ODYSSEY-WI: UMA ABORDAGEM PARA A MINERAÇÃO DE RASTROS DE MODIFICAÇÃO DE MODELOS EM REPOSITÓRIOS VERSIONADOS

Cristine Ribeiro Dantas

TESE SUBMETIDA AO CORPO DOCENTE DA COORDENAÇÃO DOS PROGRAMAS DE PÓS-GRADUAÇÃO DE ENGENHARIA DA UNIVERSIDADE FEDERAL DO RIO DE JANEIRO COMO PARTE DOS REQUISITOS NECESSÁRIOS PARA A OBTENÇÃO DO GRAU DE MESTRE EM CIÊNCIAS EM ENGENHARIA DE SISTEMAS E COMPUTAÇÃO.

Aprovada por:

_____

Prof. Cláudia Maria Lima Werner, D.Sc.

_____

Prof. Marta Lima de Queirós Mattoso, D.Sc.

_____

Prof. Karin Koogan Breitman, D.Sc.

RIO DE JANEIRO, RJ – BRASIL

MARÇO DE 2005

DANTAS, CRISTINE RIBEIRO

Odyssey-WI: Uma Abordagem Para a Mineração de Rastros De Modificação De Modelos Em Repositórios Versionados [Rio de Janeiro] 2005

XI, 115 p., 29,7 cm (COPPE/UFRJ, M.Sc., Engenharia de Sistemas e Computação, 2005)

Tese - Universidade Federal do Rio de Janeiro, COPPE

1. Manutenção de Software
2. Gerência de Configuração de Software
3. Mineração de Dados
4. Rastreabilidade

I. COPPE/UFRJ II. Título (série)

À Deus.

Agradecimentos

À Deus, em primeiro lugar.

Aos meus pais, pelo apoio financeiro e pela presença nos momentos em que necessitei de atenção.

Ao meu irmão, Alexandre Dantas, pelos conselhos e pelas broncas, que foram necessárias para que eu refletisse mais sobre como me posicionar diante da vida.

À professora Cláudia Werner, por sua orientação e seu otimismo, por ter incentivado esse trabalho e me apoiado nos momentos difíceis.

Ao Leonardo Murta, pela co-orientação extra-oficial, por sempre se mostrar disposto a ouvir, por ser extremamente paciente, por discutir cada ponto desta dissertação em detalhe, por ajudar sempre, por buscar a realização de um bom trabalho, pelos conselhos, por sempre atender o telefone nas minhas horas de desespero e por ser um grande amigo. Dedico essa tese também à você.

Às professoras Marta Mattoso e Karin Breitman por aceitarem participar desta banca.

Aos integrantes do grupo de Gerência de Configuração de Software e Projeto Odyssey-SCM, Hamilton e Luiz Gustavo, cuja participação foi realmente importante. Obrigada, Hamilton, por sua amizade e pelo apoio em um dos momentos mais difícies da minha vida.

A todos os integrantes do Projeto Odyssey que contribuíram muito para a realização deste trabalho. Em especial, a atenção dos amigos Aline Vasconcellos, Ana Paula Blois, Marco Mangan, Marco Lopes e Márcio Barros.

Ao CNPq, pelo apoio financeiro.

Resumo da Tese apresentada à COPPE/UFRJ como parte dos requisitos necessários para a obtenção do grau de Mestre em Ciências (M.Sc.)

ODYSSEY-WI: UMA ABORDAGEM PARA A MINERAÇÃO DE RASTROS DE MODIFICAÇÃO DE MODELOS EM REPOSITÓRIOS VERSIONADOS

Cristine Ribeiro Dantas

Março / 2005

Orientadora: Cláudia Maria Lima Werner

Programa: Engenharia de Sistemas e Computação

Os sistemas de software estão em constante evolução. À medida que o software evolui, os modelos de análise e projeto *UML* devem ser modificados de forma a manter a consistência do software. Estas modificações podem por sua vez conduzir modificações subseqüentes em outros elementos do modelo. Neste cenário, um dos maiores problemas é detectar quais elementos devem ser modificados em conjunto. O principal questionamento é: “Existem outros elementos que devem ser alterados em virtude de uma dada modificação?”. Para responder a esta pergunta, é necessário encontrar as relações entre os elementos de modelo existentes. Entretanto, encontrar relações sem suporte automático é uma tarefa propensa à falhas.

O objetivo desse trabalho é aplicar técnicas de mineração em repositórios de sistemas da gerência de configuração de software a fim de detectar rastros de modificação entre elementos de modelo *UML*. As técnicas de mineração de dados possibilitam que relações sejam descobertas através da análise de informações. Durante a manutenção e construção do software, tais relações podem: sugerir modificações futuras, minimizar alterações incompletas e apoiar a análise de impacto, através da identificação das possíveis conseqüências de uma modificação no software. Os rastros de modificação são apresentados com uma semântica própria que auxilia a análise da sua utilidade no processo de modificações.

Abstract of Thesis presented to COPPE/UFRJ as a partial fulfillment of the requirements for the degree of Master of Science (M.Sc.)

ODYSSEY-WI: AN APPROACH FOR MINING CHANGE TRACES FROM VERSIONED REPOSITORIES

Cristine Ribeiro Dantas

March / 2005

Advisor: Cláudia Maria Lima Werner

Department: Computer and Systems Engineering

Software is constantly changing. As software evolves, analysis and design models should be modified. These changes can lead to subsequent changes in other model elements. In this scenario, one of the main problems is to detect which elements should be changed together. The main question is “Are there other elements that may be affected by a certain change?”. One way to answer this question is finding the relations of existing *UML* model elements. However, finding relations without any automatic support is an error-prone task.

This work aims to apply data mining techniques over a software configuration management repository in order to detect change traces among model elements. Data mining techniques allow the extraction of patterns through information analysis. These patterns can help to: suggest and predict likely further changes, prevent incomplete changes, and support the execution of the impact analysis activity through the identification of the potential side-effects of a software change. These traces are detected together with context information that assists the analysis of its utility in the modification process.

Índice

Capítulo 1 – Introdução 1
1.1 – Motivação 1
1.1.1 – Desenvolvimento Baseado em Componentes 3
1.2 – Caracterização do Problema 4
1.3 – Objetivo 6
1.4 – Contexto da Dissertação 7
1.5 – Organização dos Capítulos 8
Capítulo 2 – Rastreabilidade através da Gerência de Configuração de Software 9
2.1 – Introdução 9
2.2 – Rastreabilidade 10
2.2.1 – Rastros de Pré-Especificação 12
2.2.2 – Rastros de Pós-Especificação 16
2.3 – Análise de Impacto 18
2.4 – Gerência de Configuração de Software 22
2.5 – Integração dos Espaços de Trabalho 25
2.6 – Conclusões 28
Capítulo 3 – O Uso de Técnicas de Mineração de Dados no Auxílio à Manutenção do Software 30
3.1 – Introdução 30
3.2 – Mineração de Dados em Repositórios de Software 31
3.3 – Técnicas de Mineração de Dados 35
3.4 – Mineração por Regras de Associação 38
3.4.1 – Medidas de suporte e confiança 39
3.4.2 – O algoritmo Apriori 42
3.4.3 – Generalização das Regras de Associação 45
3.5 – Conclusões 46
Capítulo 4 – Odyssey-WI: Um Mecanismo para Rastreabilidade entre Modelos 47
4.1 – Introdução 47
4.2 – Contexto de Desenvolvimento: Projeto Odyssey-SCM 48
4.2.1 – Odyssey-CCS: Sistema de Controle de Modificações 49

4.2.2 – Odyssey-VCS: Sistema de Controle de Versões Para Modelos Baseados no MOF 51
4.3 – Odyssey-WI: A Abordagem Proposta 53
4.3.1 – Características Gerais 53
4.3.1.1 – Fundamentação dos rastros 54
4.3.1.2 – Representação segundo a estrutura 5W + 1H 55
4.3.1.3 – Semântica configurável 56
4.3.1.4 – Relevância do Rastro 57
4.3.1.5 – Abrangência em relação aos tipos de artefatos 58
4.3.1.6 – Independência em Relação ao Ambiente de Desenvolvimento 59
4.3.2 – Cenário de Utilização 59
4.3.2.1 – Exemplo 62
4.3.2.2 – Configuração do Mecanismo 63
4.3.2.3 – Detecção dos Rastros 65
4.3.2.3.1 – Base para a Mineração de Dados 65
4.3.2.3.2 – Aplicação do algoritmo de Mineração de Dados 68
4.3.2.4 – Apresentação dos Rastros de Modificação 70
4.4 – Conclusões 72
Capítulo 5 – Protótipo *Odyssey-WI* 74
5.1 – Introdução 74
5.2 – Arquitetura do Projeto Odyssey-SCM 75
5.3 – Contexto de Utilização 78
5.4 – Repositório de Armazenamento 79
5.4.1 – Meta-modelo de Versionamento e Modificações 82
5.5 – Sistemas de Gerência de Configuração de Software 83
5.5.1 – Sistema de Controle de Modificações Odyssey-CCS 83
5.5.2 – Sistema de Controle de Versões Odyssey-VCS 85
5.6 – Camada de Transporte 86
5.7 – A implementação Odyssey-WI 88
5.8 – Utilização do Protótipo 91
5.8.1 – Configuração 91
5.8.2 – Visualização dos Rastros 94
5.9 – Conclusões 97
Capítulo 6 – Conclusões 99

6.1 – Considerações Finais 99
6.2 – Limitações e Trabalhos Futuros 101
6.2.1 – Reestruturação (*refactoring*) dos modelos de análise e projeto.... 102
6.2.2 – Dependência em relação aos dados dos sistemas da GCS..... 102
6.2.3 – Melhorias no suporte a componentes 102
6.2.4 – Extensão para a Pré-Rastreabilidade 103
6.2.5 – Extensão para código-fonte 103
6.2.6 – Melhorias na identificação dos rastros 103
6.2.7 – Melhorias na apresentação da semântica do rastro..... 104
6.2.8 – Suporte à avaliação dos rastros..... 105
6.2.9 – Verificação das conclusões obtidas 105
Referências Bibliográficas..... 107

Índice de Figuras

Figura 2.1 - Modelo de referência e Modelo de raciocínio (Fonte: RAMESH e JARKE, 2000). 14
Figura 2.2 - Pré e Pós Rastreabilidade (Fonte: LINDVALL e SANDAHL, 1996)..... 16
Figura 2.3 - Rastreabilidade vertical (intramodelos) e horizontal (intermodelos) (Fonte: LINDVALL e SANDAHL, 1996). 16
Figura 2.4 - Detecção de conjuntos de modificações 17
Figura 2.5 - Processo de análise de impacto 18
Figura 2.6 - Mecanismo de notificação de dependências 19
Figura 3.1 - Matriz de suporte 40
Figura 3.2 - Matriz de confiança 41
Figura 3.3 - Exemplo do algoritmo Apriori 43
Figura 3.4 - Geração de regras de associação 44
Figura 4.1 - Modelo simplificado do controle de modificações (Fonte: MURTA, 2004). 51
Figura 4.2 - Semântica do rastro de modificação 55
Figura 4.3 - Atividades envolvidas na abordagem Odyssey-WI 60
Figura 4.4 - Casos de uso do exemplo Hotelaria 62
Figura 4.5 - Componentes (a) e classes (b) do exemplo Hotelaria 63
Figura 5.1 - Parte da arquitetura do projeto Odyssey-SCM 76
Figura 5.2 - Meta-modelo do projeto Odyssey-SCM 82
Figura 5.3 - Configuração do fomulário de requisição da modificação 84
Figura 5.4 - Preenchimento das informações de uma modificação 85
Figura 5.5 - Arquivo de configuração WSDL 87
Figura 5.6 - Classes de importação e exportação de XMI no Odyssey 88
Figura 5.7 - Classes do algoritmo de mineração de dados 89
Figura 5.8 - Classes do protótipo Odyssey-WI 90
Figura 5.9 - Configuração das questões 5W + 1H 92
Figura 5.10 - Configuração da precisão do mecanismo de rastreabilidade 93
Figura 5.11 - Ação para a visualização dos rastros 95
Figura 5.12 - Visualização dos rastros de modificação recuperados do repositório 95

Índice de Tabelas

Tabela 3.1 - Tarefas realizadas por técnicas de mineração de dados 35
Tabela 3.2 - Técnicas de mineração de dados 36
Tabela 3.3 - Contexto de utilização da técnica de mineração por regras de associação. 39
Tabela 4.1 - Exemplo de coleta da semântica do rastro (Fonte: MURTA, 2004). 56
Tabela 4.2 - Descrição das modificações realizadas no exemplo Hotelaria..... 63
Tabela 4.3 - Versões dos artefatos gerados para uma modificação..... 66
Tabela 4.4 - Modificações no repositório dos sistemas de GCS 67
Tabela 4.5 - Suporte e confiança das associações encontradas na base de transações... 69
Tabela 4.6 - Semântica associada ao rastro 71
Tabela 5.1 - Arquitetura de metadados da OMG (Fonte: MATULA, 2003)..... 80

Capítulo 1 – Introdução

1.1 – Motivação

Devido à complexidade crescente dos projetos de software em relação a tamanho, sofisticação e tecnologias envolvidas, a fase de manutenção é considerada a fase mais cara nos projetos atuais, contabilizando 60% do total de esforço utilizado no processo de desenvolvimento de software. Aproximadamente, 20% de todo o esforço de manutenção é usado para consertar erros de implementação, os outros 80% são utilizados na adaptação do software em função de modificações externas[1] e na atividade de reengenharia da aplicação (PRESSMAN, 2001).

A qualidade dos artefatos produzidos durante as fases iniciais de um processo de desenvolvimento de software é fundamental para o sucesso no decorrer do projeto e para a manutenção de sua viabilidade (CONROW e SHISHIDO, 1997). A maior parte dos erros encontrados em software - 64% - está associada às fases de análise de requisito e projeto, e são, geralmente, descobertos nas fases mais avançadas como codificação e testes. O custo para correção de um erro ainda na fase de análise equivale a 1/5 do que seria na fase de testes e a 1/15 depois que o sistema estivesse em uso (KOTONYA e SOMMERVILE, 1996).

Uma característica do processo de desenvolvimento de software, que diminui a complexidade dos projetos, é sua composição em várias fases, onde cada fase apresenta um nível de abstração específico. O uso de modelos[2] em diferentes visões é fundamental nas atividades de modelagem e no refinamento sucessivo do conhecimento do domínio em direção a uma visão menos abstrata e mais próxima da visão computacional do problema. Além de servir como base para as etapas seguintes do processo, como codificação e validação, o modelo, quando utilizado em conjunto com uma notação gráfica específica, como a *UML*, se torna um meio de comunicação padronizado entre os membros da equipe.

---

[1] As modificações externas incluem fatores como modificações em requisitos funcionais, negócios, ambiente de hardware, custos, etc (PRESSMAN, 2001).

[2] Os modelos podem ser definidos como os principais artefatos da fase de análise e projeto.

No processo de desenvolvimento de software, verifica-se que as atividades relacionadas à manutenção e construção do software impõem a modificação de artefatos de software produzidos em diferentes fases. Ao modificar um requisito, modificam-se todos os artefatos relacionados a ele, desde a análise até sua implementação em código-fonte. No entanto, apesar do processo impor a modificação em artefatos de software situados em diferentes níveis de abstração e fases do processo, atualmente, as ferramentas de apoio à manutenção do software estão voltadas basicamente para o código-fonte. Para identificar as partes do código que devem ser modificadas, o engenheiro de software pode utilizar uma ferramenta que, estaticamente ou dinamicamente, analisa dependências entre artefatos (WEISER, 1984; AGRAWAL e HORGAN, 1990). Essa análise ajuda a localizar elementos específicos do código, mas não identifica todos os artefatos relevantes a uma modificação específica.

Por existir essa limitação, algumas abordagens (ZIMMERMANN et al., 2003a; YING, 2003; SHIRABAD, 2003) surgiram, tentando relacionar os elementos do código que devem ser alterados a partir da modificação de um dado artefato. No entanto, percebe-se que essas abordagens não são adequadas a uma modificação que deve ser refletida nos diferentes níveis de abstração do software. O foco limitado a código-fonte dificulta a manutenção e propicia a geração de uma documentação desatualizada e inconsistente em relação ao código-fonte.

A documentação deve ser utilizada para narrar diferentes fases do ciclo de vida do software. Na tentativa de explicar o comportamento do software, os modelos que formam a base da análise e do projeto adicionam informação ao entendimento do sistema. Os modelos de software antecipam e facilitam as modificações futuras, minimizando os erros nas fases de codificação e testes. Além disso, melhoram o entendimento do software por propor um nível de abstração mais elevado e por descrever aspectos não representados em código-fonte.

Considerando qualquer diagrama ou modelo presente na *UML*, percebe-se que durante a manutenção, as modificações realizadas em determinado modelo podem levar a modificações subseqüentes em outros artefatos do mesmo modelo *UML* ou em outros modelos *UML* relacionados. A propagação de uma modificação nos modelos tem os seguintes propósitos: (1) mantê-los atualizados e consistentes com o software, (2) estimar o que é necessário modificar, ajudando a análise de impacto na definição dos custos e da complexidade da modificação, e (3) determinar os artefatos que devem ser alterados anteriormente à implementação, o que permite que o engenheiro de software

avalie qual é a forma mais adequada de implementar a modificação. Ao considerar os efeitos de uma modificação nos modelos de análise e projeto, falhas que poderiam ser encontradas tardiamente no processo de desenvolvimento são antecipadas, o que é uma vantagem em relação às abordagens que propagam a modificação apenas em código-fonte (ZIMMERMANN et al., 2003a; YING, 2003; SHIRABAD, 2003).

Considerando o processo de desenvolvimento em suas diferentes fases, reparamos, ao partir de uma visão mais genérica e abstrata para uma visão mais específica do domínio do problema, que determinadas estruturas semânticas se repetem em diferentes fases do ciclo de vida do software. Isso significa que o mesmo conceito pode ser representado sob pontos de vista e detalhamento diferentes, dependendo da fase do processo.

Com o intuito de porporcionar uma maior reutilização dos artefatos, abordagens como o Desenvolvimento Baseado em Componentes surgiram, utilizando o componente como uma forma de uniformizar conceitos usados para representar as entidades do mundo real nos vários níveis de abstração.

1.1.1 – Desenvolvimento Baseado em Componentes

O Desenvolvimento Baseado em Componentes (DBC) surgiu como um paradigma de desenvolvimento voltado para a reutilização de software, fazendo uso de interfaces e conectores como elementos de estruturação de sistemas. Componentes, como um módulo coeso e reutilizável de software (D'SOUZA et al., 1998), fazem uso de interfaces descritas de forma contratual para definir seu comportamento com os demais artefatos do software (SZYPERSKY, 2002). Especialmente, a partir da *UML* 2.0 (Unified Modeling Language) (OMG, 2002b), os artefatos produzidos durante o processo de desenvolvimento de software, situados em diferentes níveis de abstração, estarão intimamente relacionados através do conceito de componente.

O componente é considerado uma parte independente do software (SZYPERSKI, 2002) e implementa uma ou mais interfaces, através das quais interage com outros componentes do software, formando unidades de software com capacidades específicas (PAGE-JONES, 1999).

Contudo, sempre que um componente é modificado, a modificação deve ser propagada para os demais artefatos do modelo. Devido à característica do paradigma DBC e ao fato do componente evoluir em relação à sua funcionalidade e à sua interface, dois tipos de relações se destacam: relações entre componentes e relações entre artefatos

presentes no projeto interno de cada componente. Quando a interface é modificada durante o processo de desenvolvimento, o engenheiro de software verifica as relações entre componentes. Quando a funcionalidade de um ou mais componentes é alterada, ele verifica as relações entre os diferentes artefatos *UML* que compõem o componente. Como os componentes se relacionam via interfaces, algumas dependências lógicas entre componentes podem não ser facilmente identificadas pelo engenheiro de software durante a manutenção. Por exemplo, se a classe de um componente estiver sendo frequentemente atualizada em conjunto com a classe de um segundo componente, existe uma dependência lógica entre dois artefatos *UML* que pertencem a componentes diferentes. Essa relação, por não estar documentada, não é facilmente identificada. O maior problema é que essa relação pode estar interferindo na coesão dos componentes, o que acentua a necessidade que já existia no desenvolvimento convencional, de que o software só poderá evoluir de forma consistente e com qualidade se for preservada a rastreabilidade entre os diferentes artefatos.

A rastreabilidade define relações que existem entre os diversos artefatos de um processo de desenvolvimento de software (CLELAND-HUANG et al., 2003). Na literatura, verificamos que as técnicas de rastreabilidade podem ser utilizadas para identificar todos os artefatos que devem ser atualizados quando uma modificação é introduzida, atuando em diferentes níveis de abstração e fases do processo de desenvolvimento do software, o que é fundamental nas atividades de manutenção.

1.2 – Caracterização do Problema

No processo de desenvolvimento de software, a complexidade de alteração de modelos de análise e projeto é alta (BRIAND et al., 2003). Quando o engenheiro de software modifica um modelo *UML*, ou realiza alguma alteração em código que influencia a reorganização do modelo, ele se questiona: “Quais outros artefatos do modelo devem ser alterados em virtude dessa modificação?”. Para responder a esse questionamento, é necessário encontrar relações entre artefatos *UML* que indiquem quais artefatos devem ser modificados em conjunto. Encontrar relações entre artefatos do software, sem nenhuma automatização ou mecanismo, é uma tarefa difícil, tanto durante a construção quanto na manutenção do software (SHIRABAD, 2003).

A análise das informações referentes à Gerência de Configuração de Software (GCS)[3] tem o intuito de dar subsídio para a melhoria do processo que está por trás das atividades de Engenharia de Software, provendo conhecimento indireto sobre o processo de desenvolvimento. Neste contexto, o uso de técnicas de mineração[4] de dados se mostra promissor, pois permite ir além das informações explícitas existentes nos repositórios dos sistemas da GCS. Através da análise das modificações e versões existentes no contexto dos sistemas de GCS, torna-se possível detectar automaticamente informações implícitas nos repositórios como relações semânticas entre artefatos do ambiente de desenvolvimento (BALL et al., 1997; YING, 2003; ZIMMERMANN et al., 2003a, SHIRABAD, 2003).

O uso de técnicas de mineração de dados no repositório dos sistemas da GCS pode apoiar a detecção de rastros de modificação[5], encontrando regras do tipo: "Desenvolvedores que modificam esses artefatos também modificam esses outros artefatos". Considerando os artefatos *UML*, encontra-se, por exemplo, relações entre classes e casos de uso.

No entanto, percebemos que essas relações não trazem nenhum significado semântico[6]. Para aumentar o entendimento do engenheiro de software, torna-se necessário coletar e organizar informações dos sistemas de GCS, de forma a obter um indício de como o rastro surgiu. Neste caso, pode ser utilizada a informação armazenada nos sistemas de controle de modificações que contém dados preenchidos durante a execução do processo. Por exemplo, no decorrer do processo, identifica-se no sistema de controle de modificações um novo pedido de alteração no software. Nessa requisição, declara-se em algum nível de obrigatoriedade, a descrição da modificação, prioridade e relevância da modificação. Esses dados podem ser capturados e relacionados aos rastros de modificação, trazendo um pouco da história que está por traz

---

[3] A Gerência de Configuração de Software (GCS) pode ser definida como uma abordagem disciplinada para gerenciar o processo de evolução dos sistemas (ESTUBLIER et al., 2002).

[4] As técnicas de mineração de dados podem ser aplicadas com o objetivo de extrair informações relevantes dos repositórios através de pesquisa e da determinação de padrões, classificações e associações entre elas (GOEBEL e GRUENWALD, 1999).

[5] No contexto desta dissertação, os rastros de modificação são rastros que surgiram a partir da evolução do software.

[6] A semântica ou significado atribuído ao rastro existe em função do raciocínio empregado pelo engenheiro de software na definição do rastro.

da indicação do rastro. Essa história, quando avaliada, fornece subsídio para a validação do próprio rastro e uma indicação da sua relevância para a modificação que estiver sendo efetuada.

1.3 – Objetivo

Tendo em vista os problemas e necessidades citados anteriormente, o objetivo desta dissertação é fornecer um mecanismo de detecção de rastros de modificação entre artefatos *UML*, que torne a manutenção do software, em especial no contexto de DBC, uma atividade menos complexa e mais automatizada.

Através da identificação automática de rastros de modificação realizada a partir da análise das informações existentes no contexto dos sistemas da GCS, pretende-se interferir o mínimo necessário na rotina de trabalho do engenheiro de software, proporcionando algum ganho de produtividade.

Ao indicar, no ambiente de desenvolvimento de DBC, rastros de modificação entre artefatos *UML*, este mecanismo, denominado *Odyssey-WI* (Wokspace Integration), propõe integrar os espaços de trabalho referentes à GCS e DBC através da mineração das informações existentes nos repositórios dos sistemas da GCS. No entanto, não deve se ater ao contexto de DBC, mas servir também ao desenvolvimento de software em geral.

O mecanismo deve identificar rastros de modificação entre artefatos *UML* apresentando as informações para análise do engenheiro de software. Isso significa que a partir da seleção de um artefato *UML*, o mecanismo deve ser capaz de detectar, automaticamente, rastros para outros artefatos *UML*, no auxílio às tarefas de modificação do engenheiro de software. Além disso, deve indicar, juntamente com o rastro, informações coletadas nos sistemas de GCS que forneçam algum significado semântico ao rastro, facilitando o entendimento do próprio rastro e suas futuras modificações.

Descobrir, através da mineração de dados, relações não triviais entre artefatos *UML* que precisam ser modificados em conjunto favorece o entendimento do software e minimiza certas complicações, como a introdução de novos erros e alterações incompletas durante as modificações (ZIMMERMANN et al., 2003a). O resultado da mineração pode apoiar tanto as atividades de GCS quanto de DBC, sendo útil, respectivamente, na análise de impacto das modificações e na detecção de falhas de projeto devido ao alto acoplamento entre artefatos não correlatos (ZIMMERMANN et

al., 2003b). O mecanismo auxilia a atividade de análise de impacto à medida que indica quais artefatos devem ser alterados a partir de uma modificação. A detecção de falhas de projeto é possível porque a identificação de rastros de modificação permite que o engenheiro de software visualize dependências que não foram, necessariamente, projetadas durante a concepção do software, mas que existem em função das modificações realizadas durante as atividades do processo de desenvolvimento.

1.4 – Contexto da Dissertação

O ambiente de desenvolvimento de software utilizado no contexto desta dissertação é o ambiente *Odyssey* (WERNER et al., 2003), que visa apoiar a reutilização de software através de modelos de domínio[7], Linha de Produto[8] e DBC. A reutilização de software no ambiente *Odyssey* ocorre através dos processos de Engenharia de Domínio, cujo objetivo é construir artefatos reutilizáveis voltados para problemas de domínios de conhecimento específicos, e de Engenharia de Aplicação que tem o objetivo de desenvolver aplicações, em um determinado domínio, reutilizando esses artefatos genéricos existentes. O ambiente *Odyssey* utiliza extensões de diagramas *UML* para representar o conhecimento de um domínio, e permite que esses diagramas sejam reutilizados para facilitar a construção de aplicações.

Neste ambiente, a rastreabilidade tem como objetivo facilitar a identificação dos artefatos do domínio em todos os níveis de abstração, ou seja, desde as visões de modelos conceituais e funcionais do domínio até as visões mais detalhadas, como modelos de casos de uso, classes e interação.

Neste contexto, o projeto *Odyssey-SCM* (MURTA, 2004) se propõe a prover uma abordagem de GCS voltada à evolução controlada de artefatos de alto nível de abstração e às atividades de DBC, visto que os sistemas atuais da GCS não atendem questões específicas de DBC (ESTUBLIER et al., 2002). O processo de DBC difere do processo convencional de desenvolvimento de software devido à adição de atividades relacionadas à reutilização e substituição de componentes existentes (MURTA, 2004).

---

[7] Modelo de domínio contém as características específicas do domínio, que são comuns a várias aplicações (BRAGA, 2000).

[8] Linha de produtos é um conjunto de sistemas que compartilham características gerenciáveis comuns, que satisfazem às necessidades específicas de um seguimento de mercado em particular e que são desenvolvidos sistematicamente, a partir de um conjunto comum de artefatos de software (CLEMENTS et al., 2001).

O projeto *Odyssey-SCM* visa ainda prover, um ferramental de apoio para que um maior controle na evolução dos artefatos reutilizáveis possa ser realizado.

No contexto do projeto *Odyssey-SCM*, o mecanismo *Odyssey-WI*, visa integrar os espaços de trabalho referentes à GCS e DBC, automatizando algumas tarefas relacionadas à manutenção do software.

1.5 – Organização dos Capítulos

O restante desta dissertação está dividida nos seguintes capítulos:

No **Capítulo 2**, é apresentado um estudo sobre rastreabilidade e GCS. Neste capítulo, avaliamos se é vantajosa a integração entre o ferramental para rastreabilidade e o processo de GCS como forma de obter a detecção automática de rastros de modificação entre artefatos *UML*.

No **Capítulo 3**, aprofundamos nosso conhecimento sobre as técnicas de mineração de dados, optando pela utilização de uma técnica que possa ser utilizada na descoberta dos rastros de modificação entre artefatos. A escolha de uma técnica depende da tarefa[9] específica a ser executada e dos dados disponíveis para a análise.

O **Capítulo 4** apresenta a proposta do mecanismo *Odyssey-WI*, o cenário de utilização desta proposta no apoio às tarefas de manutenção e construção do software, sua relação com os demais sistemas da GCS, além de suas principais características e funcionalidades.

No **Capítulo 5**, discutimos a implementação do protótipo *Odyssey-WI* desenvolvido no contexto do ambiente *Odyssey* e do projeto *Odyssey-SCM*, e algumas tecnologias que foram empregadas para o seu desenvolvimento, incluindo a arquitetura do qual faz parte. Ao final deste capítulo, apresentamos um exemplo de utilização do protótipo no contexto do ambiente *Odyssey*.

O **Capítulo 6** encerra a dissertação apresentando as suas principais contribuições e limitações, listando ainda os possíveis trabalhos futuros.

[9] Neste contexto, tarefa é o tipo de problema de descoberta de conhecimento a ser solucionado.

Capítulo 2 – Rastreabilidade através da Gerência de Configuração de Software

2.1 – Introdução

AMBLER (1999), de uma forma ampla, destaca que do ponto de vista da Engenharia de Software, a rastreabilidade entre os diferentes níveis de abstração (requisito, análise, projeto, código, testes e programa executável) contribui para alinhar o software às inevitáveis mudanças nas necessidades dos usuários, reduzir riscos através da captura de potenciais problemas, determinar o impacto da modificação no sistema e, conseqüentemente, proporcionar melhoria no processo.

A rastreabilidade desempenha um importante papel no processo de desenvolvimento de software, sendo amplamente reconhecida como um importante fator na gerência efetiva do desenvolvimento e da evolução do software. Mais ainda, é fundamental para o entendimento do sistema, para a análise de impacto e para a reutilização de software (DE LUCIA et al., 2004). No entanto, o sucesso da adoção de novos métodos e ferramentas no desenvolvimento de software, no que diz respeito a rastreabilidade, está intimamente relacionado com os benefícios providos e o esforço adicional necessário para atingir esses benefícios (MURTA, 2004).

A identificação e manutenção dos rastros são consideradas atividades custosas ao projeto. Por esse motivo, a adoção de mecanismos de rastreabilidade pode ser inviabilizada durante a execução das atividades de desenvolvimento de software caso não seja proposta uma forma automatizada ou semi-automatizada de obter os rastros. A solução proposta deve interferir o mínimo necessário no processo de desenvolvimento. Por esta razão, existe a necessidade de não aumentar a burocracia ao executar as atividades do desenvolvimento de software e fazer uso do mesmo ferramental que o engenheiro de software está habituado.

A Gerência de Configuração de Software (GCS) exerce um papel importante no controle da evolução do software, provendo conhecimento indireto sobre o processo de desenvolvimento. É uma disciplina indispensável para controlar a complexidade existente nos projetos de software, principalmente, quando equipes numerosas manipulam, concomitantemente, um conjunto de artefatos comuns. Neste contexto, os

repositórios dos sistemas da GCS são utilizados na descoberta de informações qualitativas e quantitativas sobre vários aspectos do desenvolvimento de software, principalmente, porque esses sistemas permitem o acompanhamento detalhado do andamento das modificações do projeto. Sendo assim, uma atenção especial deve ser despendida na integração entre o ferramental proposto para a rastreabilidade e os processos e sistemas da GCS.

Esse capítulo apresenta um estudo sobre rastreabilidade e GCS. Através desse estudo, é avaliada a necessidade da integração entre o ferramental para a rastreabilidade e os processos e sistemas da GCS na busca pela propagação de conhecimento entre a GCS e os ambientes de desenvolvimento de software. Neste contexto, a Seção 2.2 discute os vários aspectos da rastreabilidade, definindo a rastreabilidade em duas categorias: pré-rastreabilidade e pós-rastreabilidade. A Seção 2.3 apresenta um estudo sobre a análise de impacto, onde a pós-rastreabilidade se enquadra. A Seção 2.4 apresenta a GCS e a Seção 2.5 verifica as possibilidades de integração entre ambientes, i.e, entre o que é produzido através do processo de GCS e o que pode ser utilizado no processo de desenvolvimento de software.

2.2 – Rastreabilidade

A capacidade de rastreabilidade é vista como um fator de qualidade em software – uma característica que os sistemas devem apresentar como requisito não funcional (RAMESH e JARKE, 2000). Sua importância, na evolução dos sistemas, é um fato amplamente reconhecido e recomendado por diversas normas, como DoD 2167a, em 1980, que foi substituída por MIL-STD-498, anos mais tarde (DoD, 1988). Entretanto, as normas oferecem pouca orientação, os modelos e mecanismos são diversos e freqüentemente pouco compreendidos, e os modelos existentes são simples ou rígidos para lidar com uma ampla variedade de estratégias que são aplicadas na indústria (RAMESH e JARKE, 2000).

