog

Relacionamentos	
Associação	`traceto( <element>, 'association', <element> ).`
Agregação	`traceto( <element>, 'aggregation', <element> ).`
Composição	`traceto( <element>, 'composition', <element> ).`
Generalização	`traceto( <element>, 'generalization', <element> ).`
Realização	`traceto( <element>, 'implements', <element> ).`
Dependência	`traceto( <element>, 'depends', <element> ).`

Para que seja possível realizar a análise de impacto, é necessária a existência de certas funções lógicas que verificam as relações de dependência e composição. Neste sentido, são

declaradas a função *dependency*, que define quais são os tipos de construção consideradas dependência.

O presente trabalho considera que as construções *partof* e *traceto* estabelecem relações de dependência entre dois elementos de um modelo. Para realizar a consulta sobre as relações de dependência no sistema, a função *dependof* será utilizada, uma vez que ela adiciona recursividade à função *dependency*, descobrindo relações de dependência recursivamente. A função *depend1g* é utilizada para descobrir relações de dependência de primeiro grau e a função *depend2g* é utilizada para descobrir relações de dependência de segundo grau. A função *write_table* é utilizada para formatar a saída dos resultados da análise de impacto, que serão retornados para a aplicação.

Tabela 8 -Funções pré-definidas em Prolog

Tipo de função	**Implementação**

Funções de dependência entre elementos do modelo	`dependency(X,Y) :- partof(Y,X).`
	`dependency(X,Y) :- traceto(X,_,Y), not(traceto(Y,_,X)).`
	
	`dependof(X,Y) :- dependency(X,Y).`
	`dependof(X,Z) :- dependency(X,Y), dependof(Y,Z).`
	
	`depend1g(X,Y) :- dependency(X,Y).`
	`depend2g(X,Z) :- dependency(X,Y), dependency(Y,Z).`

Funções de retorno de informações	`write_table([]).` `write_table([I|IT]) :- writeln(I), write_table(IT).`

A partir dos padrões de mapeamento apresentados, será apresentado um pequeno exemplo real de artefatos de software representados em Prolog conforme os padrões definidos. Este exemplo será baseado em conceitos utilizados em um pequeno sistema acadêmico, onde tem-se alunos, matrículas e turmas. Para o exemplo não ficar muito extenso, ele será dividido em seções menores.

A primeira seção a ser demonstrada trata da definição dos artefatos do sistema, que são elementos com um nível de rastreabilidade *coarse-grained*. Aqui, tem-se artefatos como os requisitos textuais *Realizar Matrícula* e *Trancar Contrato*, da fase de especificação de requisitos. O ator *Aluno*, os use cases *Matricular* e *Trancar*, além das classes *Aluno*, Turma e *Matricula* do modelo de *design* UML. Por fim, as classes de implementação (código-fonte) *Model*, *AlunoModel*, *TurmaModel* e *MatriculaModel*. *Model* é uma super-classe de um *framework* que define o comportamento básico de um objeto de negócios *Active Record*. Desta forma, as classes filhas *AlunoModel*, *TurmaModel* e *MatriculaModel* herdam o comportamento básico que permite salvar, carregar e excluir registros do banco de dados.

Tabela 9 -Definição de artefatos em Prolog

Definição de artefatos

```
requirement('Realizar Matrícula').
requirement('Trancar Contrato').
actor('Aluno').
usecase('Matricular').
usecase('Trancar').
class('Aluno').
class('Turma').
class('Matricula').
pclass('Model').
pclass('AlunoModel').
pclass('TurmaModel').
pclass('MatriculaModel').
```

A segunda seção a ser demonstrada trata da definição dos membros de artefatos do sistema, que são elementos com um nível de rastreabilidade *fine-grained*. Aqui, tem-se a definição dos termos relevantes *aluno* e *turma*, que fazem parte do requisito *Realizar Matrícula*, dos métodos *Matricular* e Tra*ncar*, que fazem parte da classe de *design Aluno*, dos métodos *store* e *delete* que fazem parte da classe de implementação *Model* e dos métodos *Matricular* e *Trancar* que fazem parte da classe de implementação *AlunoModel*.

A segunda parte trata da declaração das relações *partof* (todo/parte) existentes entre os artefatos e seus membros. Para cada declaração de *member*, existirá exatamente uma declaração do tipo *partof*. Cabe salientar que as relações do tipo *partof* existem em função da análise de impacto, uma vez que este tipo de relacionamento corresponde à um tipo de dependência..

Tabela 10 -Definição de membros de artefatos em Prolog

Definição de membros de artefatos

```
%% declaração dos membros
member(requirement('Realizar Matrícula'), word('aluno')).
member(requirement('Realizar Matrícula'), word('turma')).
member(class('Aluno'), method('Matricular')).
member(class('Aluno'), method('Trancar')).
member(pclass('Model'), pmethod('store')).
member(pclass('Model'), pmethod('delete')).
member(pclass('AlunoModel'), pmethod('Matricular')).
member(pclass('AlunoModel'), pmethod('Trancar')).

%% relações partof
partof( requirement('Realizar Matrícula'),
        member(requirement('Realizar Matrícula'), word('aluno'))).
partof( requirement('Realizar Matrícula'),
        member(requirement('Realizar Matrícula'), word('turma'))).
partof( class('Aluno'), member(class('Aluno'), method('Matricular'))).
partof( class('Aluno'), member(class('Aluno'), method('Trancar'))).
partof( pclass('Model'), member(pclass('Model'), pmethod('store'))).
partof( pclass('Model'), member(pclass('Model'), pmethod('delete'))).
partof( pclass('AlunoModel'),
        member(pclass('AlunoModel'), pmethod('Matricular'))).
partof( pclass('AlunoModel'), member(pclass('AlunoModel'), pmethod('Trancar'))).
```

A terceira seção a ser demonstrada trata da definição dos relacionamentos entre elementos de um mesmo modelo (rastreabilidade vertical). Neste sentido, são apresentados relacionamentos entre artefatos do modelo de *design*, como é o caso do relacionamento de associação entre o ator *Aluno* e o *Use Case Matricular*, o relacionamento de composição entre a classe de design *Aluno* e a classe *Matricula*, o relacionamento de associação entre a classe de design *Matricula* e a classe *Turma.*

Também são apresentados relacionamentos entre artefatos de implementação (código-fonte), como é o caso do relacionamento de generalização entre as classes de implementação *AlunoModel*, *TurmaModel* e *MatriculaModel* em relação à classe *Model* e a relação de dependência entre as classes de implementação *AlunoModel* e *MatriculaModel* e entre as classes *MatriculaModel* e *TurmaModel*. É importante destacar novamente que os relacionamentos de nível vertical são coletados automaticamente pelo agente *RequirementEngineer* com o auxílio das classes *TextualParser*, *ModelParser* e *SourceParser*.

Figura 86. Definição da rastreabilidade vertical

Rastreabilidade vertical

```
traceto(actor('Aluno'), 'association', usecase('Matricular')).
traceto(class('Aluno'), 'composition', class('Matricula')).
traceto(class('Matricula'), 'association', class('Turma')).
traceto(pclass('AlunoModel'), 'extends', pclass('Model')).
traceto(pclass('TurmaModel'), 'extends', pclass('Model')).
traceto(pclass('MatriculaModel'), 'extends', pclass('Model')).
traceto(pclass('AlunoModel'), 'depends', pclass('MatriculaModel')).
traceto(pclass('MatriculaModel'), 'depends', pclass('TurmaModel')).
```

A quarta e última seção a ser demonstrada trata da definição dos relacionamentos entre elementos de diferentes modelos (rastreabilidade horizontal). Neste sentido, são apresentadas algumas relações entre requisitos textuais e modelos de design, como é o caso da associação entre o *Use Case Matricular* e o requisito *Realizar Matrícula* e entre o *Use Case Trancar* e o requisito *Trancar Matrícula* e entre artefatos da fase de análise e artefatos da fase de *design* de modelos UML como é o caso da associação entre o método *Matricular* da classe de design *Aluno* com o *Use Case Matricular*, e entre o método *Trancar* da mesma classe e o *Use Case Trancar*.

Também constam relacionamentos entre artefatos de implementação (código-fonte) e artefatos de *design*, como é o caso da relação entre a classe de implementação *AlunoModel* e a classe de design *Aluno*, entre as classes *TurmaModel* (implementação) e *Turma* (design) e entre as classes *MatriculaModel* (implementação) e *Matricula* (design), que são

relacionamentos *coarse-grained*. Além destes, constam os relacionamentos de dependência entre o método *Matricular* da classe de implementação *AlunoModel* e o método *Matricular* da classe de design *Aluno* e também o relacionamento entre o método *Trancar* da classe *AlunoModel* (implementação) e o método *Trancar* da classe *Aluno* (*design*), que são relacionamentos *fine-grained*. É importante destacar novamente que os relacionamentos de nível horizontal podem ser registrados manualmente via interface do sistema ou também podem ser sugeridos a partir de regras de similaridade entre termos pelo sistema.

Figura 87. Definição da rastreabilidade horizontal

Rastreabilidade horizontal

```
traceto(usecase('Matricular'), 'association', requirement('Realizar Matrícula')).
traceto(usecase('Trancar'), 'association', requirement('Trancar Contrato')).
traceto(member(class('Aluno'), method('Matricular')), 'association',
        usecase('Matricular')).
traceto(member(class('Aluno'), method('Trancar')), 'association',
        usecase('Trancar')).
traceto(pclass('AlunoModel'), 'depends', class('Aluno')).
traceto(pclass('TurmaModel'), 'depends', class('Turma')).
traceto(pclass('MatriculaModel'), 'depends', class('Matricula')).
traceto(member(pclass('AlunoModel'), pmethod('Matricular')), 'depends',
        member(class('Aluno'), method('Matricular'))).
traceto(member(pclass('AlunoModel'), pmethod('Trancar')) , 'depends',
        member(class('Aluno'), method('Trancar'))).
```

A partir da declaração dos artefatos, membros de artefatos e relacionamentos, bem como as funções de análise previamente detalhadas, é possível realizar a consulta ao modelo sobre todas as relações de dependência existentes a partir de um determinado elemento.

Na consulta a seguir, é realizada a consulta para descobrir todos os elementos dependentes do requisito textual *Realizar Matricula*. Esta operação seria realizada nos casos em que este elemento sobre altum tipo de mudança do tipo alteração ou uma exclusão, a fim de averiguar os elementos que possivelmente teriam de ser revistos a partir de uma mudança de conceitos.

Na primeira parte da tabela tem-se a declaração da função *findall*, que busca todos elementos dependentes de um determinado conceito por meio da função *dependof*. A segunda parte da tabela traz os resultados que são: os termos *aluno* e *turma* deste requisito, que possuem relação todo/parte, o *Use Case Matricular* que está associado com este requisito e o ator *Aluno*, que está associado com o *Use Case Matricular*. Também são relacionados os métodos *Matricular* da classe de *design Aluno*, que possui relação com o *Use Case Matricular* e o método *Matricular* da classe de implementação *AlunoModel* que depende do método *Matricular* da classe de *design Aluno*.

Tabela 11 -Análise de impacto

Análise de impacto
`findall(X,dependof(X, requirement('Realizar Matrícula')),List),` `write_table(List).`

`member(requirement(Realizar Matrícula), word(aluno))`
`member(requirement(Realizar Matrícula), word(turma))`
`usecase(Matricular)`
`actor(Aluno)`
`member(class(Aluno), method(Matricular))`
`member(pclass(AlunoModel), pmethod(Matricular))`

O diagrama a seguir procura demonstrar a sequência de atividades que inicia com a classificação de uma mudança por parte do *stakeholder* e é concluída pela análise de impacto e consequênte notificação dos efeitos. As atividades iniciam quando o *stakeholder* classifica uma mudança via interface da aplicação, que aciona o método *classifyChange* do agente *ImpactAnalyser*. Após a classificação ser realizada, o mesmo controlador de interface aciona o método *checkConsistency* do agente *TraceabilityManager*, que verifica a consistência da rastreabilidade. Caso alguma inconsistência tenha sido encontrada, então é acionado o método *notifyInconsistency* do agente *ProcessManager* para notificar esta ocorrência.

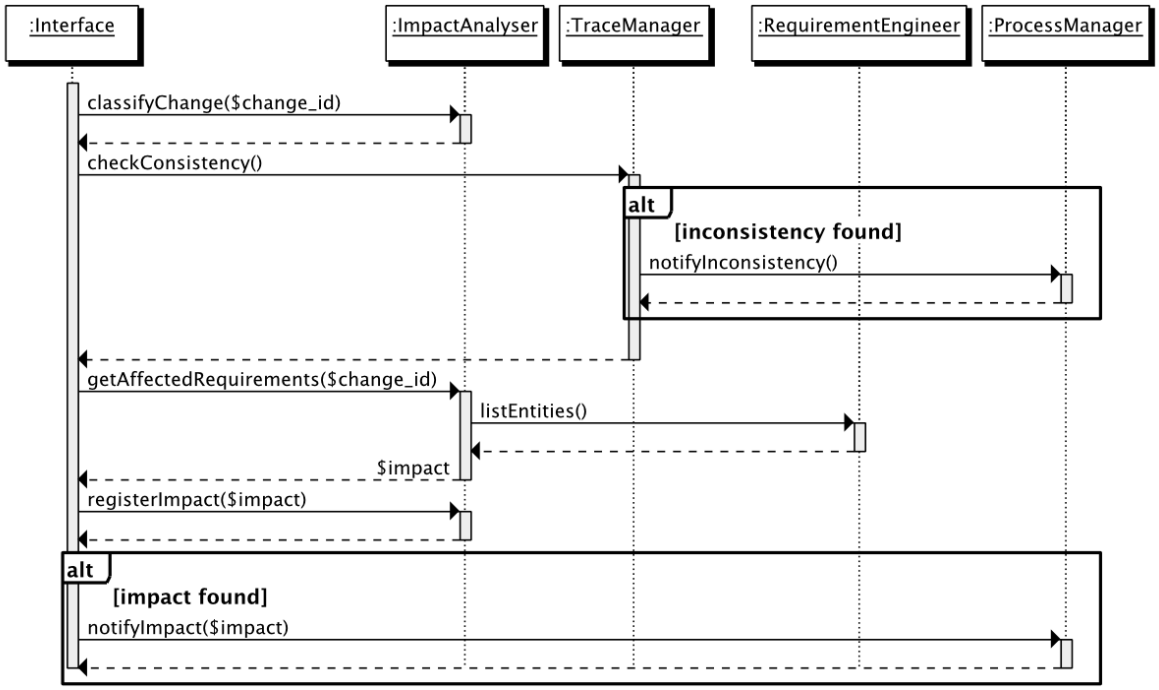


Figura 88. Sequência de atividades da análise de impacto

Após verificar a consistência, é acionado o método *getAffectedRequirements* do agente *ImpactAnalyser*, que aciona a máquina de inferência em Prolog para decobrir o conjunto de

elementos afetados pela mudança. Este método, por sua vez, utiliza o método *listEntities* do agente *RequirementEngineer* para descobrir a relação de todas entidades (artefatos, membros e relacionamentos) do sistema. Após descoberto o impacto, o mesmo é registrado por meio do método *registerImpact*. Caso realmente tenha ocorrido algum impacto, o método *notifyImpact* do agente *ProcessManager* é acionado para notificar os *stakeholders* impactados.

O agente *ProcessManager* descobre a identificação do *stakeholder* responsável pelo elemento afetado de diferentes maneiras. No caso dos requisitos textuais, a identificação do *stakeholder* responsável pelo requisito consta em um campo específico para tal, no documento de especificação. No caso de modelos de design UML, que não constam com um campo para tal informação, o sistema busca a partir do último usuário que realizou commit daquele modelo no repositório (*subversion*). E nos casos de códigos-fontes, o sistema busca o usuário responsável por determinada classe ou método a partir dos cabeçalhos *javadoc* de cada método, que contém a identificação por meio do uso da tag *@author*.

A partir da análise efetuada, o sistema parte para a notificação dos responsáveis pelos elementos afetados. Neste caso, casa um dos responsáveis receberá um e-mail como demonstrado pela figura a seguir, onde constam informações sobre a modificação como a data, a descrição de sua ocorrência e o seu causador, dados da classificação como a razão e a origem, e os elementos impactados. Também consta um *link* para a operação de resolução do impacto, que registra aquele impacto como resolvido no sistema.

AgentChange

Um Sistema MultiAgente Colaborativo para Gestão da mudança de Requisitos de Software

Notificação de Impacto	
Data da ocorrência	2009-10-21
Descrição da ocorrência	re matr
Stakeholder causador	2 - Pablo
Entidade alterada	requirement(matricular)::word(contrato)
Tipo de ocorrência	2 - Exclusion
Razão	2 - requisito faltante
Origem	4 - analise do desenvolvedor
Impactos	class(Contrato)**(pablo)** class(Contrato)::method(addTurma)**(pablo)** class(Contrato)::method(get_nome_curso)**(pablo)** class(Contrato)::method(store)**(pablo)** class(Contrato)::property(dt_registro)**(pablo)** class(Aluno)**(pablo)** pclass(Contrato)**(pablo)**
Resolva este impacto clicando aqui.	

Figura 89. Notificação de impactos por e-mail

4.7.6 RESOLUÇÃO DE IMPACTOS

Após a notificação dos impactos e a consequente resolução dos mesmos por parte dos desenvolvedores nos casos em que há a necessidade, é necessário registrar no sistema a resolução adotada. Em princípio, os impactos ficam abertos até serem registrados como resolvidos. O formulário de resolução questiona o usuário sobre o *stakeholder* responsável pela resolução, sobre a efetividade da análise de impacto e pede ainda uma descrição textual sucinta para fins de registro.

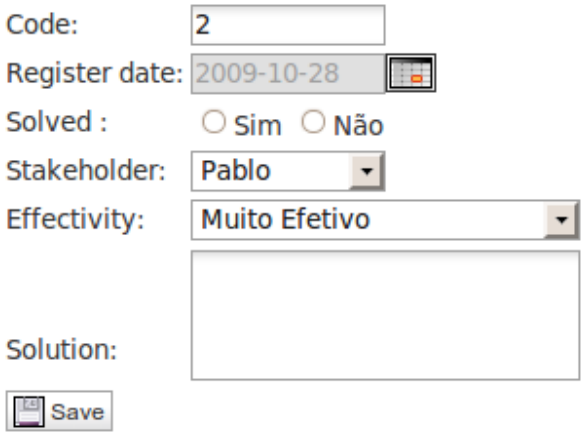


Figura 90. Formulário de resolução de impactos

4.8 PROTÓTIPO

Apesar de várias telas do protótipo desenvolvido já terem sido demonstradas no tópico anterior, no qual era detalhado o método de trabalho, neste tópico serão demonstradas as demais funcionalidades da ferramenta desenvolvida.

A figura a seguir procura demonstrar a arquitetura do protótipo. Este protótipo constitui-se em uma aplicação Web (quadro Aplicação) apoiada pelos agentes (quadro *Web Services*). Diferentemente de outras aplicações tradicionais, que comunicam-se diretamente com seu banco de dados, esta apliação comunica-se somente com os agentes, que provêem todos os serviços necessários para seu funcionamento, desde a gerência do projeto e de seus *stakeholders*, até o gerenciamento de mudanças e análise de impactos. A aplicação protótipo nada mais é do que uma interface para os recursos disponibilizados pelos *Web Services*.

A aplicação protótipo é formada por um conjunto de classes de interface que implementam o *design pattern Page Controller*, tais como *ProjectList*, *ProjectForm*, *ArtifactReport*, dentre outras. Ao todo, são cerca de 25 classes de controle de interface. A seguir, serão demonstradas algumas de suas principais funcionalidades.

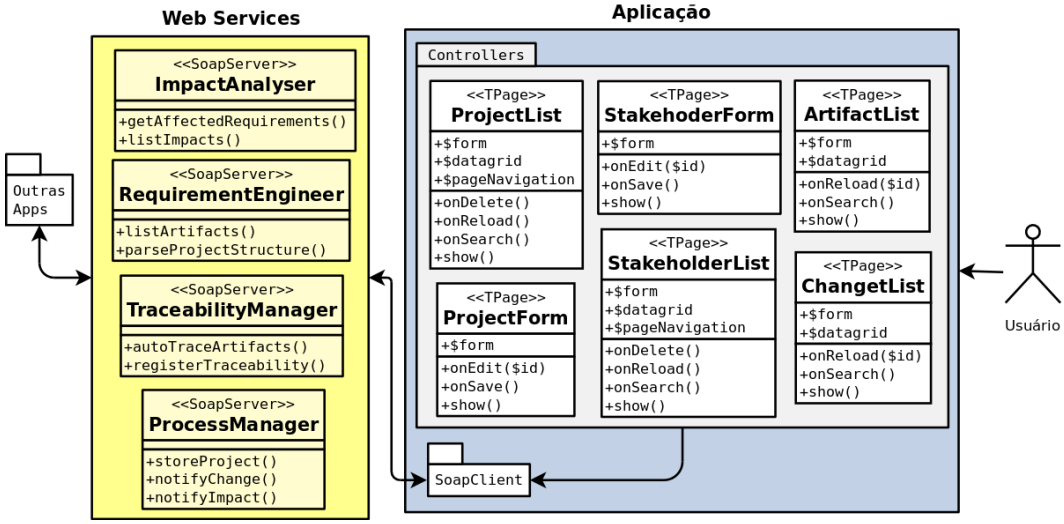


Figura 91. Arquitetura do protótipo

4.8.1 CONFIGURAÇÃO DE PROJETO

A gerência de um projeto pela ferramenta proposta inicia com a configuração de um projeto. Para tal, deve-se registrar um projeto no sistema informando seus dados no formulário de cadastro de projetos. Este formulário é dividido em três partes. Na primeira parte, são informados os dados como nome do projeto, data de criação, e a sua localização no repositório central (*subversion*). Para tal, é necessário informar ao sistema a localização dos requisitos textuais, do modelo de *design* XMI e também do código-fonte da aplicação. Estes três conjuntos de artefatos podem estar localizados em pastas diferentes de um mesmo repositório ou até mesmo em repositórios distintos. Por fim, é informada a linguagem do projeto, por meio da qual o sistema saberá qual conjunto de expressões regulares a utilizar para fins de *parsing*.

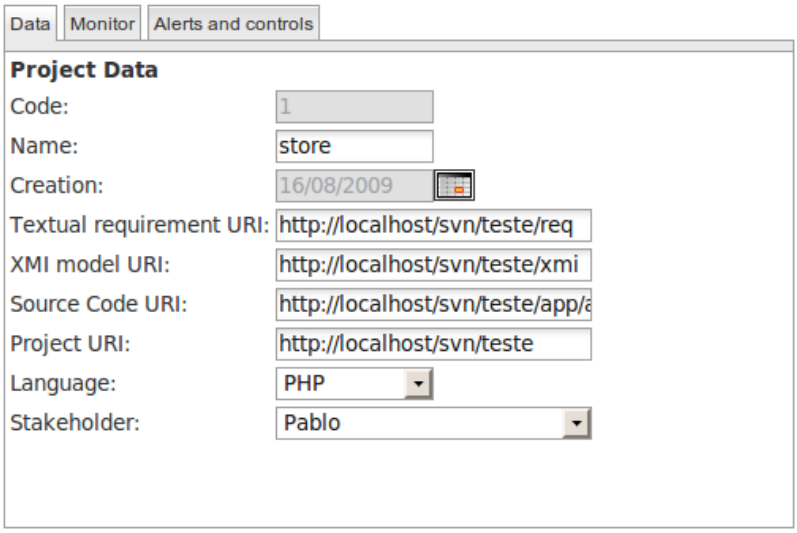


Figura 92. Configuração dos dados básicos do projeto

Após definir os dados básicos do projeto, deve-se definir a sua configuração de monitoramento. Nesta seção é definida a frequência com a qual as alterações do projeto devem ser monitoradas: a cada hora, a cada dia, a cada semana ou a cada mês. Neste sentido, ainda pode-se definir dia, hora e minuto exatos que deseja-se realizar a monitoria.

Além de definir a frequência, é possível indicar quais os tipos de artefatos devem ser monitorados no projeto, como: requisitos textuais, diagramas de caso de uso, diagramas de classe e código-fonte.

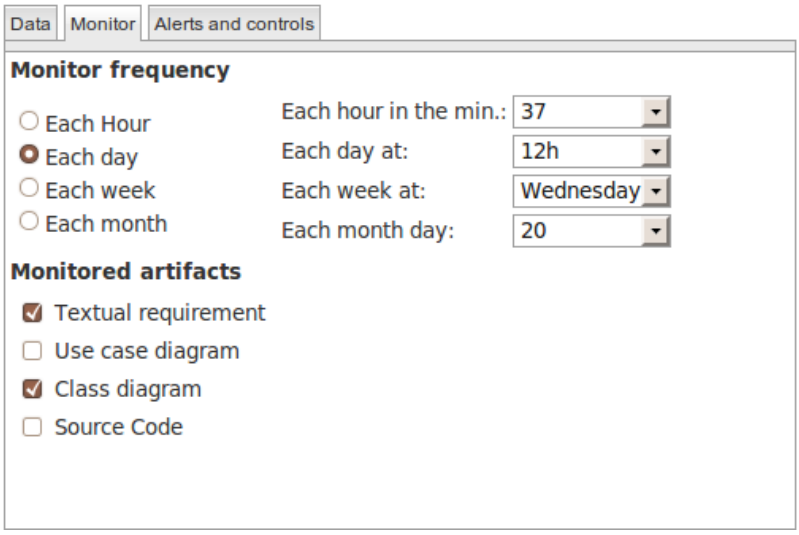


Figura 93. Configuração sobre o monitoramento do projeto

Após definir as características de monitoramento, pode-se definir quais alertas e

controles estarão habilitados no sistema. Neste sentido, é possível monitorar alterações relativas à criação, mudança e exclusão de elementos, que podem ser tanto artefatos como membros de artefatos, e também monitorar a criação e exclusão de *links* de rastreabilidade.

Além disto, é possível habilitar e desabilitar a notificação relativa à inconsistência da rastreabilidade, bem como o envio do relatório de análise de impacto aos *stakeholders* impactados.

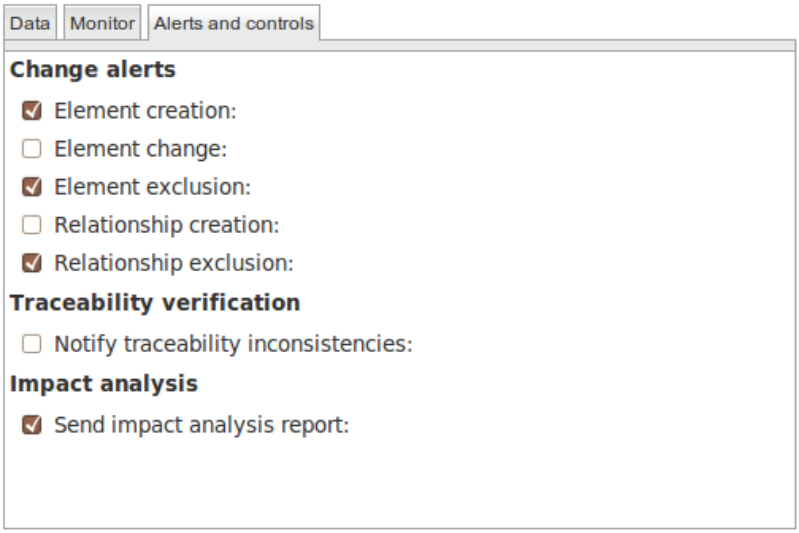


Figura 94. Configuração sobre alertas e controles do projeto

4.8.2 CADASTRO DE *STAKEHOLDERS*

O cadastro de *stakeholders* indica quem terá acesso à interface do sistema. É importante pois o login informado neste cadastro será utilizado para identificar o stakeholder em diversas etapas do processo. Como exemplo pode-se citar a análise de impacto, onde o *login* do usuário é utilizado para identificar o stakeholder responsável por cada artefato. Neste sentido, o login é informado no formulário de requisitos textuais, é utilizado para realizar *commit* dos artefatos de design e é informado nos cabeçalhos dos métodos de código-fonte para identificar o responsável por cada classe e método.

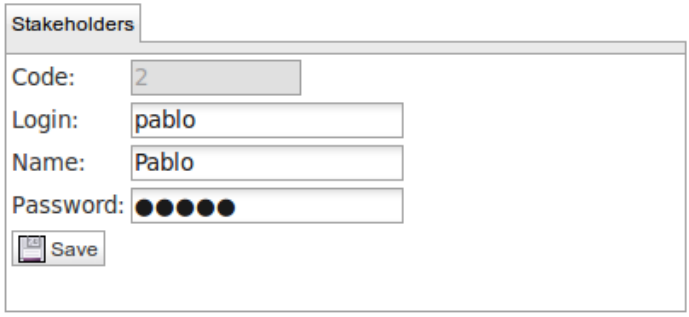


Figura 95. Cadastro de stakeholders

4.8.3 REGISTRO DE RASTREABILIDADE MANUAL

Como visto anteriormente, o agente *RequirementEngineer* é responsável pela leitura e interpretação dos artefatos do sistema. Dentre suas atribuições, está a reconstituição da rastreabilidade vertical, ou seja, o relacionamento entre elementos de um mesmo modelo. Esta rastreabilidade é lida automaticamente do modelo. No entanto, o agente *RequirementEngineer* não identifica relações de rastreabilidade horizontal. Estas podem ser registradas de forma manual, por meio do formulário exibido a seguir, ou também de forma semi-automática, como será demonstrado adiante. No formulário de registro manual, o *stakeholder* poderá registrar relacionamentos entre artefatos, entre membros e entre artefatos e membros. As relações são registradas no banco de dados e perduram mesmo entre sucessivas trocas de versões.

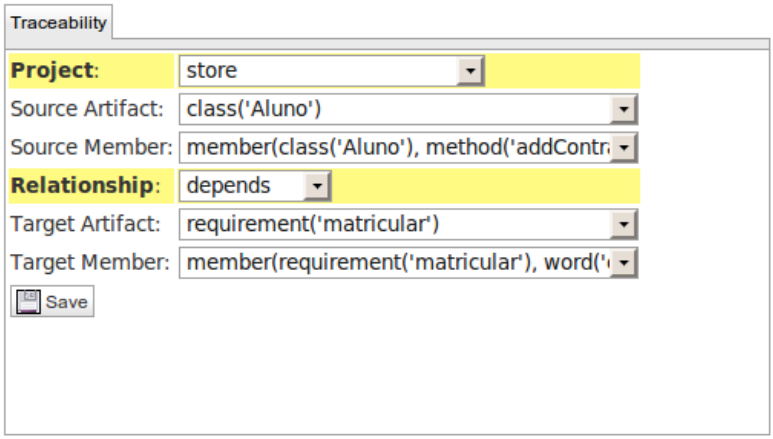
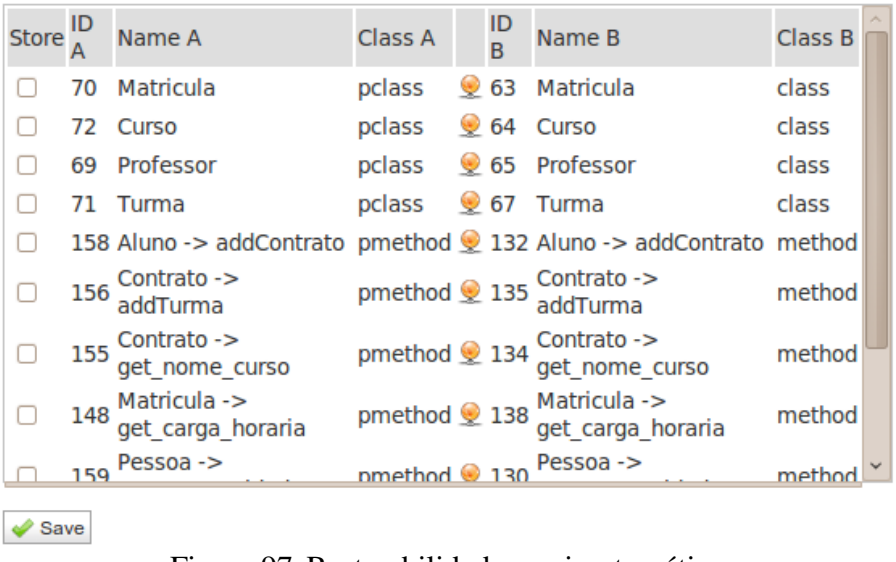


Figura 96. Registro de rastreabilidade manual

4.8.4 REGISTRO DE RASTREABILIDADE SEMI-AUTOMÁTICA

A rastreabilidade semi-automática procura detectar a rastreabilidade entre elementos de diferentes modelos (rastreabilidade horizontal) por meio da similaridade entre os termos. Desta forma, foram elaboradas algumas heurísticas para aproximar artefatos e membros de modelos diferentes. A ferramenta procura comparar de forma *case-insensitive* nomes de artefatos que provavelmente estejam relacionados, como classes de *design* UML e classes de implementação (código-fonte). A ferramenta também procura comparar membros de artefatos de *design* como métodos ou atributos com membros de artefatos de implementação. No caso dos membros de artefatos, não basta somente o membro ter o mesmo nome, mas o artefato pai também deve ter identificação igual.