De uma perspectiva cognitiva, rastros identificam elementos conceituais relacionados de uma forma significativa (RAMESH e JARKE, 2000). Sendo assim, a rastreabilidade define relacionamentos que existem entre os diversos artefatos de um processo de desenvolvimento de software (CLELAND-HUANG et al., 2003). Artefatos podem ser requisitos, módulos do código-fonte, casos de uso, classes, casos de testes, entidades que representem características e comportamentos de um sistema, entre outros.

A identificação dos rastros pode interferir diretamente na execução das atividades do desenvolvimento de software. A capacidade de registrar rastros entre elementos está relacionada ao que os rastros devem representar, quais são os elementos de rastreabilidade envolvidos e sob que condições os rastros devem ser registrados (PINHEIRO, 2000). Portanto, os rastros devem estar organizados sob algum modelo.

Atualmente, apesar dos modelos tenderem a registrar ao máximo as informações sobre o processo de desenvolvimento, relacionando-as para futura recuperação, eles não levam em consideração que para sua efetiva utilização é necessário uma discriminação de que tipo de informação os engenheiros de software precisarão acessar de forma mais relevante (RAMESH e JARKE, 2000). A literatura atual propõe mecanismos para representar diferentes tipos de relacionamentos entre artefatos, mas deixa a cargo do engenheiro de software a interpretação dos significados desses relacionamentos.

A maioria das abordagens (GOTEL e FINKELSTEIN, 1995; YING, 2003; SHIRABAD, 2003; ZIMMERMANN et al., 2003a) identifica que existe um relacionamento entre duas entidades sem ser capaz de especificar a razão deste relacionamento. A semântica ou significado atribuído ao rastro existe em função do raciocínio empregado pelo engenheiro de software na definição do rastro. A perda da semântica resulta em uma identificação subjetiva. Além disso, o conhecimento de como e por que as entidades estão relacionadas é perdido.

GOTEL e FINKELSTEIN (1994) conduziram uma extensa pesquisa sobre os problemas que afetam a rastreabilidade, identificando alguns fatores que contribuem para isso. Dentre os fatores, podemos citar: (1) métodos e práticas de desenvolvimento informais; (2) recursos insuficientes e falta de disponibilidade para o suporte a rastreabilidade; (3) ausência de clareza quanto aos papéis desempenhados na prática da rastreabilidade; (4) ausência de cooperação ou coordenação entre pessoas responsáveis por diferentes artefatos; (5) perspectiva de um custo maior em relação ao benefício na identificação e manutenção da rastreabilidade; (6) ausência de informação no decorrer da prática da rastreabilidade e (7) ausência de treinamento que torne a prática da rastreabilidade uma realidade.

Os rastros podem ser representados através de um ou mais atributos que descrevam características do relacionamento. Eles podem ser representados através de matrizes, hipertextos e técnicas baseadas em grafos. No entanto, apesar das diferentes formas de representação, o problema mais comum e que persiste até os dias atuais é a identificação e a manutenção dos rastros.

O processo de coleta de relações entre artefatos de software é continuo e se passa durante todo o desenvolvimento do projeto. A cada novo artefato adicionado em uma fase ou iteração do ciclo de vida, novas relações devem ser adicionadas para assegurar a consistência da informação. À medida que o sistema evolui, uma das maiores dificuldades é manter a infra-estrutura construída em prol da rastreabilidade atualizada e consistente com o estado atual do software. Esse tipo de dificuldade é uma realidade, principalmente quando existem restrições de custo e prazo nos projetos (CLELAND-HUANG et al., 2003).

Na literatura, alguns métodos de identificação e manutenção de rastros são manuais (EGYED, 2001; HAUMER et al., 1999), e em alguns casos, semi-automatizado (ANTONIOL et al., 1999; PINHEIRO e GOGUEN, 1996; POHL, 1996; SMITH et al., 2003). A identificação manual dos rastros é custosa e propensa à falhas, além de ser um dos problemas freqüentemente reportados na literatura (CLELAND-HUANG et al., 2003) devido a natureza iterativa do processo de desenvolvimento do software.

Existem duas formas de rastreabilidade (GOTEL e FINKELSTEIN, 1994) que serão detalhadas a seguir: pós-rastreabilidade e pré-rastreabilidade. As técnicas que facilitam a pós-rastreabilidade não são adequadas ao problema da pré-rastreabilidade e vice-versa, existindo, por esse motivo, a necessidade de separar as definições. A ausência de uma definição que aborde todos os aspectos que fazem da rastreabilidade uma área de problema ainda em aberto, demonstra que na literatura não existe um consenso do que este problema representa.

2.2.1 – Rastros de Pré-Especificação

A rastreabilidade de pré-especificação, ou pré-rastreabilidade, ou pré-rastros, está relacionada com as origens dos requisitos, produção e refinamento dos mesmos, anteriormente à especificação (GOTEL e FINKELSTEIN, 1994).

Na pré-rastreabilidade, existe a necessidade de técnicas que armazenem e recuperem informações referentes à produção e revisão dos requisitos com o intuito de facilitar o entendimento de outros profissionais. A maior dificuldade no emprego das técnicas e na engenharia de requisitos é o relacionamento entre requisito, sua origem e o raciocínio empregado na sua obtenção.

POHL (1996) descreve um protótipo de ambiente para a pré-rastreabilidade baseado em um framework tri-dimensional, onde o processo de desenvolvimento da especificação de requisitos pode ser descrito através de três dimensões ortogonais:

(1) Dimensão de representação: diz respeito à representação física da informação utilizada no processo como documentos, especificações e políticas que podem estar em diferentes níveis de formalidade e formatos;

(2) Dimensão de especificação: representa o conteúdo da especificação independente da representação, definido de acordo com guias, padrões ou modelos de domínio;

(3) Dimensão de concordância: representa os diferentes pontos de vista dos responsáveis pelos requisitos, incluindo as alternativas, as argumentações, e as decisões necessárias ao entendimento comum feitas durante a engenharia de requisitos.

O protótipo PRO-ART (POHL, 1996) contém um repositório de rastros para estruturar a informação e proporcionar a recuperação seletiva. Por ser um ambiente de engenharia de requisitos centrado em processos, PRO-ART, é composto de: (1) um ambiente de modelagem, no qual os processos são definidos e modelados, (2) um ambiente de instanciação, onde o modelo instanciado é interpretado para fins de controle e monitoramento e (3) um ambiente de gerência, onde as atividades são de fato implementadas. A identificação e o armazenamento dos rastros são acompanhados pelo ambiente de execução do processo, uma vez que as dependências criadas entre os artefatos consumidos e os artefatos produzidos nas atividades dependem da forma como o processo é realizado. Desta forma, PRO-ART resolve problemas reportados por GOTEL e FINKELSTEIN (1994), como a ausência de treinamento e de procedimentos para a prática da rastreabilidade.

RAMESH e JARKE (2000), com base em um estudo experimental, analisaram os tipos de objetos e associações usados na prática da engenharia de requisitos e organizaram um modelo de referência a partir das necessidades encontradas. O estudo revela que a eficácia e a efetividade no suporte à pré-rastreabilidade é determinada pelos tipos de objetos e associações disponíveis.

O modelo de referência de RAMESH e JARKE (2000) cobre as três dimensões da engenharia de requisitos mencionadas por POHL (1996). O modelo de referência é apresentado no primeiro modelo da Figura 2.1.

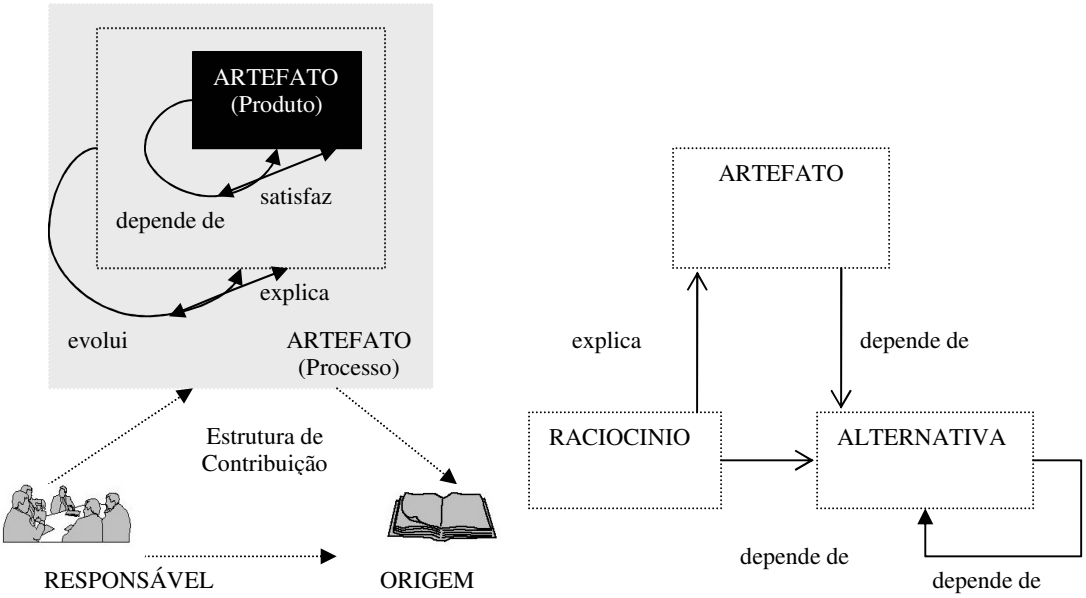

Figura 2.1 - Modelo de referência e Modelo de raciocínio (Fonte: RAMESH e JARKE, 2000)**.**

O modelo de referência de RAMESH e JARKE (2000) representa os tipos de artefatos que são consumidos ou produzidos no processo de desenvolvimento de software, i.e, diferentes elementos conceituais do ciclo de vida do software. A rastreabilidade entre os artefatos é definida através de quatro tipos de relacionamentos. Alguns estão voltados para o produto por descrever propriedades e relacionamentos entre artefatos, independente de como foram criados. Outros relacionamentos estão voltados para as ações realizadas durante o processo, indicando as evoluções e as razões que motivaram a criação ou modificação dos artefatos existentes.

Para representar as razões que motivaram a criação ou modificação dos artefatos, um modelo de raciocínio pode ser definido como na Figura 2.1, mencionando idéias, alternativas, argumentos de oposição ou consenso. Esse modelo pode ser representado através do relacionamento entre elementos e/ou derivações das classes definidas no modelo de referência. Essa característica diferencia esta abordagem das existentes como, por exemplo, RD-100 (ALFORD, 1991), que definem um *framework* uniforme (tal como IBIS[10]) para a representação do raciocínio. Nestas ferramentas, geralmente, o resultado é um esquema simples, que não detalha suficientemente os aspectos críticos do desenvolvimento de sistemas, ou um esquema complexo, que não

[10] Do inglês, *Issue Based Information Systems* (CONKLIN e BEGEMAN, 1988).

incentiva a sua utilização em todos os estágios do desenvolvimento. Diferentemente dessas abordagens, RAMESH e JARKE (2000) propõem a captura do raciocínio em deferentes níveis de detalhamento a partir de um *framework* flexível que pode ser modelado a partir do modelo de referência.

Como grande parte da informação relativa a pré-rastreabilidade é encontrada na forma de textos, diagramas e linguagem natural, PINHEIRO (2000) propõe que o *framework* provido pelas ferramentas atuais seja estendido para suportar informações não-estruturadas, em suas várias formas de representação. A partir da visão de que a informação informal é determinante nas fases iniciais do desenvolvimento, conclui que esta, apesar de complexa e dependente do contexto, exerce grande influência na engenharia de requisitos, e sua redução em favor de estruturas formais e pré-definidas pode não ser adequada.

Entretanto, como não é possível registrar corretamente em um modelo todas as relações necessárias a futuros rastros, PINHEIRO (2000) propõe a ferramenta TOORS. TOORS define um modelo abstrato composto de três linguagens: uma linguagem para definir estruturas de objetos, outra para relacionamentos e a última para expressões, cobrindo a possibilidade de registrar informações através de relações não previstas anteriormente.

A pré-rastreabilidade tem como um dos propósitos a criação e a manutenção dos relacionamentos que existem entre elementos conceituais do software e os agentes do processo de desenvolvimento responsáveis por tais elementos. As abordagens de RAMESH e JARKE (2000) e POHL (1996) mencionam a importância dos envolvidos no processo de desenvolvimento para a definição da pré-rastreabilidade. Neste ponto, a pré-rastreabilidade favorece o processo de gerência de conhecimento. A gerência de conhecimento envolve armazenamento, disseminação e atualização do conhecimento tácito e explícito da organização. A pré-rastreabilidade atua como um meio para criar, armazenar, recuperar, transferir e aplicar conhecimento nas organizações de desenvolvimento de software (RAMESH, 2002).

As abordagens para pré-rastreabilidade, que são (semi-) automáticas, são baseadas em palavras-chave e métodos de recuperação da informação. Neste caso, esses métodos aparecem como um auxílio no processo de identificação de rastros (HAYES et al., 2003).

2.2.2 – Rastros de Pós-Especificação

A rastreabilidade de pós-especificação, ou pós-rastreabilidade, ou pós-rastros, está relacionada com a evolução do requisito nas fases do ciclo de vida, a partir da especificação, considerando os demais níveis de abstração e representação, até sua implementação em código (GOTEL e FINKELSTEIN, 1994). A Figura 2.2 apresenta as diferenças entre a pós-rastreabilidade e a pré-rastreabilidade.

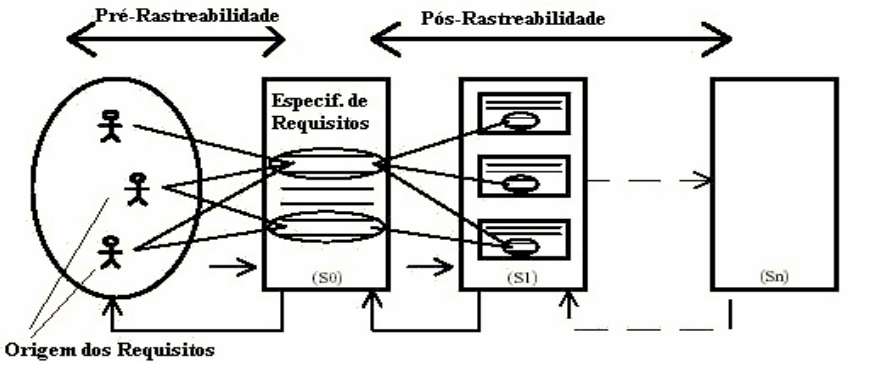

Figura 2.2 - Pré e Pós Rastreabilidade (Fonte: LINDVALL e SANDAHL, 1996).

Dentre os pós-rastros, dois tipos se destacam (LINDVALL e SANDAHL, 1996): intramodelos e intermodelos. Rastro intramodelo ou rastreabilidade vertical equivale a rastros de diferentes elementos conceituais em um mesmo nível de abstração e, rastro intermodelo ou rastreabilidade horizontal corresponde a rastros de um mesmo elemento conceitual em diferentes níveis de abstração. A Figura 2.3 exemplifica a rastreabilidade vertical e horizontal a partir das definições.

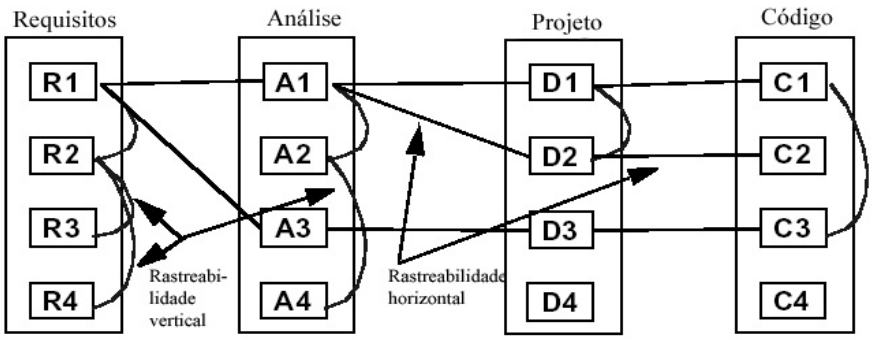

Figura 2.3 - Rastreabilidade vertical (intramodelos) e horizontal (intermodelos) (Fonte: LINDVALL e SANDAHL, 1996).

Freqüentemente, não é trivial a relação entre diferentes artefatos do sistema (SHIRABAD, 2003). Ao modificar artefatos de software, tais como funções, classes,

componentes, o engenheiro de software deve estar atento à cobertura da modificação. O rastro derivado da modificação do software é definido nesta dissertação como **rastro de modificação**. Nesse contexto, o rastro de modificação diz respeito ao pós-rastro intramodelo e intermodelo que surgiu a partir da evolução do software.

Esses rastros são utilizados para detectar conjuntos de modificações e guiar o desenvolvedor na implementação desses conjuntos, pois sempre que uma modificação é implementada, a própria implementação da modificação pode incentivar o surgimento de novas modificações, como exibido na Figura 2.4.

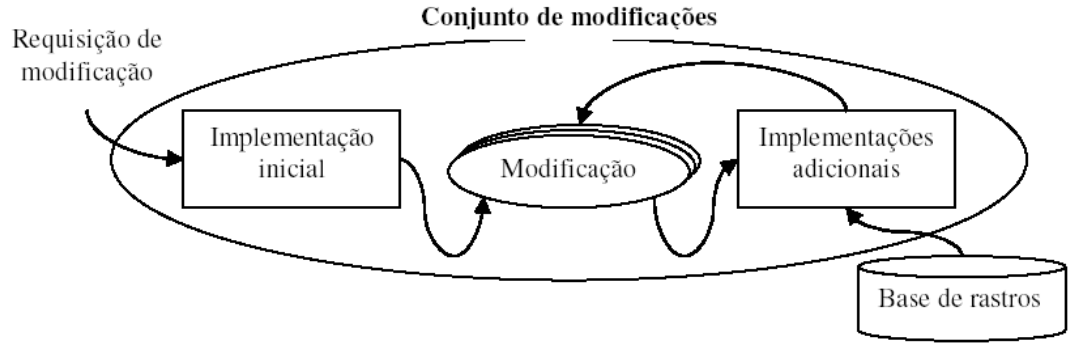

Figura 2.4 - Detecção de conjuntos de modificações

Independente da origem, a modificação introduzida em um artefato do projeto afeta, usualmente, outras partes do software. Os requisitos do projeto podem mudar por inúmeras razões. No contexto do desenvolvimento de software, as origens da modificação podem ser externas ou internas. As modificações externas incluem fatores como modificação em requisitos funcionais, negócios, ambiente de hardware, custos, etc. As modificações internas podem ser direcionadas por necessidade de reestruturação do código ou projeto, assim como devido a existência de erros (KOWALCZYKIEWICZ e WEISS, 2002). Devido às pressões de mercado, adaptação à legislação e necessidades de melhorias, as alterações no software são necessárias e podem ser classificadas em: evolutivas, corretivas, adaptativas e preventivas (PRESSMAN, 2001).

Durante a manutenção, a compreensão que se tem do software se torna um ponto crucial. A experiência e o conhecimento do engenheiro de software sobre o sistema são fontes de informação importantes, sendo, na prática, as mais utilizadas ao conduzir a análise de impacto da modificação (LINDVALL e SANDAHL, 1996). No entanto, a rotatividade da equipe faz com que o conhecimento tácito de alguns profissionais seja perdido com facilidade. Conseqüentemente, o engenheiro de software, ao realizar a

análise de impacto, tem dificuldade em encontrar relações entre artefatos através de um processo não sistemático, sem realizar uma profunda análise do sistema. Isso o torna incapaz de observar as relações entre artefatos que uma modificação pode vir a introduzir (KOWALCZYKIEWICZ e WEISS, 2002).

A pós-rastreabilidade, por abranger um conjunto de artefatos que podem estar em diferentes níveis de abstração e em diferentes fases do processo de desenvolvimento do software, favorece a análise de impacto precisa (LINDVALL e SANDAHL, 1996). Os pós-rastros são utilizados para identificar todos os artefatos que devem ser atualizados a partir da introdução de uma modificação. Entretanto, independentemente do tipo de pós-rastro, a sua detecção é considerada um problema ainda em aberto na Engenharia de Software, devido à complexidade envolvida no descobrimento dos reais motivos para o surgimento desses rastros.

A seguir, é apresentado um estudo mais aprofundado sobre a análise de impacto.

2.3 – Análise de Impacto

A análise de impacto pode ser definida como o processo de identificação de possíveis conseqüências de uma modificação no software (BOHNER, 2002). A partir da análise do impacto da modificação no software, estima-se o custo da implementação.

Segundo BOHNER (2002), o processo de análise de impacto (Figura 2.5) é iterativo.

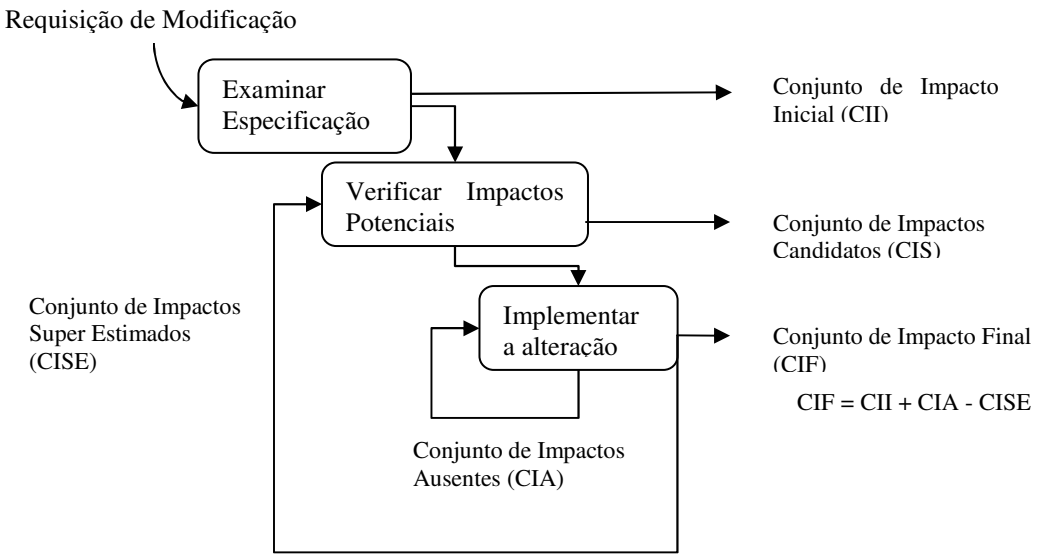

Figura 2.5 - Processo de análise de impacto

À medida que a modificação é implementada, novas modificações são descobertas, sendo que os impactos podem ser categorizados como: (1) impacto direto e (2) impacto indireto. O impacto direto corresponde ao primeiro nível de impacto obtido do grafo de dependências e ocorre quando o objeto que sofre o impacto da modificação está diretamente relacionado ao causador. O impacto indireto ocorre quando o objeto que sofre o impacto está relacionado ao causador através de um número inteiro n (onde $0 < n < \infty$) de relacionamentos intermediários, sendo por isso também denominado impacto de nível n.

O processo da Figura 2.5 se resume a examinar a requisição de modificação para encontrar um conjunto inicial de artefatos que devem ser alterados. Em seguida, verificar quais são candidatos à alteração em decorrência da alteração dos artefatos encontrados na atividade anterior. E, após esse passo, finalmente, implementar a requisição de modificação. No entanto, ao implementar, verifica-se impactos que foram super estimados, indicando relações incorretas e impactos ausentes que não foram mencionados no processo de análise. De acordo com BOHNER (2002), é preciso aumentar a acurácia no processo, encontrando um número maior de impactos válidos e menor de impactos incorretos.

O uso das relações armazenadas em um ambiente de desenvolvimento de software pode vir a automatizar a propagação da mudança e notificação de usuários. O mecanismo de notificação, como exemplificado na Figura 2.6, proporciona a distribuição do conhecimento sobre a mudança aos interessados. Ao mesmo tempo, identifica todos os artefatos nos diferentes níveis de abstração que sofrerão o impacto da mudança, facilitando a previsão de custo e tempo de realização (KOWALCZYKIEWICZ e WEISS, 2002).

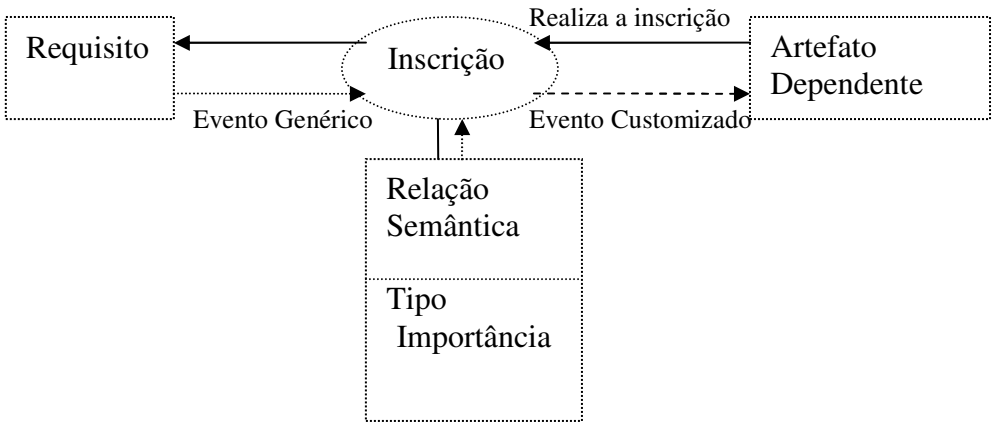

Figura 2.6 - Mecanismo de notificação de dependências

Quando uma modificação ocorre, o gerente de requisitos publica uma notificação genérica que contém informações sobre a estrutura e a semântica da modificação. O servidor de eventos customiza a mensagem genérica enviando para cada artefato dependente uma notificação de atualização. A semântica envolvida na publicação do evento de notificação armazena informações como tipo, descrição, razão, responsável e importância da modificação. A importância classifica a alteração em termos de prioridade e serve como um filtro na notificação aos interessados (CLELAND-HUANG et al., 2003).

O mecanismo de notificação resolve um dos problemas reportados por GOTEL e FINKELSTEIN (1994) no que se refere à ausência de cooperação e coordenação entre pessoas responsáveis por diferentes artefatos, o que aumenta a dificuldade de manutenção dos rastros em um estado consistente. Através desse mecanismo, os rastros são mantidos consistentes em função da coordenação na atualização de todos os artefatos influenciados por uma modificação.

Podem ser verificadas diferentes formas de trabalho para automatizar, pelo menos de forma parcial, a atividade de análise de impacto. KNETHEN e GRUND (2003) apresentam uma ferramenta denominada QuaTrace que fornece suporte semi-automático para a análise de impacto, fazendo uso de rastros entre artefatos. A QuaTrace atua apoiando a análise de impacto referente à própria modificação e às modificações originadas a partir da propagação da modificação principal. A abordagem apóia diferentes papéis no desenvolvimento de software.

Entretanto, o uso de modelos de análise e projeto em *UML* (OMG, 2003b) leva a um certo número de modelos interindependentes. Por ser a pesquisa em análise de impacto em grande maioria voltada para código fonte, BRIAND et al. (2003) definem uma abordagem para análise da propagação do impacto das modificações já efetuadas em modelos *UML*. Para atingir esse objetivo, o modelo *UML* construído para contemplar a alteração é comparado com o modelo *UML* original, fazendo uso de regras descritas em *OCL* (Object Constraint Language) (OMG, 2003b). As regras *OCL* para auxílio à analise de impacto são agrupadas em três categorias: regras de consistência, regras de classificação e regras de impacto. As regras de consistência têm por objetivo verificar se os modelos estão consistentes com relação à especificação da *UML*. Sem essa avaliação não seria possível confiar na qualidade da análise de impacto gerada. As regras de classificação têm por objetivo detectar o tipo de modificação em questão. E,

finalmente, as regras de impacto têm por objetivo identificar, para cada tipo possível de modificação, os impactos existentes.

Diferentemente de BRIAND et al. (2003), que propõem uma análise sintática para identificar impactos de modificação, LETELIER (2002) propõe um meta-modelo para rastreabilidade baseado na *UML*. Essa abordagem é independente de processo e integra especificações textuais (requisitos, por exemplo) com especificações *UML* utilizando o próprio contexto da *UML*. Do ponto de vista da Engenharia de Requisitos, o meta-modelo definido por LETELIER (2002) oferece maior facilidade de adaptação às necessidades dos projetos, uma vez que faz uso do mecanismo de extensões provido pela *UML*. A abordagem de análise sintática para identificar impactos não é tão flexível, e, apesar de exaustiva, não é abrangente, visto que não considera todos os modelos da *UML* e suas extensões.

Geralmente, as abordagens (RAMESH e JARKE, 2000; POHL, 1996; HAN, 1996) não consideram no modelo de rastreabilidade todos os documentos influenciados por uma requisição de modificação, direcionando a abordagem para relacionamentos entre tipos de entidades. Em virtude da complexidade associada aos diversos tipos de entidades e relacionamentos e da ausência de uma definição precisa dos elementos, a aplicação de meta-modelos na prática se torna difícil (LETELIER, 2002). Como conseqüência, o modelo incompleto de rastros gera uma análise de impactos não precisa e incompleta. Segundo BOHNER (2002), a granularidade das entidades gerenciadas pelas ferramentas de rastreabilidade disponíveis é insuficiente para uma análise de impacto precisa.

RAMESH e JARKE (2000) definiram um meta-modelo para a rastreabilidade, com foco em Engenharia de Requisitos, representando as diferentes classes e relacionamentos com base no modelo de referência apresentado na Seção 2.2.1. O modelo completo apresenta 31 tipos de entidades (meta-classes no meta-modelo) e 50 tipos de relacionamentos. Geralmente, o que se percebe é que as abordagens apresentam um modelo de rastreabilidade que deve ser definido pelo usuário com granularidade alta, não considerando todo o software. O modelo de rastreabilidade de KNETHEN (2001) identifica tipos de entidade no nível dos requisitos do sistema e mostra como relacionar essas entidades a outras no mesmo nível e em diferentes níveis de abstração. Dependendo da abstração na representação contida em uma requisição de modificação, o impacto é parte de diferentes abstrações e visões do sistema. O impacto abrange

requisitos, entidades referentes a diagramas em diferentes níveis de abstração e código-fonte.

Nesta seção, foram apresentadas várias abordagens para automatizar a atividade de análise de impacto. Dentre as abordagens estudadas, constatamos que existe na literatura a intenção de verificar o impacto causado por uma modificação em artefatos de análise e projeto, não ficando restrito a código-fonte. Além disso, existe a iniciativa em criar ferramental de suporte a análise de impacto fazendo com que a atividade seja automatizada.

A análise de impacto é considerada uma das atividades mais importantes no processo de GCS. A GCS formaliza todas as atividades que dizem respeito à introdução de uma modificação, sendo definida como uma abordagem disciplinada para gerenciar o processo de evolução dos sistemas, como discutido a seguir.

2.4 – Gerência de Configuração de Software

Os sistemas de software estão em constante evolução (LEHMAN, 1997). A partir dos anos 60 e 70, a Gerência de Configuração (GC) passou a considerar artefatos de software, indo além dos artefatos de hardware, formando a GCS. Mas somente no final dos anos 80, com o surgimento do padrão IEEE Std 1042 (IEEE, 1987)[11], e em meados dos anos 90, com a norma ISO 10007 (ISO, 1995)[12], a GCS foi incorporada no processo de desenvolvimento de software das organizações em geral.

Desde então, a GCS tem sido reconhecida pelas organizações como um importante fator de sucesso, influenciando a qualidade do produto, o ambiente de desenvolvimento, os usuários e a própria organização. É aplicada durante todo o ciclo de vida do software, desde a criação do software, desenvolvimento, liberação até a manutenção (DART, 1991).

A GCS surgiu da necessidade de métodos e ferramentas para maximizar a produtividade e minimizar os erros, visto os problemas relacionados à comunicação devido ao aumento da equipe e crescimento dos sistemas em tamanho e complexidade. A impossibilidade de determinar o que foi modificado, assim como quem, porque e quando foram efetuadas as modificações acentuou essa necessidade, uma vez que para

[11] IEEE 1042 (1987) é considerada a norma internacional mais completa da GCS (MURTA, 2004).

[12] ISO 10007 (1995) fornece diretrizes para a utilização da GCS na indústria, definindo também a interface de GCS com as demais áreas de gerência (MURTA, 2004).

preservar a reputação da empresa era vital a diminuição do tempo de desenvolvimento e agilizar a correção de erros (LEON, 2000).

A sua atuação ocorre como processo auxiliar de controle e acompanhamento das atividades do processo de desenvolvimento de software, sendo uma disciplina que controla e notifica as inúmeras correções, extensões e adaptações aplicadas durante o ciclo de vida do software de forma a assegurar um processo de desenvolvimento sistemático e rastreável (ESTUBLIER et al., 2002). Segundo o padrão IEEE (IEEE, 1990), a definição da GCS inclui quatro funções, sob a perspectiva gerencial: identificação da configuração, controle da configuração, acompanhamento da configuração e auditoria da configuração.

A função de **identificação da configuração** tem por objetivo possibilitar a seleção e identificação de itens de configuração (ICs) de software[13] através de: (1) descrição física e funcional e (2) um esquema de nomes e números que garante a identificação inequívoca de ICs no grafo de versões[14] e variantes[15].

A função de **controle da configuração** tem como objetivo acompanhar, de forma controlada, a evolução dos ICs selecionados na função anterior, estabelecendo as seguintes atividades: (1) requisição da modificação, iniciando um ciclo da função de controle, dado um pedido de manutenção; (2) classificação da modificação, que estabelece a prioridade do pedido em função dos demais, de acordo com a criticalidade e o nível de importância ou relevância; (3) análise da modificação, que visa relatar os impactos em esforço, cronograma e custos, definindo uma proposta de implementação da manutenção; (4) avaliação da modificação pelo Comitê de Controle da Configuração[16], que estabelece se o pedido será implementado, postergado ou rejeitado; (5) implementação da modificação, caso o pedido de modificação tenha sido aprovado na atividade anterior, utilizando os mecanismos de *check-out*[17] e *check-in*[18]; (6)

---

[13] O termo item de configuração (IC) representa a agregação de hardware, software ou ambos, tratada pela GCS como elemento único (IEEE, 1990).