A imagem a seguir demonstra o resultado de uma rodada da rastreabilidade semi-automática gerada pela aplicação proposta. No início constam várias classes de implementação (*pclass*) para as quais foram encontradas classes de *design* correspondente, como é o caso das classes *Matricula*, *Curso*, *Professor* e *Turma*. Após, constam alguns métodos de implementação, para os quais foram encontrados correspondentes no modelo de *design*, como o método *addContrato* da classe *Aluno*, o método *addTurma* da classe *Contrato*, o método *get_nome_curso* da classe *Contrato* e o método *get_carga_horaria* da classe *Matricula*, dentre outros. Após validar os relacionamentos propostos pelo sistema, o usuário poderá registrá-los no sistema.



Store	ID A	Name A	Class A		ID B	Name B	Class B
☐	70	Matricula	pclass		63	Matricula	class
☐	72	Curso	pclass		64	Curso	class
☐	69	Professor	pclass		65	Professor	class
☐	71	Turma	pclass		67	Turma	class
☐	158	Aluno -> addContrato	pmethod		132	Aluno -> addContrato	method
☐	156	Contrato -> addTurma	pmethod		135	Contrato -> addTurma	method
☐	155	Contrato -> get_nome_curso	pmethod		134	Contrato -> get_nome_curso	method
☐	148	Matricula -> get_carga_horaria	pmethod		138	Matricula -> get_carga_horaria	method
☐	159	Pessoa ->	pmethod		130	Pessoa ->	method

Save

Figura 97. Rastreabilidade semi-automática

4.8.5 GERENCIAMENTO DE MUDANÇAS

A listagem a seguir é utilizado para gerenciar as mudanças ocorridas no projeto. Neste, é possível filtrar pelo nome dos artefatos, pela versão e ainda somente por mudanças não classificadas. A listagem informa a data e o artefato modificados, o tipo de modificação e a mensagem registrada no momento do *commit* realizado pelo usuário.

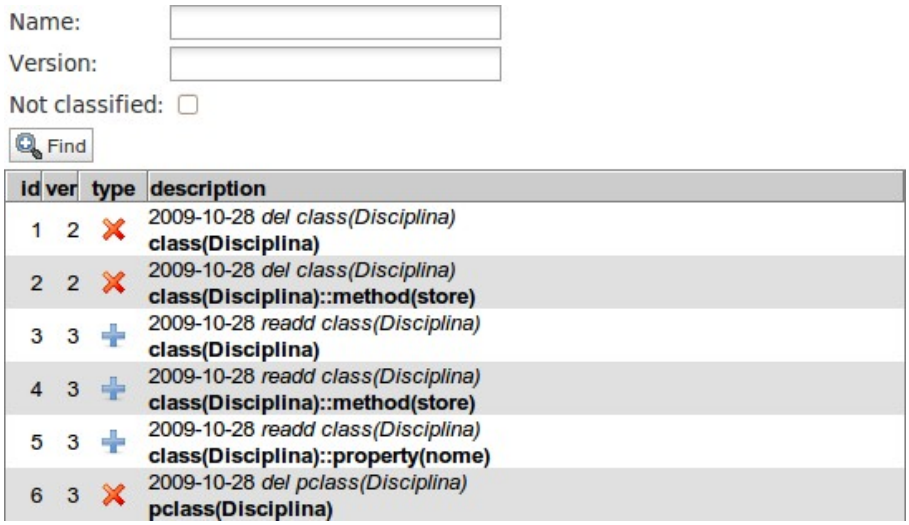


Figura 98. Gerenciamento de mudanças

4.8.6 GERENCIAMENTO DE IMPACTOS

A listagem a seguir é utilizada para listar os impactos ainda não resolvidos no sistema. Ela exibe, da mesma forma que a listagem de mudanças, a data e o artefato modificados, o tipo de modificação e a mensagem registrada no momento do *commit* realizado pelo usuário. A seguir são identificados os artefatos afetados pela mudança. O número na frente do artefato indica o nível de relacionamento que o artefato possui com o artefato alterado. Nível 1 significa que os artefatos são relacionados diretamente e nível 2 significa que o artefato relaciona-se com algum artefato de nível 1.

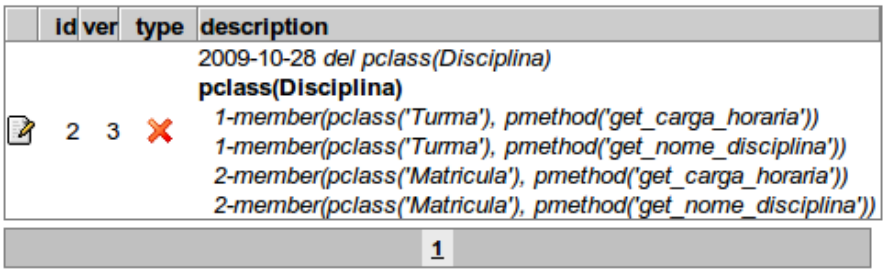


Figura 99. Gerenciamento de impactos

4.8.7 RELATÓRIO DE ARTEFATOS

O relatório de artefatos pode ser utilizado para listar todos os artefatos coletados pelo agente *RequirementEngineer* em uma determinada versão do sistema sendo monitorado. É possível realizar um filtro pelo nome do artefato e pela versão. São exibidas algumas informações como o identificador interno do artefato (*ID*), a versão do sistema, o nome do artefato e a chave do artefato, identificando se o mesmo é um requisito, um ator, um caso de uso, uma classe de *design*, uma classe de implementação, dentre outros.

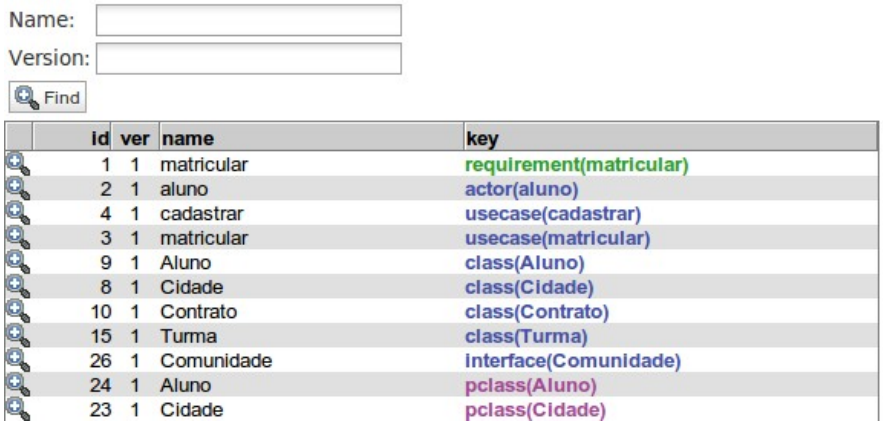


Figura 100. Relatório de artefatos

4.8.8 RELATÓRIO DE MEMBROS

No relatório a seguir, pode-se visualizar todos os membros de artefatos encontrados em uma determinada versão do sistema. Neste caso, podem ser vistos membros como palavras (*word*) de requisitos (*requirement*), métodos (*method*) de classes de design (*class*), dentre

outros.

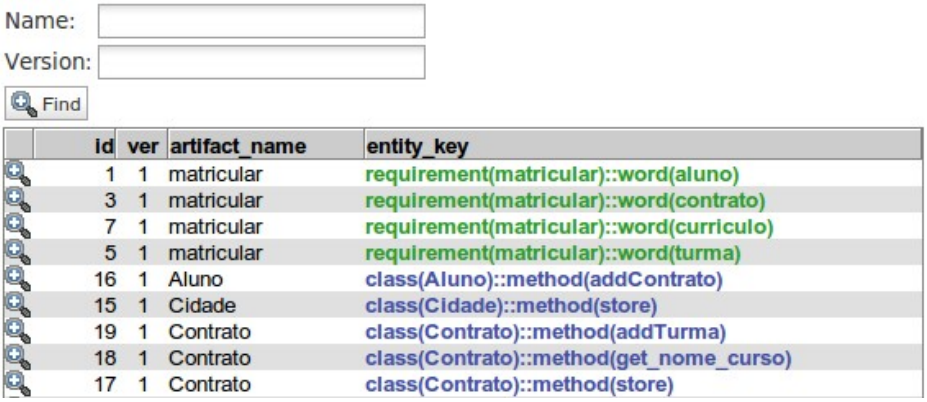


Figura 101. Relatório de membros de artefatos

4.8.9 RELATÓRIO DE RELACIONAMENTOS

No relatório a seguir, podem ser visualizadas as relações entre os artefatos e membros do sistema. Como visto anteriormente, as relações verticais são lidas automaticamente dos modelos e as horizontais podem ser registradas manualmente, bem como por um processo semi-automático. Neste exemplo, pode-se visualizar a relação de associação entre o ator *aluno* e o *use case matricular*, a relação de composição entre as classes de *design Aluno* e *Contrato*, dentre outros.

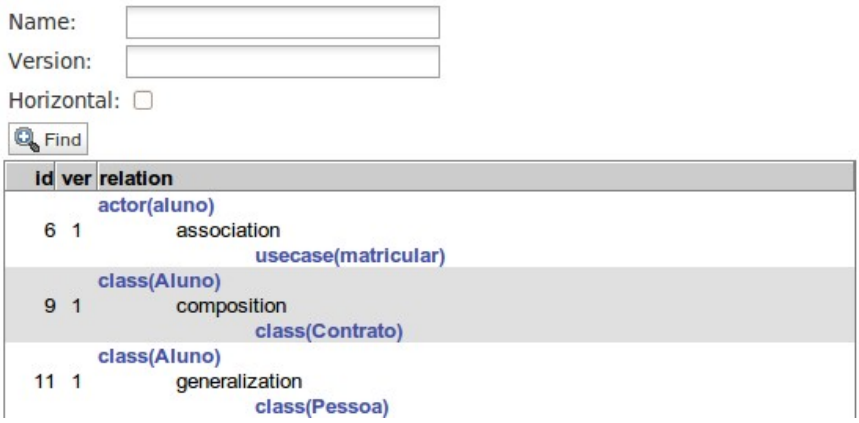


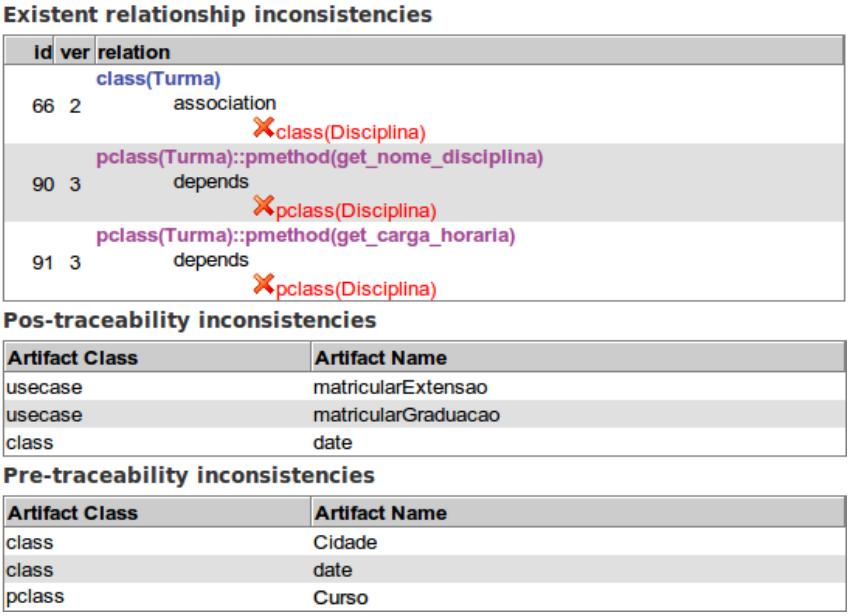
Figura 102. Relacionamentos entre os artefatos e membros do sistema

4.8.10 RELATÓRIO DE INCONSISTÊNCIAS

Da mesma forma em que o *stakeholder* é notificado sobre as inconsistências da

rastreabilidade após classificação de mudanças, o mesmo poderá consultar o sistema por meio de sua interface e solicitar um relatório de inconsistências detectadas na versão atual. Como já dito anteriormente, as inconsistências se dividem em três partes: relacionamentos entre artefatos excluídos, elementos sem pós-rastreabilidade e elementos sem pré-rastreabilidade.

A figura a seguir procura demonstrar o relatório de inconsistências. A primeira parte demonstra os relacionamentos entre artefatos já excluídos. Neste caso, pode-se citar o relacionamento de associação entre as classes *Turma* e *Disciplina*. Este relacionamento foi listado pois a classe *Disciplina* encontra-se excluída na atual versão do sistema. A segunda parte da figura demonstra as inconsistências de pós-rastreabilidade. Neste caso, foram relacionados artefatos como o *use case MatricularGraduacao* e a classe *date* em virtude destes não terem relação alguma com artefatos de fases subsequêntes do projeto. Por fim, são listadas as inconsistências de pré-rastreabilidade. Neste caso, foram relacionados a classe de design *Cidade* e a classe de projeto *Curso*, por exemplo, por não possuírem relação alguma com artefatos de fases anteriores do ciclo de desenvolvimento.



Existent relationship inconsistencies

id	ver	relation
		class(Turma)
66	2	association
		✘class(Disciplina)
		pclass(Turma)::pmethod(get_nome_disciplina)
90	3	depends
		✘pclass(Disciplina)
		pclass(Turma)::pmethod(get_carga_horaria)
91	3	depends
		✘pclass(Disciplina)

Pos-traceability inconsistencies

Artifact Class	Artifact Name
usecase	matricularExtensao
usecase	matricularGraduacao
class	date

Pre-traceability inconsistencies

Artifact Class	Artifact Name
class	Cidade
class	date
pclass	Curso

Figura 103. Relatório de inconsistências

4.8.11 GRAFO DE RASTREABILIDADE

A informação da rastreabilidade, tanto vertical quanto horizontal, permite que a ferramenta proposta possa gerar um grafo de rastreabilidade. Este grafo de rastreabilidade

representa os artefatos e os membros de artefatos do sistema e suas interligações. No grafo de rastreabilidade, os nodos representam os artefatos, que são elementos com granularidade *coarse-grained*, como uma classe de design ou um caso de uso e também os membros de artefatos, que são elementos *fine-grained*, como um método ou uma propriedade de uma classe. Já as arestas representam os links de rastreabilidade existente entre os elementos.

Os elementos são renderizados com cores diferentes e formatos diferentes. Elementos em forma de elipse representam artefatos e elementos em forma de retângulo representam membros de artefatos. Elementos em verde representam artefatos ou membros pertencentes à etapa de levantamento de requisitos e representam requisitos textuais, bem como termos relevantes dentro destes requisitos. Elementos em violeta representam artefatos ou membros pertencentes à etapa de projeto, como classes de *design* UML, bem como seus métodos e atributos. Por último, elementos em azul representam artefatos ou membros de implementação, como classes encontradas no código-fonte da aplicação, bem como seus métodos e atributos. O grafo de rastreabilidade é gerado pela aplicação por meio do uso biblioteca *open-source* Graphviz (2010), desenvolvida pelos laboratórios da AT&T.

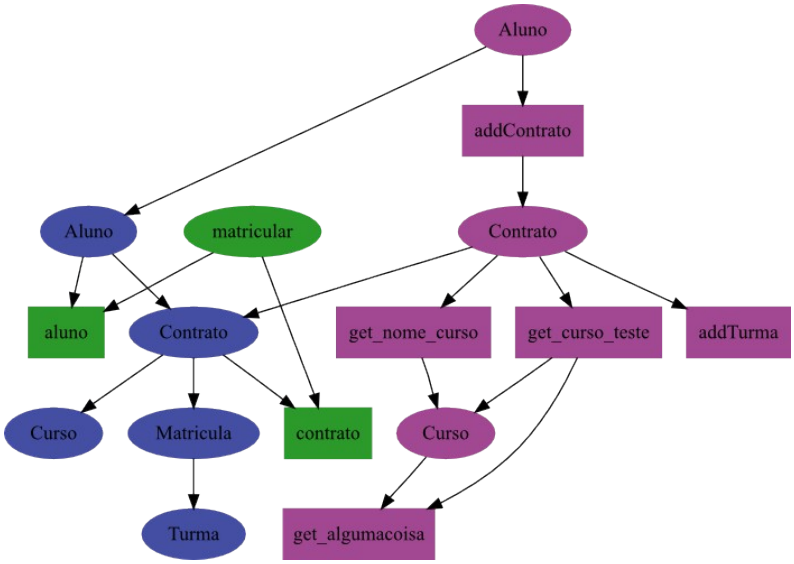


Figura 104. Grafo de rastreabilidade

4.8.12 DASHBOARD

A figura a seguir exibe o *dashboard* do sistema. O *dashboard* oferece um panorama gerencial sobre o estado dos principais elementos sendo monitorados. Neste sentido, o

dashboard permite visualizar a quantidade total de mudanças detectadas no sistema, a quantidade de mudanças aiinda não classificadas pelos *stakeholders*, a quantidade geral de impactos gerados até o momento, bem como a quantidade de impactos ainda não resolvidos. O *dashboard* permite interatividade, ou seja, no momento em que o usuário clicar sobre alguma das estatísticas, será direcionado à tela correspondente. Como exemplo, pode-se citar que ao clicar sobre o botão *Classified Changes*, que que consta o número de mudanças não classificadas, o mesmo será direcionado à tela que permite a classificação das mudanças ainda pendentes.

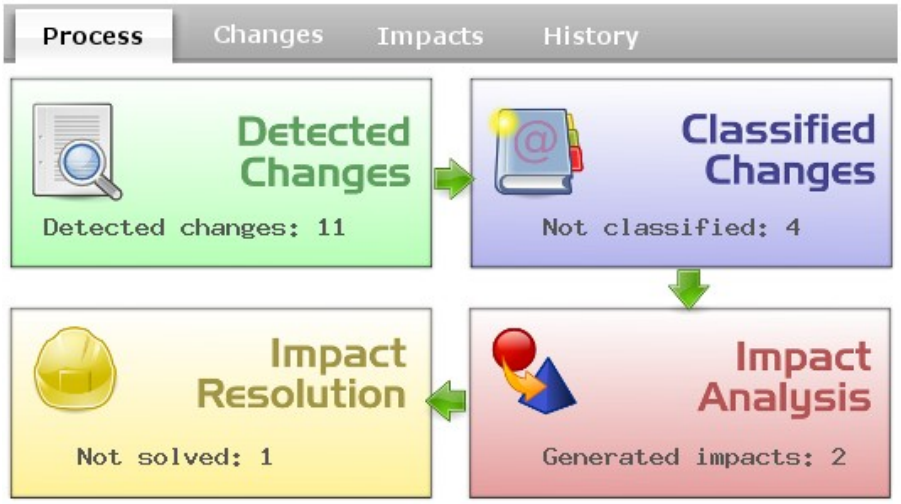


Figura 105. Dashboard da aplicação

4.8.13 ESTATÍSTICAS DE MUDANÇAS

O gráfico a seguir exibe as estatísticas de mudanças ocorridas no projeto. As estatísticas são geradas a partir das métricas identificadas por Nurmuliani, Zowghi e Fowell (2004). Neste sentido, é apresentado o número de mudanças por razão (falta de requisitos, melhoria funcional, correção, estratégia de produto), o número de mudanças ocorridas por origem (relatório de análise, gerência do projeto, observações de *marketing*), o número de mudanças ocorridas por tipo (adição, exclusão) e o número de mudanças ocorridas por *stakeholder*.

Figura 106. Estatísticas de mudanças

4.8.14 ESTATÍSTICAS DE IMPACTOS

Da mesma maneira que são exibidas as estatísticas de mudanças, as estatísticas de impacto também são geradas a partir das métricas identificadas por Nurmuliani, Zowghi e Fowell (2004). Como nem toda mudança irá gerar impactos, o gráfico a seguir exibe as estatísticas de impactos ocorridas no projeto.

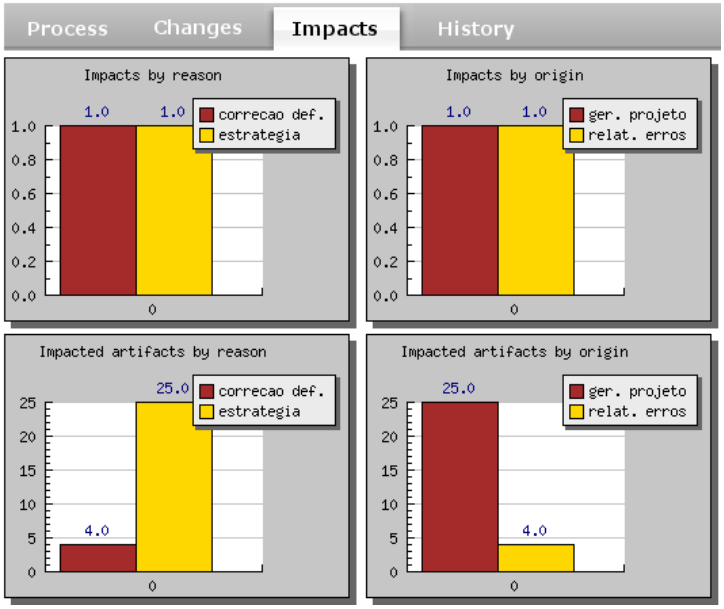


Figura 107. Estatísticas de impactos

Cada impacto ocorrido pode afetar um diferente número de artefatos. Neste sentido, é apresentado o número de impactos por razão (falta de requisitos, melhoria funcional, correção, estratégia de produto), o número de impactos ocorridos por origem (relatório de análise, gerência do projeto, observações de *marketing*), o número de artefatos impactados por razão e o número de artefatos impactados por origem.

4.8.15 HISTÓRICO DO PROJETO

O gráfico a seguir procura demonstra o histórico do projeto. O histórico procura demonstrar a evolução das versões do sistema ao longo do tempo. Neste sentido são exibidos o número total de mudanças geradas, o número de mudanças ainda não classificadas, o número total de impactos e o número de impactos ainda não resolvidos, para cada versão do sistema.

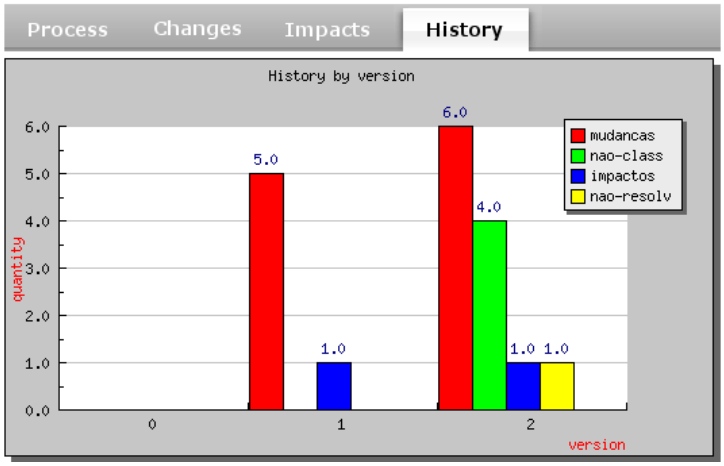


Figura 108. Histórico do projeto

4.9 CENÁRIOS

Para realizar a validação da aplicação antes da realização de seu estudo de caso, foi elaborada uma série de cenários, contendo algumas das principais situações relativas à gerência da mudança e análise de impacto.É importante ressaltar que o protótipo desenvolvido não foi testado somente com estes cenários, mas também com um número incontável de variação destes. Neste sentido, procurou-se observar o comportamento da aplicação em

situações envolvendo artefatos de diferentes fases do processo de desenvolvimento e diferentes níveis de granularidade. A exposição a seguir procura exibir apenas um pequeno exemplo de cada situação.

4.9.1 INCONSISTÊNCIA NA RASTREABILIDADE

O primeiro cenário criado procura demonstrar uma possível inconsistência de rastreabilidade. Inconsistências de rastreabilidade podem ser basicamente de três tipos: inconsistência de rastreabilidade, onde um relacionamento já existente em versão anterior da aplicação é quebrado pela exclusão de um dos elementos relacionados, inconsistências de pós-rastreabilidade, onde um elemento não possui relacionamento com nenhum outro elemento de etapas posteriores do ciclo de desenvolvimento e inconsistência de pré-rastreabilidade, onde um elemento não posusi relacionamento com outro elemento de etapas anteriores.

Neste cenário a seguir, o *stakeholder* cria dois requisitos textuais: *Realizar Matrícula* e *Trancar Matrícula*, contendo as descrições destas atividades. Posteriormente desenvolve um modelo de casos de uso, no qual cria dois casos de uso: *Matricular* e *Trancar*. Após, estabelece um relacionamento horizontal entre o requisito *Realizar Matricula* e o caso de uso *Matricular* de forma manual. É importante lembrar que a ferramenta oferece suporte semi-automático para detectar *links* de rastreabilidade quando há similaridade de nomenclaturas. Neste caso em específico, o requisito *Trancar Matricula* não encontra relacionamento com elemento algum de etapas subsequêntes. Da mesma forma, o caso de uso *Trancar* encontra-se isolado, não implementando nenhum requisito já mapeado. Ao representar este cenário, o protótipo deverá indicar uma inconsistência de pós-rastreabilidade para o requisito *Realizar Matricula* e uma de pré-rastreabilidade para o caso de uso *Trancar*. Esta inconsistência deverá ser relatada logo após a primeira classificação de mudanças que contiver este cenário.

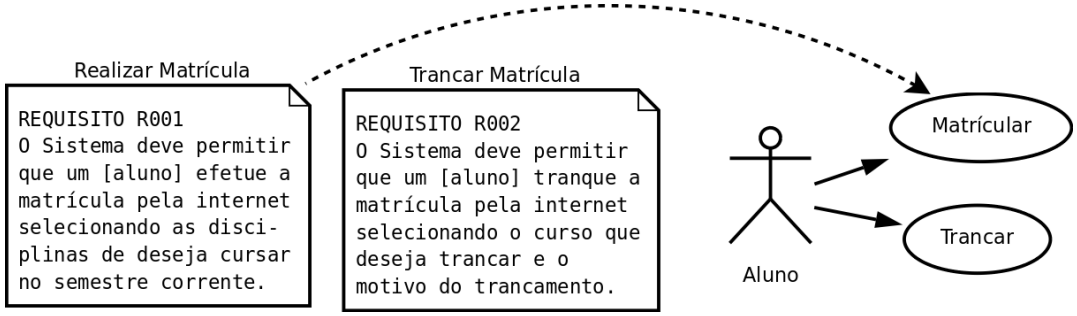


Figura 109. Cenário 1 - Inconsistência na rasterabilidade

4.9.2 IMPACTO CAUSADO POR EXCLUSÃO

O segundo cenário criado procura demonstrar uma exclusão de artefato. Neste caso, ilustrado pela figura a seguir, após criar o requisito textual *Realizar Matricular* e o caso de uso *Matricular*, o *stakeholder* cria a classe de *design Aluno*, com seus métodos *trancar* e *matricular* e após, implementa a classe *Aluno* com seu método *matricular*. Como o relacionamento entre o requisito *Realizar Matricula* e o caso de uso *Matricular* já estava definido, é estabelecido o relacionamento entre o caso de uso *Matricular* e o método *matricular* da classe de *design Aluno* e após, entre o método *matricular* da classe de *design Aluno* com o método *matricular* da classe de implementação *Aluno*.

Após este cenário definido, é excluído o caso de uso *Matricular*. Neste caso, os elementos que dependem de sua definição devem ser descobertos pela análise de impacto e seus respectivos *stakeholders* notificados. Neste exemplo específico, devem ser identificados: o ator *Aluno*, que possui associação direta com este caso de uso, o método *matricular* da classe de *design Aluno*, que depende diretamente do caso excluído e o método *matricular* da classe implementada *Aluno,* que depende do método matricular da classe de *design Aluno*. Para alertar os stakeholders, o agente *ProcessManager* irá obter o responsável pelo modelo de *design* com base no usuário que realizou a última submissão deste modelo para o repositório centralizado. Já em relação ao código-fonte, o agente irá obter o responsável com base nos cabeçalhos de documentação *javadoc*, que contém os responsáveis por cada classe e cada método do código-fonte, mantando um nível de granularidade apropriado.

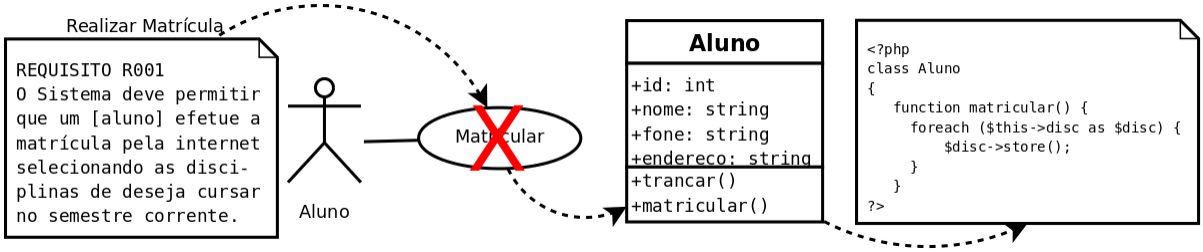


Figura 110. Cenário 2 – Exclusão de artefato

4.9.3 IMPACTO CAUSADO POR MODIFICAÇÃO

O terceiro cenário elaborado compreende a alteração de estruturas de implementação

(código-fonte), uma situação que envolve somente elementos de um mesmo modelo, classificada como rastreabilidade vertical. Apesar de parecer simples em um primeiro momento, descreve uma situação que ocorre com bastante frequência durante o desenvolvimento de projetos de sistemas e envolve situações complexas do ponto de vista da engenharia de software. Este cenário descreve um dos vários tipos de dependência entre estruturas de programação existentes. Sabe-se que a orientação a objetos permite um grande número de relacionamentos de dependência como: associação, agregação, composição, herança, dentre outros. É importante destacar que todos estes são interpretados pelo agente *RequirementEngineer* de forma totalmente automática para as linguagens PHP e Java com base em um conjunto de expressões regulares contendo as expressões de cada linguagem.