[14] O termo versão representa o estado do IC em um determinado momento do desenvolvimento de software (LEON, 2000).

[15] O termo variante representa uma versão funcionalmente equivalente a outra, mas projetada para ambiente de hardware ou software distintos (LEON, 2000).

[16] Do inglês, *Configuration Control Board* (CCB) (IEEE, 1990).

[17] O termo *check-out* representa o processo de requisição, aprovação e cópia de ICs do repositório para o espaço de trabalho do desenvolvedor (LEON, 2000).

verificação da modificação, que, através dos testes de sistema, valida a proposta de implementação levantada na análise de impacto; e (7) geração de configuração de referência[19], que pode ser liberada[20] para o cliente em função da sua importância e questões de marketing associadas.

A análise de impacto é considerada uma das atividades mais importantes no processo de GCS. A partir dos laudos elaborados nessa atividade é possível tomar as decisões referentes à aprovação ou não dos pedidos de modificação.

A função de **acompanhamento da configuração** permite que as informações geradas nas funções anteriores sejam armazenadas e acessadas, em função de necessidades específicas, através de relatórios gerenciais, de estimativas do projeto ou da aplicação de métricas voltadas para a melhoria do processo.

A função de **auditoria da configuração** ocorre quando a configuração de referência gerada na função de controle da configuração é selecionada para ser liberada para o cliente. Esta função abrange as atividades de: (1) auditoria funcional, realizada através da revisão de planos, dados, metodologia e resultados de testes, assegurando que o que foi especificado foi de fato cumprido; e (2) auditoria física, que certifica a completude em termos do que ficou acertado em cláusulas contratuais.

Contudo, sob a perspectiva de desenvolvimento, a GCS é dividida em três sistemas principais (MURTA, 2004): controle de modificações, controle de versões e controle de construções[21] e liberações.

O sistema de controle de modificações executa a função de controle da configuração de forma sistemática, armazenando todas as modificações geradas durante o processo de desenvolvimento de software. As informações das modificações realizadas são relatadas aos participantes interessados e autorizados conforme definido na função de acompanhamento da configuração.

---

[18] O termo *check-in* representa o processo de revisão, aprovação e cópia de ICs do espaço de trabalho do desenvolvedor para o repositório de versões (LEON, 2000).

[19] O termo configuração de referência (*baseline*) representa um conjunto de ICs formalmente aprovados, que serve de base para as etapas seguintes de desenvolvimento (IEEE, 1990).

[20] O termo liberação (*release*) representa a notificação e liberação formal de uma configuração de referência do software para o cliente (IEEE, 1990).

[21] O termo construção (*building*) representa o procedimento de geração do sistema para uma configuração alvo (LEON, 2000).

O sistema de controle de versões permite que os ICs sejam identificados conforme a função de identificação da configuração estabelece, e que evoluam de forma distribuída, concorrente e disciplinada, não corrompendo o sistema caso diversas requisições de modificação sejam tratadas em paralelo.

O sistema de controle de construções e liberações do produto automatiza o complexo processo de transformação dos diversos artefatos de software que compõe um projeto no sistema executável propriamente dito, de forma aderente aos processos, normas, procedimentos, políticas e padrões definidos para o projeto. Além disso, estruturam as configurações de referência selecionadas para a liberação, conforme necessário para a execução da função de auditoria da configuração.

Para que estes recursos sejam efetivamente utilizados, alguns serviços de integração de espaços de trabalho entre GCS e Ambientes de Desenvolvimento de Software devem estar presentes. Essas técnicas visam prover ao desenvolvedor de software um ambiente que forneça os recursos de GCS sem afetar de forma profunda a sua rotina de trabalho (MURTA, 2004), conforme discutido a seguir.

2.5 – Integração dos Espaços de Trabalho

As atividades da Engenharia de Software lidam com problemas complexos que podem ser agravados se forem acrescentadas as preocupações referentes ao controle de versões, modificações e construções e liberações. Em função dessa complexidade, diferentes técnicas de integração dos espaços de trabalho são utilizadas. Alguns ambientes de programação (*IDE* – Integrated Development Environment) integram o controle de versões e construções. *Eclipse* (ECLIPSE FOUNDATION, 2004) e *NetBeans* (NETBEANS COMMUNITY, 2004) são exemplos de *IDEs* que incorporam o controle de versões, permitindo que o desenvolvedor acesse versões de arquivos de código-fonte sem ter que sair do *IDE*. O mesmo acontece com controle de construções, onde Ant (ANT, 2005) é um exemplo amplamente adotado em *IDEs* voltados à programação Java.

Um outro aspecto de integração poderia ser a apresentação das características do produto ou processo no espaço de trabalho do desenvolvedor. Por armazenar, através das modificações, aspectos interessantes referentes à evolução do software, os sistemas de GCS são considerados fontes de dados para técnicas de análise retrospectiva. Vários autores (BALL et al., 1997; EICK et al., 2001; GRAVES et al., 2000; MOCKUS et

al., 2003) exploram os repositórios de sistemas de GCS analisando propriedades do processo de desenvolvimento.

No contexto do processo de desenvolvimento de software, os sistemas de GCS disponibilizam dados que podem ser utilizados na descoberta de informações qualitativas e quantitativas sobre vários aspectos do desenvolvimento de software (MOCKUS e VOTTA, 2000). De acordo com GRAVES et al. (2000) e a partir destes dados, é possível obter medidas que estimem a probabilidade de falha do software. As medidas de processo como número de vezes que o código foi alterado são mais úteis para estimar a taxa de falha do que as medidas de produto baseadas em linhas de código. Esse é o motivo da não aplicabilidade de algumas métricas de complexidade de software nesse contexto e da preferência do uso de outras medidas como: potencial de falha no módulo[22], freqüência de modificações por módulo, esforço de efetuar uma alteração e acoplamento entre módulos.

Alguns trabalhos na literatura (GRAVES et al., 2000; GRAVES e MOCKUS, 1998) propõem estimar o esforço da modificação no software a partir de variáveis como tempo, tamanho, proprietário e descrição do propósito da modificação. A partir da análise das alterações individuais realizadas no software é possível verificar a influência de alguns fatores no processo de desenvolvimento cuja contribuição não pode ser estimada no nível de produto, visto que o produto agrega vários tipos de modificações. Utilizando os dados da GCS, são obtidas estimativas de esforço úteis na identificação de módulos complexos que apresentam um alto custo de modificação e por isso são candidatos à reestruturação, ou módulos cuja alteração é dependente da experiência de diferentes desenvolvedores.

EICK et al. (2001) definem que a dificuldade de modificar o código caracteriza a sua depreciação. Com base nisso, são definidos índices de depreciação que identificam fatores de risco. Para cumprir esse propósito, são utilizadas medidas que possam refletir o custo, o tempo de modificação e a qualidade do código gerado. Além disso, é utilizado o histórico das modificações realizadas nos últimos anos em um projeto de software. Caso os índices identifiquem a depreciação do código, a manutenção preventiva[23] é considerada por eles uma possível solução.

---

[22] O termo módulo referencia uma coleção de arquivos do sistema (GRAVES et al., 2000).

[23] A manutenção preventiva é também conhecida como reengenharia de software (PRESSMAN, 2001).

No caso da manutenção preventiva, são necessários métodos e técnicas para reestruturar o sistema e assegurar a manutenibilidade. As métricas baseadas em código como acoplamento e coesão são utilizadas para medir a complexidade estrutural do sistema. Ao invés de utilizar linhas de código, é possível se concentrar em blocos de código como programas ou módulos como forma de simplificar o problema e descobrir dependências lógicas entre módulos. Geralmente, o engenheiro de software conhece os módulos que devem ser alterados em função de um tipo de modificação. Por essa razão, a abordagem de GALL et al. (1998) se propõe a encontrar padrões de alteração a partir de informações como número de versão, módulos e subsistemas presentes nos relatos de modificação.

Analisando todas as *releases* geradas durante o processo de desenvolvimento, verifica-se que alguns módulos sempre são alterados em conjunto. A partir dessa análise, identifica-se a dependência que existe entre módulos, i.e, o acoplamento lógico entre módulos. O acoplamento lógico é identificado quando a análise extrai a mesma seqüência de modificação para dois módulos. A seqüência de modificação pode ser representada por uma tupla <1 2..n> que contém todas as *releases* nas quais arquivos de determinados módulos do sistema modificaram sua versão.

No entanto, como a validade do acoplamento pode ser questionada, GALL et al. (1998) propõe verificar se as modificações realizadas na geração das *releases* contêm similaridades nos dados de tipo da modificação, classe do problema e número da requisição. Com base nas modificações, as dependências são validadas evitando a suposição de serem apenas uma coincidência.

Enquanto alguns autores (EICK et al., 2001; GRAVES et al., 2000; MOCKUS et al., 2000; MOCKUS et al., 2003) se concentram nas características de produto como depreciação, acoplamento e coesão de módulos durante a manutenção do software, DRAHEIM e PEKACKI (2003) adotam a perspectiva de processo justificando que a qualidade do produto depende da qualidade do processo e de fatores como cooperação que influenciam o processo. A cooperação pode ser identificada analisando as modificações que foram compartilhadas. Seguindo essa abordagem, são analisadas as atividades dos desenvolvedores durante o processo de desenvolvimento, permitindo que métricas sejam estabelecidas e executadas sobre o histórico de versões do projeto.

BALL et al. (1997) propuseram que artefatos que sofrem modificações em conjunto podem estar relacionados semanticamente. A partir dos relatos das alterações,

a medida $CD_{mr} \div \sqrt{C_{mr} \times D_{mr}}$ expressa a distância ou afinidade entre classes em C++, onde C_{mr} é o número de alterações que influenciaram a classe C e CD_{mr} o número de alterações que influenciaram C e D em conjunto. Se a classe C e D são sempre modificadas em conjunto, a probabilidade é 1 e nesse caso, o número de modificações que influenciaram C e D isoladamente é igual a CD_{mr}.

A partir dessa análise, BALL et al. (1997) observam efeitos de algumas decisões de projeto, como baixa coesão e alto acoplamento nas classes que sofrem o mesmo tipo de modificação. Esse tipo de resultado pode ser útil tanto durante a análise de impacto, etapa do processo de controle de modificações, quanto na aplicação de manutenção preventiva.

Considerando os resultados obtidos pelos diversos autores, verificamos que é possível detectar automaticamente relações semânticas entre artefatos no ambiente de desenvolvimento utilizando como fonte de informação os repositórios de GCS.

2.6 – Conclusões

Esse capítulo apresentou o conteúdo básico do estudo realizado sobre o processo de GCS e suas diversas atividades, além dos conceitos e abordagens existentes na literatura sobre a rastreabilidade, verificando a necessidade da integração entre o ferramental para a rastreabilidade e os processos de GCS.

A ausência de ferramental que suporte a identificação automática ou semi-automática dos pós-rastros é amplamente relatada na literatura (CLELAND-HUANG et al., 2003). À medida que o software cresce em tamanho e complexidade, o volume de artefatos aumenta, fazendo com que o número de interdependências e relações cresça exponencialmente. Esse fato torna inviável a manutenção e identificação dos rastros de forma manual em grandes projetos.

Como visto neste capítulo, o ferramental adequado para a rastreabilidade deve prover uma forma automatizada ou semi-automatizada de obter os pós-rastros, utilizando recursos disponíveis dentro do processo de desenvolvimento de software. Através da análise das modificações e versões existentes no contexto dos sistemas de GCS, é possível detectar automaticamente relações semânticas entre artefatos (BALL et al., 1997; ZIMMERMANN et al., 2003a; YING, 2003; SHIRABAD, 2003) e apresentá-las no contexto do ambiente de desenvolvimento de software. Conseqüentemente, as técnicas que automatizam a rastreabilidade podem fazer uso dos

repositórios de sistemas de GCS, de forma a apoiar o surgimento gradativo de pós-rastros intermodelos e intramodelos.

A partir desta motivação, apresentamos, no Capítulo 3, uma forma de extrair rastros de modificação que auxiliem na análise de impacto das modificações, no entendimento do software e nas futuras manutenções.

Capítulo 3 – O Uso de Técnicas de Mineração de Dados no Auxílio à Manutenção do Software

3.1 – Introdução

No Capítulo 2, vimos que os repositórios dos sistemas da GCS contêm informações sobre a evolução do software que são úteis para o processo de desenvolvimento de software. Além disso, conceituamos rastros em duas categorias: pré-rastros e pós-rastros. Os pós-rastros, foco de pesquisa desta dissertação, ainda podem ser classificados em intramodelos e intermodelos.

Quando artefatos *UML* estão sendo alterados, é importante detectar quais outros artefatos podem também necessitar de alterações. Freqüentemente, não é trivial a relação entre diferentes artefatos do sistema (SHIRABAD, 2003). Descobrir e entender relações não triviais entre artefatos favorece o entendimento do software e evita complicações como a introdução de novos erros e alterações incompletas durante a manutenção. A análise de impacto, apresentada no Capítulo 2, dita a importância desse conhecimento durante a manutenção e construção do software.

Para responder a questões do tipo "Quais artefatos devem ser alterados se este artefato *UML* for modificado?", é necessário encontrar rastros de modificação entre os artefatos *UML*, com o intuito de sugerir ao engenheiro de software quais artefatos precisam ser modificados em conjunto. O uso de técnicas de mineração de dados pode apoiar na detecção desses rastros, encontrando regras do tipo: "Desenvolvedores que modificam esses artefatos também modificam esses outros".

A descoberta de conhecimento em bases de dados, ou mineração de dados, representa o processo de extração de relações implícitas, ou de alto nível, a partir de um conjunto de informações relevantes (CHEN et al., 1996). A mineração de dados ou análise de dados é uma linha de pesquisa, reconhecida por pesquisadores de diferentes áreas como Banco de Dados, Gerência de Conhecimento, Inteligência Artificial, Aprendizado de Máquina e Estatística. Suas técnicas podem ser utilizadas na gerência da informação, processamento de consultas, tomada de decisões e em diversas aplicações que disponibilizem serviços personalizados a clientes, aumentando as chances de negócio (CHEN et al., 1996).

Esse capítulo apresenta o estudo realizado sobre técnicas de mineração de dados em repositórios dos sistemas da GCS, apresentando alguns trabalhos relacionados ao tema. As abordagens, apresentadas na Seção 3.2, diferem em relação a granularidade, ao tipo e a importância do resultado na execução da tarefa de modificação. Na Seção 3.3, é apresentada uma visão geral das técnicas de mineração de dados. Nesta seção, uma técnica é escolhida para ser utilizada pela abordagem proposta nesta dissertação. A Seção 3.4 discute a técnica selecionada. Ao final, a Seção 3.5 conclui o capítulo.

3.2 – Mineração de Dados em Repositórios de Software

Ao modificar artefatos do software tais como funções, classes e componentes, o engenheiro de software deve estar atento à cobertura da modificação. Neste contexto, alguns autores (SHIRABAD, 2003; ZIMMERMANN et al., 2003a; YING, 2003) questionam quais arquivos ou rotinas seriam relevantes a uma determinada modificação.

Atualmente, engenheiros de software têm dificuldade em entender o software, principalmente quando precisam alterar o código em função de uma modificação no requisito. O grafo de dependências extraído do código tem o propósito de ajudar no entendimento do código, indicando dependências estáticas entre elementos do código-fonte. No entanto, por não descrever a razão que levou a dependência e ser restrito ao código, o grafo de dependências não satisfaz totalmente as necessidades dos engenheiros de software quando a questão é a compreensão das relações existentes entre os artefatos do software.

O código-fonte pode ser armazenado no repositório de um sistema de controle de versões. O repositório contém, para cada arquivo armazenado, informações como: a data da criação do arquivo e as revisões ao longo do tempo que alteram, eventualmente, o tamanho do arquivo. As revisões indicam as linhas do código afetadas pela alteração. O repositório associa, na revisão do arquivo, a data exata da ocorrência da alteração, o comentário do desenvolvedor indicando a razão da modificação e, em alguns casos, a lista de outros arquivos que fazem parte da modificação descrita no comentário.

Alguns autores (ZIMMERMANN et al., 2003a; FISCHER et al., 2003; YING, 2003) têm utilizado as características de sistemas de controle de versões de software, como *CVS* (FOGEL e BAR, 2001), para recuperar as versões do produto e relacioná-las as modificações que as originaram no projeto. O comentário inserido na operação de *check-in*, por exemplo, pode referenciar a modificação assim como descrever indiretamente seu propósito.

CHEN et al. (2001) recuperam os comentários do repositório do *CVS*, que descrevem, geralmente, o raciocínio envolvido na implementação ou alteração de um fragmento de código, e associam esses comentários às modificações no código-fonte. Desta forma, indexam o código e orientam os desenvolvedores na localização das linhas de código referentes a uma determinada característica particular do software. A proposta é justamente prover um entendimento melhor da versão atual do código a partir de versões anteriores.

Existe, na literatura, o intuito de mapear as modificações no nível de linha de código para modificações em elementos como funções ou estruturas de dados através de uma técnica que construa automaticamente um grafo de dependência com base nas alterações já realizadas em código. Neste caso, a adição da razão da dependência poderia ser obtida do próprio repositório, através da análise das revisões de todos os arquivos, realizadas durante o ciclo de vida do software (HASSAN e HOLT, 2003).

HASSAN e HOLT (2003) propõem o mapeamento das modificações realizadas em código para subsistemas e a construção de um grafo de dependências estático (ou modelo funcional). A partir da comparação realizada entre o modelo conceitual original e o modelo evoluído do software, encontram dependências que podem ser classificadas em três categorias: convergente, ausente e divergente. A categoria convergente consiste de dependências que existem no software e têm sua existência esperada pelo engenheiro de software. A categoria ausente consiste de dependências que o engenheiro de software espera encontrar no modelo final, mas que a arquitetura revela que, de fato, não existem. Isso pode ocorrer devido a modificações na arquitetura, ou remoção de algumas características do sistema ou ainda, devido à falta de conhecimento do engenheiro de software que está avaliando o sistema. A categoria divergente consiste de dependências inesperadas que existem na arquitetura do software que pode ocorrer devido a características não documentadas que foram adicionadas ao software. Para cada dependência, são associadas informações provenientes dos sistemas da GCS, indicando possíveis razões para as diferenças encontradas entre os modelos.

Mesmo utilizando técnicas diferentes na identificação de dependências, algumas abordagens compartilham propósitos e o mesmo nível de granularidade quanto aos artefatos relacionados. Por exemplo, ZIMMERMANN et al. (2003b) relacionam variáveis ou métodos localizados no código-fonte e, a partir das dependências encontradas, analisam a arquitetura do software. A análise da arquitetura envolve verificar o grau de modularidade com base no acoplamento inter e intra-elemento.

No entanto, existem abordagens que embora apresentem uma granularidade maior em relação ao artefato, são particularmente importantes em projetos onde o código-fonte existe em diferentes linguagens e plataformas, e onde as análises estáticas e dinâmicas realizadas em código não ajudam muito o engenheiro de software nas tarefas de manutenção. YING (2003), por exemplo, complementa as análises em código, encontrando relações entre arquivos do código-fonte como forma de ajudar o engenheiro de software a identificar o que deve ser alterado em conjunto.

A descrição detalhada presente nos repositórios dos sistemas da GCS fornece uma grande oportunidade para a análise de estudos empíricos sobre a evolução do software. O uso destes repositórios tem a vantagem de não impor a adição de custos extras ao projeto, uma vez que os sistemas de GCS, como explicado no Capítulo 2, estão difundidos na indústria, já fazendo parte do processo de desenvolvimento de software (HASSAN e HOLT, 2003).

FISHER et al. (2003) utilizam o repositório de sistemas de GCS com o intuito de identificar o relacionamento entre características do software ou *features*. Eles definem *feature* como característica ou comportamento de uma parte do software. As *features* são unidades que descrevem o software sob a perspectiva de vários usuários além de desenvolvedores e mantenedores. A análise de sua evolução identifica informações sobre como foi feita a implementação das próprias *features*, além de facilitar a análise de impacto em relação às *features* e ao projeto arquitetural do software.

A idéia básica é, primeiramente, selecionar os arquivos alterados em função de cada modificação e testar as relações entre as modificações. Os arquivos alterados em conjunto, em mais de uma modificação, podem estar sendo alterados para solucionar problemas similares. Neste caso, a distância entre modificações pode ser expressa como o número de arquivos modificados para consertar ambos os problemas. Quanto mais arquivos em comum, mais similares são os problemas. FISHER et al. (2003) relacionam as *features* aos arquivos através da instrumentação do código, obtendo, por conseguinte, o relacionamento de *features* e as modificações efetuadas. A partir desta informação, é possível obter um mapeamento das relações entre *features,* que pode orientar futuras implementações ou alterações e, como dito anteriormente, a própria análise de impacto.

Embora os sistemas de GCS forneçam dados para a análise das relações entre arquivos (YING, 2003; SHIRABAD, 2003) ou elementos do código-fonte

(ZIMMERMANN et al., 2003a), CUBRANIC e MURPHY (2003) utilizam os relatos de modificação, os documentos de projeto e a lista de mensagens com o intuito de sugerir aos desenvolvedores artefatos pertinentes à tarefa de modificação que está sendo executada. Eles procuram, desta forma, maior abrangência, acessando a memória do grupo de trabalho presente na documentação do projeto e em mensagens onde geralmente se discute situações relacionadas a problemas que ocorrem durante o desenvolvimento de software. No entanto, eles relacionam arquivos relevantes a uma tarefa de modificação, propondo uma granularidade maior em relação a outras abordagens (ZIMMERMANN et al., 2003a), que relacionam elementos de código-fonte como variáveis e métodos.

O acoplamento definido na ferramenta HIPIKAT de CUBRANIC e MURPHY (2003) não é somente evolutivo como definido na abordagem de ZIMMERMANN et al. (2003a). Para CUBRANIC e MURPHY (2003), dois artefatos podem estar relacionados se eles fazem referência à mesma modificação, se aparecem juntos no e-mail do projeto ou se os comentários anexados no ato de *check-in* no sistema de controle de versões contém textos similares. Além disso, CUBRANIC e MURPHY (2003) necessitam de manutenções similares ocorridas no passado para identificar e recomendar artefatos que devem ser modificados em conjunto, o que é uma diferença em relação às abordagens (YING, 2003; ZIMMERMANN et al., 2003a) vistas anteriormente.

Nesta seção, foram mencionadas várias abordagens para a extração e análise de informações existentes nos repositórios dos sistemas da GCS. As abordagens têm o intuito de detectar automaticamente relações semânticas entre artefatos através da análise das modificações. O uso de mineração de dados é promissor, neste contexto, pois permite ir além das informações explícitas existentes no repositório.

As técnicas de mineração de dados são utilizadas por diversos autores na Engenharia de Software com o intuito de: (1) melhorar o entendimento do software, extraindo especificações de alto nível (COHEN, 1995), (2) decompor o software em subsistemas coesos para ajudar o engenheiro de software nas atividades de reengenharia e manutenção (MONTES DE OCA e CARVER, 1998), (3) avaliar a arquitetura do software (ZIMMERMANN et al., 2003b), (4) descobrir padrões de reutilização de métodos e classes (MICHAIL, 2000), (5) avaliar, diretamente ou indiretamente, o impacto de uma modificação no software, recomendando arquivos de código-fonte que

devem ser modificados em conjunto durante a atividade de manutenção (SHIRABAD, 2003; YING, 2003).

3.3 – Técnicas de Mineração de Dados

A mineração de dados pode ser considerada como uma parte do processo de Descoberta de Conhecimento em Banco de Dados (*KDD - Knowledge Discovery in Databases*). O termo *KDD* é utilizado para representar o processo de extração de conhecimento de alto nível em bases de dados (GOEBEL e GRUENWALD, 1999).

Recentemente, a capacidade de geração e coleta de dados tem crescido rapidamente. O crescimento dos dados e das bases de dados gera a necessidade urgente de técnicas e ferramentas que possam, de forma automática e inteligente, transformar um enorme conjunto de dados em algum tipo de conhecimento.

De acordo com a definição de BERRY e LINOFF (1997), mineração de dados é a exploração e a análise, por meio automático ou semi-automático, de grandes quantidades de dados, com a finalidade de descobrir padrões, i.e, regras significativas.

No entanto, não existe uma técnica que resolva todos os problemas de mineração de dados. Diferentes métodos servem para diferentes propósitos, cada um oferecendo vantagens e desvantagens. De acordo com HARRISON (1998) a escolha de uma técnica de mineração de dados depende da tarefa específica a ser executada e dos dados disponíveis para a análise. As técnicas de mineração de dados podem ser aplicadas a tarefas como, por exemplo, classificação, estimativa, associação, segmentação e sumarização. Essas tarefas são apresentadas, de forma resumida, na Tabela 3.1.

Tabela 3.1 - Tarefas realizadas por técnicas de mineração de dados

(Fonte: DIAS, 2001; CHEN et al., 1996).

Tarefa	Descrição	Exemplos
Classificação	Constrói um modelo de algum tipo que possa ser aplicado a dados não classificados a fim de categorizá-los em classes. O objetivo é descobrir a relação entre um atributo cujo valor será previsto e um conjunto de atributos usados para a previsão, classificando os dados com base no valor de certos atributos.	- Classificar pedidos de crédito. - Identificar a melhor forma de tratamento de um paciente.
Estimativa (ou Regressão)	Usada para definir um valor para alguma variável contínua desconhecida.	- Estimar a probabilidade de que um paciente morrerá baseando-se nos resultados de diagnósticos médicos.
Associação	Usada para determinar quais itens tendem a ser adquiridos juntos em uma mesma transação.	- Determinar que produtos costumam ser colocados juntos em carrinhos de supermercado.

Segmentação (ou *Clustering*)	Processo de partição de um conjunto de dados heterogêneos em vários grupos ou subgrupos mais homogêneos. O objetivo é agrupar, primeiramente, dados em um conjunto de classes com base em algum princípio e, posteriormente, derivar um conjunto de regras com base nesta classificação.	- Agrupar clientes com comportamento de compra similar. - Agrupar seções de usuários Web para prever comportamento futuro de usuário.
Sumarização	Envolve métodos para encontrar uma descrição compacta para um subconjunto de dados. Dentre as ferramentas que utilizam essa técnica, podemos citar as de processamento analítico (OLAP).	- Tabular o significado e desvios padrão para todos os itens de dados. - Derivar regras de síntese.

Com base nas tarefas citadas na Tabela 3.1, é apresentado um resumo das técnicas de mineração de dados existentes (Tabela 3.2). A familiaridade com as técnicas é necessária para que se possa escolher uma delas. Dentre as técnicas apresentadas, citamos algumas que são normalmente utilizadas.

Tabela 3.2 - Técnicas de mineração de dados

(Fonte: DIAS, 2001; CHEN et al., 1996).

Técnica	Descrição	Tarefas
Descoberta de Regras de Associação	Estabelece uma correlação estatística entre itens de um conjunto de dados.	- Associação
Árvores de Decisão	Hierarquização dos dados baseada em estágios de decisão (nós) e na separação de classes e subconjuntos.	- Classificação - Regressão
Raciocínio Baseado em Casos ou CBR	Baseado no método do vizinho mais próximo combina e compara atributos para estabelecer uma hierarquia de semelhanças.	- Classificação - Segmentação
Algoritmos genéticos	Métodos gerais de busca e otimização, inspirados na Teoria da Evolução, onde a cada nova geração, soluções melhores têm mais chance de ter "descendente" e assim evoluir.	- Classificação - Segmentação
Redes Neurais Artificiais	Modelos inspirados na fisiologia do cérebro, onde o conhecimento é fruto do mapa das conexões neuronais e dos pesos dessas conexões.	- Classificação - Segmentação
Processamento Analítico	Refere-se ao conjunto de processos para criação, gerência e manipulação de dados multidimensionais para análise e visualização pelo usuário em busca de uma maior compreensão destes dados. Apóia a tomada de decisões, permitindo analisar tendências e padrões em grandes quantidades de dados com base no histórico.	- Sumarização

Dentre as técnicas apresentadas, concluímos que alguns parâmetros devem ser considerados na escolha da técnica, como, por exemplo, a tarefa a ser desempenhada pela técnica e as características dos dados do repositório. A adequação da técnica às

características dos dados minimiza possíveis dificuldades na transformação dos dados. Neste caso, torna-se necessário: (1) verificar a classe do problema a ser resolvido, ou traduzir o problema em uma série de tarefas de mineração de dados; (2) compreender a natureza dos dados disponíveis em termos de conteúdo e significado, considerando possíveis relações entre os registros da base (BERRY e LINOFF, 1997) e (3) selecionar, a partir das características dos dados em análise, uma das técnicas de mineração de dados que minimize as dificuldades para a obtenção do resultado.

O objetivo do estudo das técnicas de mineração nesta dissertação é encontrar rastros de modificação que indiquem as relações entre os artefatos de software que foram modificados em conjunto. Com base neste propósito, verificamos que duas técnicas de mineração de dados se destacam: descoberta de regras de associação e classificação por árvores de decisão.

Atualmente, a descoberta de regras de associação parece ser a técnica de mineração mais utilizada em pesquisas que buscam objetivos semelhantes com o desta dissertação (ZIMMERMANN et al., 2003a; YING, 2003). Essa técnica, por ser direcionada para a descoberta de regras de associação em bases de dados, pode ser aplicada no contexto desse trabalho, dado a sua aplicabilidade para o problema em questão, ao tipo de resultado retornado e às suas características.

Outra técnica possível é a técnica de árvore de decisão aplicada ao problema da classificação, já mencionada em algumas abordagens (SHIRABAD, 2003). SHIRABAD (2003) divide o conjunto de dados em categorias, a partir de um conjunto de heurísticas, e constrói um modelo capaz de rotular futuros exemplos em uma dessas categorias. Ao definir uma relação de relevância[24] entre dois artefatos de software, ele destaca duas categorias: relevantes e não relevantes. Para classificar a relevância entre dois artefatos do software é necessário definir categorias, ou classes, e associar cada instância do conceito à classe.

O problema de aprendizado de conceitos de relevância pode ser visto como um problema de classificação. No entanto, a técnica de árvore de decisão aplicada a esse problema deve considerar valores discretos para os atributos. Como visto na Tabela 3.1,

[24] A relação de relevância, em termos gerais, mapeia um conjunto de elementos em um valor discreto que tem o significado de quão relevantes esses elementos são entre si.

esses atributos[25] são usados para categorizar um novo exemplo. A abordagem proposta nesta dissertação pretende utilizar valores contínuos como, por exemplo, medidas que indicam a frequência de um elemento na base de dados. A técnica de regras de associação considera esse valor durante a mineração. Ele é importante para o problema da manutenção porque indica se a relação entre dois artefatos deve realmente ser levada em conta durante uma determinada modificação. Por esse motivo, optou-se por usar a técnica de regras de associação ao invés da classificação.

Existem diversos algoritmos que implementam a técnica de regras de associação, por exemplo, *FP-growth, Apriori* e diversas derivações do *Apriori* como *AprioriTID* e *AprioriHybrid*, dentre outros. Como a abordagem proposta nesta dissertação pretende ser independente em relação ao tipo de algoritmo de regras de associação utilizado, qualquer um desses algoritmos poderia ser usado com o propósito de mineração. Dentre eles, o *Apriori* foi escolhido por ser amplamente conhecido na literatura, além de já ser utilizado em diversas abordagens (dentre elas, ZIMMERMANN et al., 2003a; MICHAIL, 2000).

3.4 – Mineração por Regras de Associação

A mineração por regras de associação encontra relações entre itens de dados que ocorrem freqüentemente em transações[26] na base de dados (ZIMMERMANN et al., 2003a). Considerando um universo de transações T onde cada transação é representada por um conjunto de itens, e sendo ε um universo de itens de dados, diz-se que cada transação t ($t \in T$) está contida em ε, i.e, $t \subseteq \varepsilon$. Exemplificando, se ε for um conjunto de itens disponíveis em um supermercado, *t* representa uma compra efetuada.

---

[25] Os atributos no trabalho de SHIRABAD (2003) representam propriedades do software cujos valores podem ser adquiridos do código-fonte ou dos dados de manutenção. Essa consideração, contudo, torna a abordagem dependente da linguagem de programação e do ambiente em que o código foi desenvolvido, além de dificultar a sua aplicação em um contexto de trabalho mais abrangente.

[26] As operações executadas sobre uma base de dados, em geral, ocorrem dentro de uma transação. A transação corresponde à unidade de trabalho que contém uma operação ou conjunto de operações que foram executadas em bloco, e que ao final, é considerada completa, caso tenha sido executada com sucesso, ou desfeita, caso tenha ocorrido algum erro no seu processamento. Por exemplo, a operação de compra indica quais itens em um supermercado foram comprados juntos. Essa operação única executada sobre a base de dados de um supermercado pode ser definida como uma transação.

Nesse exemplo, o universo de transações (T) pode ser analisado para encontrar um padrão de compra que indique quais itens são freqüentemente comprados juntos no supermercado. Esse padrão de compra pode ser representado como uma regra de associação. Por exemplo, a informação de que clientes que compram computadores também compram softwares pode ser representada como uma regra de associação da forma: *Computador* → *Software.*

Seja X e Y conjuntos de itens da base de dados e *t* uma transação pertencente ao universo T, é dito que uma transação t contém X, se somente se, $X \subseteq t$. Com base nisso, uma regra de associação pode ser caracterizada como uma implicação X→Y onde $X \subset \varepsilon, Y \subset \varepsilon, X \cap Y = \emptyset$. Na regra, o conjunto de itens X é definido como antecedente da regra e o conjunto de itens Y é definido como conseqüente. É possível ter interesse apenas em regras que contenham um item específico, aparecendo em X ou Y.

No contexto desta dissertação, os itens de dados são artefatos *UML* como, por exemplo, casos de uso, classes, componentes e pacotes. A base de dados, neste caso, é um repositório que armazena todas as alterações realizadas nos artefatos do software durante o processo de desenvolvimento, onde cada transação na base é uma modificação implementada através de uma ou mais operações de *check-in* no sistema de controle de versões. A Tabela 3.3 diferencia os dois contextos de utilização de mineração por regras de associação mencionados. O primeiro é o contexto clássico de compras em um supermercado, e o segundo, o contexto desta dissertação.