Neste exemplo em específico, foram criadas três classes de implementação: *Aluno*, *Matricula* e *Turma*. Cada uma destas classes implementa o *design pattern Active Record* e trata do registro de objetos na base de dados. A classe *Aluno* possui o método *Matricular*, que por sua vez realiza o registro de uma matrícula e depende da classe *Matricula*, mais especificamente de seu método *store*. Já o método store da classe *Matricula* depende da classe *Turma*, uma vez que realiza chamada de seu método *ReservaVaga*. Neste caso, se estabelece uma relação de dependência com base na associação. O agente *RequirementEngineer* não monitora somente inclusões e exclusões, mas também alterações em classes e métodos, por meio da comparação de seu conteúdo entre sucessivas versões. Desta forma, uma alteração no código-fonte do método *ReservaVaga* da classe *Turma*, deve resultar em impactos no método *store* da classe *Matricula* e também no método *Matricular* da classe *Aluno*. Cabe ressaltar que os *stakeholders* responsáveis por cada elemento impactado serão obtidos por meio da interpretação dos cabeçalhos *javadoc* de cada método, que contém o *login* do *stakeholder* responsável por cada uma das estruturas.

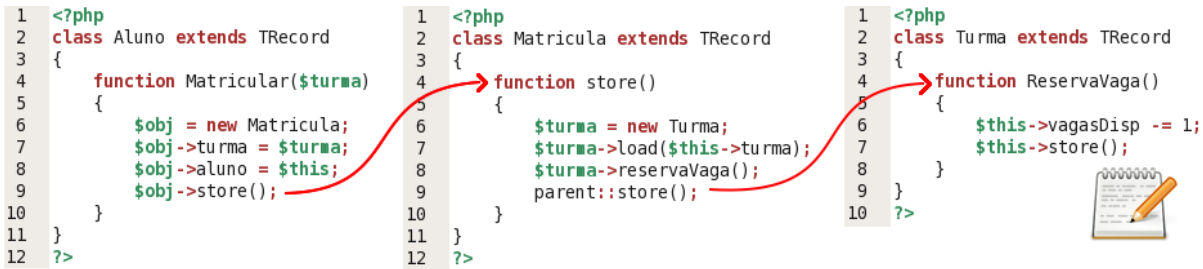


Figura 111. Cenário 3 – Modificação de artefato

5 ANÁLISE COMPARATIVA ENTRE OS TRABALHOS

5.1 COMPARAÇÃO ENTRE MODELOS

Neste capítulo, será realizada a comparação entre os modelos estudados no capítulo 3 e o modelo proposto no presente trabalho.

5.1.1 CRITÉRIOS DE COMPARAÇÃO

Com base nos critérios apontados por Hoffmann et al. (2004), Lang e Duggan (2001) e Cleland-Huang et al. (2007), definiu-se um grupo de 12 critérios a serem utilizados a fim de se comparar os modelos avaliados frente ao modelo proposto. A seguir constam os critérios, bem como uma explicação detalhada de cada um deles.

Tabela 12 -Critérios de comparação entre modelos

Critério	Descrição
A	Suporta relação com documentos externos: Permitir relacionar artefatos ou requisitos com documentos externos, tais como textos ou planilhas;
B	Suporta requisitos textuais: Suporta a representação de especificação textual de requisitos;
C	Suporta documentos UML: Suporta artefatos de design, como diagrama de classes ou use cases;
D	Suporta código-fonte: Suporta a representação de arquivos contendo código-fonte, classes e métodos;
E	Suporta rastreabilidade vertical: Permite rastreabilidade entre requisitos de um mesmo modelo;
F	Suporta rastreabilidade horizontal: Permite rastreabilidade entre requisitos de modelos diferentes;
G	Suporta diferentes projetos: Permite representar requisitos de projetos distintos;
H	Suporta configuração de granularidade: Permite configurar o nível de granularidade da rastreabilidade;
I	Suporta representação de mudanças: Permite representar as mudanças ocorridas;
J	Suporta representação de impactos: Permite representar os impactos causados pelas mudanças;
K	Suporta versionamento (*baseline*): Permite representar as versões dos requisitos;
L	Permite relacionar *stakeholders* e artefatos: Permite relacionar artefatos com *stakeholders*;

A partir dos critérios selecionados para comparação, na tabela a seguir são definidos os modelos a serem comparados.

Tabela 13 -Modelos a serem comparados

Método	Descrição
M1	Modelo de Ramesh
M2	Modelo de Lang e Duggan
M3	Modelo de Letelier
M4	Modelo de Briand, Labiche e O'Sullivan
M5	Modelo de De Grande e Martins
M6	Modelo de Kassab, Ormandjieva e Daneva
M7	Modelo de Ghazi
M8	Modelo de Leal, Figueiredo e De Souza
MP	Modelo Proposto

5.1.2 TABELA COMPARATIVA

Com base nos critérios e modelos selecionados, é estabelecida uma matriz contendo a comparação dos modelos conforme os critérios selecionados, como pode ser visto na tabela a seguir. Cada ponto de interseção representa um critério atendido. Nas linhas tem-se os modelos (M1-MP) e nas colunas os critérios de comparação (A-L).

Tabela 14 -Comparação entre modelos

	A	B	C	D	E	F	G	H	I	J	K	L
M1	x	x			x				x			x
M2		x	x		x	x			x			x
M3		x	x		x	x		x				x
M4			x		x			x	x	x	x	
M5	x	x			x		x		x	x	x	x
M6	x	x	x		x	x	x	x				x
M7	x	x	x	x	x	x	x					x
M8		x	x		x	x					x	
MP	x	x	x	x	x	x	x	x	x	x	x	x

5.1.3 ANÁLISE QUALITATIVA

Por meio da matriz de comparação entre os modelos estudados e o modelo proposto, pode-se perceber que não existe critério algum que seja atendido apenas por um dos modelos. Entretanto, alguns modelos oferecem uma cobertura maior do que outros.

Dentre os modelos estudados, podemos perceber que somente dois modelos, o modelo de Ghazi (2008) e o modelo proposto, estão preparados para representar com uma maior abrangência artefatos de software de fases distintas do ciclo de desenvolvimento, que são: requisitos textuais, modelos de design e artefatos de código-fonte.

O critério "suportar a rastreabilidade vertical" foi o único critério atendido por todos os modelos estudados. Já o critério "suportar a rastreabilidade horizontal" foi atendido por seis dos modelos estudados. É importante notar que, para suportar a rastreabilidade horizontal, é necessário suportar a representação de artefatos de pelo menos duas fases distintas do desenvolvimento de software.

Um critério essencial adotado pelo presente trabalho, "a configuração da granularidade", que permite relacionamentos de rastreabilidade *coarse-grained* bem como *fine-grained* é atendida por quatro dos modelos estudados.

Cinco modelos permitem a representação de mudanças ocorridas, mas este número cai para apenas três modelos quando se observar a representação dos impactos sofridos.

Quatro modelos suportam versionamento dos artefatos, existe uma aproximação entre estes modelos em relação aos modelos que suportam análise de impacto, mas o número analisado é inexpressivo para provar algo.

Sete modelos suportam a relação entre artefatos e stakeholders, destes cinco suportam a relação entre artefatos e documentos externos.

Nenhum dos modelos atendeu menos de cinco critérios. Nenhum dos modelos, com exceção do modelo proposto, atendeu mais de oito critérios. A seguir, um gráfico indicando a quantidade de critérios atendida por cada modelo.

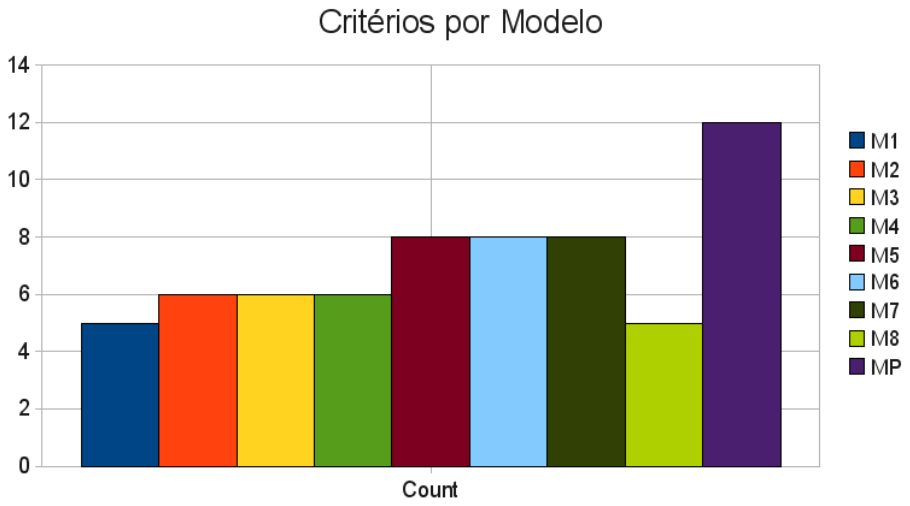


Figura 112. Critérios atendidos por modelo

5.2 COMPARAÇÃO ENTRE MÉTODOS

Neste capítulo, será realizada a comparação entre os métodos estudados no capítulo 3 e o método proposto no presente trabalho.

5.2.1 CRITÉRIOS DE COMPARAÇÃO

Com base nos critérios apontados por Hoffmann et al. (2004), Lang e Duggan (2001), Cleland-Huang et al. (2007) e Fasano (2007), definiu-se um grupo de 20 critérios a serem utilizados a fim de se comparar os métodos avaliados frente ao método proposto. A seguir constam os critérios, bem como uma explicação detalhada de cada um deles.

Tabela 15 -Critérios de comparação entre métodos

Critério	Descrição
A	Suporta o gerenciamento do projeto, permitindo configurar as suas características, como linguagem, localização, *stakeholders*;
B	Suporta o gerenciamento da mudança, permitindo a detecção das alterações e classificação das mesmas;
C	Suporta rastreabilidade entre artefatos de um projeto (vertical ou horizontal);
D	Suporta a realização de análise de impacto;

Critério	Descrição
E	Suporta notificação de eventos aos *stakeholders* quando ocorrem mudanças;
F	Suporta a propagação de eventos, permitindo a notificação recursiva de eventos aos *stakeholders* conforme a rastreabilidade;
G	Suporta utilização on-line pela *web*;
H	Suporta integração com outras ferramentas como editores de código ou repositórios;
I	Suporta rastreabilidade *fine-grained*;
J	Suporta versionamento dos artefatos;
K	Suporta representação de requisitos textuais;
L	Suporta representação de modelos UML;
M	Suporta representação de código-fonte;
N	Permite recuperação da rastreabilidade vertical, ou seja a rastreabilidade entre artefatos de um mesmo modelo;
O	Permite recuperação da rastreabilidade horizontal, ou seja, a rastreabilidade entre artefatos de diferentes modelos;
P	Permite *parsing* de estruturas complexas (requisitos textuais, UML, código-fonte) e a consequente compreensão de suas estruturas internas;
Q	Oferece gerenciamento pró-ativo, não necessitando que o usuário acione a aplicação para obter os resultados;
R	Suporta acompanhamento do processo, permitindo que o usuário acompanhe cada fase e possa tomar decisões como aprovar uma mudança ou não (*workflow*);
S	Oferece estatísticas gerenciais como gráficos com indicadores que demonstram as métricas sobre o processo;
T	Suporta adaptabilidade e extensibilidade, permitindo que o potencial da ferramenta seja expandido para novas aplicações sem que a mesma precise ser reescrita.

A partir dos critérios selecionados para comparação, são definidos os métodos a serem comparados conforme tabela a seguir.

Tabela 16 -Métodos a serem comparados

Método	Descrição
M1	Método ADAMS
M2	Método Event Based Traceability
M3	Método SIGERAR
M4	Método iACMTool
M5	Método Chianti
M6	Método de Leal, Figueiredo e De Souza
M7	Método RM-Tool
M8	Método ONTrace
MP	Método Proposto

5.2.2 TABELA COMPARATIVA

A partir dos critérios e métodos selecionados para comparação, é definida uma matriz contendo a comparação dos métodos conforme os critérios selecionados, como pode ser visto na figura a seguir. Cada ponto de interseção representa um critério atendido. Nas linhas tem-se os critérios (A-T) e nas colunas os métodos (M1-MP).

Tabela 17 -Comparação entre métodos

	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T
M1	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x				
M2		x	x	x	x	x	x	x			x					x	x	x		
M3	x	x	x	x	x	x	x			x	x	x						x		
M4		x	x	x	x	x		x		x		x		x		x		x	x	
M5		x	x	x		x		x	x				x	x		x			x	
M6			x	x			x	x		x	x	x		x	x	x				
M7	x	x	x		x		x	x		x	x							x	x	
M8			x					x	x			x		x	x	x				
MP	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x

5.2.3 ANÁLISE QUALITATIVA

Dentre os métodos estudados, apenas quatro suportam a configuração do projeto. Sete métodos suportam a gerência da mudança, ou seja, apenas dois métodos: método de Leal, Figueiredo e De Souza (2008) e método ONTrace não possuem este recurso. Sete métodos também suportam a análise de impacto, sendo que as exceções são os métodos RM-Tool e ONTrace.

Todos métodos estudados suportam a rastreabilidade entre os artefatos de software, seja vertical ou horizontal. Seis métodos suportam notificação de eventos, sendo que um número igual à este suporta a propagação de eventos sobre a informação de rastreabilidade. Quando analisados somente métodos que possuem estes dois recursos ao mesmo tempo, tem-se cinco métodos que atendem aos critérios simultaneamente.

Seis métodos suportam a utilização pela web. Apenas um não suporta integração com outras ferramentas. Apenas quatro métodos suportam a rastreabilidade *fine-grained* e seis

suportam versionamento.

Apenas dois métodos possuem abrangência maior e suportam todos tipos artefatos propostos simultaneamente: requisitos textuais, modelos de design UML e código-fonte, que são o método ADAMS e o método proposto. Coincidentemente, dentre todos métodos estudados, o método ADAMS é o que mais se aproxima do método proposto em termos de cobertura, ou seja, pela quantidade de critérios atendidos.

Dentre os métodos estudados, seis deles suporta a recuperação da rastreabilidade vertical. Se analisarmos destes, quantos suportam a recuperação da rastreabilidade horizontal, este número diminui para quatro métodos. Os métodos que não possuem a recuperação da rastreabilidade horizontal são justamente aqueles que suportam a representação de artefatos de apenas uma fase do ciclo de desenvolvimento.

Apenas dois métodos: SIGERAR e RM-Tool não suportam o *parsing* de estruturas complexas e um número muito reduzido de trabalhos, apenas Event Based Traceability e o trabalho proposto suportam o gerenciamento pró-ativo.

Sete trabalhos oferecem algum tipo de indicador estatístico para informações gerenciais, mas apenas o trabalho proposto atende ao último critério, que é de permitir extensibilidade e adaptabilidade sem a necessidade de reescrever o software e também sem precisar conhecer profundamente suas estruturas internas.

Dos trabalhos estudados, o que atendeu a menor quantidade de critérios foi o método ONTrace, com sete critérios atendidos. O trabalho que atendeu o maior número de critérios após o trabalho estudado foi o método ADAMS, com dezesseis critérios atendidos. A figura a seguir procura demonstrar a quantidade de critérios atendida por cada métodos estudado.

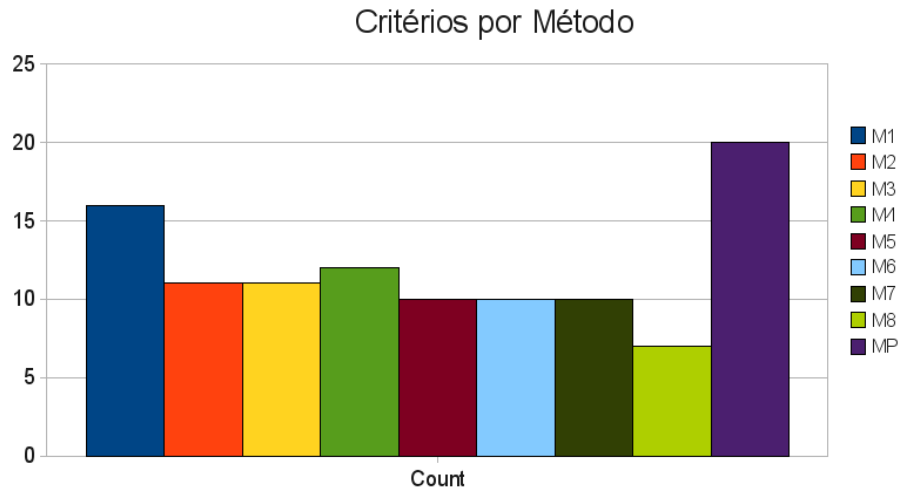


Figura 113. Critérios atendidos por método

6 ESTUDO DE CASO

Neste capítulo será apresentado com se deu a validação da aplicação proposta por meio de um estudo de caso que ocorreu por meio da implantação do sistema em uma organização externa.

Antes da realização deste estudo de caso, ocorreu uma espécie de validação interna, durante o seu próprio estágio de desenvolvimento. Para tal, foi preparado um sistema fictício com requisitos, classes de *design* e classes de implementação com base nos cenários pré-determinados para que a aplicação proposta pudesse monitorar este sistema e gerenciar as mudanças e impactos nele ocorridas. É importante ressaltar que a validação interna ocorreu como forma de prova dos conceitos desenvolvidos e não trabalhou com uma massa de dados muito grande, ou seja, as análises foram qualitativas e não quantitativas.

A validação real da aplicação ocorreu após o desenvolvimento da mesma em uma organização externa. A organização selecionada trata-se de uma instituição de ensino superior privada, na qual está sendo desenvolvido um novo sistema para controle administrativo da mesma (ERP). O desenvolvimento deste sistema administrativo iniciou abril de 2008 e tem prazo previsto para término em abril de 2010. A validação do protótipo ocorreu entre os dias 27/11/2009 e 06/01/2010. O desenvolvimento do sistema avaliado está em fase final. Desta forma, a maioria das modificações registradas ocorrem em decorrência de observações de falhas e sugestões de melhorias a partir de testes realizados pelos usuários do sistema, o que poderá ser notado nos resultados.

6.1 CENÁRIO

O ambiente de desenvolvimento da instituição referida utiliza-se somente de ferramentas livres tais como: Linux, Apache, PHP, PostgreSQL e Subversion facilitando questões de implantação e integração com a ferramenta proposta, uma vez que ela é desenvolvida nas mesmas tecnologias. Além disto, ressalta-se a utilização da orientação a objetos, de um framework de desenvolvimento e a separação de aspectos de desenvolvimento por meio da adoção do framework MVC. Tais práticas habilitam este ambiente como cenário para um estudo de caso.

Algumas dificuldades foram encontradas durante a implementação e execução do

estudo de caso. Inicialmente, o ambiente não possuia instalada a ferramenta Prolog no servidor, o que foi sanado posteriormente. Além disto, não havia o costume da equipe de desenvolvimento utilizar cabeçalhos padrão de documentação no estilo *javadoc* em todo código-fonte. Desta forma, em alguns casos em que não foi possível identificar o desenvolvedor responsável por um determinado método, optou-se por centralizar a notificação no gerente do projeto, para que este tome as devidas providências.

O tamanho do projeto também mostrou-se um problema. Ao todo o sistema possui mais de mil classes distribuídas em três camadas: modelo, visualização e controle. Como o número de classes era muito grande e o projeto estava em pleno desenvolvimento, inicialmente ocorriam muitas notificações de mudança. Este excesso de notificações, tornou-se ingerenciável, uma vez que a equipe dispunha de pouco tempo disponível para outras tarefas, como envolver-se na implantação do protótipo. Como forma de resolver paliativamente esta questão, optou-se por limitar a aplicação do sistema à camada de modelo (*Model*) do padrão MVC na aplicação. A camada de modelo possui as classes que concentram maior responsabilidade sobre os dados aos quais manipula, e são as mais utilizadas ao longo do sistema. Impactos ocorridos na camada de modelo tendem a ser mais desastrosos do que impactos ocorridos em outras camadas do sistema.

Por fim, a instituição escolhida para o estudo de caso não possui como prática manter a especificação de requisitos textuais ao longo do desenvolvimento do mesmo. A partir do momento em que o desenvolvimento do sistema atinge um nível de satisfatório em que os desenvolvedores já assimilaram boa parte dos requisitos elaborados, os esforços concentram-se na construção da aplicação em si e não mais na atualização da especificação textual. Desta forma, praticamente não ocorrem mudanças em requisitos textuais a partir de um certo ponto do desenvolvimento.

A instituição escolhida adota a utilização de um modelo de classes muito similar ao banco de dados. Isto se dá em virtude da adoção do padrão de projeto (*design pattern*) *Table Data Gateway*, no qual cada tabela é representadas por classes que controla os mecanismos de persistência à mesma. Além disto, percebeu-se que, em muitos casos, ocorriam modificações na base de dados sem que esta modificação fosse replicada no modelo de *design*. Apesar de não ser uma prática correta sob o ponto de vista da engenharia de software, ocorre com certa frequência e acaba ocasionando alguns problemas de inconsistência entre modelos. Como forma de resolver esta questão, propõ-se para trabalhos futuros que, além de monitorar

requisitos textuais, modelo de *design* e código-fonte, que possam também monitorar mudanças ocorridas em nível de banco de dados. Desta forma, com seriam monitoradas as modificações das estruturas das tabelas, com a adição ou remoção de colunas, o que pode trazer reflexos (impactos) ao modelo de *design*.

Por razões de privacidade, não serão expostos aqui nenhum dos artefatos de desenvolvimento da ferramenta avaliada, tais como requisitos textuais ou modelos de *design*.

6.2 RESULTADOS

Como já dito anteriormente, a validação do protótipo ocorreu entre os dias 27/11/2009 e 06/01/2010. Neste período ocorreram quinze releases de versões contendo correções e melhorias. A tabela a seguir procura demonstrar o tamanho do projeto. No total são 2071 elementos, contendo 1383 artefatos, 5804 membros e 12884 relacionamentos. Se considerarmos somente a camada de modelo da aplicação, este número se reduz para 1142 elementos, sendo estes 180 artefatos, 598 membros e 364 relacionamentos.

Tabela 18 -Dados sobre o tamanho do projeto

Total		**Camada Model**	
Artefatos	1383	Artefatos	180
Membros	5804	Membros	598
Relações	12884	Relações	364
Elementos	20071	Elementos	1142

Ao longo do período de monitoramento do projeto, ocorreram 51 mudanças gerenciadas pela ferramenta proposta. Estas mudanças geraram 9 impactos, sendo que estes impactos atingiram 13 elementos (artefatos ou membros). A seguir, as mudanças ocorridas serão analisadas separadamente.

O primeiro número a ser avaliado é a quantidade de mudanças ocorridas no período, conforme a classificação de razão e origem idealizada por Nurmuliani, Zowghi e Fowell (2004) e realizada pelo *stakeholder*. A imagem a seguir contém dois gráficos produzidos pela ferramenta proposta, que exibem o número de mudanças ocorridas classificadas por razão (gráfico da esquerda) e por origem (gráfico da direita).

Conforme pode ser visualizado no primeiro gráfico, ocorreram 18 mudanças motivadas por melhorias de design, 13 mudanças motivadas por requisitos faltantes, 11 mudanças motivadas por correções de defeitos, 6 motivadas por melhorias funcionais e 3 por especificação errada de requisitos. Esperava-se um volume maior de mudanças motivadas por correções de erros, tendo em vista que o projeto está em fase de validação e testes. Houve uma surpresa em relação à quantidade de mudanças em virtude de melhorias de *design*. Este número leva a crer que foi necessário melhorar algumas estruturas da aplicação para acomodar algumas mudanças inesperadas. O número de requisitos faltantes também ficou alto. Este número é facilmente explicado pois neste projeto, a etapa de análise teve de ser bastante superficial em razão do curto período disponível para o desenvolvimento do projeto. Desta forma, muitas demandas dos usuários foram sendo levantadas durante o próprio desenvolvimento do sistema.

Conforme pode ser visualizado no segundo gráfico, ocorreram 11 mudanças originadas em discussões técnicas, 9 originadas em relatórios de erros realizados pelo usuário, 8 originadas por discussões com o cliente, 8 originadas por análise do desenvolvedor, 6 originadas em revisões da especificação do software, 4 originadas durante revisões de *design*, 3 originadas pela equipe de gerenciamento do projeto e 2 originadas pela equipe de engenharia. O grande número de mudanças originadas em discussões técnicas explica por que houveram tantas mudanças em função de melhorias de design, como analisado anteriormente. A grande quantidade de mudanças originadas por relatórios de erros gerados pelo usuário explica-se facilmente uma vez que o usuário do sistema avaliado atualmente encontra-se em fase de testes do mesmo, reportando continuamente os erros ocorridos. Boa parte das mudanças ocorreram também em função de discussões pontuais com o cliente, envolvendo a redefinição de processos e também análise particular do desenvolvedor.

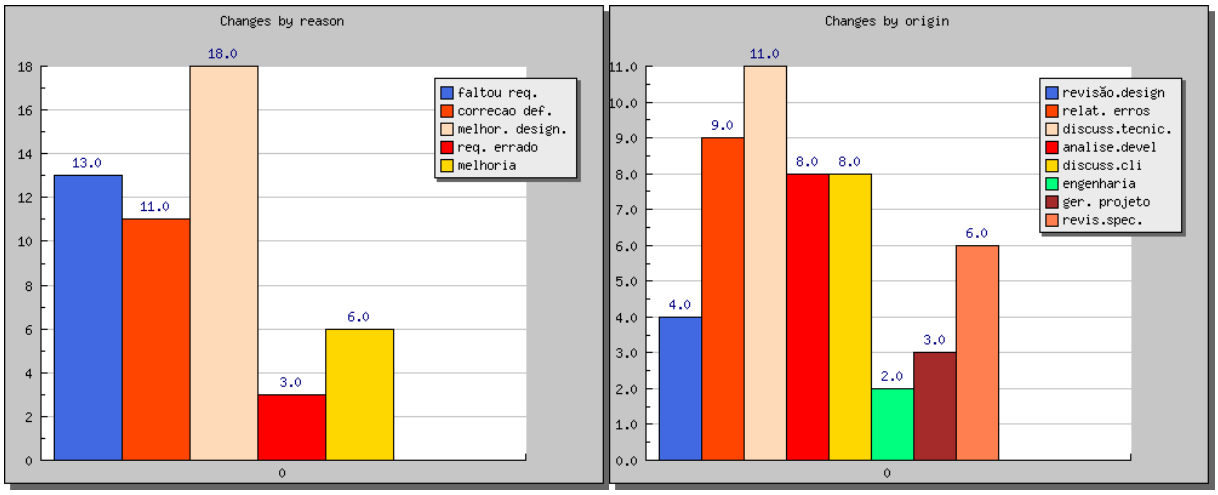


Figura 114. Estatísticas de mudanças

A tabela a seguir procura demonstrar a análise de correlação entras razões e as origens de mudanças. Como pode ser visto, de todas as mudanças que ocorreram, 8 foram motivadas por melhorias de *design*, ou seja, melhoria nas estruturas do projeto, originadas pela análise do desenvolvedor. Desta mesma origem (melhorias de *design*), ainda tem-se 4 originadas por discussões técnicas, 4 por revisões de *design,* 4 por revisões na especificação e 2 por solicitações da equipe de engenharia.

Ainta tem-se 8 mudanças motivadas pela falta de requisitos e originadas em discussões com o cliente. Desta mesma origem, outras 3 foram originadas pela equipe de gerenciamento do projeto e 2 pela revisão da especificação. Além destas, outras 7 mudanças foram motivadas por correções de defeito e originadas em discussões técnicas, sendo que desta mesma origem, outras 4 mudanças foram originadas por relatório de erros do usuário. Por fim, 4 melhorias funcionais foram originadas por revisões da especificações e 2 por relatórios de erros do usuário e 3 definições erradas de requisitos foram originadas por relatórios do usuário.

A análise combinada permite concluir superficialmente que dois grandes focos de trabalho durante o período monitorado pelo sistema foram: a melhoria do *design* do sistema, originada por discussões técnicas e análises individuais de membros da equipe e a correção de defeitos e melhorias funcionais, originadas principalmente por discussões com o cliente e relatórios de erro. Grande parte das melhorias de *design* foram realizadas para suportar melhorias funcionais.

Tabela 19 -Análise combinada de razões e origens de mudanças

Razão	Origem	Quantidade	Percentual
Melhoria de design	Analise de desenvolvedor	8	15,7 %
Requisito faltante	Discussao com o Cliente	8	15,7 %
Correcao de defeito	Discussao tecnica	7	13,7 %
Correcao de defeito	Relatorios de erros	4	7,9 %
Melhoria de design	Discussao tecnica	4	7,9 %
Melhoria de design	Revisao de design	4	7,9 %
Melhoria funcional	Revisao de Especificacao	4	7,9 %
Requisito faltante	Gerenciamendo do Projeto	3	5,8 %
Requisito Errado	Relatorios de erros	3	5,8 %
Melhoria de design	Chamado da Engenharia	2	3,9 %
Requisito faltante	Revisao de Especificacao	2	3,9 %
Melhoria funcional	Relatorios de erros	2	3,9 %

Nem toda mudança gera necessariamente um impacto no sistema. Os próximos gráficos procuram demonstrar os impactos gerados no sistema e quais foram as principais razões e origens que levaram à estes impactos. Neste sentido, como pode ser observado na figura a seguir, obteve-se coincidentemente um resultado homogêneo no qual o número de razões e origens causadoras dos impactos reduziu-se significativamente. Desta forma, cabe salientar que de 9 impactos gerados, 3 foram motivados pela falta de requisitos, 3 pela correção de defeitos e 3 por definições erradas de requisitos. Em relação às origens, 3 impactos foram gerados em discussões técnicas, 3 por relatórios de erros e 3 por relatórios de erros e 3 por discussões com o cliente.

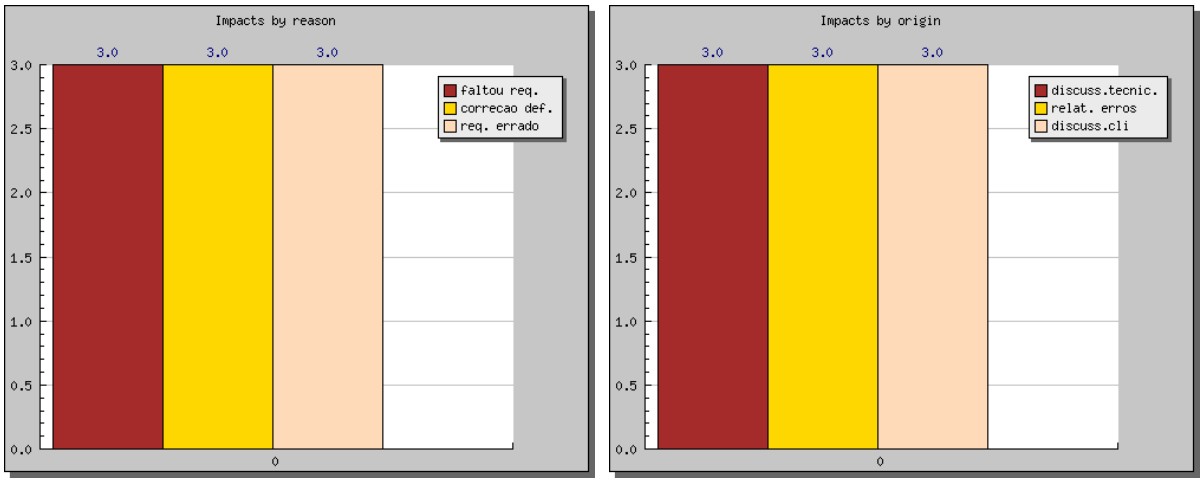


Figura 115. Estatísticas de impactos ocorridos

O número de impactos nem sempre é muito significativo, uma vez que um impacto

pode afetar uma quantidade diferente de elementos no sistem (artefatos ou membros). Neste caso, é necessário analisar não somente o número de impactos, mas também o número de elementos afetados por estes impactos. Neste sentido, os gráficos da imagem a seguir procuram transmitir o número de elementos afetados conforme as classificações de razão e origem adotadas. Assim, verificou-se que dos 13 elementos afetados, 6 foram em virtude de definições erradas de requisitos, 4 pela falta na definição de requisitos e 3 pela correção de defeitos. Estes números permitem supor inicialmente que a falta de definição ou a definição errada de requisitos produz um impacto significativo ao longo do desenvolvimento do sistema, afirmação que está em sintonia com o pensamento dos principais autores da área da engenharia de software.