Tabela 3.3 - Contexto de utilização da técnica de mineração por regras de associação.

Contexto	Item de Dado	Transação	Base de Dados	Resultado da Mineração
Supermercado	Mercadoria	Uma compra efetuada no supermercado	Vendas efetuadas	Relações de compra (e.g: Computador → Software)
Manutenção de Software	Artefatos de software	Uma modificação implementada no software	Repositório dos sistemas da GCS	Rastro de modificação

3.4.1 – Medidas de suporte e confiança

A relevância das relações encontradas entre itens de uma base de dados é expressa através de duas medidas: suporte e confiança. Nesta dissertação, a relevância é baseada no número de modificações realizadas em dois artefatos de software. Contando

o número das modificações realizadas em cada par de artefatos, obtemos a freqüência do par de elementos no total de transações, i.e, a medida de suporte.

Sendo, $\varepsilon = \{e_1, e_2, e_3, e_4, e_5, e_6\}$ equivalente a um conjunto de itens da base (artefatos) que foram modificados durante o desenvolvimento do software, e t_i ($t_i \subseteq \varepsilon$, onde $1 \leq i \leq n$ e $n \geq 2$), uma transação qualquer na base de dados que contém um conjunto de artefatos alterados em virtude de uma modificação, temos, para cada par de artefatos quaisquer $(e_j, e_k) \in \varepsilon \times \varepsilon$, a medida de suporte igual a:

$$S_{ej,ek} = \frac{n(\{t_i : e_j \in t_i \wedge e_k \in t_i\})}{n(T)}$$

Em $S_{ej,ek}$, o número de ocorrências de (e_j, e_k) é igual a $n(\{t_i : e_j \in t_i \wedge e_k \in t_i\})$ e n(T) equivale ao total de transações na base.

Quando a medida é aplicada, o resultado corresponde a matriz de suporte S (Figura 3.1) que indica quão freqüente é a relação de cada par de elementos no conjunto total de transações.

S	e_1	e_2	e_3	e_4	e_5	e_6
e_1	8/9	6/9	0	0	0	0
e_2	6/9	7/9	0	0	0	0
e_3	0	0	1	3/9	0	0
e_4	0	0	3/9	3/9	2/9	1/9
e_5	0	0	0	2/9	1	8/9
e_6	0	0	0	1/9	8/9	8/9

Figura 3.1 - Matriz de suporte

Na Figura 3.1, os elementos e_1, e_2,..., e_6 são itens da base de dados e o número total de transações na base é 9. Repare que as transações t_1, t_2,..., t_n, representadas por t_i, não são conjuntos disjuntos, isto quer dizer que diferentes transações podem ter modificado os mesmos elementos da base.

A matriz de suporte S mostra que do total de modificações realizadas, oito alteraram o elemento e_1. Dentre essas oito modificações, seis também modificaram e_2.

A matriz indica que quanto maior for o número de modificações comuns entre dois elementos, maior será a medida de suporte.

Entretanto, a matriz de suporte S é simétrica e não reflete se uma modificação em e_1 gerou impacto em e_2 ou vice-versa. Somente a medida de confiança fornece essa informação, respondendo a questão: "De todas as modificações realizadas em um elemento e_j, quantas também impactaram um elemento e_k?". A medida de confiança indica o número relativo de ocorrências de dois elementos no conjunto de transações, sendo definida da forma:

$$C_{ej,ek} = \frac{n(\{t_i : e_j \in t_i \wedge e_k \in t_i\})}{n(\{t_i : e_j \in t_i\})} \text{ ou } C_{ej,ek} = \frac{S_{ej,ek}}{S_{ej,ej}}$$

Calculando a medida de confiança entre pares de elementos, encontra-se a matriz de confiança C (Figura 3.2).

C	e_1	e_2	e_3	e_4	e_5	e_6
e_1	1	6/8	0	0	0	0
e_2	6/7	1	0	0	0	0
e_3	0	0	1	3/9	0	0
e_4	0	0	1	1	2/3	1/3
e_5	0	0	0	2/9	1	8/9
e_6	0	0	0	1/8	1	1

Figura 3.2 - Matriz de confiança

Dada uma matriz de suporte S, calcula-se a matriz de confiança C dividindo cada linha da matriz S pelo valor do elemento correspondente na diagonal. O valor nulo na matriz indica que não existe relação entre os elementos. Conclui-se que uma regra de associação tem como medida de confiança o equivalente à probabilidade condicional de e_k em relação à e_j, ou $P(e_k|e_j)$. Em outras palavras, das oito modificações em e_1, 6/8 ou 75% gerou impacto em e_2.

Basicamente, a diferença da medida de confiança para a medida de suporte, em termos de cálculo, é que a medida de suporte calcula a freqüência de um elemento e_j (ou

par de elementos (e_j, e_k)) levando em consideração o número total de transações na base, i.e, n(T), enquanto a medida de confiança calcula a freqüência de um elemento e_k considerando apenas as transações onde um segundo elemento e_j aparece, i.e, $n(\{t_i : e_j \in t_i\})$.

Caso exista um limite δ para a seleção dos resultados gerados na matriz de suporte ou confiança, utiliza-se:

$F^{M,t} = 0$ se $M_{ej, ek} < \delta$;

$F^{M,t} = F_{ej, ek}$ se $M_{ej, ek} \geq \delta$.

Neste caso, F equivale a matriz resultante do filtro e M equivale a matriz de suporte (S) ou confiança (C), anteriormente definidas, que poderão ser filtradas, dado que o limite de seleção de elementos é maior que zero. Nos algoritmos de mineração de dados, é comum utilizar valores de suporte maiores ou iguais que um limite pré-definido como base para o cálculo da matriz de confiança.

Os limites definidos para as medidas de suporte e confiança são utilizados pelos algoritmos de mineração de dados para descartar regras de associação que não são freqüentes ou que não são relevantes para o problema que a mineração está tratando. Vários algoritmos são apresentados na literatura com o intuito de encontrar todas as regras de associação (AGRAWAL e SRIKANT, 1994). No entanto, existe um alto custo em gerar todas as combinações de elementos com suporte acima de um certo limite definido. Dentre os algoritmos de mineração estudados, é destacado, nesta dissertação, o *Apriori.*

3.4.2 – O algoritmo Apriori

Na literatura de técnicas de mineração de dados, um conjunto de itens que contém k elementos é denominado *k-itemset* ou um *itemset* de tamanho igual a k, i.e, n(*k-itemset*) = k. A freqüência de ocorrência de um *itemset* na base de dados equivale à medida de suporte ou ao número de transações na base de dados que contém o *itemset.* Um *k-itemset* satisfaz um limite pré-definido δ se a sua freqüência de ocorrência na base de dados é maior ou igual a δ. O conjunto de *k-itemset* que satisfaz esta condição é denominado no algoritmo de L_k.

Basicamente, o algoritmo *Apriori* propõe uma busca em largura por conjuntos candidatos às regras de associação, gerando, à cada iteração, um *k-itemset* a partir de um *(k-1)-itemset* (onde $k \geq 2$), e conseqüentemente, L_k a partir de L_{k-1}. A cada iteração, um

conjunto C_k é gerado, onde cada elemento de C_k tem um valor de suporte. Inicialmente, são considerados em C_1 todos os elementos pertencentes ao conjunto de elementos ε de uma base de dados. Por exemplo, sendo $\varepsilon=\{e_1,e_2,e_3\}$, os elementos $\{e_1\}$, $\{e_2\}$ e $\{e_3\}$ fazem parte do conjunto C_1. Nas iterações seguintes, C_k corresponderá à junção dos conjuntos de *(k-1)-itemsets*, desconsiderando eventuais repetições.

Na geração de L_k, os elementos de C_k que não satisfazem a condição do suporte ser maior que o mínimo definido são descartados. Para reduzir o espaço de busca, o algoritmo considera que todo conjunto deve satisfazer o mínimo definido para suporte, baseado na premissa que se um *(k-1)-itemset* não satisfaz a condição da medida de suporte então qualquer *k-itemset,* que contenha os elementos de *(k-1)-itemset,* também não satisfaz. Logo, o valor do suporte não aumenta ao adicionar qualquer elemento a um conjunto. Todo o processamento continua até que não seja possível gerar novos elementos em L_k com suporte maior ou igual ao definido como limite. A Figura 3.3 exemplifica o algoritmo passo a passo.

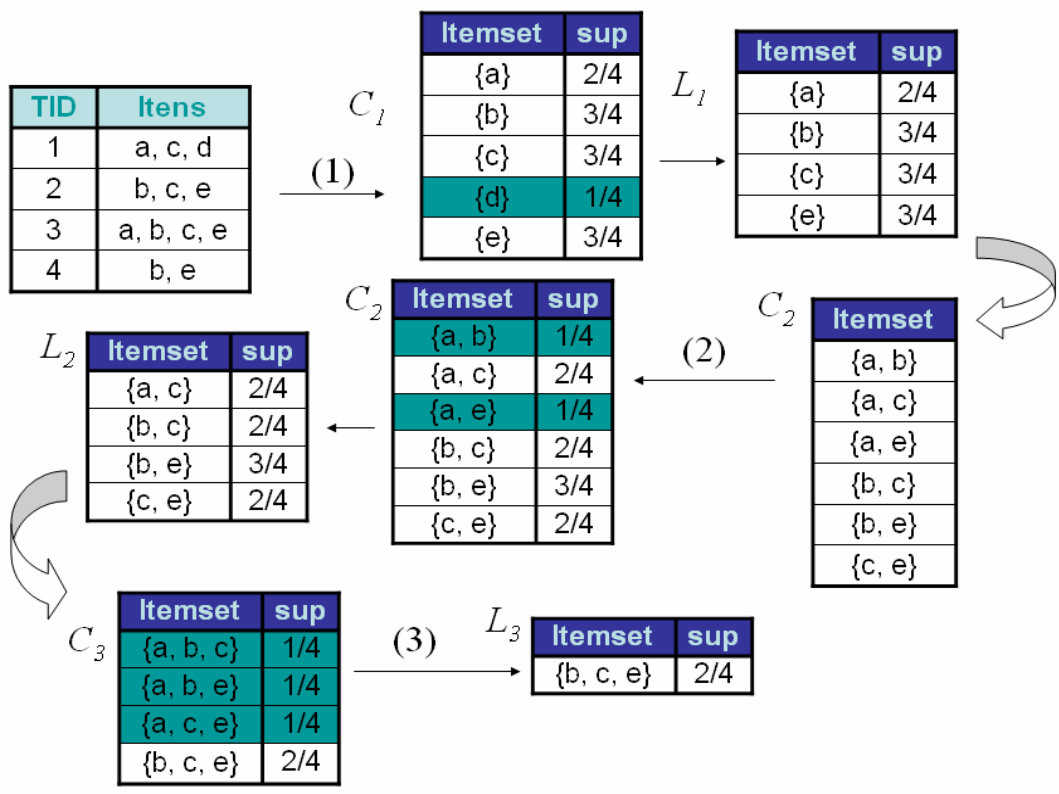

Figura 3.3 - Exemplo do algoritmo Apriori

Neste exemplo, o suporte mínimo equivale a 2/4 e *TID* representa o identificador de cada transação na base de dados. Na Figura 3.3 utilizamos **sup** como abreviação para a palavra suporte e $\varepsilon=\{a,b,c,d,e\}$.

Na primeira iteração do algoritmo *Apriori* da Figura 3.3, a base C_1 é gerada considerando todos os elementos presentes na base de dados. Percorrendo a base C_1, o algoritmo calcula as medidas de suporte para cada *1-itemset,* desconsiderando, na geração do conjunto L_1, os conjuntos que não satisfazem a condição da medida de suporte ter um valor maior que o mínimo. Para descobrir C_2, o algoritmo faz a junção dos elementos presentes em L_1, obtendo *itemset* de tamanho 2 ou *2-itemsets*. Esse processo continua até a geração de L_3. No exemplo, o algoritmo continua até a geração de L_3 onde obtém o *itemset* {b, c, e}, após o descarte dos conjuntos que não respeitaram a condição referente ao suporte.

No algoritmo *Apriori*, o número de conjuntos em C_k pode ser exponencial a depender do número de elementos. A cada passo, o algoritmo percorre toda a base, o que, conseqüentemente, faz com que todo o processo percorra k vezes a base, se k for o maior tamanho para o conjunto candidato à regra de associação. Para atenuar esse problema, ele reduz o número de conjuntos candidatos ao descartar aqueles que possivelmente não resultam em uma regra de associação. Entretanto, a redução no número de conjuntos candidatos depende do suporte mínimo ser suficiente para descartar conjuntos que não satisfaçam o limite definido.

Para gerar as regras de associação a partir dos conjuntos obtidos, AGRAWAL e SRIKANT (1994) utilizam a medida de confiança. As regras de associação podem ser geradas a partir da combinação dos elementos presentes nos conjuntos de L_k (resultado do processamento anterior). Sendo assim, para cada subconjunto X contido em L_k, existe a regra X → Y, onde Y é o complemento de X em L_k. Esta regra é válida somente se a medida de confiança da regra for maior ou igual ao limite mínimo definido. A Figura 3.4 mostra todas as regras geradas para o exemplo anterior. Nesta figura, o cálculo foi detalhado apenas para o primeiro caso.

Regra Associação	confiança
{b, c} → e	2/2 = 100%
{b, e} → c	2/3 = 67%
{c, e} → b	2/2 = 100%
b → {c, e}	2/3 = 67%
c → {b, e}	2/3 = 67%
e → {b, c}	2/3 = 67%

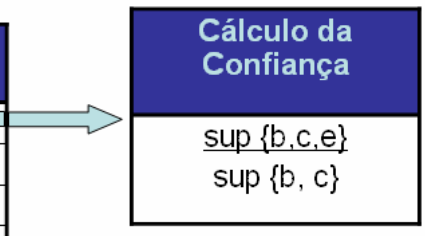

Figura 3.4 - Geração de regras de associação

Na Figura 3.4, foram calculadas as medidas de confiança para cada regra gerada, tendo como base o suporte, tal como definido, anteriormente, na Seção 3.4.1. Neste caso, a probabilidade condicional pode ser expressa da forma:

$$C_{(X \rightarrow Y)} = \frac{S_{X,Y}}{S_X}$$

onde X e Y são conjuntos de elementos.

3.4.3 – Generalização das Regras de Associação

As regras de associação têm sido utilizadas na descoberta de padrões de reutilização de classes e métodos. Com o intuito de ajudar o engenheiro de software a construir aplicações a partir de uma biblioteca particular, MICHAIL (2000) utiliza uma outra abordagem de regras de associação para encontrar padrões que indiquem algumas informações relevantes que as abordagens clássicas de regras de associação desconsideram. A diferença em relação às demais, é que nessa abordagem, as associações respeitam diferentes níveis na hierarquia não considerando apenas os elementos presentes nas transações, mas também seus ancestrais. Conseqüentemente, ela produz um número maior de regras, às vezes redundantes, em comparação com outras abordagens, o que faz com que seja recomendado o uso de técnicas de descarte de elementos para que apareçam somente as regras interessantes. MICHAIL (2000) considera cinco relacionamentos: herança de classe, instanciação de classe, invocação de método, invocação implícita e sobreposição de método. Neste caso, a abordagem é aplicada a qualquer linguagem orientada a objeto, considerando classes e métodos nas regras de associação.

Generalizar as regras de associação levando em conta a hierarquia na qual o elemento está inserido tem sido um dos focos dos trabalhos de SRIKANT e AGRAWAL (1995). A partir de um conjunto de transações e uma taxonomia, o objetivo é encontrar regras de associação onde os elementos podem estar em qualquer nível da taxonomia. O conjunto de taxonomias é representado por **T**, sendo cada taxonomia um grafo acíclico. Se existir em **T**, uma ligação entre dois elementos, como, por exemplo, **p** e **c**, é dito que **p** representa a generalização de **c** e **c** representa a especialização de **p**. Uma regra de associação generalizada é uma implicação na forma X → Y onde as seguintes condições são respeitadas: $X \subset \varepsilon, Y \subset \varepsilon, X \cap Y = \emptyset$ e não existe elemento em Y que seja ancestral de algum elemento em X.

A diferença básica deste algoritmo para o *Apriori*, sendo este uma derivação do último, é que este considera outros tipos de regras de associação, onde para cada elemento na transação são adicionados todos os seus ancestrais.

3.5 – Conclusões

Esse capítulo apresentou o estudo realizado sobre técnicas de mineração de dados e algumas abordagens que aplicam as técnicas estudadas no contexto da manutenção do software. Essas abordagens exploram os repositórios de sistemas da GCS sugerindo alguma indicação sobre os artefatos que podem ser modificados em conjunto. No entanto, percebe-se que elas ainda estão voltadas para código-fonte, não fornecendo apoio às modificações que acontecem no nível de modelo *UML*. Quando o engenheiro de software modifica um modelo *UML* ou realiza alguma alteração em código que influencia a reorganização do modelo, ele se questiona: "Quais outros artefatos *UML* devem ser alterados em virtude dessa modificação?". Para responder esse questionamento, é necessário encontrar relações entre artefatos *UML*, ou seja, rastros de modificação, que indiquem quais artefatos devem ser modificados em conjunto. Atuando no nível de modelo *UML,* as chances de encontrar falhas de projeto nas fases iniciais do processo de desenvolvimento são maximizadas.

A abordagem apresentada no Capítulo 4 tem o intuito de encontrar esses rastros. Para encontrar relações entre artefatos *UML*, a técnica de regras de associação pode ser utilizada. Devido a sua aplicabilidade para o problema em questão, ao tipo de resultado retornado e às suas características, optamos por utilizá-la. Já em relação aos algoritmos estudados, o *Apriori* foi escolhido.

O *Apriori* usa duas medidas, suporte e confiança. Embora sua versão original relate problemas referentes ao seu desempenho em grandes volumes de dados, existem, na literatura, várias propostas de otimização, onde *AprioriTID* e *AprioriHybrid* (AGRAWAL e SRIKANT, 1994) são exemplos. Além disso, o *Apriori* pode ser adaptado para tratar propósitos específicos do problema em questão.

Embora a proposta desta dissertação seja trabalhar com artefatos *UML*, não existe a necessidade de utilizar a hierarquia da *UML* ou qualquer taxionomia para obter regras de associação entre seus elementos. Isso se deve, como veremos no Capítulo 4, às características dos dados armazenados no repositório pelo sistema de controle de versões, que já considera a hierarquia na qual o artefato *UML* está inserido.

Capítulo 4 – Odyssey-WI: Um Mecanismo para Rastreabilidade entre Modelos

4.1 – Introdução

No Capítulo 2, foram apresentados os conceitos e os sistemas da Gerência de Configuração de Software (GCS) para o desenvolvimento convencional. Neste estudo, averiguamos a possibilidade de integração entre esses sistemas e ambientes de desenvolvimento de software, concluindo que informações da GCS podem ser utilizadas para prover conhecimento sobre o processo de desenvolvimento. Esses sistemas armazenam dados, que se extraídos, podem servir de base para a detecção de rastros de modificação que indiquem quais artefatos foram modificados em conjunto. Esses rastros, como visto no Capítulo 3, podem ser obtidos a partir da aplicação de técnicas de mineração de dados.

Contudo, os sistemas atuais da GCS não atendem requisitos específicos do Desenvolvimento Baseado em Componentes (DBC) (ESTUBLIER et al., 2002). Apesar da existência de uma ampla infra-estrutura de GCS para o desenvolvimento convencional, percebe-se que a integração desta infra-estrutura com paradigmas específicos de desenvolvimento de software ainda é carente (ESTUBLIER et al., 2002). Com a mudança no foco de desenvolvimento, surgem novos problemas referentes a GCS (LEBSACK et al., 2001) como o aumento da complexidade do processo de manutenção e evolução, que se acentua devido à necessidade freqüente de adaptar os artefatos antes e após a sua efetiva reutilização (MURTA, 2004).

No contexto do ambiente *Odyssey*, o projeto *Odyssey-SCM* (MURTA, 2004), tenta suprir a ausência de uma infra-estrutura de GCS para DBC, provendo um maior controle na evolução de seus artefatos. Apesar do controle sobre as atividades de DBC introduzir uma certa sobrecarga no trabalho original, devido a atividades até então inexistentes, acredita-se que essas atividades trazem benefícios à evolução do software, aumentando a produtividade e diminuindo a quantidade de erros, na medida em que o desenvolvimento se dá em um ambiente controlado e organizado.

Durante o desenvolvimento, os sistemas de GCS são fundamentais para prover controle sobre os artefatos produzidos e modificados por diferentes desenvolvedores.

Ao modificar um requisito, devem ser modificados todos os artefatos de software relacionados a ele, desde a análise até a sua implementação em código-fonte.

Como visto anteriormente, as técnicas de rastreabilidade podem ser utilizadas, durante o desenvolvimento, para identificar todos os artefatos que devem ser atualizados em função de uma modificação introduzida no software. Nesse contexto, portanto, torna-se de grande valia a existência de mecanismos de rastreabilidade que facilitem a manutenção no software, indicando quais artefatos, situados em diferentes níveis de abstração, devem ser modificados. Os rastros de modificação trazem à tona o entendimento de como o software foi estruturado e como evoluiu, além de fornecer um indício sobre a interação entre componentes e os elementos de modelagem presentes em sua composição.

Este capítulo apresenta a proposta do mecanismo *Odyssey-WI*, no contexto do projeto *Odyssey-SCM*, que tem como objetivo principal apoiar os desenvolvedores, fornecendo suporte automatizado na detecção de rastros de modificação entre artefatos *UML* relacionados através do conceito de componente.

O capítulo está organizado da seguinte forma: a Seção 4.2 apresenta o projeto *Odyssey-SCM* e os sistemas da GCS cujos dados são analisados pelo mecanismo. Na Seção 4.3, é apresentada a abordagem *Odyssey-WI*. Nessa seção, incluímos suas características e funcionalidades, além de um cenário de utilização. Comparações com outras abordagens pesquisadas e considerações finais sobre o mecanismo são apresentadas na Seção 4.4.

4.2 – Contexto de Desenvolvimento: Projeto Odyssey-SCM

O projeto *Odyssey-SCM* (MURTA et al., 2004) provê uma abordagem de GCS voltada para a evolução controlada de artefatos de alto nível de abstração e para as atividades do processo de DBC. Esse projeto é composto por:

(1) Processos e normas (*Odyssey-SCMP*[27]) de GCS para o contexto de DBC (DANTAS et al., 2003).

(2) Sistema de Controle de Modificações (*Odyssey-CCS*[28]), que propõe uma abordagem configurável e flexível para o processo de controle de modificações,

[27]SCMP: Do inglês, *Software Configuration Management Process.*

[28] CCS: Do inglês, *change control system.*

atenta para a interação entre as diferentes equipes de um ambiente de DBC (LOPES, 2005).

(3) Sistemas de Controle de Construções e Liberações (*Odyssey-BRCS*[29]), que fornece suporte às atividades de construção e liberação, correlacionando todas as partes que compõe um componente, desde a especificação até testes e serviços executáveis.

(4) Sistema de Controle de Versões (*Odyssey-VCS*[30]), que propõe uma abordagem configurável para o controle de versões de modelos baseados no *MOF*[31], como, por exemplo, a *UML* (OLIVEIRA et al., 2004).

(5) Sistemas que integram os espaços de trabalho (DANTAS et al., 2004) utilizando as informações de GCS no apoio ao desenvolvimento. Neste contexto, se enquadra a abordagem proposta nesta dissertação (*Odyssey-WI*[32]), que aplica técnicas de mineração de dados sobre o repositório do sistema de controle de versões *Odyssey-VCS* e modificações *Odyssey-CCS* para detectar rastros de modificação intramodelos e intermodelos entre artefatos *UML*. Considerando o paradigma de DBC, o sistema expõe as dependências existentes entre os artefatos que se relacionam, internamente ou externamente, com o componente.

Nas seções seguintes, explicaremos em maiores detalhes os sistemas de controle de versões *Odyssey-VCS* e de controle de modificações *Odyssey-CCS* utilizados como fonte de dados pelo mecanismo *Odyssey-WI*.

4.2.1 – Odyssey-CCS: Sistema de Controle de Modificações

Os sistemas de controle de modificações existentes para o desenvolvimento de software, geralmente, implementam um processo específico de GCS, não possibilitando a customização do processo. Como o processo pode sofrer adaptações em função da organização, é importante que os sistemas de controle de modificações possibilitem

---

[29]BRCS: Do inglês, *Build and Release Control System.*

[30] VCS: Do inglês, *version control system*.

[31] *MOF* (*Meta Object Facility*) (OMG, 2002a) é um padrão da *Object Management Group* (*OMG*) para descrição de meta-modelos e modelos independente de plataforma, sendo a UML (2003b), CWM (*Common Warehouse Metamodel*) (OMG, 2003a) e o próprio MOF exemplos de meta-modelos MOF.

[32] WI: Do inglês, *Workspace Integration*.

customizações futuras, onde adaptações no processo ocorram sem prejudicar os processos em execução.

De acordo com MURTA (2004), os processos de GCS relacionados com DBC ainda estão imaturos e por isso tendem a sofrer modificações até que uma maior estabilidade seja alcançada. Ainda não existe um consenso em relação a quais informações devem ser coletadas em cada formulário de cada atividade do processo. Também não existem normas que estabeleçam que atividades são obrigatórias e quais atividades são opcionais na adoção de GCS em DBC. Em virtude disso, é essencial possibilitar a adaptação do sistema de controle de modificações a novos processos e permitir a configuração da coleta de informações em função das necessidades de outros sistemas da organização.

O sistema *Odyssey-CCS* (LOPES, 2005) propõe uma abordagem configurável e extensível para o controle de modificações, permitindo a configuração da coleta das informações e das atividades. A utilização do *Odyssey-CCS* consiste, principalmente, na etapa inicial de preparação do ambiente e na execução do processo de controle de modificações. A etapa de preparação do ambiente é executada pelo gerente de configuração responsável por implantar no *Odyssey-CCS* a norma de gerência de configuração adotada na organização, por exemplo, IEEE Std 1042 (1987) ou ISO 10007 (1995).

O responsável pela GCS deve criar os formulários necessários e, para cada formulário, criar os campos que compõem esse formulário. Esses formulários podem ser utilizados nas diversas atividades do processo de GCS, com intuitos diferentes (MURTA, 2004). Do ponto de vista de análise de impacto, eles podem, por exemplo, apoiar à análise de causa através de campos específicos para manutenções corretivas.

Posteriormente, o processo deverá ser modelado, composto de atividades onde cada atividade será associada aos respectivos formulários. Cada atividade do processo agrega novos conhecimentos ao ciclo de vida das modificações, como visto na Seção 2.4. Na atividade de requisição da modificação, por exemplo, informações como a razão e a gravidade da modificação, entre outras, podem ser necessárias à execução das demais atividades do processo. Somente após a conclusão da fase de preparação, o sistema de controle de modificações poderá ser executado.

Portanto, para possibilitar a configuração do processo, o *Odyssey-CCS* permite que os formulários sejam configuráveis em função das necessidades do processo e facilmente adaptáveis caso o processo precise evoluir. Cada instância do formulário

corresponde à documentação requerida pela atividade do processo. Assim como cada instância do campo presente no formulário de uma atividade do processo diz respeito a uma informação necessária à execução do processo de modificação. Para cada execução do processo completo de controle de modificações, é criado um objeto que representa a modificação propriamente dita. A modificação reúne todos os documentos produzidos através do preenchimento dos formulários durante as atividades de GCS. A Figura 4.1 exibe um modelo simplificado dos conceitos envolvidos no sistema.

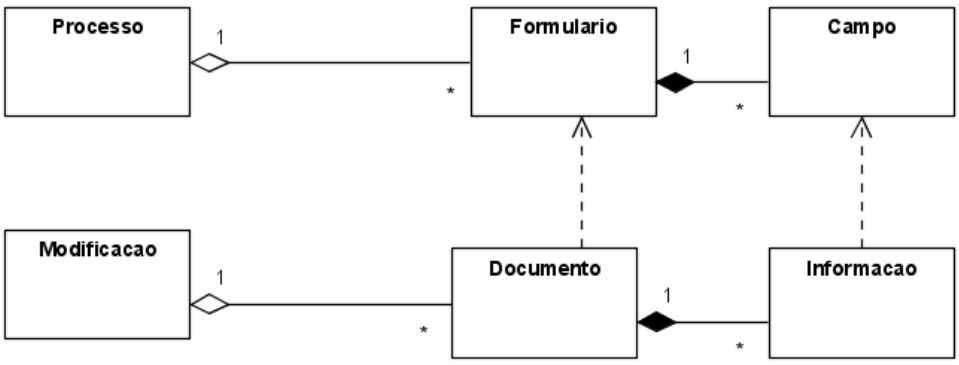

Figura 4.1 - Modelo simplificado do controle de modificações (Fonte: MURTA, 2004).

Ao término do ciclo de vida de uma modificação, o objeto que representa a modificação contém todo o conhecimento referente ao processo, organizado hierarquicamente através das estruturas de atividades, formulários e campos. Esse conhecimento é útil porque possibilita a execução das funções de acompanhamento e auditoria da configuração, além de permitir identificar deficiências no próprio processo de desenvolvimento de software.

Como o sistema de controle de modificações *Odyssey-CCS* armazena o conhecimento referente ao processo de GCS, as informações nele registradas podem ser capturadas pelo mecanismo de rastreabilidade e utilizadas para contextualizar o rastro no processo de desenvolvimento.

4.2.2 – Odyssey-VCS: Sistema de Controle de Versões Para Modelos Baseados no MOF

Atualmente, o foco dos sistemas de controle de versões está no código-fonte. No entanto, quando se versiona artefatos de alto nível de abstração como, por exemplo, modelos *UML*, utilizando a mesma perspectiva de sistemas de arquivos, os resultados obtidos são pouco satisfatórios. Um arquivo é um item de configuração indivisível, que

é persistido como elemento único nos sistemas atuais de controle de versões. Os artefatos pertencentes a *UML* estão em níveis diferentes de uma mesma hierarquia de composição, por exemplo, métodos e atributos fazem parte de classes, e classes fazem parte de pacotes, que por sua vez, fazem parte de modelos. Se o modelo for versionado como um único elemento, não será possível acompanhar a evolução individual de cada elemento de modelagem, como, casos de uso e classes. Por esta razão, uma nova perspectiva em termos de versionamento deve ser adotada para satisfazer a estrutura dos modelos de análise e projetos (MURTA, 2004).

Para possibilitar o relacionamento entre esses diversos elementos foram definidos, no contexto do projeto *Odyssey-SCM*, os conceitos de grão de comparação e grão de versionamento, além de uma nova abordagem baseada nesses conceitos. O grão de versionamento representa os elementos que necessitam ser versionados na hierarquia de composição, tornando-se itens de configuração quando enviados para o repositório. Em um cenário onde os itens de configuração são modelos *UML*, o grão de versionamento poderia ser elementos como, por exemplo, classes e caso de uso. O grão de comparação estabelece artefatos na hierarquia de composição que devem ser utilizados como parâmetros na identificação de conflitos quando a operação de junção[33] (*merge*) é realizada. Portanto, se dois desenvolvedores modificam, concorrentemente, um elemento que é grão de comparação, um conflito ocorre. Esse conflito deve ser resolvido com base na política de resolução de conflitos fornecida pelo próprio sistema.

Odyssey-VCS (OLIVEIRA et al., 2004) é um sistema de controle de versões, independente de *IDE*, que atua sobre modelos descritos por meta-modelos *MOF* (OMG, 2002a), como, por exemplo, a *UML*, fazendo uso de políticas configuráveis para os grãos de comparação e versionamento.

O versionamento proposto pelo *Odyssey-VCS* permite o acompanhamento da evolução individual de cada elemento do modelo, sendo, ainda, uma abordagem extensível, por considerar qualquer modelo baseado no *MOF*. Para controlar as versões de um determinado meta-modelo *MOF*, é necessário estabelecer a hierarquia de composição dos seus elementos e determinar quais são os grãos de comparação e

[33] O tratamento de conflito conhecido como junção (*merge*) une os trabalhos efetuados em paralelo por dois ou mais desenvolvedores sobre um mesmo IC e produz uma nova versão do IC que contém a soma desses trabalhos (MURTA, 2004).

versionamento. Utilizando essa informação, o sistema *Odyssey-VCS* é capaz de lidar com modelos criados a partir desse meta-modelo.

Como a *UML* é um modelo *MOF*, a utilização do *Odyssey-VCS* como sistema de controle de versões permite o controle individual dos elementos de modelo *UML* que foram configurados como grãos de versionamento. Com base nas versões criadas para cada artefato, torna-se possível à detecção de rastros intramodelos e intermodelos no contexto de DBC.

4.3 – Odyssey-WI: A Abordagem Proposta

Nesta seção, é apresentada uma proposta de mecanismo de suporte a rastreabilidade, denominado *Odyssey-WI*, que utiliza o repositório dos sistemas de controle de versões *Odyssey-VCS* e modificações *Odyssey-CCS* como base para a aplicação de técnicas de mineração de dados e identificação de rastros entre artefatos *UML*.

Este mecanismo se propõe a identificar rastros de modificação entre artefatos *UML*, através da integração entre os ambientes de DBC e GCS. Esse procedimento é realizado de forma não intrusiva[34], utilizando dados já armazenados nos sistemas de controle de versões e modificações da GCS, atuando em conjunto com o ambiente de modelagem. A partir da seleção de um artefato *UML*, o mecanismo é capaz de detectar, automaticamente, rastros para outros artefatos *UML*, no auxílio às tarefas de modificação do engenheiro de software. Além disso, indica juntamente com o rastro, informações coletadas nos sistemas de GCS que forneçam algum significado semântico, facilitando o entendimento do próprio rastro e suas futuras modificações.