Quando analisados os impactos gerados conforme a sua origem, pode-se notar que 6 impactos foram originados por mudanças registradas em relatórios de erros do usuário, 4 em discussões com o cliente e 3 em discussões técnicas. Conclui-se superficialmente que os piores maiores foram detectados somente pelo usuário final da aplicação.

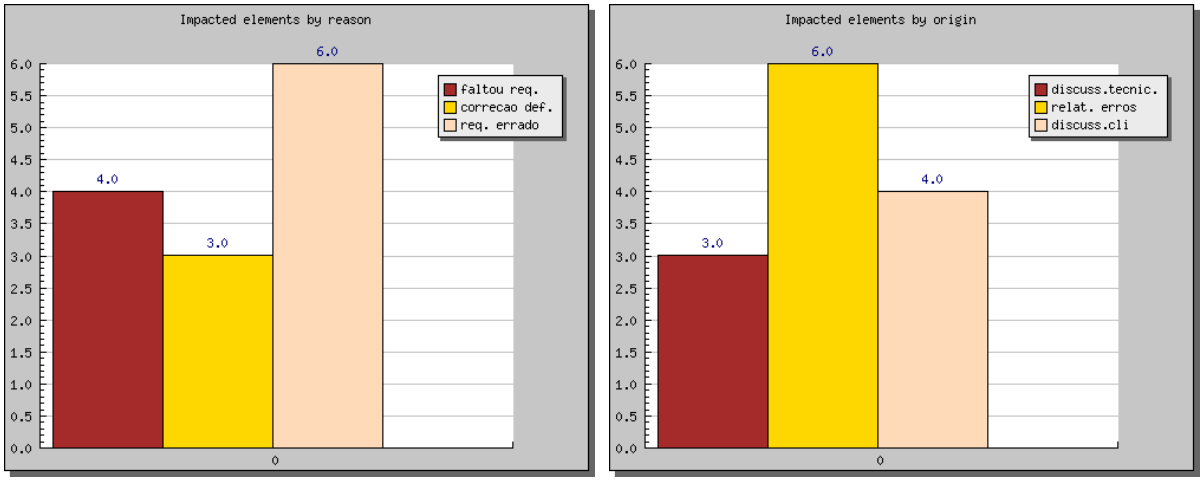


Figura 116. Estatísticas de elementos impactados

A tabela a seguir procura demonstrar a análise de correlação entras razões e as origens de imapctos. Desta forma, pode-se perceber que os impactos motivados por definições erradas de requisitos foram verificados em relatórios de erros do cliente, os impactos motivados pela falta da definição de requisitos foram originados em discussões com o cliente e impactos motivados pela correção de defeitos foram originados em discussões técnicas.

Tabela 20 -Análise combinada de razões e origens de impactos

Razão	Origem	Quantidade	Percentual
Requisito Errado	Relatorios de erros	3	33,33 %
Requisito faltante	Discussao com o Cliente	3	33,33 %
Correcao de defeito	Discussao tecnica	3	33,33 %

Por fim, procura-se elaborar um panorama geral da situação do projeto de desenvolvimento com base nas informações coletadas durante o período de funcionamento do protótipo. Neste sentido, a figura a seguir demonstra quatro variáveis diferentes: a quantidade total de mudanças realizadas, as mudanças ainda não classificadas, o número total de impactos gerados e o número de impactos ainda não resolvidos. Com base no gráfico a seguir, não pode-se afirmar que, durante o período de análise, o projeto tem demonstrado alguma tendência de estabilização de mudanças e impactos, sendo que os números sofreram certa elevação na fase final do período de avaliação. Esta elevação deveu-se em grande parte à uma intensificação dos testes por parte dos usuários neste mesmo período, uma vez que a instituição avaliada iria entrar em período de férias no período imediatamente subsequente ao término desta análise.

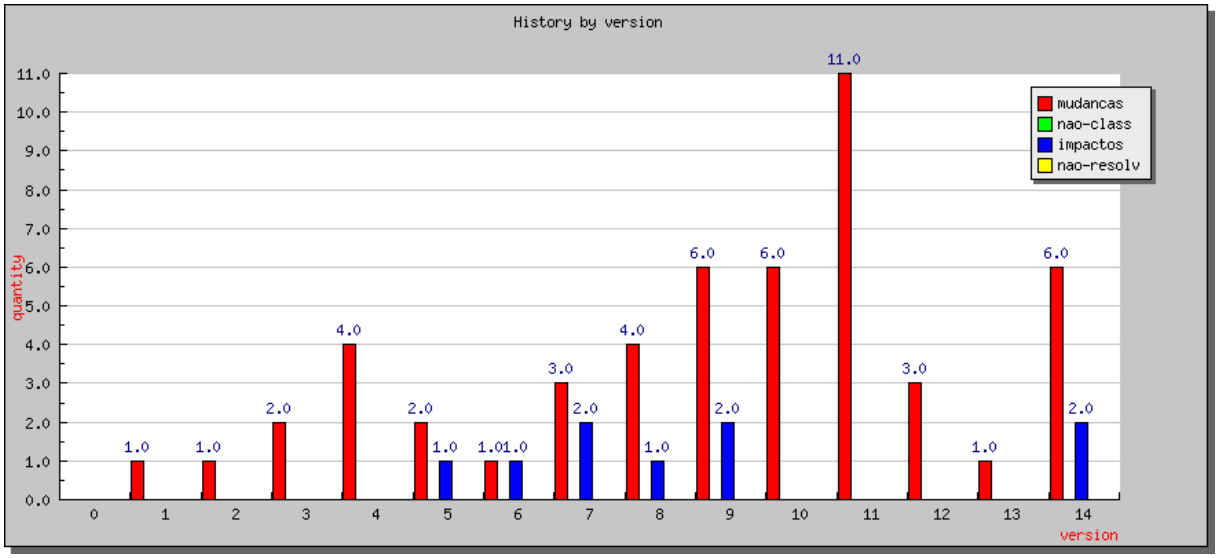


Figura 117. Histórico de mudanças do projeto

7 CONCLUSÕES

A gestão da mudança de requisitos é uma atividade que exerce papel fundamental em relação à qualidade no processo de desenvolvimento de software. A gestão da mudança está diretamente relacionada com a análise de impactos, que permite dimensionar os custos de uma mudança. A análise de impactos, por sua vez, está relacionada com a informação de rastreabilidade, que provê subsídios para o correto dimensionamento dos impactos gerados por uma mudança.

Para a criação de um ambiente que possibilite a implementação de uma efetiva gestão de mudanças, é necessário o desenvolvimento de um modelo que contemple todos os aspectos relacionados à mudança de requisitos, como o suporte à artefatos de qualquer fase do processo de desenvolvimento, o suporte à rastreabilidade (horizontal e vertical), a configuração da granularidade, a gestão dos impactos, o versionamento dos requisitos, dentre outros critérios. A criação de um modelo que suporte os critérios apontados permite o desenvolvimento de um método que explore todo o potencial da gerência da mudança de requisitos.

Neste sentido, a maior contribuição deste trabalho está em desenvolver um modelo e um método voltados à gestão da mudança de requisitos com suporte à rastreabilidade e à análise de impactos, que suprem um conjunto de requisitos não encontrados em sua totalidade em nenhum dos trabalhos avaliados.

O modelo proposto no presente trabalho permite a representação de artefatos de todo o ciclo de vida de um projeto de software, desde análise até o desenvolvimento, bem como todas as interligações possíveis entre estes artefatos. Este modelo possui ainda, flexibilidade suficiente para representar relações com diferentes níveis de granularidade no projeto. Além disto, o modelo representa importantes conceitos envolvidos em diferentes processos como a representação de mudanças, de impactos e o versionamento dos artefatos. A flexibilidade, bem como a prática de sua implementação possibilitam a criação de uma série de aplicações, não restringindo-se somente ao método aqui proposto.

O método proposto no presente trabalho é concretizado por um protótipo que implementa todos as características suportadas pelo modelo proposto. Este protótipo apóia-se sobre a informação de rastreabilidade para suportar a gestão da mudança de requisitos, bem como a análise de impactos. Dentre suas características principais estão o suporte à notificação e propagação de eventos, a integração com outras ferramentas, a interpretação de diversos de artefatos de software pertencentes à diferentes fases do ciclo de desenvolvimento,

a recuperação da informação de rastreabilidade horizontal, e o gerenciamento proativo desempenhado por gerentes de software. Além disto, possibilita a criação de outras aplicações para gestão da mudança de requisitos, que venham a implementar funcionalidades adicionais para esta aplicação, visto que tal método destaca-se por permitir extensibilidade por meio de *Web Services* que expõem todas as funcionalidades descritas no presente trabalho. Desta forma, trabalhos futuros podem explorar a criação de novas funcionalidades sem a necessidade de reescrever os recursos já existentes na ferramenta.

7.1 TRABALHOS FUTUROS

O trabalho aqui proposto cumpriu plenamente as metas que foram tracadas em seu planejamento. Entretanto, no decorrer de seu desenvolvimento e de sua validação, percebeu-se que alguns novos recursos poderiam ser implementados, dentre eles, merecem destaque:

- Integrar a ferramenta proposta com alguma ferramenta de gerência de projetos a fim de obter automaticamente informações sobre os papeis, *stakeholders*, datas de *releases*, para que se possa emitir gráficos com informações gerenciais dimensionadas conforme o planejamento original do projeto;
- Procurar realizar algum tipo de integrar *on-line* da ferramenta proposta com repositórios do tipo SVN. Desta forma, as mudanças poderiam ser detectadas no exato momento em que elas ocorrerem, diferentemente do trabalho proposto, no qual um agente vasculha o repositório com certa periodicidade;
- Utilizar técnicas como a de *data mining* sobre o modelo proposto, a fim de se descobrir informações sobre tendências de mudanças que possam ser utilizadas par ao planejamento futuro do projeto;
- Melhorar o algorítmo para recuperação da informação de rastreabilidade por meio da aplicação de técnicas específicas para o reconhecimento de expressões e processamento de linguagem natural;
- Criar novas categorias de agentes, voltados a monitorar outros aspectos do gerenciamento de requisitos não previstos, como mudanças em bancos de dados;

REFERÊNCIAS

ARNOLD, R. S., BOHNER, S. A. “Impact Analysis - Towards a Framework for Comparison”. In Proceedings of the Conference on Software Maintenance. 1993. D. N. Card, Ed. IEEE Computer Society, Washington, DC, 292-301.

BATISTA, E., CARVALHO, A. "Uma taxonomia facetada para técnicas de elicitação de requisitos", 11/2003, 6th International Workshop on Requirements Engineering, Vol. 1, pp.48-62, Piracicaba, SP, Brasil, 2003

BECK, Kent, ANDRES, Cynthia. Extreme Programming Explained : Embrace Change (2nd Edition). 224 pages. Publisher: Addison-Wesley Professional; 2 edition (November 16, 2004).

BIANCHI, A., VISAGGIO, G., FASOLINO, A. R. An Exploratory Case Study of the Maintenance Effectiveness of Traceability Models. In Proceedings of the 8th international Workshop on Program Comprehension (June 10 - 11, 2000). IWPC. IEEE Computer Society, Washington, DC, 149.

BOHNER, S. A. Impact analysis in the software change process: a year 2000 perspective. In Proceedings of the 1996 international Conference on Software Maintenance (November 04 - 08, 1996). ICSM. IEEE Computer Society, Washington, DC, 42-51

BREITMAN, K., LEITE, J.C.S.P. Ontology as a Requirements Engineering Product. Proceedings of the 11th IEE International Requirements Engineering Conference. 1090-705X/ 03, 2003 IEEE Computer Society.

BREITMAN, K., LEITE, J.C.S.P., Berry, M. Supporting scenario evolution. Requirements Engineering, Springer London, v.10 n.2, p.112-131, May 2005.

BRIAND, L., LABICHE, Y., O'SULLIVAN, L. "Impact Analysis and Change Management of UML Models". Software Maintenance, IEEE International Conference on, vol. 0, no. 0, pp. 256, 19th IEEE International Conference on Software Maintenance (ICSM'03), 2003.

BRITO, R. A., VASCONCELOS, A. M. L. Integrando Groupware e Gerenciamento de Requisitos no Suporte à Engenharia de Requisitos Distribuída. In: IX Workshop on Requirements Engineering, 2006, Rio de Janeiro. Cadernos do IME - Série Informática, 2006. v.20. p.84-92.

BRUEGGE, B., DE LUCIA, A., FASANO, F., TORTORA, G. Supporting Distributed Software Development with fine-grained Artefact Management. In Proceedings of the IEEE international Conference on Global Software Engineering (October 16 - 19, 2006). ICGSE. IEEE Computer Society, Washington, DC, 213-222.

CAIRE, G. Agent oriented analysis using MESSAGE/UML. In Ciancarini, P., Wooldridge, M. (Eds.) Agent-Oriented Software Engineering, Berlin: Springer-Verlag, 2001.

CHAVDA, K. F. Anatomy of Web Service, Journal of Computing Sciences in Colleges, Kennesaw State University, January 2004.

CHUNG, J., LIN, K., MATHIEU, R. Web Services Computing: Advancing Software Interoperability. IEEE Computer, 35-37, 2003.

CIMITILE, A., FASOLINO, A. R., VISAGGIO, G. A Software Model for Impact Analysis: A Validation Experiment. InProceedings of the Sixth Working Conference on Reverse Engineering (Oct. 06 - 08, 1999). WCRE. IEEE Computer Society, Washington, DC, 212.

CLELAND-HUANG, J., CHANG, C. K., CHRISTENSEN, M. Event-Based Traceability for Managing Evolutionary Change. IEEE Trans. Softw. Eng. 29, 9 (Sep. 2003), 796-810.

CLELAND-HUANG, J., SETTIMI, R., ROMANOVA, E., BERENACH, B., CLARK, S. Best practices for automated traceability, IEEE Computer Magazine. Vol.40, no. 6, p. 27-35. 2007.

CLELAND-HUANG, J., HAYES, J. H., DOMEL, J. M. Model-based traceability. In Proceedings of the 2009 ICSE Workshop on Traceability in Emerging Forms of Software Engineering (May 18 - 18, 2009). International Conference on Software Engineering. IEEE Computer Society, Washington, DC, 6-10.

DÁRIO, Cláudia. Uma Metodologia Unificada para o Desenvolvimento de Sistemas Orientados a Agentes, M.S. Thesis – DCA-FEEC- UNICAMP (2005).

DAVIS, Alan M., NURMULIANI, N., PARK, S., ZOWGHI, D. Requirements Change: What's the Alternative? Computer Software and Applications Conference, Annual International, vol. 0, no. 0, pp. 635-638, 2008 32nd Annual IEEE International Computer Software and Applications Conference, 2008.

DE BORTOLI, Lis Ângela., PRICE, Ana Maria de Alencar. O Uso de Workflow para Apoiar a Elicitação de Requisitos. Workshop em Engenharia de Requisitos WER 2000: 22-37

DE GRANDE, J. I., MARTINS, L.E.G. SIGERAR: uma Ferramenta para Gerenciamento de Requisitos. Proceedings of the IX Workshop on Requirements Engineering (WER'2006). Rio de Janeiro - Brasil, Julho/2006. pp. 1-9.

DE LOACH. An Overview of the Multiagent Systems Engineering Methodology. Proceedings of the First International Workshop on Agent-Oriented Software Engineering, 10th June 2000, Limerick, Ireland.

DE LUCIA, A., FASANO, F., FRANCESE, R., TORTORA, G. ADAMS: an artefact-based process support system. In Proceedings of 16th International Conference on Software Engineering and Knowledge Engineering, pages 31--36, Banff, Alberta, Canada, 2004. KSI Press.