4.3.1 – Características Gerais

As características gerais da abordagem foram definidas com base na revisão da literatura e nos sistemas de GCS apresentados na Seção 4.2. São elas: Fundamentação dos Rastros (Seção 4.3.1.1), Representação segundo a estrutura 5W+1H (Seção 4.3.1.2), Semântica Configurável (Seção 4.3.1.3), Relevância do Rastro (Seção 4.3.1.4),

[34] Do ponto de vista do usuário, a utilização dos sistemas de GCS faz com que o mecanismo de rastreabilidade faça uso de informações já armazenadas nesses sistemas durante o processo de desenvolvimento do software. Por isso não é necessário nenhum tipo de intervenção ou pedido de novas informações no decorrer do processo.

Abrangência em relação aos tipos de artefatos (Seção 4.3.1.5) e Independência em relação ao Ambiente de Desenvolvimento (Seção 4.3.1.6).

4.3.1.1 – Fundamentação dos rastros

Atualmente, os mecanismos de rastreabilidade têm como desvantagem a falta de semântica nas relações encontradas entre os artefatos de software, que deixam a cargo do engenheiro de software o registro do raciocínio que originou o rastro. Contudo, considerando a realidade dos projetos atuais, onde os cronogramas são justos e os custos escassos, registrar manualmente esse tipo de informação não é prático nem viável em grandes projetos (HASSAN e HOLT, 2003).

A importância da semântica das relações introduzidas no software é comentada em várias abordagens. MURPHY et al. (1995) argumentam a necessidade de relacionar o raciocínio e as idéias produzidas durante a fase de projeto ao código. O relato de um estudo experimental, avaliando a necessidade da documentação do projeto durante a atividade de análise de impacto, constatou que de dois sistemas, um teve uma significativa melhora em relação a corretude e a velocidade das modificações realizadas, quando teve acesso à documentação do projeto (BRATTHALL et al., 2000). Com o intuito de melhorar a documentação, algumas abordagens sugerem a recuperação de padrões de projeto a partir do código-fonte como uma forma de avaliar que decisões foram tomadas durante a fase de implementação (KELLER et al., 1999).

A rastreabilidade indica que existe uma relação entre dois ou mais artefatos. Contudo, esta informação não é suficiente para o completo entendimento do software. Para recuperar o entendimento, pode ser necessário ter que consultar participantes do projeto ou realizar um estudo mais aprofundado. Nesse processo, algumas dúvidas podem surgir como: “Existe um bom motivo para esta relação existir?" ou "Será que essa relação representa algum mal entendido introduzido por algum desenvolvedor?".

A semântica pode significar que a modificação que originou a relação entre os artefatos foi realizada por um desenvolvedor experiente, o que é um possível indicativo de qualidade. Mas também pode indicar ausência desta, caso a relação entre os artefatos tenha sido introduzida apenas para consertar um erro crítico a ser resolvido em um curto espaço de tempo. A partir desse ponto, consideramos que seria de grande valia para a tarefa de manutenção e avaliação do rastro encontrado pelo mecanismo, se existisse, anexado ao rastro, a sua semântica ou algum indício do seu significado.

4.3.1.2 – Representação segundo a estrutura 5W + 1H

Alguns trabalhos na literatura (DOURISH e BELLOTTI, 1992; GUTWIN, 1997, SOHLENKAMP, 1998, VIEIRA, 2003) se baseiam em um conjunto de seis questões para identificar informações que aumentem o entendimento da atividade que está sendo executada, ou da informação que está sendo apresentada. Essas questões são: o que (*What*), quem (*Who*), quando (*When*), onde (*Where*), como (*How*) e por quê (*Why*). A estrutura definida como 5W+1H caracteriza o que é necessário saber para que determinada atividade ou informação seja compreendida. A semântica de cada uma dessas perguntas, e o que elas se propõem a responder, pode variar dependendo do contexto onde as mesmas são utilizadas.

Uma vez identificados os tipos básicos de questionamento, verificamos que é possível fundamentar o rastro, relacionando as informações coletadas do repositório dos sistemas de controle de versões e de modificações. O repositório desses sistemas armazena detalhes relacionados às modificações realizadas no software, que podem ser utilizados no apoio ao entendimento das relações encontradas (HASSAN e HOLT, 2003). Essas informações auxiliam futuras manutenções, identificando não apenas o raciocínio empregado durante a implementação das modificações como também as pessoas envolvidas nesse processo.

É importante frisar que, embora correlacionar as informações de ambos os sistemas não indique propriamente a razão do rastro, as ocorrências capturadas formam todo o histórico dos rastros que foram identificados e servem de base para futuras análises e para a validação do próprio rastro. A Figura 4.2 exemplifica as seis questões básicas que, se respondidas, ajudam a fundamentar o rastro.

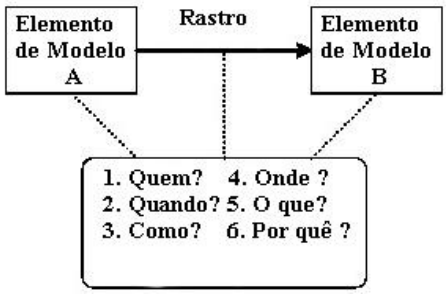

Figura 4.2 - Semântica do rastro de modificação

Nesse caso, verificamos que, para cada requisição de modificação, é possível detectar informações referentes a quem implementou a modificação, quando foi

implementada, como foi implementada, onde foi necessário alterar para implementá-la, o que foi realmente feito e por quê a modificação foi necessária.

Conforme visto no Capítulo 2, no processo de controle de modificações, segundo as normas IEEE (1987) e ISO (1995), são estabelecidas as seguintes atividades, para que os itens de configuração de software possam evoluir de forma controlada: (1) requisição da modificação; (2) classificação da modificação; (3) análise das modificações; (4) avaliação da modificação; (5) implementação, (6) verificação da implementação e (7) geração da configuração de referência. Averiguando as informações que constam usualmente nos formulários de requisição de modificações e nos registros de *check-in* e *check-out,* ocorridos em virtude da execução das atividades relacionadas acima, constata-se que as respostas às seis questões poderiam ser obtidas como exibido na Tabela 4.1.

Seguindo a estrutura 5W+1H, o mecanismo pode categorizar a informação recuperada dos sistemas de controle de versões e modificações, melhorando o entendimento do rastro encontrado. Contudo, vale ressaltar que essa característica torna a abordagem dependente da qualidade do que é registrado pelos desenvolvedores.

Tabela 4.1 - Exemplo de coleta da semântica do rastro (Fonte: MURTA, 2004)**.**

Questão	Descrição
Quem?	Obtido nos registros de *check-in* do sistema de controle de versões e na atividade de implementação do sistema de controle de modificações.
Quando?	Obtido nos registros de *check-in* do sistema de controle de versões e na atividade de implementação do sistema de controle de modificações.
Como?	Obtido no laudo de análise de impacto gerado a partir da atividade de análise, no sistema de controle de modificações.
Onde?	Obtido nos registros de *check-in* no sistema de controle de versões, que relatam quais partes de quais artefatos foram modificadas.
O que?	Obtido no laudo de verificação gerado a partir da atividade de verificação da modificação, que relata o que realmente foi feito, e é necessário para aprovar ou não a integração da modificação na configuração de referência. Também podendo ser recuperado do comentário registrado na operação de *check-in* no sistema de controle de versões.
Por quê?	Obtido no documento de requisição da modificação, que informa o motivo da modificação e serve como base para a continuidade do processo de controle de modificação.

4.3.1.3 – Semântica configurável

A modificação, segundo o sistema de controle de modificações *Odyssey-CCS*, reúne todos os documentos produzidos através do preenchimento dos formulários durante a execução das atividades do processo definido, concentrando aspectos

interessantes referentes à evolução do software. Partindo do princípio que o sistema de controle de modificações utilizado é configurável em relação ao processo e em relação aos formulários das atividades que compõem o processo, verificamos que é viável associar as seis questões, apresentadas na Seção 4.3.1.2, aos campos definidos nos formulários do processo do sistema de controle de modificações.

Isso traz vantagens ao mecanismo de rastreabilidade, uma vez que torna a semântica do rastro configurável, fazendo com que as respostas aos questionamentos levantados através da estrutura 5W+1H possam ser obtidas a partir de uma configuração inicial. Com isso, o engenheiro de software visualiza nas respostas o conteúdo das informações necessárias ao seu entendimento.

Além das informações registradas no sistema de controle de modificações, podem ser acrescentadas as informações referentes ao sistema de controle de versões em algumas questões como exemplificado na Tabela 4.1 da Seção 4.3.1.2.

A abordagem proposta para o mecanismo de rastreabilidade pode apresentar essa característica, uma vez que: (1) o sistema de controle de modificações, utilizado como fonte de dados, configura e armazena o conhecimento referente ao processo, através de uma estrutura hierárquica composta de atividades, formulários e campos e (2) o sistema de controle de versões, também utilizado como fonte de dados, armazena a cada operação de *check-in* informações como, por exemplo, data da operação e usuário responsável, comentário com informações sobre a implementação, além dos artefatos que foram versionados em função do *check-in*.

4.3.1.4 – Relevância do Rastro

Relacionar um significado ou uma medida de relevância ao rastro é uma proposta de grande valia para os mecanismos de rastreabilidade, porque a partir dessa informação torna-se possível avaliar a utilidade ou importância do rastro na atividade que está sendo executada. A relevância do rastro pode ser obtida através das medidas de suporte e confiança utilizadas pelos algoritmos de mineração de dados e apresentadas na Seção 3.4.1 do Capítulo 3.

A medida de suporte indica a (co-) ocorrência de elementos, i.e, que do total de modificações realizadas, x% contém o(s) elemento(s). A medida de confiança indica o quão forte é a presença de um elemento em função de outro, o que equivale a dizer que em y% das vezes que um elemento é modificado, outro também é modificado.

Logo, essas duas medidas fornecem um indicador de quão freqüente é a alteração de dois elementos, sendo que a primeira medida demonstra a freqüência em relação às modificações como um todo, e a segunda medida demonstra a freqüência em relação às modificações que contêm um elemento em particular. A partir desses dois valores, o rastro pode ser avaliado quanto a sua utilidade no contexto da modificação que está sendo realizada.

4.3.1.5 – Abrangência em relação aos tipos de artefatos

Atualmente, a literatura tem sugerido mecanismos de suporte a rastreabilidade voltados para código-fonte (YING, 2003, ZIMMERMANN et al., 2003; SHIRABAD, 2003). No entanto, percebe-se que é importante não apenas atuar no nível de código-fonte, mas também nos modelos de análise e projeto. Ao propagar a modificação para os diferentes níveis de abstração, consideramos artefatos *UML* e não somente o produto final representado pelo código-fonte.

Com a adoção do *MOF* como meta-modelo da *UML* e da atuação do sistema de controle de versões *Odyssey-VCS* na evolução de modelos baseados no *MOF,* tornou-se possível controlar versões de elementos de modelos *UML.* No entanto, o *Odyssey-VCS* atua sobre qualquer modelo baseado no *MOF*, o que significa que não é restrito a *UML*. Essa característica o torna extensível a outros modelos. Alguns trabalhos na literatura têm definido modelos *MOF* para código Java (MATULA, 2003) e XML/DTD (SANTOS et al., 2003). Portanto, é possível estender o versionamento de modelos *UML* para código, definindo um modelo *MOF* para código.

As características do *Odyssey-VCS* favorecem a utilização de seu repositório como base para a proposta de um mecanismo de rastreabilidade por dois motivos: (1) por atuar de forma abrangente nos artefatos produzidos no processo de desenvolvimento do software e (2) por fornecer aos elementos do rastro (itens de configuração do sistema de controle de versões) uma granularidade que depende do grão de versionamento definido.

Desta forma, a granularidade das informações presentes no rastro é variável, a depender do que foi definido como grão de versionamento, ou seja, depende de quais artefatos contém informação sobre versão. Neste caso, o mecanismo de rastreabilidade pode representar rastros levando em conta um grão maior considerando, por exemplo, classes e componentes, ou um grão menor chegando ao nível de métodos e atributos de classes. Além disso, é capaz de relacionar elementos de diferentes modelos *MOF*. Se o

sistema de controle de versões atuar, tanto no modelo *MOF* para código Java, quanto no modelo *UML*, o mecanismo de rastreabilidade relacionará artefatos *UML* com artefatos de código-fonte Java.

Essa característica torna o mecanismo de rastreabilidade extensível a diferentes tipos de artefatos e o diferencia das abordagens atuais (YING, 2003; ZIMMERMANN et al., 2003a) que não chegam a relacionar elementos de modelo *UML* com artefatos de código-fonte utilizando sistemas da GCS.

4.3.1.6 – Independência em Relação ao Ambiente de Desenvolvimento

Os desenvolvedores realizam seu trabalho através de um ambiente de desenvolvimento que permite a edição e criação de modelos de software. O uso de sistemas de controle de versões neste ambiente permite que os elementos de modelos ou artefatos de software produzidos na ferramenta de modelagem sejam controlados de forma disciplinada.

Durante o processo de modificação, o desenvolvedor indica que uma modificação será realizada e seleciona a versão do artefato na qual pretende trabalhar, efetuando *check-out* dos artefatos. Após realizar as modificações necessárias nos artefatos, armazena no repositório, através do processo de *check-in*, uma nova versão do artefato, identificado no sistema de controle de versões como um item de configuração.

Neste cenário, o mecanismo de rastreabilidade necessita saber apenas qual versão de item de configuração o desenvolvedor pretende alterar. Com base nesta informação, o mecanismo é capaz de indicar outros artefatos para a modificação que está sendo realizada, como veremos mais adiante na Seção 4.3.2.

Portanto, não há motivo para que o mecanismo seja dependente do ambiente de desenvolvimento ou ferramental de modelagem utilizado pelo desenvolvedor. Essa característica torna o mecanismo reutilizável em qualquer ambiente de desenvolvimento, não sendo também intrusivo na rotina de trabalho do desenvolvedor. O fato de não ser intrusivo está relacionado à utilização das informações já registradas nos sistemas de controle de versões e modificações no dia-a-dia do desenvolvimento, que evita a interrupção do trabalho.

4.3.2 – Cenário de Utilização

Esta seção descreve o funcionamento da abordagem proposta. A Figura 4.3 apresenta uma visão geral de sua utilização no desenvolvimento de software. A

abordagem foi concebida para permitir a utilização do mecanismo de rastreabilidade dentro de um ambiente de desenvolvimento de software que permita a criação, dentre outros artefatos, de modelos de software baseados na *UML*.

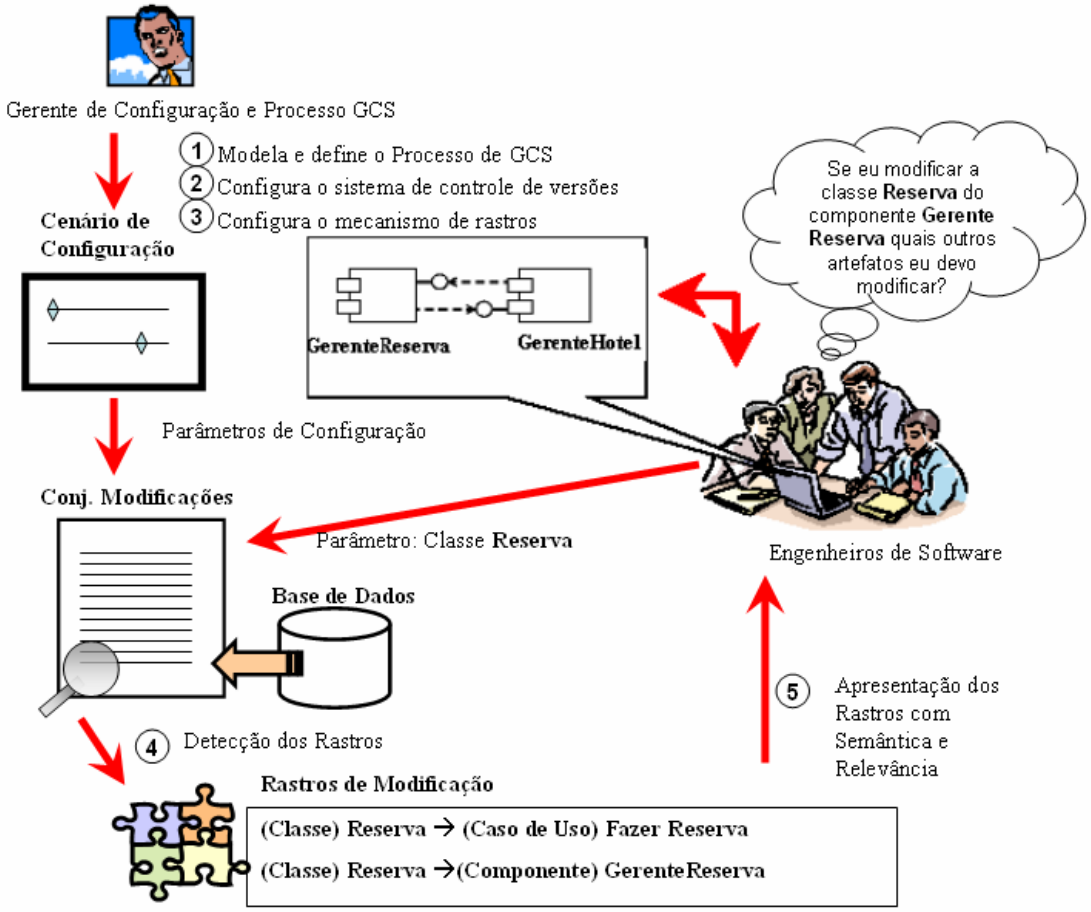

Figura 4.3 - Atividades envolvidas na abordagem Odyssey-WI

O mecanismo tem como pré-requisito a utilização dos sistemas *Odyssey-VCS* e *Odyssey-CCS* durante o desenvolvimento, e necessita de uma configuração inicial para a sua execução. Essa configuração deve ser realizada pelo gerente de configuração responsável por implantar o processo da GCS na organização, cabendo a ele:

A- Em relação ao processo de GCS:

(1) Modelar o processo de controle de modificações;

(2) Definir os formulários que devem ser preenchidos durante o processo;

(3) Definir os campos que deverão constar em cada formulário;

(4) Incluir o meta-modelo da *UML* como modelo *MOF* ao configurar o sistema de controle de versões; e

(5) Configurar grãos de versionamento e comparação no sistema de controle de versões.

B- Em relação ao mecanismo de rastro:

(6) Relacionar os campos do sistema de controle de modificações às questões da estrutura 5W+1H, visto que essas questões servirão para contextualizar o rastro, fornecendo a ele algum significado semântico;

(7) Definir um nível de precisão para os rastros que serão obtidos. Essa última atividade é detalhada na Seção 4.3.2.2.

A etapa de configuração consiste na definição de alguns parâmetros necessários à execução do mecanismo, para que este possa realizar a mineração de dados no repositório dos sistemas de GCS.

Depois de realizada a configuração e iniciado o desenvolvimento do software, o engenheiro de software indica que uma modificação será realizada e seleciona a versão do artefato na qual pretende trabalhar, efetuando *check-out* dos artefatos. Neste momento, caso necessite de orientação, é executado o mecanismo de rastreabilidade.

Para identificar quais outros artefatos poderão ser modificados, em função da alteração de um artefato, o mecanismo extrai do repositório as modificações realizadas no software, e realiza sobre esse conjunto a mineração dos dados. O resultado da mineração depende da configuração inicial definida e é direcionado à modificação que está sendo realizada pelo engenheiro de software, ou seja, considera o artefato que está sendo alterado por ele. A precisão definida pelo gerente de configuração pode retornar um conjunto maior ou menor de rastros. A precisão pode ser baixa, trazendo relações que não são, necessariamente, freqüentes no repositório, mas que existem em maior número, ou alta, trazendo relações que constam no repositório com freqüência, e que, geralmente, existem em menor número.

A precisão é definida para cada uma das medidas explicadas na Seção 4.3.1.4, que são utilizadas pelos algoritmos de mineração de dados quando a técnica de mineração utilizada extrai regras de associação. No Capítulo 3, discutimos na Seção 3.4.1 a utilização dessas medidas para a obtenção das regras.

Por fim, os rastros são organizados e apresentados de forma a guiar o engenheiro de software na modificação que está sendo realizada. Os rastros são apresentados com informação de relevância, como discutido na Seção 4.3.1.4, e com uma semântica associada. A semântica é apresentada de acordo com a configuração também realizada

pelo gerente, contendo as informações dos campos do sistema de controle de modificações preenchidas durante o processo de desenvolvimento.

As seções seguintes apresentam os aspectos gerais desse mecanismo, divididos em três etapas. Para facilitar a compreensão, apresentamos um exemplo na Seção 4.3.2.1. A Seção 4.3.2.2 apresenta a atividade de configuração do mecanismo. A seção seguinte, 4.3.2.3, apresenta a detecção dos rastros e a última seção, 4.3.2.4, discute a apresentação dos rastros.

4.3.2.1 – Exemplo

O exemplo utilizado se encontra na literatura (CHEESMAN e DANIELS, 2000). Trata-se de um sistema voltado para o cadastro de clientes, hotéis, quartos e tipos de quartos. Além desses elementos, o sistema deve ser capaz de fazer reservas e oferecer consultas aos seus usuários. Os principais casos de uso desse sistema são ilustrados na Figura 4.4, onde duas classes de usuários interagem com o sistema: administradores e hóspedes. A função do administrador é manter os dados atualizados e consistentes para que consultas e reservas possam ser realizadas por hóspedes.

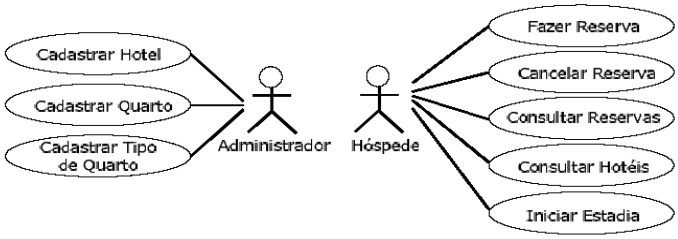

Figura 4.4 - Casos de uso do exemplo Hotelaria

De acordo com o trabalho de TEIXEIRA (2003), podem ser identificados quatro componentes neste exemplo: “Gerente de Reserva”, “Gerente de Tipos de Quarto”, “Gerente de Hotéis” e “Gerente de Hóspedes”, listados na Figura 4.5(a). A geração destes componentes é realizada com base nas classes definidas no modelo da Figura 4.5(b):

- O componente “Gerente de Hóspedes” é composto das classes “Cliente” e “Hóspede”.
- O componente “Gerente de Reserva” é composto da classe “Reserva”.
- O componente “Gerente de Tipos de Quarto” é composto da classe “Tipo de Quarto”.

- O componente “Gerente de Hotéis” é composto das classes “Hotel” e “Quarto”.

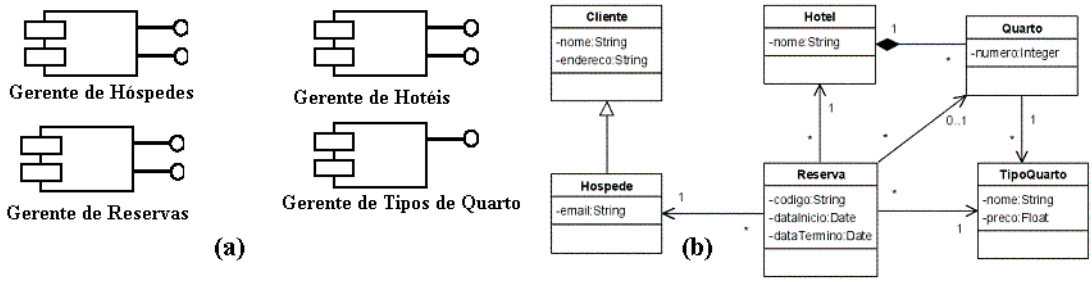

Figura 4.5 - Componentes (a) e classes (b) do exemplo Hotelaria

Para mostrar como a abordagem funciona, assume-se que seis alterações foram requeridas no sistema *Odyssey-CCS*. Essas requisições de modificações são listadas na Tabela 4.2. A partir dessas requisições, novas características foram inseridas no sistema do exemplo.

Tabela 4.2 - Descrição das modificações realizadas no exemplo Hotelaria

Modificação	Descrição
1	Ao efetuar a reserva, verificar a disponibilidade de quartos no hotel para o tipo de quarto selecionado no período determinado para a reserva.
2	Ao consultar a reserva de um hóspede, visualizar o tipo de quarto reservado, assim como sua característica e preço na temporada.
3	Não existindo quarto disponível durante a efetuação da reserva, verificar se esse tipo de quarto se encontra disponível em outro hotel (cadastrado no sistema) da região.
4	Ao efetuar a reserva de um hóspede, não disponibilizar o quarto reservado para futuras reservas (considerar o período reservado).
5	Ao cancelar a reserva de um hóspede, disponibilizar o quarto para nova reserva (considerar o período reservado).
6	Relacionar todos os hotéis da rede hoteleira que já teve o cliente como hóspede, e apresentar como consulta para o cliente.

4.3.2.2 – Configuração do Mecanismo

Como vimos na Seção 4.3.1.3, as informações, que constam usualmente nos formulários das requisições de modificações e nos registros de *check-in* e *check-out* do sistema de controle de versões, fornecidas durante as atividades do processo de desenvolvimento, podem ser utilizadas para prover algum significado semântico ao rastro. No entanto, para que isso seja possível, é necessário que o gerente de

configuração atribua um campo do formulário do sistema de controle de modificações a cada uma das questões da estrutura 5W + 1H.

Essas associações fornecem um significado a cada uma das informações anexadas ao rastro. Para exemplificar o procedimento de configuração, relacionamos cada questão da seguinte forma:

- Quem? → Responsável pela Implementação
- Quando? → Data da verificação da Implementação
- Onde? → Itens alterados pela Implementação
- O que? → Mudanças ocasionadas pela Implementação
- Como? → Alternativas de Implementação
- Por quê? → Descrição da Modificação

Esses campos têm sua origem no processo modelado no sistema *Odyssey-CCS*, conforme visto nas Seções 4.2.1 e 4.3.1.2.

O mecanismo além de incorporar uma maior flexibilidade quanto às informações que são apresentadas junto com o rastro também permite configurar uma precisão para o resultado que será apresentado.

A precisão[35] pode assumir os seguintes valores: baixa, média ou alta. A precisão baixa traz relações que não são, necessariamente, freqüentes no repositório dos sistemas de GCS. Portanto, traz um número maior de rastros, onde relações incorretas podem aparecer.

Por exemplo, se uma única modificação, em 100 já realizadas, altera a classe Hotel em um primeiro momento, gerando uma versão deste elemento, e posteriormente, em um segundo momento, desfaz a alteração realizada, o sistema de controle de versões associa uma nova versão para a última alteração, mesmo que, no decorrer do processo, o estado da classe Hotel não tenha sido alterado. Esse dado de versão, armazenado no repositório, traz, com a utilização do mecanismo, um rastro em relação a outros elementos, alterados durante a mesma modificação, que não deveria existir.

[35] Assumimos que a precisão alta é referente a valores de suporte e confiança, ambos, altos. O inverso vale para a precisão baixa. Se uma única modificação inseriu um rastro entre dois elementos e se ambos não foram mais modificados no desenvolvimento, a confiança (como visto no Capítulo 3) será de 100% e o suporte será 1% se o total de modificações realizadas for 100. Este caso não é retornado pelo mecanismo, quando a configuração da precisão é alta. Mas se a precisão for baixa, pode vir a ser, principalmente, se inicialmente o valor atribuído à medida de confiança for muito baixo, como 0,01.

Esse tipo de rastro só aparece no resultado final, representando 1% no total de modificações, se a precisão for baixa. Ao contrário, a precisão alta traz somente as relações que são freqüentes. Com uma precisão alta, o rastro do exemplo anterior, não apareceria.

4.3.2.3 – Detecção dos Rastros

Como o repositório dos sistemas de controle de versões e modificações armazena todas as alterações realizadas nos artefatos do software durante o processo de desenvolvimento, torna-se necessário para a abordagem selecionar o conjunto de dados no qual a mineração de dados deverá atuar para detectar os rastros. Uma primeira etapa da detecção consiste na seleção dos dados e composição da base. Uma segunda etapa é a análise da base através da aplicação da mineração de dados. O resultado final são os rastros.

A detecção dos rastros ocorre toda vez que o engenheiro de software desejar saber qual artefato modificar em função da alteração de um artefato em particular.

4.3.2.3.1 – Base para a Mineração de Dados

Como o processo de detecção de rastros depende do que foi registrado no repositório, devemos assumir que algumas modificações já foram implementadas durante o desenvolvimento, e no decorrer do processo, foram utilizados os sistemas de controle de versões *Odyssey-VCS* e de modificações *Odyssey-CCS*.

No processo de controle de modificações, a atividade de implementação se inicia quando o desenvolvedor seleciona a modificação que deseja implementar. Com base no exemplo da Seção 4.3.2.1, digamos que o desenvolvedor escolha a modificação 1. Ao implementá-la, o desenvolvedor criou e alterou artefatos como demonstra a Tabela 4.3. Vale ressaltar que o modelo é sempre modificado quando qualquer elemento contido nele é alterado. Portanto, ele sempre é versionado. No entanto, para evitar repetições, optou-se por omiti-lo na Tabela 4.3 e na Tabela 4.4.

Tabela 4.3 - Versões dos artefatos gerados para uma modificação

Tipo de Artefato	Elementos Modificados Representação: Artefato (número da versão considerando cada operação de *check-in*)
Pacotes	Pacote de Casos de Uso (v 1), Pacote de Classes (v 2), Pacote de Componentes (v 2).
Casos de Uso	Fazer Reserva (v 1).
Classes	Quarto (v 1), Hotel (v 1), Reserva (v 2), Hóspede (v 1), Cliente (v 1) e Tipo de Quarto (v 1).
Componentes	Gerente de Reserva (v 2), Gerente de Hotéis (v 1), Gerente de Hóspedes (v 1) e Gerente de Tipo de Quarto (v 1).

Cada modificação, quando implementada, gera um conjunto de versões de artefatos. O sistema de controle de versões permite que estes artefatos sejam obtidos do repositório através do processo de *check-out*, modificados dentro do espaço de trabalho do desenvolvedor, e depois retornados ao repositório através do processo de *check-in*. Contudo, a realidade mostra que nos projetos, vários *check-ins* são necessários durante um certo período de tempo, para que a implementação seja concluída. Como cada operação de *check-in* gera uma única versão do artefato, concluímos que foram necessários dois *check-ins* para que a implementação da primeira modificação da Tabela 4.2 fosse concluída.

Vale ressaltar que os pacotes, neste exemplo, também foram versionados devido à hierarquia de composição dos elementos. Nesse processo, como os sistemas *Odyssey-VCS* e *Odyssey-CCS* estão integrados, acessando um único repositório, cada versão está obrigatoriamente associada à modificação que a originou.

Seguindo o processo e após completar a implementação das seis modificações listadas na Tabela 4.2, o desenvolvedor gerou as versões dos artefatos da Tabela 4.4 em cada uma das modificações.

As versões, representadas entre parênteses na Tabela 4.4, estão de acordo com o versionamento global adotado pelo sistema de controle de versões. No versionamento global, a nova versão de um artefato, ao se fazer um *check-in*, está associada a todos os elementos que mudaram com aquele *check-in.* No exemplo, a última configuração gerada para a modificação 5, gerou a versão 7 para os artefatos que foram modificados no último *check-in*. Os artefatos que estão com a versão 6 fazem parte da configuração anterior do repositório realizada após o primeiro *check-in* dessa modificação. Com o versionamento global, percebemos exatamente o que mudou de uma configuração para outra.

Tabela 4.4 - Modificações no repositório dos sistemas de GCS

N°	Elementos Modificados Representação: Artefato (número da versão considerando cada operação de *check-in*)			
	Pacotes	**Casos de Uso**	**Classes**	**Componentes**
1	Pacote de Casos de Uso (v 1), Pacote de Classes (v 2), Pacote de Componentes (v 2).	Fazer Reserva (v 1).	Cliente (v 1), Tipo de Quarto (v 1), Quarto (v 1), Hotel (v 1), Reserva (v 2), Hóspede (v 1).	Gerente de Reserva (v 2), Gerente de Hotéis (v 1), Gerente de Hóspedes (v 1), Gerente de Tipo de Quarto (v 1).
2	Pacote de Casos de Uso (v 3), Pacote de Classes (v 3), Pacote de Componentes (v 3).	Consultar Reserva (v 3).	Tipo de Quarto (v 3), Reserva (v 3).	Gerente de Tipo de Quarto (v 3), Gerente de Reserva (v 3).
3	Pacote de Casos de Uso (v 4), Pacote de Classes (v 4), Pacote de Componentes (v 4).	Fazer Reserva (v 4).	Reserva (v 4), Hotel (v 4).	Gerente de Reserva (v 4), Gerente de Hotéis (v 4).
4	Pacote de Casos de Uso (v 5), Pacote de Classes (v 5), Pacote de Componentes (v 5).	Fazer Reserva (v 5).	Reserva (v 5), Hotel (v 5).	Gerente de Reserva (v 5) e Gerente de Hotéis (v 5).
5	Pacote de Casos de Uso (v 6), Pacote de Classes (v 7), Pacote de Componentes (v 7).	Cancelar Reserva (v 6).	Reserva (v 6), Hotel (v 7).	Gerente de Reserva (v 6), Gerente de Hotéis (v 7).
6	Pacote de Casos de Uso (v 8), Pacote de Classes (v 8), Pacote de Componentes (v 8).	Consultar Hotéis (v 8)	Hotel (v 8), Hóspede (v 8)	Gerente de Hotéis (v 8) e Gerente de Hóspedes (v 8)

Cada linha da Tabela 4.4 corresponde a uma modificação realizada no repositório. Durante o processo, sempre que uma implementação é iniciada, o engenheiro de software seleciona uma versão de um artefato para a alteração, realizando uma operação de *check-out*.

Ao selecionar as modificações que devem ser analisadas para a detecção de rastros, consideramos três condições:

(1) A análise[36] para a detecção de rastros deve ser feita sobre a base, considerando modificações que ocorreram até a versão selecionada.

[36] Os rastros que poderão ajudar o engenheiro de software na atividade de implementação devem estar condizentes com a versão do artefato selecionado, i.e, devem ser coerentes com as modificações realizadas anteriormente à versão selecionada. Sendo assim, por exemplo, se um artefato Reserva está na versão 6, ao realizar o *check-out* da versão 5 para alteração, verificamos rastros de modificações realizadas no repositório anteriormente à versão 5. Neste caso, é desconsiderada a versão 6.

(2) Devem ser selecionadas apenas modificações concluídas, i.e, verificadas no processo de controle de modificações, que estejam em uma configuração de referência. As modificações em um estágio anterior a este podem retornar à atividade de implementação, caso alguma correção tenha que ser realizada, alterando o conjunto de elementos modificados. Por esse motivo, devem ser desconsideradas.

(3) Apenas as modificações que contém o artefato, que será alterado pelo engenheiro de software, devem constar na base.

Esta última condição retorna uma base onde apenas as modificações que contém o artefato que está sendo modificado pelo desenvolvedor aparecem. Isso faz sentido, uma vez que procuramos por rastros que indiquem quais outros artefatos devem ser alterados no decorrer desta modificação.

Assumindo que todas as seis modificações da Tabela 4.4 estão concluídas e que o desenvolvedor selecionou a versão 6 da classe Reserva[37], ao fazer *check-out*, concluímos que apenas as cinco primeiras modificações da Tabela 4.4 serão selecionadas para análise dos rastros, i.e, apenas as modificações já concluídas, que têm a classe Reserva com versão até a 6 deverão ser selecionadas no repositório dos sistemas de GCS.

4.3.2.3.2 – Aplicação do algoritmo de Mineração de Dados

A aplicação de mineração de dados pode ser utilizada para responder perguntas como "Desenvolvedores que modificam esse elemento modificam também quais outros?". Em termos gerais, a mineração por regras de associação encontra relações entre elementos, através da procura por conjuntos de elementos freqüentemente encontrados em transações[38] na base de dados. Essas relações, na literatura, são denominadas regras de associação. As modificações selecionadas na seção anterior compõem a base de transações cujos dados serão analisados no processo de mineração.

Dentre os algoritmos de mineração estudados no Capítulo 3, optamos por utilizar um algoritmo baseado no *Apriori*. O algoritmo *Apriori* faz uso de duas medidas: suporte e confiança. Para a aplicação do algoritmo, é necessário configurar de antemão valores iniciais para essas medidas. Esses valores devem ser atribuídos anteriormente à

[37] Neste caso, o suporte da classe Reserva é igual a 83%.

[38] A base de dados, neste caso, é um repositório que armazena todas as alterações realizadas nos artefatos do software durante o processo de desenvolvimento, onde cada transação na base é uma modificação implementada através de uma ou mais operações de *check-in* no sistema de controle de versões.

aplicação da técnica e detecção dos rastros pelo gerente de configuração. Como a mineração de dados trabalha com elementos freqüentemente encontrados no repositório, a medida de suporte serve para descartar elementos que não são freqüentes.

O resultado da aplicação da mineração na base de transações é uma regra de associação, representada da forma: antecedente → conseqüente. No contexto dessa dissertação, antecedente é o elemento selecionado para a modificação pelo engenheiro de software e conseqüente é, segundo a técnica de mineração, o que deve ser modificado em conjunto com o elemento antecedente. Inicialmente, o algoritmo de mineração recebe três parâmetros para o processamento: (1) número total de modificações realizadas no repositório da GCS, (2) base de transações contendo o elemento antecedente, e (3) medidas de suporte e confiança fornecidas pelo gerente de configuração (como visto na Seção 4.3.2.2).

O algoritmo percorre a base de transações e retorna todos os elementos associados ao antecedente, **calculando** para cada associação o suporte e a confiança. O cálculo do suporte leva em consideração o número total de modificações realizadas no repositório dos sistemas de controle de versões e modificações, uma vez que representa a freqüência do elemento no total de alterações realizadas. A medida de confiança, por definição, leva em consideração o conjunto das transações que contém o antecedente. O algoritmo assume que na base estão presentes transações que incluem o antecedente como elemento, usado neste processamento como referência para a mineração. A Tabela 4.5 ilustra o resultado do cálculo para a base exibida na Tabela 4.4, considerando a classe Reserva como antecedente. Na Tabela 4.5, foi omitido o modelo do sistema de Hotelaria por este apresentar suporte e confiança em 100%.

Tabela 4.5 - Suporte e confiança das associações encontradas na base de transações

N°	Conseqüente	Suporte	Confiança
1	Pacote de Casos de Uso	83%	100%
2	Pacote de Classes	83%	100%
3	Pacote de Componentes	83%	100%
4	Gerente de Reserva	83%	100%
5	Hotel	67%	80%
6	Gerente de Hotéis	67%	80%
7	Fazer Reserva	50%	60%
8	Tipo Quarto	33%	40%
9	Gerente de Tipo Quarto	33%	40%
10	Cliente	17%	20%
11	Hóspede	17%	20%
12	Gerente de Hóspedes	17%	20%
13	Quarto	17%	20%
14	Cancelar Reserva	17%	20%
15	Consultar Reserva	17%	20%

O algoritmo utiliza as medidas de suporte e confiança fornecidas pelo gerente de configuração com a finalidade de **descartar** relações não relevantes. Isso acontece ao final do processamento. Caso o limite para suporte seja 50% e para a confiança seja 60%, são descartadas as associações de 8 a 15 e retornadas aquelas com medidas acima do limite como resultado final da mineração.

Esse algoritmo foi baseado no *Apriori*, e adaptado, basicamente, devido aos propósitos da abordagem. Dentre as adaptações realizadas podemos destacar:

(1) Restringe a base, de início, considerando apenas as transações que contém o antecedente como elemento, trabalhando com um conjunto menor de dados.

(2) Considera um único conseqüente no resultado, evitando combinações de elementos e recalculo da medida de suporte no decorrer do processamento, percorrendo a base apenas uma vez.

(3) Descarte (de acordo com medidas fornecidas) apenas ao final do processamento, possibilitando a reutilização dos resultados para outros valores de medidas de suporte e confiança.

4.3.2.4 – Apresentação dos Rastros de Modificação

Para que exista um entendimento maior quanto à utilidade do rastro para a modificação que será realizada pelo desenvolvedor, é necessário associar informações que forneçam semântica ao mesmo. Como apenas a apresentação do rastro não é suficiente durante a tarefa de manutenção, a relevância e a semântica do rastro também devem ser apresentadas.

A relevância é baseada nas medidas de suporte e confiança e obtida na etapa de detecção dos rastros. Já a semântica é obtida a partir da execução de dois passos:

(1) Recuperação das modificações relacionadas ao conjunto de elementos presentes no rastro.

(2) Recuperação das informações registradas no repositório a partir de cada uma das modificações encontradas no passo anterior.

Para exemplificar como o mecanismo recupera a semântica do rastro, voltamos ao exemplo da Seção 4.3.2.1. No exemplo, o conjunto de elementos {Reserva, Hotel} do rastro Reserva → Hotel (suporte: 67%, confiança: 80%), aparece nas modificações 1, 3, 4 e 5 da Tabela 4.5. A partir do conjunto de modificações {1, 3, 4 e 5}, o mecanismo recupera as informações registradas nos sistemas de controle de versões e modificações.

Os registros de *check-in* do sistema de controle de versões *Odyssey-VCS* armazenam informações de quais artefatos, neste processo, retornaram ao repositório (*Where*), de quando este foi realizado (*When*) e por quem (*Who*). O gerente de configuração, como visto na Seção 4.3.2.2, define os campos do formulário do sistema de controle de modificações *Odyssey-CCS* para cada uma das questões da estrutura 5W+1H. Com base nessas informações, o mecanismo incorpora no rastro a semântica da Tabela 4.6, apresentando, em algumas questões, informações armazenadas pelos dois sistemas.

Na Tabela 4.4, as modificações 1 e 5 tiveram duas operações de *check-in* realizadas durante a atividade de implementação. No entanto, optamos por exibir na questão "Quando" (*When*) o último *check-in* realizado. Essa informação indica quando a tarefa de implementação foi finalizada.

Tabela 4.6 - Semântica associada ao rastro

Questão 5W + 1H	**Semântica do Rastro**
Quem?	***Odyssey-VCS:*** Hamilton ***Odyssey-CCS:*** Hamilton
Quando?	***Odyssey-VCS:*** 20/02/2005 ***Odyssey-CCS:*** 21/02/2005
Onde?	***Odyssey-CCS:*** Gerente de Hotéis e Gerente de Reserva ***Odyssey-VCS:*** Hotel, Reserva, Gerente de Hotéis, pacote de Classes, Cancelar Reserva, pacote de Casos de Uso, Gerente de Reserva, pacote de Componentes, Modelo Hotelaria.
O que?	***Odyssey-CCS:*** Método efetuaReserva() modificado, verificando se teve cancelamento no período, além da implementação dos métodos sugeridos.
Como?	***Odyssey-CCS:*** Método cancelaReserva() no gerente de Reserva deve ser implementado. Considerar quarto alocado para a reserva, retornado pelo método existeQuarto() no Gerente de Hotel.
Por quê?	***Odyssey-CCS:*** Ao cancelar a reserva de um hóspede, disponibilizar o quarto para nova reserva (considerar o período reservado).

É importante frisar que algumas informações são preenchidas pelo engenheiro de software no sistema de controle de modificações durante o processo de desenvolvimento, enquanto outras são automaticamente registradas pelo sistema de controle de versões. Como visto na Seção 4.3.1.2, as informações referentes às questões "Quem", "Quando" e "Onde" são obtidas dos registros de *check-in* do sistema de controle de versões. Por isso, independente do que o engenheiro de software registre no sistema de controle de modificações durante o processo, existe a garantia de que as informações referentes às questões "Quem", "Quando" e "Onde" serão recuperadas pelo mecanismo.

Pelo fato do mecanismo incorporar na semântica do rastro informações advindas de dois sistemas da GCS torna-se possível realizar o cruzamento de algumas informações em cada uma das questões citadas anteriormente. Por exemplo, como a questão "Quem" é respondida com os dados recuperados de dois sistemas, o engenheiro de software pode verificar se o desenvolvedor que implementou a modificação e gerou a versão do novo artefato no sistema de controle de versões foi realmente o desenvolvedor indicado no sistema de controle de modificações para implementá-la. Na Tabela 4.6 percebe-se que a informação recuperada do Odyssey-VCS está consistente com a informação recuperada do Odyssey-CCS para a questão "Quem".

O cruzamento dos dados dos sistemas da GCS ajuda a encontrar inconsistências nas informações do rastro. Caso as informações apresentadas estejam inconsistentes, poderá estar ocorrendo o registro inadequado de informações no sistema de controle de modificações.

4.4 – Conclusões

A atividade de manutenção exige um certo nível de entendimento do software. Na maioria dos casos, ter uma documentação atualizada durante a tarefa de entendimento do software, que representa 50% a 90% do esforço de manutenção, pode ser significativo (STANDISH, 1984). Na literatura, várias abordagens (YING, 2003; ZIMMERMANN et al., 2003a, SHIRABAD, 2003; HASSAN e HOLT, 2003) têm sido propostas com o intuito de ajudar este processo de entendimento, atuando em código-fonte. No entanto, devido a sua atuação restrita, não atendem a contento as tarefas de manutenção, que devem propagar todas as modificações realizadas em código para os modelos de análise e projeto.

Essa necessidade pode ser atendida por mecanismos de rastreabilidade que identifiquem rastros de modificação entre artefatos *UML* e apresentem as informações para análise ao engenheiro de software, interferindo o mínimo necessário no seu trabalho.

A proposta do mecanismo *Odyssey-WI* é detectar, através da aplicação de técnicas de mineração nos repositórios de sistemas da GCS, rastros entre artefatos *UML* que devem ser modificados em conjunto, fornecendo juntamente com o rastro, informações que facilitem o entendimento e as futuras modificações.

Nessa proposta, a escolha em utilizar o sistema de controle de versões *Odyssey-VCS* se deve a sua atuação em qualquer modelo *MOF*. Essa característica não restringe

a proposta do mecanismo de rastreabilidade apenas aos modelos da *UML,* permitindo que iniciativas na definição de modelos *MOF*, como, por exemplo, para código-fonte Java, estendam os artefatos identificados no rastro a todos os produtos do processo de desenvolvimento de software. Além disso, a abordagem de políticas configuráveis do sistema de controle de versões torna a granularidade dos artefatos utilizados no rastro também configurável. No entanto, a dissertação teve seu foco nos artefatos *UML* apesar da abordagem ser extensível a outros artefatos como elementos de código-fonte.

A escolha do sistema de controle de modificações *Odyssey-CCS* se deve não somente à sua integração com o sistema de controle de versões, mas também à abordagem configurável e extensível que este sistema provê ao processo de controle de modificações. Ao estruturar o conhecimento referente ao processo, através de uma hierarquia composta de atividades, formulários e campos, esse sistema provê flexibilidade ao próprio processo. Devido a essa flexibilidade, esse conhecimento pode ser relacionado ao rastro também de forma flexível e configurável.

O fato de ser extensível e configurável torna essa abordagem diferente das, até então, encontradas na literatura (YING, 2003; ZIMMERMANN et al., 2003a; SHIRABAD, 2003; HASSAN e HOLT, 2003).

Diferentemente de algumas abordagens (LETELIER, 2002; RAMESH e JARKE, 2000) que propõem um meta-modelo para associar a rastreabilidade entre elementos, a proposta, apresentada neste capítulo, têm por objetivo, atuar de forma automática, descobrindo relações existentes no repositório através de mineração de dados. A aplicação desta proposta, no contexto do DBC, possibilita algumas vantagens interessantes para o engenheiro de software. A identificação automática dos rastros permite a detecção de falhas de projeto. Nesse sentido, é possível realizar uma análise arquitetural do software encontrando dependências que indiquem um alto acoplamento entre artefatos não correlatos. O mecanismo também favorece a atividade de análise de impacto das modificações, minimizando complicações como a introdução de novos erros e alterações incompletas durante a modificação.

Veremos no Capítulo 5 como a proposta foi implementada no ambiente *Odyssey*, mostrando um exemplo de utilização da ferramenta e, ainda, como essas vantagens poderiam ser obtidas.

Capítulo 5 – Protótipo *Odyssey-WI*

5.1 – Introdução

No Capítulo 4, foi apresentada a proposta do mecanismo de rastreabilidade entre artefatos, denominado *Odyssey-WI*, cujo propósito é auxiliar o engenheiro de software durante as atividades de construção e manutenção do software através da detecção e notificação de rastros de modificação que indiquem quais artefatos *UML* devem ser modificados em conjunto. Para tornar essa proposta viável na prática, reduzindo o tempo e o esforço das atividades relacionadas à DBC, apresentamos, neste capítulo, a implementação de um protótipo deste mecanismo, no contexto do ambiente de desenvolvimento *Odyssey* (WERNER et al., 2003).

O ambiente *Odyssey* visa apoiar a reutilização de software através da Engenharia de Domínio, Linha de Produto e DBC. Neste contexto, a rastreabilidade tem como principal objetivo facilitar a identificação dos artefatos do domínio em todos os níveis de abstração, ou seja, desde as visões conceituais e funcionais do domínio ao longo de visões mais detalhadas como modelos de casos de uso, classes, componentes e interação. Neste ambiente, os diversos modelos têm na rastreabilidade uma forma de facilitar o entendimento do domínio e a sua reutilização. No entanto, apesar das vantagens que a rastreabilidade traz ao ambiente, os rastros entre os artefatos não são capturados de forma automática, fazendo com que o analista de domínio seja obrigado a relacionar explicitamente os artefatos em diferentes níveis de abstração.

Nas seções a seguir, são detalhadas as características atuais do ambiente e de como o mecanismo de rastreabilidade *Odyssey-WI* pode ser aplicado neste contexto, indicando, automaticamente, rastros de modificação entre artefatos *UML*, encontradas nas modificações freqüentemente realizadas durante as atividades do processo de desenvolvimento de software. Esses rastros podem não somente melhorar o entendimento do domínio como indicar falhas de projeto, caso seja encontrada uma dependência que interfira na coesão de um componente, ou um acoplamento entre artefatos que não deveria existir. Dentre as vantagens que o mecanismo traz com a detecção de rastros, pode ser citado ainda o auxílio à atividade de análise de impacto através da indicação do que deve ser modificado em conjunto. A indicação de

modificações futuras, como dito anteriormente, minimiza erros e inconsistências no software.

Este capítulo está organizado em oito seções, além desta introdução. Na Seção 5.2, é apresentada parte da arquitetura do projeto *Odyssey-SCM* no qual o protótipo está inserido. As seções seguintes detalham cada uma das camadas desta arquitetura. A Seção 5.3 discute o ambiente *Odyssey* e a Seção 5.4, o repositório utilizado no contexto do projeto. A seção seguinte 5.5 menciona os sistemas da GCS, indicando sua utilização por um desenvolvedor durante o processo de desenvolvimento de software. A Seção 5.6 descreve a camada de transporte utilizada pelo protótipo *Odyssey-WI* e a Seção 5.7, a implementação do protótipo *Odyssey-WI* propriamente dito. A Seção 5.8 apresenta um exemplo de utilização do protótipo. Por último, na Seção 5.9, são apresentadas as considerações finais.

5.2 – Arquitetura do Projeto Odyssey-SCM

Esta seção apresenta parte da arquitetura proposta para o projeto *Odyssey-SCM* (MURTA et al., 2004). Como foi visto no Capítulo 4, esse projeto é composto de cinco partes e propõe uma abordagem abrangente para a GCS em ambientes DBC: Processos e normas (*Odyssey-SCMP*), Sistema de Controle de Modificações (*Odyssey-CCS*), Sistemas de Controle de Construções e Liberações (*Odyssey-BRCS*), Sistema de Controle de Versões (*Odyssey-VCS*) e Sistema de Integração dos espaços de trabalho de GCS e DBC (*Odyssey-WI*).

A arquitetura do projeto e a elaboração da primeira versão do processo[39] de GCS para DBC (*Odyssey-SCMP*), tratando questões específicas de DBC nas funções de identificação e controle da configuração (DANTAS et al., 2003), foram realizadas pelo grupo do próprio projeto. O grupo, como um todo, participou da elaboração de toda a infra-estrutura, incluindo, neste caso, as ferramentas de apoio para a importação e exportação de *XMI* (XML MetaData Interchange) (OMG, 2003c) pelo ambiente de desenvolvimento *Odyssey* (Seção 5.6), e definição do meta-modelo para persistência dos dados do projeto (Seção 5.4.1). Cada camada desta arquitetura, apresentada na Figura 5.1, é posteriormente detalhada nas seções seguintes.

[39] A partir da análise dos processos propostos pela IEEE e ISO (IEEE, 1990; ISO 1995), foram identificados pontos de adaptação que possibilitariam acrescentar suporte ao DBC.

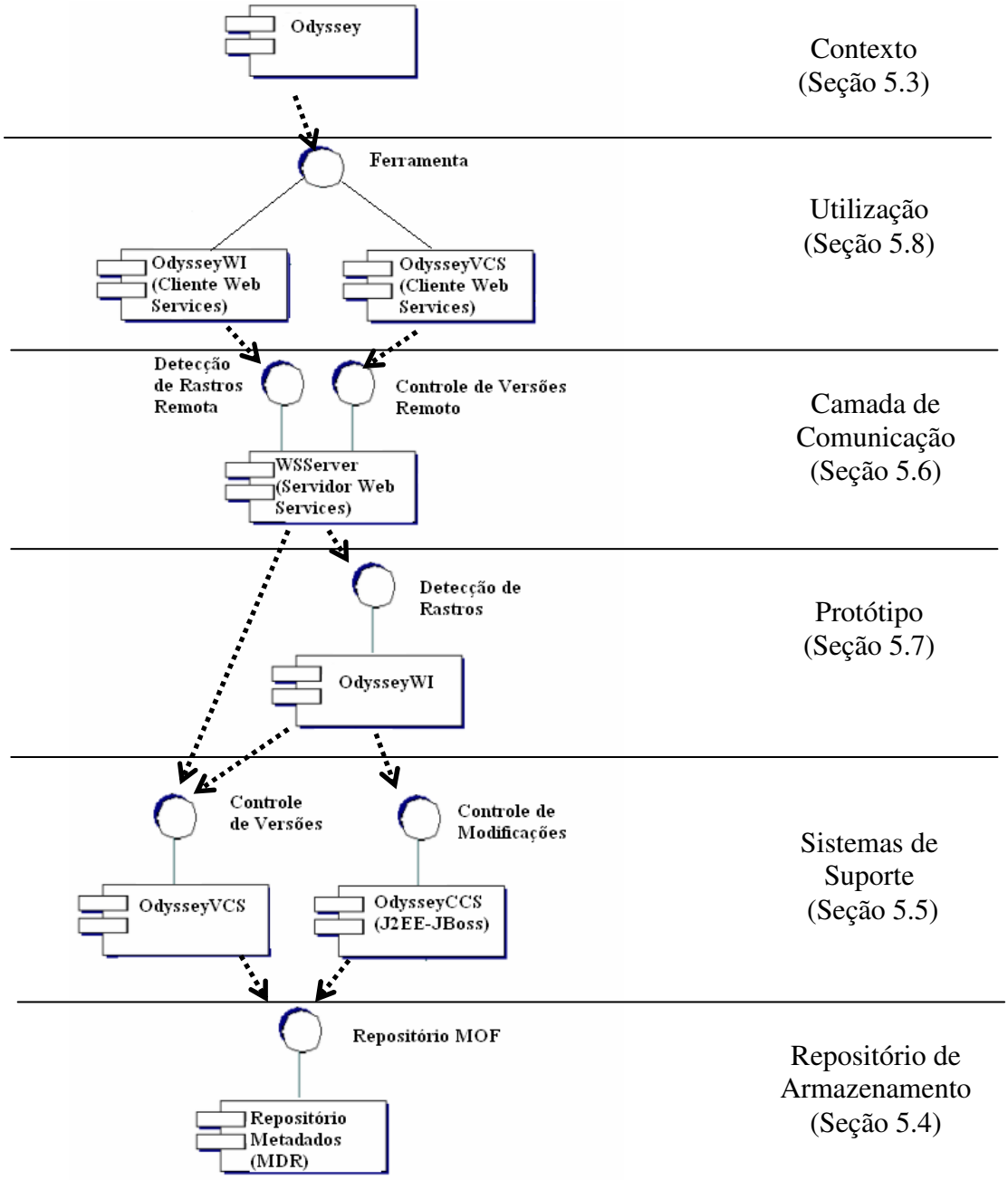

Figura 5.1 - Parte da arquitetura do projeto Odyssey-SCM

As ferramentas estão disponíveis para uso no ambiente *Odyssey* através de um mecanismo denominado carga dinâmica de componentes (MURTA et al., 2004). Este mecanismo permite a instalação de componentes sob demanda. Além das funcionalidades essenciais, o ambiente *Odyssey* oferece uma série de componentes (por exemplo, clientes *Odyssey-VCS* e *Odyssey-WI*) opcionais que podem ser instalados dinamicamente. A lista de componentes é carregada a partir do ambiente através de um arquivo em formato *XML* (W3C, 2004) disponível em um servidor específico. Com base na lista, é permitido selecionar os componentes que se deseja instalar no ambiente e

recuperar do servidor aqueles que foram selecionados para a instalação, através da utilização do protocolo http para a comunicação entre o servidor e o ambiente.

Considerando que, tanto o sistema de controle de versões como o mecanismo de rastreabilidade são independentes em relação a *IDE* usada para o desenvolvimento, as ferramentas do lado cliente, para serem utilizadas no *Odyssey*, devem ter um meio do desenvolvedor acessar o serviço remoto, referente ao controle de versões e a detecção de rastros de modificação. Ambos os serviços são providos na interface do componente *WSServer*. Esses serviços podem ser acessados por qualquer ferramenta que pretende atuar como cliente em uma *IDE*. No *Odyssey*, o mecanismo de carga dinâmica, como explicado anteriormente, exige que o cliente implemente a interface *Ferramenta*, que oferece, basicamente, alguns menus com opções para que o usuário possa interagir com a ferramenta na *IDE*.

Neste cenário, a camada de transporte entre as ferramentas e a *IDE* utiliza *Web Services* (W3C, 2005) como meio de transporte e como uma oportunidade de disponibilizar os serviços via Internet. O formato de representação dos dados que trafegam nessa camada de transporte segue a especificação *XMI*. A utilização de *Web Services* e de *XMI* na camada de transporte torna as ferramentas aptas a serem utilizadas por qualquer outro cliente.

Os serviços do componente *WSServer* acessam tanto os métodos relacionados ao controle de versões de artefatos como, por exemplo, *check-in* e *check-out*, implementados na interface do sistema de controle de versões *Odyssey-VCS*, quanto os métodos relacionados à detecção de rastros de modificação, implementados no mecanismo de rastreabilidade *Odyssey-WI*.

Os sistemas de controle de versões *Odyssey-VCS* e de modificações *Odyssey-CCS* fazem parte do contexto do projeto *Odyssey-SCM*. Como vimos no Capítulo 4, o mecanismo de rastreabilidade necessita da utilização de ambos os sistemas para o seu funcionamento. A utilização de sistemas de GCS é essencial para que o mecanismo consiga relacionar quais modificações implicaram em quais versões no repositório, e assim detectar os rastros. O uso específico dos sistemas *Odyssey-VCS* e *Odyssey-CCS* deve-se ao fato destes proverem uma maior flexibilidade em relação ao tipo de artefato versionado (como elementos de modelo *UML*) e em relação à configuração do processo de controle de modificações. O sistema *Odyssey-CCS* está disponível via Internet, permitindo que o usuário, seja o próprio desenvolvedor ou cliente, registre novas

requisições de modificação, ou informações sobre o andamento de uma modificação específica independente de sua localização.

O projeto *Odyssey-SCM* faz uso de um repositório comum que utiliza a ferramenta *NetBeans MDR* (Metadata Repository) (MATULA, 2003) para persistir as versões de modelos criados no contexto de meta-modelos *MOF* e as modificações que motivaram essas versões. Na Figura 5.1, o componente que representa o repositório é acessado pelos componentes *Odyssey-VCS* e *Odyssey-CCS*.

5.3 – Contexto de Utilização

Odyssey (WERNER et al., 2003) é um ambiente de reutilização que oferece ferramentas para o apoio automatizado do desenvolvimento para reutilização e com reutilização. A infra-estrutura suporta as atividades de Engenharia de domínio, definidas por um processo próprio chamado *Odyssey-DE* (BRAGA, 2000), assim como as atividades de Engenharia de Aplicação, através do processo *Odyssey-AE* (MILLER, 2000). Seu enfoque principal está em especificar e estruturar modelos de domínio que possam ser reutilizados posteriormente pela equipe de desenvolvimento. O conhecimento contido nos modelos de domínio abrange desde elementos conceituais até artefatos de projeto (como, por exemplo, classes, tipos de negócio e componentes), possibilitando que sejam reutilizados durante o desenvolvimento de novas aplicações, reduzindo tempo e esforço nas atividades do processo.

O ambiente *Odyssey* é composto de ferramentas que procuram automatizar as diversas etapas definidas pelos processos *Odyssey-AE* e *Odyssey-DE*. Podemos citar ferramentas para a documentação de componentes (MURTA, 2001), especificação e instanciação de arquiteturas específicas de domínios (XAVIER, 2001), camada de mediação e navegador inteligente (BRAGA, 2000), apoio à engenharia reversa (VERONESE e NETTO, 2001), ferramentas de notificação de críticas em modelos *UML* (DANTAS, 2001), suporte a padrões de projeto (DANTAS et al., 2002) e ferramentas de modelagem e execução de processos (MURTA, 2002), entre outras. Toda essa infra-estrutura é implementada utilizando-se a linguagem Java (SUN, 2004).

Como a reutilização de artefatos é um aspecto importante no ambiente *Odyssey*, nesta seção é apresentada uma breve discussão sobre essa funcionalidade, além de como o próprio ambiente organiza os diferentes artefatos produzidos nas fases do ciclo de vida do software.

Nesse ambiente, o editor de diagramas permite a construção de modelos *UML*, além de um modelo de *features*, que é baseado no método *FODA* (KANG et al., 1990). Este modelo apresenta uma estrutura hierárquica que permite a modelagem de conceitos e funcionalidades, além de similaridades e diferenças, que facilita a reutilização das características do domínio.

No *Odyssey*, a reutilização dos artefatos de um domínio em novas aplicações é uma estratégia que está diretamente relacionada à estrutura interna que armazena esses artefatos (TEIXEIRA, 2003). Essa estrutura é caracterizada por níveis decrescentes de abstração relacionados através de rastros. Como os elementos utilizados nos modelos de domínio variam entre diferentes níveis de abstração, o ambiente mantém uma rastreabilidade própria e pré-definida entre os elementos semânticos como uma forma de contextualizá-los e relacioná-los dentro do domínio (MILLER, 2000; BRAGA, 2000). A criação de uma nova aplicação está relacionada a essa estrutura porque, quando o usuário do ambiente seleciona algumas *features* de seu interesse no domínio, ele está, indiretamente, fazendo um "recorte" do modelo de domínio e trazendo todos os artefatos que são rastreáveis a partir deles. Neste procedimento, as características e os relacionamentos anteriormente especificados no domínio são mantidos, permitindo que ajustes possam ser feitos para refletir detalhes da nova aplicação (MILLER, 2000).

Como os artefatos podem ser reutilizados em uma ou mais aplicações e como muitas aplicações compartilham características semelhantes entre si, foi definido, neste ambiente, que os componentes devem ser projetados no contexto do domínio e relacionados a outros artefatos em diferentes níveis de abstração através de rastros. Essa definição mantém a coerência com a proposta básica de DBC, pois permite que componentes sejam disponibilizados para novas aplicações, incentivando a reutilização (TEIXEIRA, 2003).

5.4 – Repositório de Armazenamento

O repositório *MDR* (MATULA, 2003) é um módulo da *IDE Netbeans* (NETBEANS COMMUNITY, 2005) que implementa um repositório *MOF*, capaz de armazenar qualquer meta-modelo *MOF* (OMG, 2002a) e instâncias desse meta-modelo. No contexto do projeto *Odyssey-SCM*, o repositório *MDR* é o componente utilizado para gerenciar dados de modelos *MOF*.

MOF é uma especificação da *OMG* (Object Management Group) que define uma linguagem abstrata para descrever modelos. Contudo, vale ressaltar que não é uma

gramática, mas uma linguagem usada para descrever uma estrutura de objetos. O *MOF* é também um *framework* extensível que conta com uma arquitetura em quatro camadas (MATULA, 2003), mostrada na Tabela 5.1. Esse *framework* é extensível porque permite que novos padrões de metadados sejam adicionados.

Tabela 5.1 - Arquitetura de metadados da OMG (Fonte: MATULA, 2003).

Nível MOF	Termos Utilizados	Exemplos
M3	Meta-Metamodelo	Modelo *MOF* (e.g: estrutura que define entidades como classes, atributos e operações).
M2	Meta-modelo	Meta-modelos: *CWM* e *UML* (e.g: entidade classe composta de atributos e operações).
M1	Modelo, Metadados[40]	Esquemas Relacionais, Instâncias do meta-modelo CWM e Modelos *UML* (e.g: modelo de classes, onde Classe Pessoa apresenta os atributos nome, idade e sexo).
M0	Instância de Modelo	Dados e Objetos (e.g: objeto João de idade 50 e sexo masculino).

A instância de uma camada é sempre modelada por uma instância da camada imediatamente superior. Por exemplo, na camada M0, os objetos (instâncias de classe) são representados por modelos *UML*, como modelo de classes, que estão na camada M1. Por sua vez, a camada M1 é definida pelo meta-modelo *UML*, camada M2, que utiliza os construtores básicos como classes, relacionamentos, entre outros. Este meta-modelo é uma instância do modelo *MOF*, que é chamado também de meta-metamodelo.

O modelo *MOF*, que está situado na quarta camada (M3) da arquitetura de metadados exemplificada na Tabela 5.1, é um modelo orientado a objetos, descrito em termos dos seus próprios conceitos, e possui um conjunto de elementos utilizados na construção dos meta-modelos, incluindo regras para o seu uso. São exemplos destes elementos, classes, associações, pacotes, tipos de dados, constantes, entre outros. A especificação *MOF* o descreve de forma narrativa e através do uso de notação *UML*, tabelas e expressões *OCL*. A *UML* é utilizada apenas por possuir uma representação gráfica para modelos, o que facilita a leitura e compreensão dos leitores. Vale ressaltar, contudo, que a *UML* não define a semântica do modelo *MOF* que está definida, por completo, na especificação *MOF*. A linguagem padrão *OCL* é utilizada para definir as restrições dos meta-modelos definidos com base no *MOF* e o padrão *XMI* para intercâmbio de metadados e meta-modelos entre ferramentas. A especificação *XMI*

[40] Metadado foi originalmente definido como a informação sobre o dado. Por exemplo, nos modelos de dados, considera-se metadado, os esquemas de banco de dados incluindo as descrições dos campos e formatos dos registros.

define um conjunto de regras que mapeiam os meta-modelos *MOF* e os metadados em documentos *XML*.

Além de ser um repositório *MOF*, *MDR* se baseia na especificação *JMI* (Java Metadata Interface) (DIRCKZE, 2002), que possibilita a geração de interfaces em linguagem Java, a partir de meta-modelos *MOF*, e a utilização dessas interfaces para acessar os elementos persistidos no repositório. Por exemplo, o meta-modelo da *UML* é descrito a partir do *MOF*. Utilizando *JMI*, é possível manipular os elementos da *UML* (i.e, classe, caso de uso, pacote, etc.), que são instâncias do meta-modelo *MOF*, como objetos Java, através das interfaces geradas (*APIs*). A especificação *JMI* permite um nível de interoperabilidade através de uma *API* específica de uma determinada plataforma. Considerando o exemplo da *UML*, ferramentas que manipulam e acessam informações sobre modelos *UML* podem, através da *API* gerada, acessar as informações sobre o modelo de forma programática. A vantagem na utilização da *API* é que ela permite que a ferramenta que a utiliza seja extensível, possibilitando que novas ferramentas utilizem a mesma *API* para acesso aos dados.