DE LUCIA, A., FASANO, F., OLIVETO, R., TORTORA, G. ADAMS Re-Trace: A Traceability Recovery Tool. InProceedings of the Ninth European Conference on Software Maintenance and Reengineering (March 21 - 23, 2005). CSMR. IEEE Computer Society, Washington, DC, 32-41.

DE LUCIA, A., OLIVETO, R., ZUROLO, F., DI PENTA, M. Improving Comprehensibility of Source Code via Traceability Information: a Controlled Experiment. In Proceedings of the 14th IEEE international Conference on Program Comprehension (June 14 - 16, 2006). ICPC. IEEE Computer Society, Washington, DC, 317-326.

DE LUCIA, A., FASANO, F., OLIVETO, R. Traceability Management for Impact Analysis. Frontiers of Software Maintenance, Beijing, China, IEEE, 2008, pp. 21-30.

DURÁN, A., BERNÁRDEZ, B., RUIZ, A., TORO, M. (1999): A requirements elicitation approach based in templates and patterns, Proceedings of the Workshop in Requirements (WER99), Buenos Aires.

EGROUPWARE. Página Web. Disponível na World Wide Web em http://www.egroupware.org. Última visita realizada em 27/12/2009.

ESPINDOLA, R., LOPES, L., PRIKLADNICKI, R., AUDY, J. Uma Abordagem Baseada em Gestão do Conhecimento para Gerência de Requisitos em Desenvolvimento Distribuído de Software. Proc. VIII Workshop on Requirements Engineering (WER 05), Cidade do Porto, Portugal, 2005, in Portuguese.

FALBO, R., PEZZIN, J., SCHAMBACH, M. OplA: Uma Metodologia para o Desenvolvimento de Sistemas Baseados em Agentes e Objetos; Actas de las IV Jornadas Iberoamericanas de Ingeniería del Software e Ingeniería del Conocimiento, 2004. Madrid, ES.

FANTECHI, A., SPINICCI, E. A Content Analysis Technique for Inconsistency Detection in Software Requirements Documents. 245-256 Anais do WER05 - Workshop em Engenharia de Requisitos, Porto, Portugal, Junho 13-14, 2005. 2005, ISBN 972-752-079-0.

FASANO, Fausto. Fine-grained management of software artefacts. In Software Maintenance, 2007. ICSM 2007. IEEE International Conference on, 2007, pp. 507-508.

FOWLER, Martin. Patterns of Enterprise Application Architecture. Addison-Wesley, 2002.

GHAZI, H. E. MV - TMM: A Multi View Traceability Management Method. In Proceedings of the 2008 32nd Annual IEEE international Computer Software and Applications Conference (July 28 - August 01, 2008). COMPSAC. IEEE Computer Society, Washington, DC, 247-254.

GIORGINI, P., KOLP, M., MYLOPOULOS, J, PISTORE, M. The Tropos Methodology: An Overview. In: Bergenti, M.-P. Gleizes and F. Zambonelli (Eds) Methodologies And Software

Engineering For Agent Systems, Kluwer Academic Publishing (NewYork), December, 2003.

GRAPHVIZ. Página Web. Disponível na World Wide Web em http://www.graphviz.org. Última visita realizada em 04/01/2010.

HANSEN, R., SANTOS, C., CRESPO, S., LANIUS, G., MASSEN, F. Web Services: An Architecture Overview. First Seminar on Advanced Research in Electronic Business, Rio de Janeiro, Brasil, 2002.

HAYES, J., DEKHTYAR, A., SUNDARAM, S. Advancing Candidate Link Generation for Requirements Tracing: The Study of Methods. IEEE Transactions on Software Engineering, vol. 32, no. 1, pp. 4-19, January, 2006.

HAZAN, C., LEITE, J.C.S.P. Indicadores a para Gerência de Requisitos, in WER 2003 - VI Workshop em Engenharia de Requisitos, São Paulo, Brazil.

HOFFMANN, M., KUHN, N., WEBER, M., BITTNER, M. Requirements for Requirements Management Tools. In Proceedings of the Requirements Engineering Conference, 12th IEEE international (Sep. 06 - 10, 2004). RE. IEEE Computer Society, Washington, DC, 301-308.

HORRIAN, H., MAHMUD, S., KARTHIKEYAN, S. Requirements Engineering in Agile methods. Dept. of Computer Science, University of Calgary, Canada, 2003.

HUANG, L., SONG, Y."Dynamic Impact Analysis Using Execution Profile Tracing. Software Engineering Research, Management and Applications, ACIS International Conference on, vol. 0, no. 0, pp. 237-244, Fourth International Conference on Software Engineering Research, Management and Applications (SERA'06), 2006. IEEE Computer Society 2006, ISBN 0-7695-2656-X. pp.237~244.

IMTIAZ. S., IKRAM, N. Impact Analysis from Multiple Perspectives: Evaluation of Traceability Techniques. Software Engineering Advances, International Conference on, vol. 0, no. 0, pp. 457-464, 2008 . The Third International Conference on Software Engineering Advances, 2008.

KAIYA, H., SAEKI, M. Ontology Based Rerquirements Analysis: Lightweight Semantic Processing Aproach. Proceedings of the Fifth International Conference on Quality Software (QSIC'05) 1550-6002/05, 2005 IEEE Computer Society.

KAIYA, H., SAEKI, M. Using Domain Ontology as Domain Knowledge for Requirements Elicitation. Proceedings of the 14th IEEE International Requirements Engineering Conference (RE'06), 0-7695-2555-5/06, 2006 IEEE Computer Society.

KASSAB, M., ORMANDJIEVA, O., DANEVA, M. A Traceability Metamodel for Change Management of Non-functional Requirements. In Proceedings of the 2008 Sixth international

Conference on Software Engineering Research, Management and Applications (August 20 - 22, 2008). SERA. IEEE Computer Society, Washington, DC, 245-254.

KNETHEN, Antje von. Change-Oriented Requirements Traceability: Support for Evolution of Embedded Systems. In Proceedings of the international Conference on Software Maintenance (Icsm'02) (October 03 - 06, 2002). ICSM. IEEE Computer Society, Washington, DC, 482.

KNETHEN, Antje von. A trace model for system requirements changes on embedded systems. In Proc. of the 4th Int. Workshop on Principles of Software Evolution, pages 17–26, New York, NY, USA, 2001. ACM Press.

KOTONYA, G., SOMMERVILLE, I. Requirements Engineering: Processes and Techniques. John Willey & Sons Ltd, 1998.

KUNG, D., GAO, J., HSIA, P., WEN, F., TOYOSHIMA, Y., CHEN, C. Change Impact Identification in Object Oriented Software Maintenance. Proceedings of the International Conference on Software Maintenance, ICSM 1994, Victoria, BC, Canada, September 1994. p.202-211. IEEE Computer Society. ISBN 0-8186-6330-8.

LAM, W., LOOMES, M., SHANKARARAMAN, V. Managing Requirements Change Using Metrics and Action Planning. In Proceedings of the Third European Conference on Software Maintenance and Reengineering (March 03 - 05, 1999). CSMR. IEEE Computer Society, Washington, DC, 122.

LANG, M., DUGGAN, J. A Tool to Support Collaborative Software Requirements Management. Requirements Engineering Journal. Vol. 6. No. 3. 2001.

LEAL, M., FIGUEIREDO, M. C., DE SOUZA, C. R. B. Uma abordagem semi-automática para a manutenção de links de rastreabilidade. In: 11th Workshop on Proceedings of the 11th Workshop on Requirements Engineering, p. 47-58. 2008.

LEE, C., GUADAGNO, L., JIA, X. An Agile Approach to Capturing Requirements and Traceability, Proc. of the 2nd International Workshop on Traceability in Emerging Forms of Software Engineering, October 7, 2003, Montreal, Canada, pp.17-23.

LEITE, J. C. S. P. Extreme Requirements. In: Jornadas de Ingeniería de Requisitos Aplicada, 2001, Sevilha. Jornadas de Ingeniería de Requisitos Aplicada, 2001. p. 1-13.

LEITE, J. C. S. P., DOORN, J. H., HADAD, G. D. S., KAPLAN, G. N. Scenario Inspections. Requirements Engineering Journal 10.1007/s00766-003-0186-9 – Springer Verlag - London Limited – 2004.

LEMOS, A., DE SOUZA, C. R. B. Engenharia de requisitos em ambientes distribuídos de desenvolvimento de software: resultados preliminares de um estudo etnográfico. Proc. 11th

WER'08, Barcelona, Spain, September 12-13, 2008, pp. 85-95.

LETELIER P. A Framework for Requirements Traceability in UML-based Projects. In Proc. of 1st International Workshop on Traceability in Emerging Forms of Software Engineering. In conjunction with the 17th IEEE International Conference on Automated Software Engineering, Edinburgh, U.K., September 2002

LLOYD, W. J., ROSSON, M. B., ARTHUR, J. D. Effectiveness of Elicitation Techniques in Distributed Requirements Engineering. Requirements Engineering, IEEE International Conference on, vol. 0, no. 0, pp. 311, 10th Anniversary Joint IEEE International Requirements Engineering Conference (RE'02), 2002.

MÄDER, P., GOTEL, O., PHILIPPOW, I. Getting back to basics: Promoting the use of a traceability information model in practice. In Proceedings of the 2009 ICSE Workshop on Traceability in Emerging Forms of Software Engineering (May 18 - 18, 2009). International Conference on Software Engineering. IEEE Computer Society, Washington, DC, 21-25.

MANNIO, M., NIKULA, U. Requirements Elicitation Using a Combination of Prototypes and Scenarios. 283-296 Anais do WER01 - Workshop em Engenharia de Requisitos, Buenos Aires, Argentina, Novembro 22-23, 2001, p.283-296.

MARTINS, L., DALTRINI, B. Activity Theory: a Framework to Software Requirements Elicitation. Anais do WER99 - Workshop em Engenharia de Requisitos, Buenos Aires, Argentina, Setembro 9-10, 1999, pp 30-44.

MARTINS, L., DALTRINI, B. Organizando o Processo de Elicitação de Requisitos Utilizando o Conceito de Atividade. 297-317 Anais do WER01 - Workshop em Engenharia de Requisitos, Buenos Aires, Argentina, Novembro 22-23, 2001. 2001

NOLL, R. P., RIBEIRO, M. B. Ontological Traceability over the Unified Process. In Proceedings of the 14th Annual IEEE international Conference and Workshops on the Engineering of Computer-Based Systems (March 26 - 29, 2007). ECBS. IEEE Computer Society, Washington, DC, 249-255.

NURMULIANI, ZOWGHI D., FOWELL S. Analysis of Requirements Volatility during Software Development Life Cycle, proceedings of the Australian Software Engineering Conference (ASWEC), April 13-16, Melbourne, Australia, 2004.

NUSEIBEH, B., EASTERBROOK, S. Requirements Engineering: A Roadmap. Proceedings of International Conference on Software Engineering (ICSE-2000), 4-11 June 2000, Limerick, Ireland.

O'NEAL, J., CARVER, D. Analyzing the Impact of Changing Requirements. Software Maintenance, IEEE International Conference on, vol. 0, no. 0, pp. 190, 17th IEEE

International Conference on Software Maintenance (ICSM'01), 2001.

OLIVETO, R., ANTONIOL, G., MARCUS, A., HAYES, J. Software artefact traceability: the never-ending challenge. In Proceedings of 23rd IEEE International Conference on Software Maintenance, pages 485--488, Paris, France, 2007. IEEE Press.

PADGHAM, L., WINIKOFF, M. A Methodology for Developing Intelligent Agents. Proceedings of AOSE 2002, 2002.

PAETSCH, F., EBERLEIN, A., MAURER, F. Requirements Engineering and Agile Software Development. Proceedings of the Twelfth IEEE International Workshops on Enabling Technologies: Infrastructure for Collaborative Enterprises (WETICE'03). 2003 IEEE.

PINHEIRO, F.A.C., GOGUEN, J.A. An Object-Oriented Tool for Tracing Requirements. icre, pp.219, Second International Conference on Requirements Engineering (ICRE'96), 1996.

PINHEIRO, F.A.C. An Object-Oriented Library for Tracing Requirements. WER99 - II Workshop on Requirements Engineering, event of XXVIII JAIIO:Jornadas Argentinas de Informatica e Investigacion Operativa, Buenos Aires, Argentina, pp 187-197, September 1999.

PINHEIRO, F.A.C. Formal and Informal Aspects of Requirements Tracing. Anais do WER00-Workshop em Engenharia de Requisitos, Rio de Janeiro-RJ, Brasil, Julho 13-14, 2000, pp1-21.

PINHEIRO, F.A.C. Requirements Traceability. Capítuo 5: Julio Cesar S. P. Leite, Jorge H. Doorn. (Org.). Perspectives on Software Requirements. Boston: Kluwer Academic Publishers, 2004, v. 1, p. 91-110.

POŽGAJ, Ž. SERTIĆ, H. BOBAN, M. Effective requirement specification as a precondition for successful software development project, Proceedings of the 25th International Conference of Information Technology Interfaces ITI 2003, Cavtat, June 16-19, 2003, p. 669-674. Rad u zborniku citiran u bazi INSPEC Computer and Control Abstract.

QUEILLE, J., VOIDROT, J., WILDE, N., and MUNRO, M. The Impact Analysis Task in Software Maintenance: A Model and a Case Study. In Proceedings of the international Conference on Software Maintenance H. A. Müller and M. Georges, Eds. IEEE Computer Society, Washington, DC, 234-242. 1994.

RAMESH, B., POWERS, T., STUBBS, C., EDWARDS, M. Implementing requirements traceability: a case study. In Proceedings of the Second IEEE international Symposium on Requirements Engineering (March 27 - 29, 1995). RE. IEEE Computer Society, Washington, DC, 89.

RAMESH, B., DWIGGINS, D., DEVRIES, G., EDWARDS, M. Towards requirements traceability models. In: Proceedings of the 1995 International Symposium and Workshop on

Systems Engineering of Computer-Based Systems. IEEE Computer Society Press, Los Alamitos, CA, pp. 229-232. 1995.

REN, X., SHAH, F., TIP, F., RYDER, B. G., CHESLEY, O. Chianti: a tool for change impact analysis of java programs. InProceedings of the 19th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (Vancouver, Canada, October 24 - 28, 2004). OOPSLA '04. ACM, New York, NY, 432-448.

SALEM, A. M. Improving Software Quality through Requirements Traceability Models. In Proceedings of the IEEE international Conference on Computer Systems and Applications (March 08 - 08, 2006). AICCSA. IEEE Computer Society, Washington, DC, 1159-1162.

SAYÃO, M., LEITE, J. C. Rastreabilidade de Requisitos. Revista de Informática Teórica e Aplicada, v. XIII, p. 57-86, 2006.

SEYBOLD, C., MEIER, S., GLINZ, M. Evolution of Requirements Models by Simulation. Proceedings of the 7th International Workshop on Principles of Software Evolution (IWPSE'04), Kyoto, Japan, 2004. 43-48.

TIOBE. Página Web. Disponível na World Wide Web em http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html. Última visita realizada em 29/12/2009.

TOGNERI, D. F., FALBO, R. A., MENEZES, C. S. Supporting Cooperative Requirements Engineering with an Automated Tool. Proceedings of the V Workshop on Requirements Engineering, pp. 240 – 254, Valencia, Spain, November 2002.

VAUGHAN, Nichalos. WEB Services: Beyond the Hype. IEEE Computer, 35:2, 2002.

WOOLDRIDGE, Michael. An Introduction to Multiagent Systems. Published in February 2002 by John Wiley & Sons (Chichester, England). ISBN 0 47149691X. 348p.

ZONG, Y., ZHI-XUE, Wang, YING-YING, Yang, YUE, Wu, YING, Liu. Towards a Multiple Ontology Framework for Requirements Elicitation and Reuse. 31st annual International Computer Software and Applications Conference (COMPSAC 2007) 0-7695-2870-8/07 2007 IEEE Computer Society.

ZHANG, Y., WITTE, R., RILLING, J., HAARSLEV, V. Ontological approach for the semantic recovery of traceability links between software artefacts, in IET Software, 2008, Vol. 2, No. 3, pp. 185-203, Montreal, Canada, 2008.