O repositório *MDR* foi escolhido por ser código aberto, gerar interfaces *JMI* e por apresentar os objetos do repositório *MOF* de uma forma gráfica, facilitando a compreensão. O *MDR* implementa a versão 1.3 do *MOF* que é base para a *UML* 1.4 (OMG, 2003b), disponibilizada pela OMG. A implementação do *MDR* voltada para o *MOF* 1.3 limita a criação do repositório e a geração de interfaces Java apenas para a versão 1.4 da *UML*. A versão 1.4 é limitada quanto à representação de alguns artefatos de software como, por exemplo, componentes em tempo de desenvolvimento.

Com o lançamento oficial da versão 2.0 da *UML* pela OMG será possível representar componentes seguindo o paradigma de DBC. A *UML* 2.0 (OMG, 2002b), baseada no *MOF* 2.0 (ainda em fase de elaboração), inclui um novo tipo de representação de componentes em sua especificação, além de novos elementos como conectores. A especificação propõe melhorias em relação a *UML* 1.x, permitindo que componentes sejam modelados de forma análoga à definida pelo paradigma de DBC. Dentre essas melhorias, a nova versão permite especificar eficientemente modelos *UML* em plataformas específicas de desenvolvimento como, por exemplo, J2EE/EJB (SUN, 2005) e .NET/COM (MICROSOFT, 2005). No entanto, como essa especificação ainda não foi aprovada, adotamos no projeto *Odyssey-SCM*, como forma de representação de componente, a notação de classe com estereótipo <<*componente*>> e a versão 1.4 da

UML. Essa decisão atende, inicialmente, nossas perspectivas de trabalho neste contexto de desenvolvimento.

5.4.1 – Meta-modelo de Versionamento e Modificações

Para persistir as versões dos elementos de modelos *MOF* e as modificações registradas pelo componente *Odyssey-CCS*, foi elaborado um modelo *MOF* (M2) para persistência das informações do projeto *Odyssey-SCM*, como mostrado de forma simplificada na Figura 5.2.

Esse modelo *MOF*, uma vez instanciado no repositório *MDR*, passa a armazenar dados que serão utilizados para análise assim como para a detecção de rastros, atuando como o repositório dos sistemas da GCS apresentados na Seção 5.5.

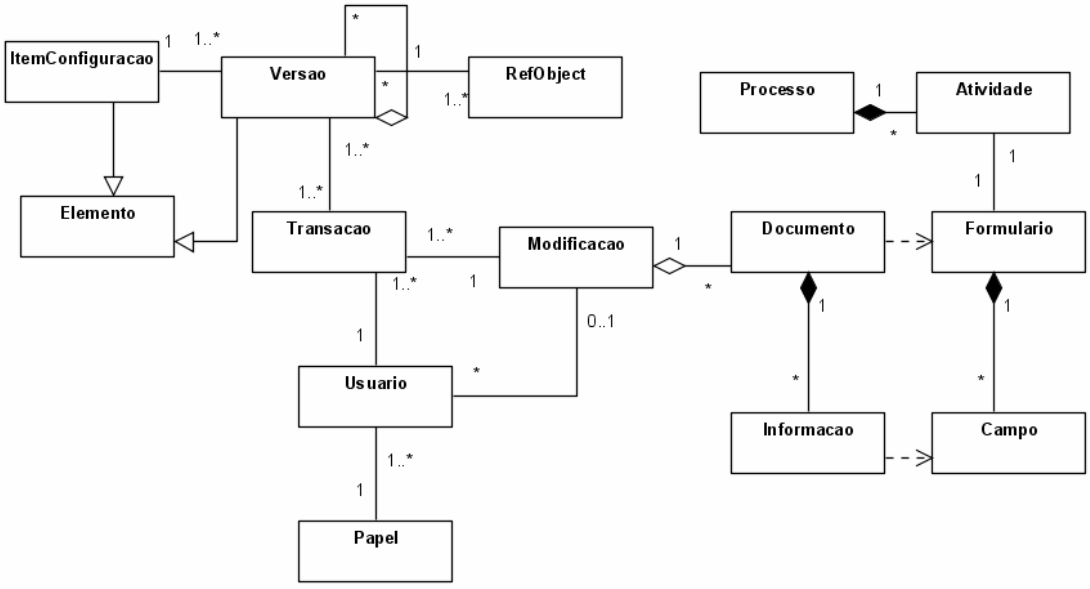

Figura 5.2 - Meta-modelo do projeto Odyssey-SCM

Na Figura 5.2, todo elemento de modelo que tem sua evolução controlada pelo sistema de controle de versões é representado no repositório como uma instância de *RefObject*. Como o projeto *Odyssey-SCM* pretende trabalhar no nível *MOF*, essa classe representa um elemento do modelo *MOF*. Isso permite que qualquer elemento do modelo *MOF* se torne um item de configuração no repositório e tenha informações de versionamento registradas.

Nesse modelo da Figura 5.2, um item de configuração pode ser um elemento de modelo *UML*, como casos de uso, classes ou componentes, que pode ter ao longo de sua história várias versões criadas a partir de operações de *check-in*. Cada operação

realizada durante a evolução deste elemento é registrada na classe *Transação*, que contém as operações de *check-in* e *check-out* para cada instância de versão criada.

Quando é realizado o *check-in*, várias versões são geradas para uma modificação que esteja sendo implementada por um usuário. Como a implementação de uma modificação não está restrita a uma única operação de *check-out* e *check-in*, o procedimento de cópia do elemento a ser alterado para o espaço de trabalho do desenvolvedor (*check-out*) e retorno ao repositório (*check-in*) pode ocorrer inúmeras vezes. Cada vez que esse procedimento ocorre sobre uma instância de versão, o usuário responsável é registrado. Posteriormente, quando for avaliada a história das modificações no processo de desenvolvimento, todas essas informações serão úteis.

Além das informações de versionamento, as informações sobre a modificação são registradas em um conjunto de documentos, preenchidos a partir de formulários, durante o processo de controle de modificações. Esses formulários são definidos para cada atividade do processo e configurados pelo sistema de controle de modificações, conforme visto na Seção 4.2.1 do Capítulo 4.

5.5 – Sistemas de Gerência de Configuração de Software

A seguir, são apresentados, em maiores detalhes, os componentes *Odyssey-VCS* e *Odyssey-CCS*, presentes na arquitetura do projeto *Odyssey-SCM*. Como dito anteriormente, esses componentes são utilizados pelos desenvolvedores durante o processo de desenvolvimento de software, armazenando no repositório dados essenciais para a análise dos rastros.

5.5.1 – Sistema de Controle de Modificações Odyssey-CCS

Como visto no Capítulo 4, a utilização do componente *Odyssey-CCS* consiste, principalmente, nas etapas referentes à preparação do ambiente e execução do processo de controle de modificações. A etapa de preparação do ambiente é executada pelo gerente de configuração responsável por implantar no *Odyssey-CCS* a norma de GCS adotada na organização, por exemplo, IEEE Std 1042 (1987) ou ISO 10007 (1995).

Inicialmente, o gerente de configuração deve modelar o processo no sistema de controle de modificações, definindo suas atividades e, no contexto do processo, os formulários de cada atividade. Cada formulário deve ser estruturado de forma a conter um conjunto de campos, que podem ser opcionais ou obrigatórios, definidos pelo próprio gerente de configuração.

A Figura 5.3 exibe o sistema de controle de modificações *Odyssey-CCS*, exemplificando como um formulário é criado para a atividade de requisição da modificação de um processo aderente à norma IEEE Std 1042 (1987). Cada formulário criado é armazenado no modelo apresentado na Seção 5.4.1, juntamente com seus respectivos campos.

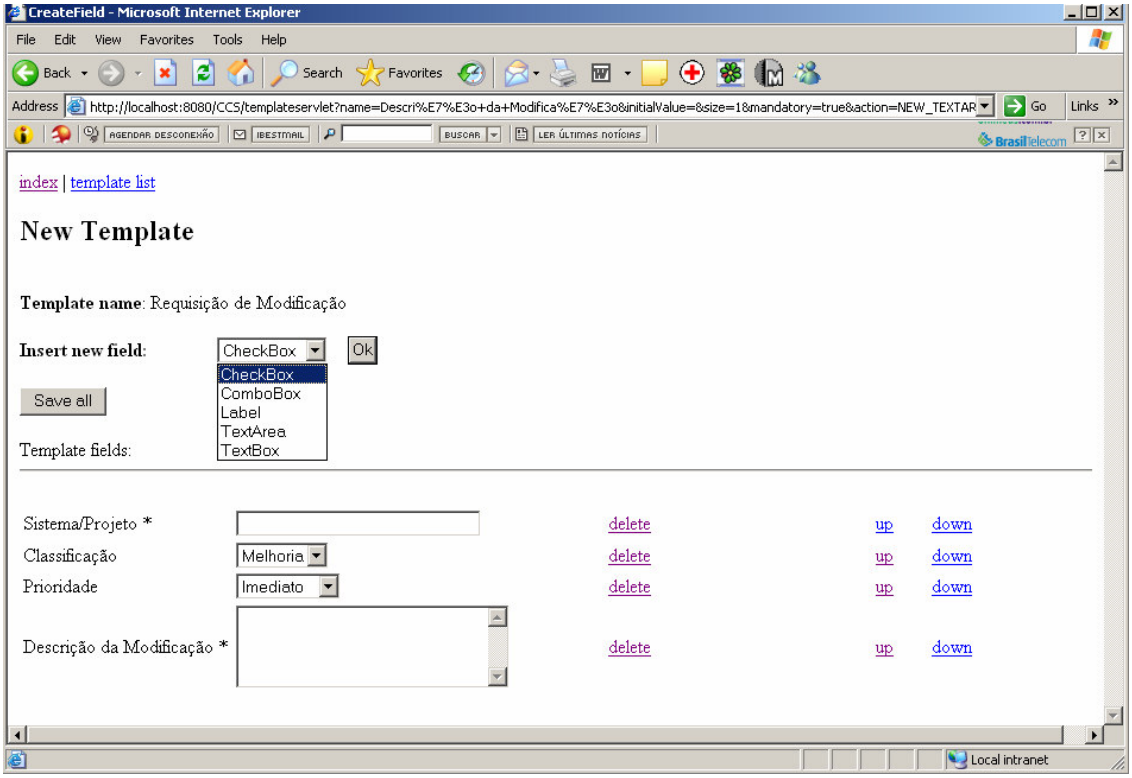

Figura 5.3 - Configuração do fomulário de requisição da modificação

Durante o processo de desenvolvimento de software, as modificações são registradas através do componente *Odyssey-CCS*, seguindo o processo de controle de modificações definido e modelado neste componente. Somente após a conclusão desta fase de configuração, o sistema de controle de modificações é executado. Para cada execução do processo completo do controle de modificações, é criado um objeto que representa no repositório a modificação propriamente dita. A cada atividade do processo, formulários são preenchidos e informações inseridas no repositório. A Figura 5.4 exemplifica o registro de uma requisição de modificação durante o processo.

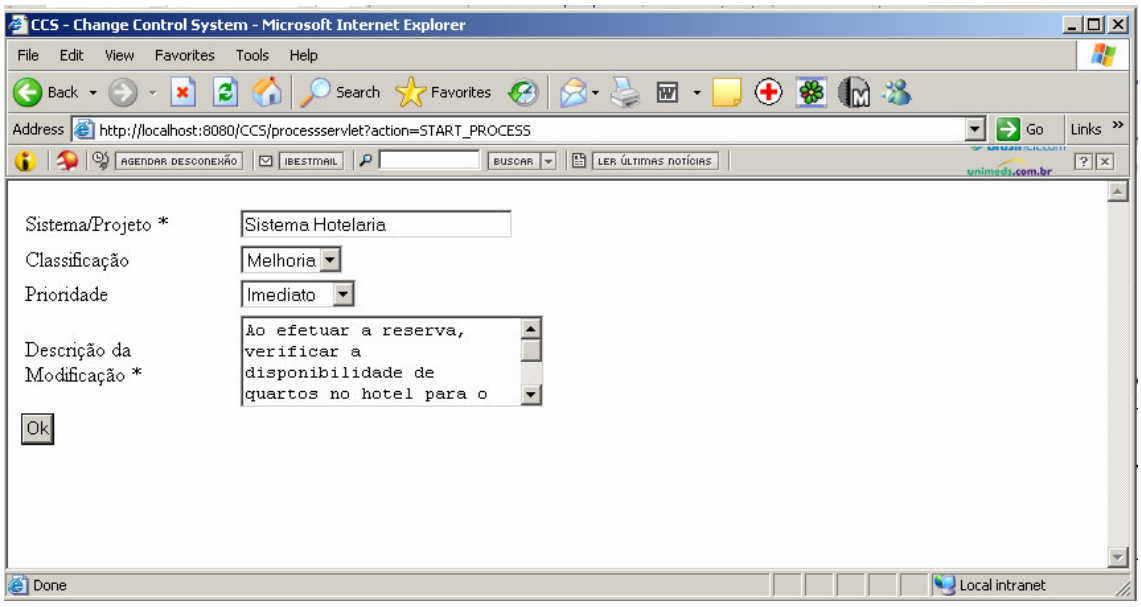

Figura 5.4 - Preenchimento das informações de uma modificação

Segundo a norma IEEE Std 1042 (1987), por exemplo, o processo de controle de modificações considera anteriormente à atividade de implementação, o registro das informações das atividades de requisição, classificação, análise e avaliação pelos responsáveis de cada atividade. Além de ser possível informar os detalhes sobre a implementação através deste componente, cabe também ao componente *Odyssey-VCS* o registro de informações sobre o procedimento da implementação da modificação.

5.5.2 – Sistema de Controle de Versões Odyssey-VCS

Durante o processo, a modificação será encaminhada para o desenvolvedor responsável por sua implementação, caso seja aprovada. Após o encaminhamento, a implementação é iniciada somente quando um desenvolvedor autorizado seleciona a modificação no ambiente de desenvolvimento. A partir deste momento, toda operação de *check-out* ou *check-in* realizada por este desenvolvedor estará associada a esta modificação. O primeiro passo para a implementação é realizar o *check-out* de uma versão do modelo *UML* no componente *Odyssey-VCS*.

O versionamento de modelos armazenados em repositórios *MOF*, proposto pelo *Odyssey-VCS*, permite o acompanhamento da evolução individual de cada elemento do modelo. Ao realizar *check-out* de uma versão do modelo *UML*, os artefatos que compõem esse modelo são copiados do repositório para o ambiente de desenvolvimento, no caso o ambiente *Odyssey*, discutido na Seção 5.3.

Depois de realizado o *check-out*, o desenvolvedor cria novos artefatos ou modifica os que convêm, e realiza a operação de *check-in* no sistema de controle de versões, que gera uma nova versão no repositório para cada artefato modificado ou criado. Esse procedimento é repetido até que a tarefa de implementação esteja concluída. Em cada *check-in*, o componente *Odyssey-VCS* armazena o responsável pela operação e a data da mesma, além de comentários sobre a atividade, que forneçam alguma referência do que foi exatamente feito no decorrer da implementação.

Após o registro das informações sobre versões, o desenvolvedor conclui a atividade de implementação no componente *Odyssey-CCS*, passando para a atividade de verificação da modificação. Se os testes nesta última atividade do processo aprovarem a implementação, a modificação pode ser considerada finalizada.

Todas as informações registradas por esse componente e pelo *Odyssey-CCS* servem de base para futuras análises, sendo vitais para a detecção de rastros entre elementos de modelo *MOF* identificados no repositório como itens de configuração.

5.6 – Camada de Transporte

A comunicação entre cliente e servidor, como exemplifica a Figura 5.1, é realizada através de *Web Services* (W3C, 2005). *Web Service* é definido como um conjunto de protocolos e padrões que permite a comunicação entre aplicações via uma rede, geralmente, a Internet. Essa comunicação, baseada em padrões e independente de plataforma, permite que as aplicações descrevam o que fazem através de serviços, e que possam assim chamar e utilizar os serviços de outras aplicações. Toda a troca de dados necessária entre os dois lados da comunicação é feita através de documentos *XML* e da especificação *SOAP* (Simple Object Access Protocol), que descreve como chamar um serviço e processar sua resposta.

SOAP define uma notação baseada em *XML* para solicitar a execução de um método de um objeto no servidor e outra para definir o formato da resposta. Como protocolo aberto, o *SOAP* define uma maneira uniforme de comunicação onde o *XML* é o formato de representação dos dados.

O serviço é descrito no *WSDL* (Web Service Description Language), que é uma linguagem baseada em *XML* usada tanto para descrever os serviços web quanto para definir como estes podem ser invocados. Uma vez descrito o serviço usando o *WSDL*, é preciso publicar de alguma forma a sua existência via rede. É através da especificação *UDDI* (Universal Description Discovery and Integration) que serviços disponíveis são

publicados e encontrados, funcionando como uma base de serviços. A Figura 5.5 exemplifica o arquivo *WSDL* na parte em que descreve os serviços do componente *Odyssey-WI*. Nessa figura, o serviço é descrito de forma a disponibilizar todos os métodos localizados no componente *WSServer* para as aplicações via rede.

```
<service name="OdysseyWI" provider="java:RPC">
  <parameter name="allowedMethods" value="*"/>
  <parameter name="scope" value="application"/>
  <parameter name="className" value="br.ufrj.cos.lens.odyssey.tools.wiserver.WSServer"/>
</service>
```

Figura 5.5 - Arquivo de configuração WSDL

Na camada de transporte, a utilização do Axis-Apache (ASF, 2004), que é uma implementação do protocolo *SOAP*, atendeu os propósitos do *Odyssey-SCM* que utiliza a web como meio de transporte. O Axis-Apache foi criado para ser usado em conjunto com o servidor de aplicações Apache Tomcat (ASF, 2004), que atua no tratamento de requisições via http e no desenvolvimento de aplicações Java para web como um container para *Servlet* (SUN, 2005) e *JSP* (JavaServer Page) (SUN,2005).

Utilizando essas duas tecnologias, o Apache Tomcat recebe uma requisição de serviço do cliente (no caso, *Odyssey*) e encaminha para o Axis-Apache. Este, por sua vez, identifica a classe responsável por tratar o serviço solicitado com base no que foi publicado no arquivo de configuração *WSDL*. Uma vez encontrado, o componente *WSServer* recebe a solicitação, retornando o resultado após o processamento.

Neste cenário, toda a comunicação entre cliente e servidor é baseada em documentos *XML*. Por esse motivo, optou-se em utilizar o *XMI* (OMG, 2003c) como especificação na representação dos dados. Como visto na Seção 5.4, *XMI* é uma especificação que define regras que permitem representar modelos baseados no *MOF* em formato de documento *XML*. O cliente *Odyssey* permite que desenvolvedores em ambientes distribuídos compartilhem objetos de modelo *UML* através de *XMI*. A importação e exportação de artefatos *UML* (Figura 5.6) no *Odyssey* utiliza uma instância do repositório *MDR* em memória que contém o meta-modelo da *UML*, e um mecanismo de tratadores (classe *HandlerController*) onde cada tratador tem a responsabilidade de fazer o mapeamento (através da classe *Mapper*) da representação de objetos *Odyssey* em *UML* e vice-versa. Através de duas interfaces *XMIWriter* e *XMIReader*, esse mecanismo disponibiliza os serviços de escrita e leitura de *XMI*, respectivamente. Esses serviços podem ser utilizados por componentes instalados no *Odyssey*, assim como pelo próprio *Odyssey*.

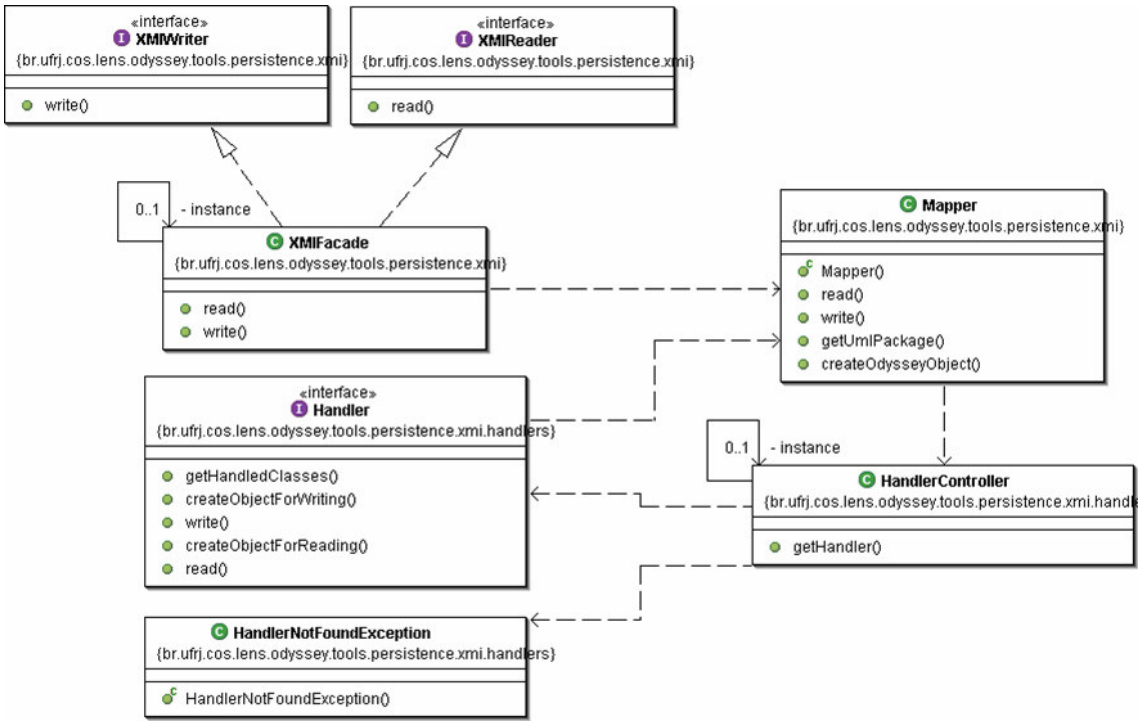

Figura 5.6 - Classes de importação e exportação de XMI no Odyssey

O cliente *Odyssey-VCS* utiliza o serviço de escrita para transmitir ao servidor os artefatos que devem ser versionados no *check-in* e usa o serviço de leitura para importar para o espaço de trabalho do desenvolvedor uma versão de um artefato do repositório durante a operação de *check-out*. Cada artefato *UML* identificado no *XMI* é um item de configuração no repositório. Ao importar e exportar os elementos do modelo para formato *XML* utiliza-se o identificador *MOF* como referência deste elemento no repositório. Como, através desse identificador, é possível encontrar uma instância do elemento no repositório, o cliente *Odyssey-WI* o utiliza para encontrar o elemento *UML* selecionado no *Odyssey*. Com base nesta informação, é possível descobrir rastros entre esse elemento *UML* e os demais elementos que são itens de configuração no repositório.

5.7 – A implementação Odyssey-WI

A partir da seleção de um artefato *UML* no *Odyssey*, o desenvolvedor pode utilizar o protótipo, caso necessite de orientação, para identificar rastros de modificação entre elementos de modelo. Como visto anteriormente, os sistemas da GCS armazenam informações referentes às modificações realizadas no software, que servem de base para análises posteriores. Esta seção descreve como foram implementados no protótipo os serviços de mineração e a apresentação dos rastros de modificação.

O serviço de mineração foi implementado com base no algoritmo *Apriori* (visto na Seção 3.4.2). Esse algoritmo (representado no modelo da Figura 5.7) considera que um elemento foi selecionado para alteração na *IDE* e que este elemento será base para a detecção dos rastros. Na literatura de mineração de dados, esse elemento é o antecedente da regra de associação.

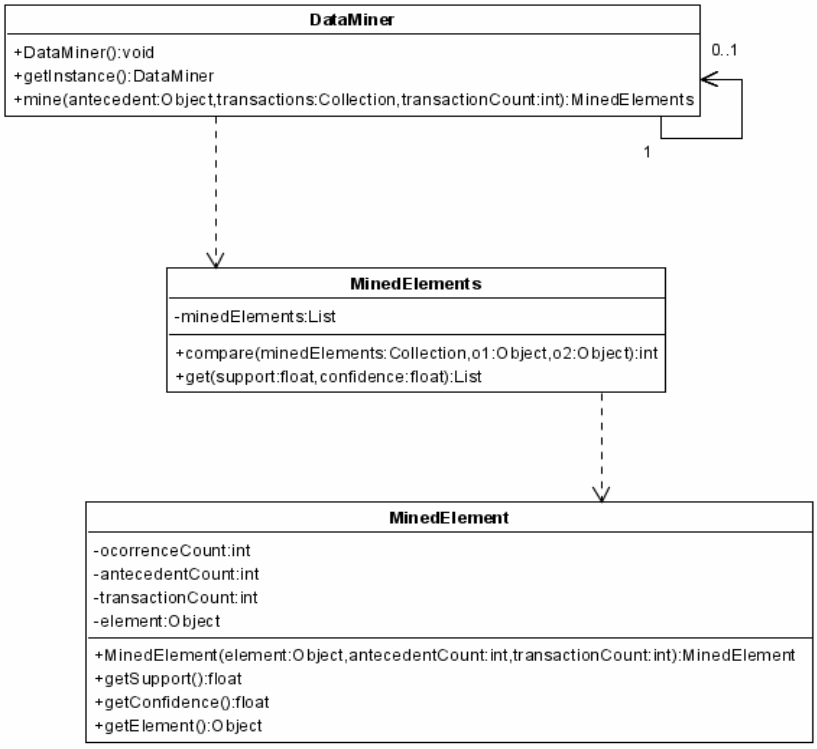

Figura 5.7 - Classes do algoritmo de mineração de dados

A classe *DataMiner* (Figura 5.7) implementa o padrão *Facade* (GAMMA et al., 1995), que torna possível o acesso ao algoritmo em um único ponto. Esta classe apresenta, para fins de mineração, o método *mine (antecedent: Object, transaction: Collection, transactionCount: int): MinedElements*. Esse método recupera todos os elementos modificados durante a alteração do antecedente (argumento *antecedent*), assumindo que devem ser analisadas apenas as modificações que contém esse elemento (argumento *transactions*) no total de modificações realizadas (argumento *transactionCount*). O retorno é um conjunto representado pela classe *MinedElements*. Cada elemento desse conjunto é uma instância da classe *MinedElement* e apresenta, em função de sua associação com o antecedente, medidas de suporte e confiança calculadas conforme descrito na Seção 3.4.1.

Durante o processamento, o algoritmo calcula as medidas para todos os elementos encontrados, indicando sua freqüência neste conjunto (confiança) e a

freqüência deste elemento no conjunto total de modificações do repositório (suporte). Somente após o processamento, são recuperados através do método *get (support: float, confidence: float): List* da classe *MinedElements*, aqueles elementos que têm o suporte e confiança maiores que os definidos nos argumentos do método. Para facilitar o uso do algoritmo de mineração por outras abordagens, a implementação foi feita de forma independente, não referenciando classes específicas do componente *Odyssey-WI.*

A chamada ao método *mine* e o acesso a classe *DataMiner* ocorre na classe *WIFacade* (Figura 5.8), que implementa o padrão *Facade*. Nesta classe ocorre todo o acesso ao repositório, incluindo a coleta dos dados para análise do algoritmo de mineração. A classe recupera do repositório, as modificações já finalizadas onde o antecedente aparece. Esse elemento é representado no repositório pelo identificador *MOF* e a partir dele é possível coletar todos os outros elementos que fizeram parte das suas modificações.

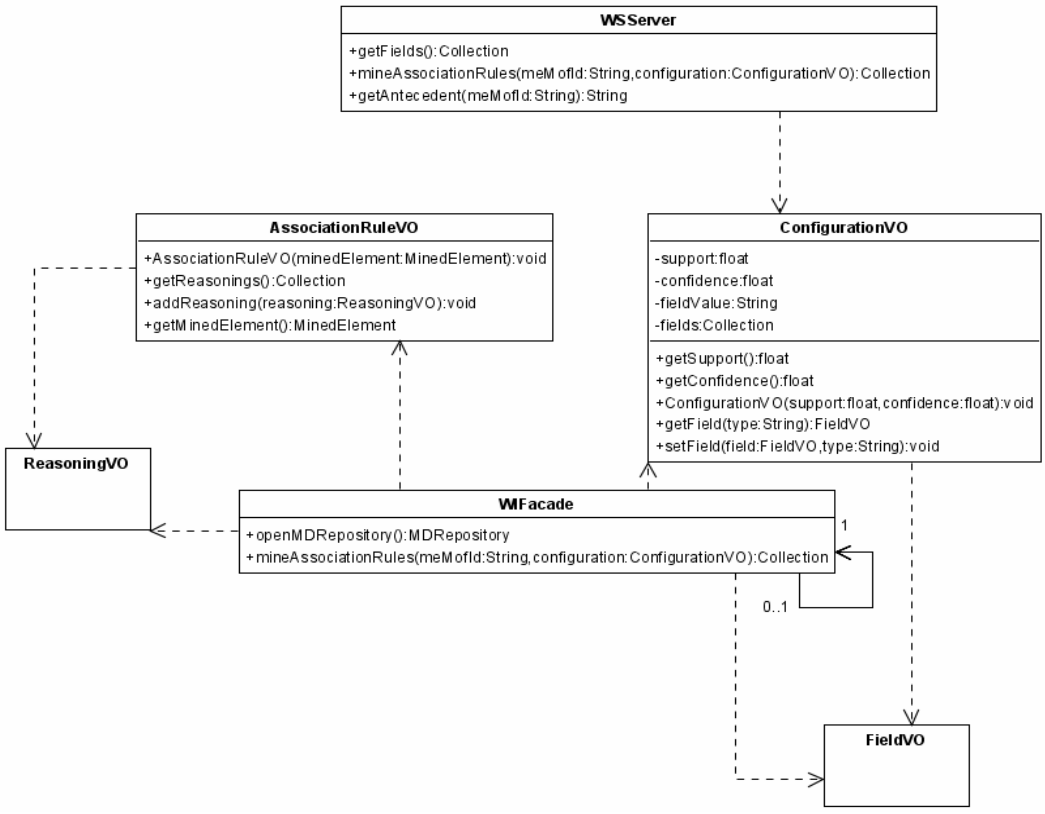

Figura 5.8 - Classes do protótipo Odyssey-WI

É de responsabilidade do método *mineAssociationRules (meMofId: String, configuration: ConfigurationVO): Collection* na classe *WIFacade* a execução do algoritmo de mineração de dados. Esse método tem como argumento a informação sobre as medidas de suporte e confiança (argumento *configuration*), fornecidas pelo

gerente de configuração. Essas medidas, como visto na Seção 4.3.2.2, definem o quão preciso deverá ser o resultado da mineração. No protótipo, toda a configuração foi designada a classe *ConfigurationVO*. Essa classe também é responsável por guardar o mapeamento entre as questões da estrutura 5W + 1H e os campos dos formulários do sistema de controle de modificações.

O método *mineAssociationRules* em *WIFacade* retorna um conjunto de regras de associação. Uma regra de associação é um objeto da classe *AssociationRuleVO*, onde cada regra tem uma semântica associada. A semântica é recuperada do repositório com base na configuração armazenada na classe *ConfigurationVO*.

Para que a configuração do protótipo seja atribuída à classe *ConfigurationVO*, a classe *WIFacade* deve coletar todos os campos armazenados no repositório pelo sistema de controle de modificações durante a modelagem e definição do processo. Somente a partir disso, o gerente de configuração terá como realizar o mapeamento entre os campos existentes nos formulários de modificação com as questões da estrutura 5W+1H.

Os métodos da classe *WIFacade*, que devem ser acessados pelo cliente (no caso, *Odyssey*) para a configuração e para a execução da mineração de rastros, estão disponíveis na classe *WSServer*. Esta classe acessa os métodos de *WIFacade* retornando para o cliente o resultado. Quando algum serviço é solicitado à classe *WSServer*, esta o delega para a classe *WIFacade*, sem que o cliente tenha conhecimento.

5.8 – Utilização do Protótipo

Conforme visto ao longo desta dissertação, a detecção de rastros de modificação entre artefatos *UML* pode tornar a manutenção do software, no contexto de DBC, uma atividade menos complexa e mais automatizada.

Nas subseções seguintes são detalhadas as atividades de configuração e utilização do protótipo no ambiente *Odyssey*, tomando como exemplo o sistema de Hotelaria, definido na Seção 4.3.2.1 do Capítulo 4.

5.8.1 – Configuração

Uma vez tendo configurado o processo de controle de modificações (apresentado na Seção 5.5.1), o gerente de configuração pode associar um campo de um dos formulários definidos para o processo a uma questão da estrutura 5W+1H do mecanismo de rastreabilidade. O objetivo dessa configuração, como já foi dito

anteriormente, é prover ao mecanismo de rastreabilidade uma forma de coletar informações específicas do sistema *Odyssey-CCS* para compor a semântica do rastro. A Figura 5.9 mostra como é realizado esse procedimento no ambiente *Odyssey*.

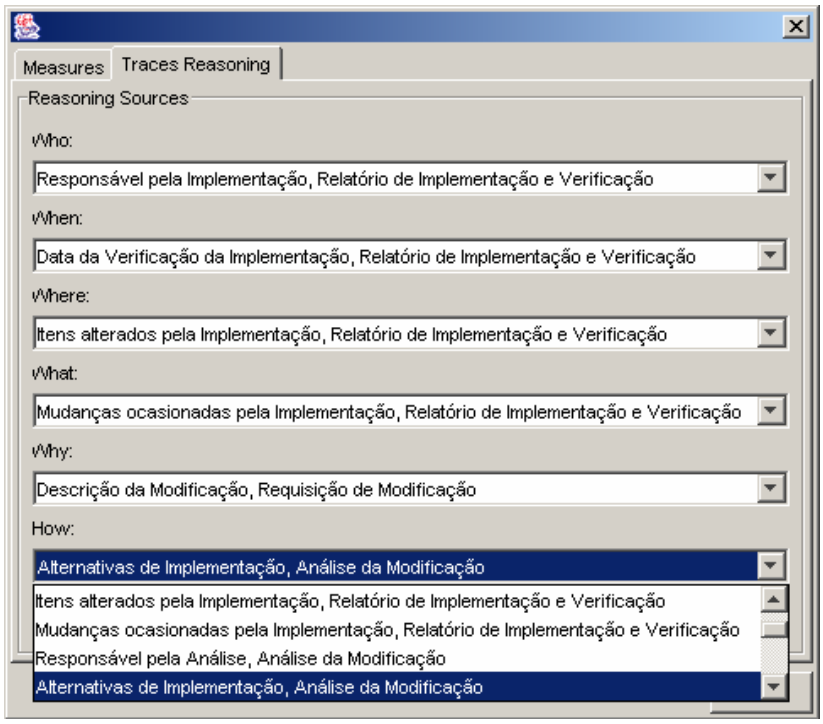

Figura 5.9 - Configuração das questões 5W + 1H

O gerente de configuração responsável pelo processo de GCS define, para cada questão, um campo dentre os formulários do sistema de controle de modificações. Na seleção do campo, aparece para escolha, o nome do campo e o nome do formulário que o contém. No exemplo, o responsável pela implementação foi atribuído à questão "Quem" (*Who*), como a pessoa mais indicada para responder dúvidas a respeito dos rastros originados pelo mecanismo. A questão "Quando" (*When*) foi preenchida com a data da verificação da implementação. Esse campo dá uma indicação de quando a modificação que deu origem ao rastro foi implementada. Na questão "Onde" (*Where*), o mesmo gerente selecionou o campo que define os itens afetados na implementação da modificação. Embora a semântica também recupere os artefatos modificados devido à operação de *check-in* no sistema de controle de versões, o campo referente ao sistema de controle de modificações, atribuído à questão, auxilia a identificar, de um modo geral, o que mudou. Já que a semântica está associada a informações de ambos os sistemas, é possível realizar comparações com o que foi preenchido pelo engenheiro de

software durante a execução do processo de gerência de modificação e o que, de fato, foi realizado na implementação. A questão “O que” (*What*) está relacionada ao campo “Mudanças ocasionadas pela Implementação”, que relata as informações do que foi feito no software em função da modificação. Na questão “Por quê” (*Why*) foi associada à descrição da modificação presente no formulário de Requisição da Modificação, que descreve o problema que se pretende resolver com a alteração. Essa informação fornece um indício do motivo que, possivelmente, levou à criação do rastro. Na última questão “Como” (*How*), um possível mapeamento seria relacionar o campo “Alternativas de Implementação” do formulário de Análise da Modificação. Neste campo, o engenheiro de software, durante a execução do processo, descreve o que pode ser feito na implementação para que a modificação seja atendida.

Após essa configuração, deve ser atribuído um tipo de precisão (baixa, média ou alta) para o resultado que se deseja obter com o mecanismo de rastreabilidade (Figura 5.10). Essa configuração é necessária para que o desenvolvedor tenha, durante a tarefa de implementação, rastros coerentes com um tipo de precisão.

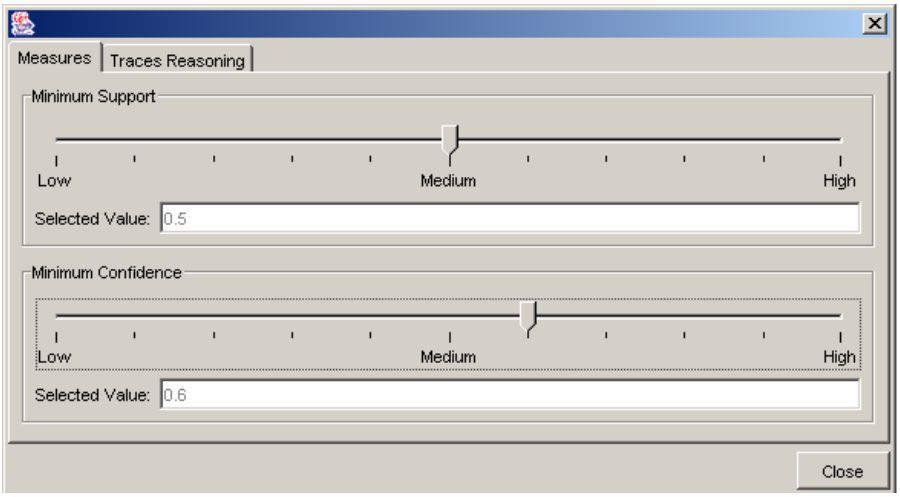

Figura 5.10 - Configuração da precisão do mecanismo de rastreabilidade

Na Figura 5.10, são definidos valores entre 0 e 1 para as medidas de suporte e confiança. Neste caso, 0.5 indica um valor médio, 0 indica o menor valor possível e 1, o maior, respectivamente. No entanto, vale ressaltar que o valor de uma medida pode ser 0.5, mas isso não indicar que a precisão dessa medida é média. Conforme visto na Seção 3.4.1, uma medida de suporte em 0.5, ou 50%, traz, ao final da mineração de dados, rastros que apareceram em 50% das modificações realizadas no software. Esse valor

pode, portanto, significar uma precisão alta para a medida de suporte, mas não ter esse mesmo significado para a medida de confiança. A medida de confiança com valor em 0.5 indica que apenas 50% das modificações de um artefato gerou impacto em outro artefato do software, o que pode ser considerado baixo em termos de precisão. Neste exemplo, foi definido para o suporte, o valor de 50% e para a confiança o valor de 60%.

Contudo, vale ressaltar que o tamanho da base de transações está diretamente relacionada com o tamanho do projeto, e conseqüentemente, com as modificações que foram implementadas no software. Esse é um fator que influencia a definição das medidas no mecanismo. Ao incorporar um número maior de artefatos ao projeto, o suporte de um artefato específico tende a diminuir. Esse fator deve ser levado em conta durante a configuração das medidas.

Outro fator que influencia o resultado obtido com o mecanismo é a estabilidade do próprio artefato. Se do total de modificações realizadas no software, um número pequeno de alterações foi realizado em um artefato, este terá um suporte menor em relação aos demais artefatos existentes no repositório.

A definição do valor mais apropriado para as medidas está sob responsabilidade do gerente de configuração. No entanto, somente com o uso do mecanismo e com a análise das informações por ele retornadas, é que o gerente de configuração será capaz de indicar uma precisão adequada.

5.8.2 – Visualização dos Rastros

Durante a tarefa de implementação, o desenvolvedor que fez o *check-out* do modelo tem a opção de verificar os rastros de um artefato em relação aos demais no *Odyssey*. Para que tenha acesso a essa informação, ele deve selecionar o artefato que está sendo alterado e selecionar a opção de menu (Figura 5.11) que apresenta os rastros ao engenheiro de software. A Figura 5.11 mostra os artefatos trazidos do repositório para o ambiente *Odyssey,* através da operação de *check-out.* Neste processo, foi recuperada, do repositório, a última versão do modelo do sistema de Hotelaria.

Selecionando a classe “Reserva” e acessando, com o botão direito do mouse, a funcionalidade de mineração dos rastros de modificação através da opção de menu *Odyssey-WI* ➔ *Show Traces*, aparece a janela da Figura 5.12, que apresenta os rastros ao engenheiro de software.

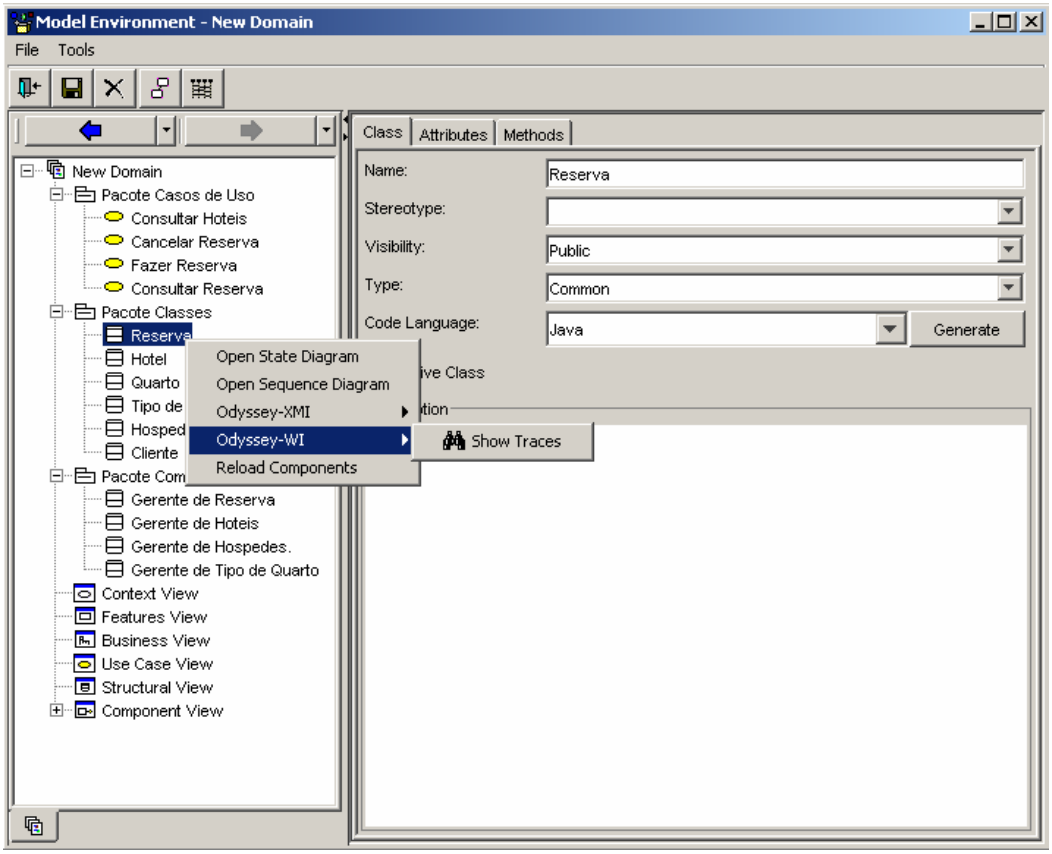

Figura 5.11 - Ação para a visualização dos rastros

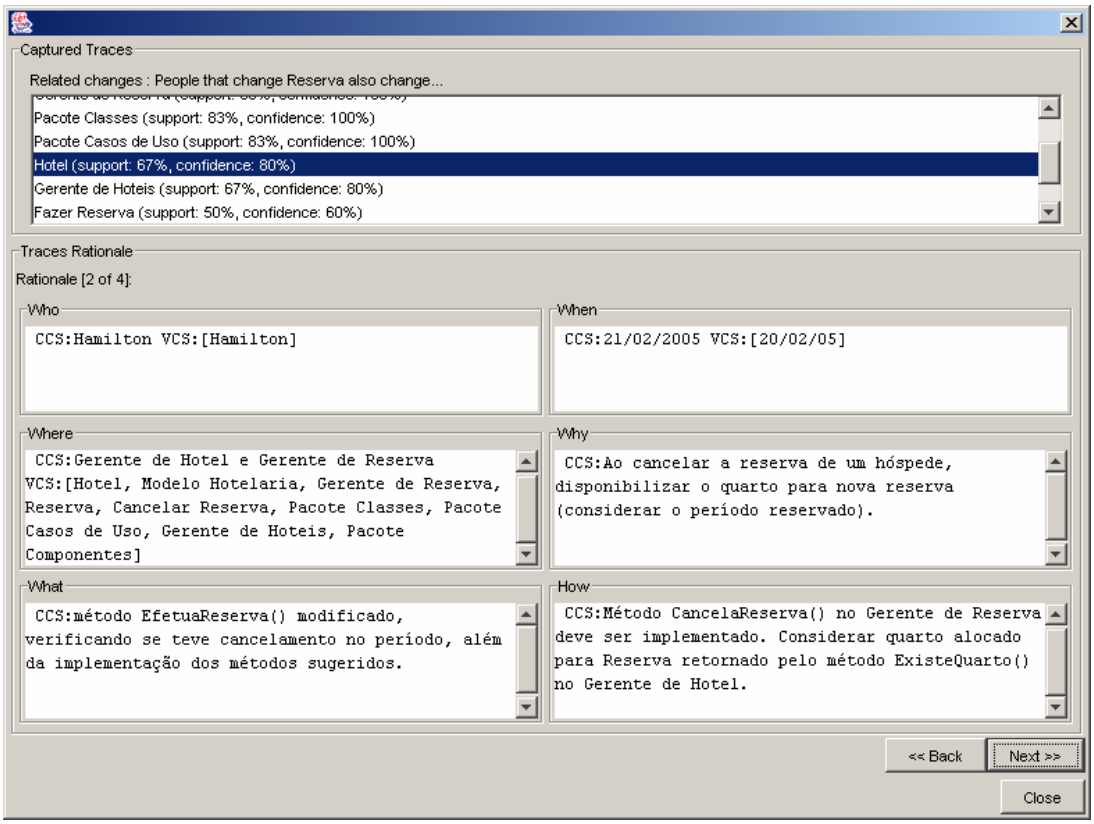

Figura 5.12 - Visualização dos rastros de modificação recuperados do repositório

Esses rastros indicam sugestões sobre quais artefatos devem ser modificados em em conjunto com a classe "Reserva". A janela apresentada na Figura 5.12 ilustra os rastros na parte superior. Ao clicar em um dos rastros, aparece na parte inferior a semântica a ele relacionada. A semântica é composta por um conjunto de informações provenientes dos sistemas de controle de versões e modificações que tem em comum a modificação que motivou o surgimento do rastro. Essa informação tem sua origem no mapeamento feito pelo gerente da configuração entre cada questão da estrutura 5W+1H e os campos dos formulários do processo (apresentado na Seção 5.5.1) e no que foi registrado no ato de *check-in* no sistema de controle de versões (apresentado na Seção 5.5.2). Como o rastro pode ter sido descoberto com base em mais de uma modificação, clicando "*Next*" avançamos para a próxima modificação relacionada ao rastro e "*Back*" retornamos para a anterior na lista. A semântica e a relevância do rastro (indicada no próprio rastro) são necessárias para que o desenvolvedor valide a sua utilidade na tarefa de implementação. Todo o processamento de mineração no repositório é realizado neste momento, mostrando como resultado todos os rastros encontrados para o artefato *UML* selecionado.

Na Figura 5.12, aparecem os mesmos rastros da Seção 4.3.2.3. Dentre os rastros apresentados, existe o rastro para o modelo do sistema de Hotelaria, que sempre é modificado quando um dos seus elementos é alterado. Além deste, temos: o componente "Gerente de Reserva" com 100% de confiança, a classe "Hotel" e o componente "Gerente de Hotel" com 80% de confiança, o caso de uso "Fazer Reserva" com 60% de confiança e os pacotes de casos de uso, classes e componentes com 100%. Esse último valor ficou em 100% porque durante as modificações, sempre, pelo menos um caso de uso, uma classe e um componente, foram alterados em conjunto com a classe "Reserva". Outros rastros foram detectados. Contudo, esses rastros foram descartados devido à configuração apresentada na Seção 5.8.1, que definiu o valor de 50% para a medida de suporte e 60% para a medida de confiança.

Quanto maior for o número de modificações incorporadas no software no decorrer do processo de desenvolvimento, maior será o volume de dados armazenados no repositório dos sistemas da GCS. Com um número maior de transações para a análise, o mecanismo será capaz de encontrar rastros de modificações com medida de suporte baixo, mas ainda assim relevantes no contexto da modificação que estiver sendo efetuada, i.e, com medida de confiança alta. Esses rastros poderão informar relações não

esperadas pelo engenheiro de software. Portanto, à medida que novas modificações são incorporadas, maior será o volume de dados para a análise e menos triviais serão as relações recuperadas pelo mecanismo.

Os rastros apresentados auxiliam a manutenção da classe "Reserva", principalmente porque mostram quais artefatos foram freqüentemente modificados em conjunto com esta classe. Com base nesta informação, o engenheiro de software pode decidir se esses artefatos serão modificados novamente ou não. No exemplo, a alteração das classes "Hotel" e "Reserva" em conjunto, deve ser analisada de acordo com semântica apresentada no rastro. A semântica indica o motivo de alterações anteriores em ambos os artefatos. Com base nesta avaliação, é possível decidir se essa alteração deve ou não ocorrer novamente.

Analisando a semântica associada ao rastro de "Reserva" em relação à "Hotel", verifica-se que a pessoa indicada para a alteração no sistema de controle de modificações foi a que, de fato, realizou a implementação. Além disso, verifica-se que todo o histórico da alteração é relatado, conforme sugerido na configuração da Seção 5.8.1, facilitando a compreensão do que pode ter originado o rastro.

Vale ainda ressaltar que, com base nas informações apresentadas pelo mecanismo, pode-se avaliar, no caso de DBC, se existe algum tipo de comportamento modelado no software que interfere no conceito de coesão de um componente. Se o engenheiro de software concluir que ao alterar uma classe de um componente, necessariamente, deve-se alterar a classe de um segundo componente, então uma possível reengenharia pode ser necessária.

5.9 – Conclusões

Apresentamos neste capítulo a implementação da proposta de um mecanismo de rastreabilidade, denominado *Odyssey-WI,* que visa a detecção e notificação de rastros que indiquem quais artefatos devem ser modificados em conjunto. Com esse objetivo e com base nas informações armazenadas pelos sistemas da GCS do projeto *Odyssey-SCM*, que têm seu foco em DBC, o protótipo identifica rastros de modificação, guiando o desenvolvedor nas tarefas de construção e manutenção do software.

Para exemplificar como o mecanismo pode ser utilizado no contexto do ambiente de desenvolvimento *Odyssey*, adotamos o modelo do domínio Hotelaria, discutido no Capítulo 4. Embora o mecanismo de rastreabilidade proposto seja abrangente quanto ao tipo de artefato (qualquer elemento de modelo *MOF*) que pretende

relacionar ao realizar a mineração dos dados do repositório, exemplificamos sua atuação em um ambiente de modelagem de artefatos *UML*.

Neste ambiente, a detecção automática de rastros traz diversas vantagens e permite que o engenheiro de software visualize dependências que não foram, necessariamente, projetadas durante a concepção do software, mas que existem em função das modificações realizadas durante as atividades do processo de desenvolvimento. Ao utilizar esta abordagem no *Odyssey*, torna-se possível a comparação entre o que foi projetado pelo analista de domínio e relacionado por ele no ambiente *Odyssey* (rastros previstos) e o que foi identificado no mecanismo de rastreabilidade após diversas modificações (rastros detectados).

Desta comparação, percebe-se que algumas relações entre artefatos *UML* não foram consideradas durante as modificações realizadas no software, ou que as relações encontradas em freqüentes modificações no software não foram previstas inicialmente. Esse tipo de comparação auxilia o desenvolvedor a verificar falhas no projeto que indiquem que deve ser feita uma reengenharia. Embora essa comparação não seja provida pelo mecanismo, ela pode ser realizada através da observação do que o mecanismo recuperou como rastro e o que foi definido pelo engenheiro de software no ambiente de desenvolvimento.

Além disso, o rastro detectado pelo mecanismo pode auxiliar na atividade de análise de impacto, como visto no Capítulo 2. Nesta atividade, deve-se preencher informações como, por exemplo, a pessoa mais indicada para a implementação da modificação, os artefatos sugeridos para a modificação da requisição além do custo e tempo estimados. Com o uso do mecanismo, é possível verificar qual a pessoa mais indicada para um determinado tipo de modificação, bastando para isso, avaliar dentre as modificações realizadas quem, em geral, realiza alterações em determinado artefato. Além disso, é possível verificar quais artefatos foram alterados em uma específica modificação. Sendo assim, a semântica, apresentada pelo mecanismo, pode auxiliar tanto a avaliação do rastro para a modificação que será realizada, quanto as atividades da GCS.

Capítulo 6 – Conclusões

6.1 – Considerações Finais

Conforme discutido nesta dissertação, grande parte das abordagens que procuram automatizar o suporte à rastreabilidade de software atuam em código-fonte. Estas abordagens, geralmente, não têm preocupações referentes à manutenção dos modelos de análise e projeto. No entanto, quando a documentação não é atualizada durante o processo de desenvolvimento do software, ficando comprometida frente às restrições de tempo, custo e nível de conhecimento de alguns membros da equipe, um número maior de falhas aparece durante a manutenção (EICK et al., 2002; HASSAN e HOLT, 2003). O processo, neste caso, se torna complexo, custoso, difícil de gerenciar e de entender. Por esta razão, essas abordagens não atendem a contento as tarefas de manutenção que devem propagar todas as modificações realizadas em código para os modelos de análise e projeto.

A importância da automatização na detecção dos rastros é também relatada na literatura (CLELAND-HUANG et al., 2003) como um problema na maioria dos mecanismos de rastreabilidade atuais. Como o processo de desenvolvimento é iterativo e as evoluções constantes, torna-se importante manter os rastros atualizados e consistentes com o estado atual do software, auxiliando futuras manutenções no projeto. Ao exigir do engenheiro de software a identificação ou a manutenção manual dos rastros entre artefatos do software tornamos a atividade de manutenção custosa e propensa à falhas (CLELAND-HUANG et al., 2003).

Outro ponto observado pelas abordagens existentes na literatura (HASSAN e HOLT, 2003) é a ausência de semântica nas relações identificadas pelos mecanismos, que dificulta a interpretação do rastro durante as atividades do desenvolvimento e traz dúvidas quanto à sua utilidade e aplicação.

Para suprir essas necessidades, foi desenvolvida uma proposta, e construído um mecanismo para detectar, automaticamente, rastros de modificação entre artefatos *UML*. A partir da análise das informações existentes nos sistemas de GCS através da aplicação de mineração de dados, os rastros são identificados e apresentados ao engenheiro de software de forma não intrusiva, interferindo o mínimo necessário na sua rotina de trabalho.

Esse mecanismo fornece as seguintes características:

- Fundamentação configurável dos rastros de modificação identificados. O rastro é apresentado ao engenheiro de software com informações extraídas do repositório dos sistemas de controle de modificações e versões, seguindo a estrutura 5W+1H. Essa informação serve de base para o engenheiro de software analisar a importância do rastro capturado e sua utilidade para as modificações que estiverem sendo realizadas. No entanto, cabe ao gerente de configuração da organização a escolha do tipo de informação que se deseja visualizar no rastro para que se tenha um melhor entendimento do seu significado.
- Utilização de técnicas de mineração por regras de associação, apoiando a detecção de rastros de modificação, encontrando regras do tipo: "Desenvolvedores que modificam esses artefatos também modificam esses outros artefatos", indo além das informações explícitas existentes nos repositórios.
- Detecção automática e não intrusiva de pós-rastros intramodelos e intermodelos, aproveitando informações já registradas nos sistemas de controle de versões e modificações. Os rastros de modificação detectados entre artefatos *UML* auxiliam a manutenção dos modelos de análise e projeto no decorrer do processo de desenvolvimento do software, mantendo a documentação atualizada e consistente. Estes rastros, no nível de modelo *UML,* antecipam falhas de projeto, o que minimiza as chances destas falhas serem tardiamente encontradas no processo.
- Abordagem extensível para mineração de elementos de software, a partir de novas definições de modelos *MOF* como, por exemplo, para código-fonte. Devido à própria atuação do sistema de versões utilizado no contexto desse trabalho, o mecanismo se torna apto a identificar rastros entre artefatos produzidos em todas as fases do ciclo de vida do software.
- Independência em relação ao ambiente de desenvolvimento. Essa característica torna o mecanismo reutilizável em qualquer ambiente de desenvolvimento de software e ferramentas *CASE*.
- Abordagem ampla que não está restrita ao contexto de DBC, podendo ser utilizada no desenvolvimento de software em geral.

No contexto de DBC, podemos dizer que o principal objetivo deste trabalho consiste em prover a integração entre os espaços de trabalho de GCS e DBC. Esse objetivo foi alcançado, através do uso de técnicas de mineração de dados no repositório dos sistemas de controle de versões e modificações. A aplicação das técnicas de mineração de dados foi necessária à detecção dos rastros de modificação. Acredita-se que, no ambiente DBC, o emprego de novas funcionalidades, como a detecção automática de rastros entre artefatos, pode auxiliar o engenheiro de software a perceber como elementos de modelo de análise e projeto estão relacionados.

Considerando as contribuições listadas anteriormente, foram observados alguns benefícios que poderiam ser obtidos com a utilização dessa abordagem. Esses benefícios podem ser comprovados através da realização de estudos de caso. São eles:

- Apoio às atividades do DBC, permitindo a detecção de falhas de projeto, devido ao alto acoplamento entre artefatos não correlatos, que serve de base para uma reengenharia dos componentes. A identificação de rastros de modificação, pelo mecanismo, que retratem relações entre artefatos de software que não deveriam existir, permite que o engenheiro de software realize uma reengenharia do projeto. Essas relações podem existir em função das modificações realizadas freqüentemete no decorrer das atividades do processo de desenvolvimento.
- Apóio às atividades da GCS, favorecendo a atividade de análise de impacto das modificações. O rastro de modificação indica quais artefatos podem sofrer o impacto de uma modificação, caso um determinado artefato seja alterado. Desta forma, o rastro auxilia o engenheiro de software a estimar quais artefatos deverão ser alterados em função de uma modificação, minimizando complicações como a introdução de novos erros e alterações incompletas durante a modificação, o que torna o trabalho de manutenção menos complexo e propenso à falhas.

6.2 – Limitações e Trabalhos Futuros

Essa dissertação apresenta, ainda, algumas limitações, descritas a seguir, bem como alguns trabalhos que podem ser realizados futuramente, dando continuidade à pesquisa realizada.

6.2.1 – Reestruturação (*refactoring*) dos modelos de análise e projeto

Como a abordagem identifica rastros de modificação a partir da análise dos dados armazenados pelos sistemas da GCS, percebemos que a exclusão de um artefato do modelo não exclui a história do artefato armazenada. Essa é uma questão que pode afetar os rastros identificados, principalmente, no caso onde a presença do artefato excluído é constante nas transações capturadas pelo mecanismo. Se o elemento que foi excluído aparecer no rastro, a sugestão de modificação estará incorreta. Da mesma forma, ao mover ou renomear um artefato, a semântica é trocada. Havendo a troca, não é possível garantir a coerência do histórico do artefato armazenado no repositório da GCS. Devido a isso, podem aparecer, como rastros de modificação, algumas relações incorretas entre artefatos.

Essa é uma limitação do mecanismo. No entanto, à medida que o repositório incorporar novos dados, a freqüência desse artefato nas transações diminuirá, contribuindo para a obtenção de um maior número de rastros corretos.

6.2.2 – Dependência em relação aos dados dos sistemas da GCS

Devemos ressaltar que a proposta é dependente da qualidade do que é registrado pelos desenvolvedores no sistema de controle de modificações. Nos projetos *open-source* os comentários anexados ao registro de *check-in* ou as informações registradas no sistema de controle de modificações, geralmente, servem como meio para comunicar as características novas do projeto e o progresso de alguns desenvolvedores. Nesses projetos, o estímulo para a entrada de comentários úteis e corretos existe em função do nível de compromisso dos desenvolvedores. Esse comprometimento em termos de documentação favorece futuras análises e apóia o próprio engenheiro de software durante as atividades do desenvolvimento. Pode-se assumir que um comprometimento similar deve existir para que a abordagem apresentada seja útil.

6.2.3 – Melhorias no suporte a componentes

Atualmente utilizamos o *MOF* 1.3 (OMG, 2002a) e *UML* 1.4 (OMG, 2003b), por uma limitação do *Netbeans MDR* que está sendo utilizado para a persistência dos dados. A partir do lançamento da *UML* 2.0 e da definição do *MOF* 2.0, a parte de componentes pode ser melhor incorporada a esta pesquisa. Ao utilizar a *UML* 2.0 estaremos aptos a versionar no sistema de controle de versões, por exemplo, as interfaces de um componente, assim como o próprio componente, e a partir desse

versionamento estaremos ampliando o conjunto de artefatos que podem ser capturados pelo mecanismo de rastreabilidade.

6.2.4 – Extensão para a Pré-Rastreabilidade

O mecanismo poderia ser estendido para a pré-rastreabilidade. No entanto, para que isso seja possível, é necessário estudar uma forma de armazenar artefatos, derivados de atividades anteriores à especificação de requisitos no repositório dos sistemas da GCS, i.e, definir um modelo *MOF* que represente esses artefatos. Neste estudo, pode-se levar em conta o meta-modelo já definido por LETELIER (2002). Esse meta-modelo considera tipos de relacionamentos e informações já discutidas na literatura e presentes no modelo de referência de RAMESH e JARKE (2000), sendo capaz de armazenar o raciocínio empregado na elaboração dos requisitos e os artefatos produzidos anteriormente à especificação, relacionados a seus respectivos responsáveis.

6.2.5 – Extensão para código-fonte

O mecanismo poderia ser estendido para incorporar no rastro de modificação elementos de código-fonte. Para que isso seja possível, é necessário que o sistema de controle de versões seja configurado para utilizar o modelo *MOF* de código-fonte e que sejam definidos grãos de versionamento e comparação para este novo modelo *MOF*.

À medida que o sistema de controle de versões for sendo utilizado no processo de desenvolvimento, as versões dos elementos do código serão armazenadas no repositório da GCS, de acordo com o grão de versionamento definido. Como esta informação poderá ser recuperada do repositório, o mecanismo poderá relacionar artefatos de código-fonte, apresentando estes relacionamentos como rastros de modificação.

6.2.6 – Melhorias na identificação dos rastros

Seria interessante que a base de rastros não fosse gerada a cada pedido e que fosse possível realizar algum tipo de consulta em função das medidas fornecidas para suporte e confiança e do elemento selecionado para a alteração. Para que isso fosse possível, com a base já criada, o algoritmo de mineração deveria associar incrementalmente as modificações que fossem realizadas via operação de *check-in*. Desta forma, uma implementação incremental do algoritmo de mineração de dados

(CHEUNG et al., 1996) atualizaria a lista de transações à medida que ocorresse o *check-in*, modificando a base de rastros após a execução do algoritmo.

Além disso, outro ponto importante seria disponibilizar para o engenheiro de software a configuração da precisão dos rastros. Contudo, seria adequado que o conhecimento referente à mineração não fosse pré-requisito para essa configuração. Para atender essa questão, é necessário um estudo de como a combinação de determinados valores de suporte e confiança pode trazer resultados satisfatórios aos três níveis de precisão: baixo, médio e alto. Essa combinação poderia ser feita pelo gerente de configuração.

Desta forma, ao invés do gerente de configuração indicar uma precisão para os rastros, ele indicaria possíveis combinações de valores para as medidas de suporte e confiança. A partir disso, o engenheiro de software, ao invés de obter rastros para uma única precisão pré-definida, ganharia a opção de selecionar a precisão, em tempo de execução, de acordo com o resultado que quer obter, mesmo sem ter o conhecimento do que as medidas por traz da configuração significam.

6.2.7 – Melhorias na apresentação da semântica do rastro

Atualmente, o mecanismo apresenta as informações extraídas do sistema de controle de modificações e versões, estruturando da forma 5W + 1H. No entanto, seria interessante que essas informações, ao invés de seguir essa estrutura, fossem sumarizadas de forma a proporcionar uma análise estatística dos dados. Por exemplo, se 10% das modificações realizadas para o surgimento do rastro foram implementadas por "Hamilton" e 90% por "Leonardo", então, conclui-se que "Leonardo" é a pessoa mais adequada a responder dúvidas a respeito do que o rastro significa.

Essa característica traria um sumário das informações coletadas nos sistemas da GCS que dizem respeito à semântica do rastro, trazendo a informação de uma forma mais resumida para análise do engenheiro de software. A mineração de dados pode ser utilizada para encontrar novas relações que não sejam, necessariamente, entre artefatos de software. Seria interessante que ela pudesse ser aplicada no repositório dos sistemas da GCS relacionando outras informações referentes às modificações realizadas no software.

6.2.8 – Suporte à avaliação dos rastros

Para aumentar a qualidade do que é identificado pelo mecanismo e encontrar um número maior de rastros válidos, poderíamos automatizar a comparação entre os rastros identificados manualmente pelo engenheiro de software (rastros previstos) e os rastros obtidos pelo mecanismo (rastros detectados).

Na literatura, alguns trabalhos (YING, 2003; ZIMMERMANN et al., 2003a; DE LUCIA et al., 2004) avaliam o resultado obtido no mecanismo de rastreabilidade com uma possível solução. Essa solução, se explicitamente mencionada pelo engenheiro de software, pode ser usada como forma de avaliação. Por exemplo, se um rastro recuperado pelo mecanismo também tiver sido indicado na solução, então esse rastro pode ser classificado como correto. Caso ele não tenha sido relatado pelo engenheiro de software, poderá ser avaliado como um rastro ausente na documentação ou um rastro incorreto no mecanismo. Já quando um rastro é documentado, mas não é encontrado pelo mecanismo, significa que a relação entre os elementos não tem sido, de fato, respeitada durante as modificações, o que pode ser um sinal de que uma reengenharia é necessária. Esse tipo de suporte poderia ser acrescentado à proposta, o que facilitaria a avaliação do rastro, tanto para a modificação que estivesse sendo efetuada, quanto para o software como um todo.

6.2.9 – Verificação das conclusões obtidas

Ao final deste trabalho, concluímos que o uso do mecanismo pode facilitar as tarefas relacionadas à manutenção do software. Contudo, essa conclusão pode ser vista como uma premissa que deve ser verificada através de estudos de caso. É importante frisar, a necessidade da aplicação da proposta em projetos reais, onde a complexidade presente nos modelos possa, de fato, representar um problema para a manutenção. Somente a partir disso, poderia ser verificada a utilidade e a qualidade dos rastros de modificação detectados pelo mecanismo.

Além disso, poderia ser avaliado se o uso de regras de associação como técnica de mineração de dados foi realmente adequado para a abordagem e se os resultados extraídos do repositório, através desta técnica, foram satisfatórios para a manutenção do software. Isso significa realizar uma análise sobre os rastros de modificação detectados pelo mecanismo como forma de avaliar se eles poderiam ser descobertos sem a utilização de mineração de dados, ou ainda, se representam, de fato, uma informação

relevante (i.e, não trivial) para a manutenção do software. Para que isso possa ser realizado, será necessário um repositório com um grande volume de dados armazenados no decorrer do processo de desenvolvimento.

Referências Bibliográficas

AGRAWAL, H., HORGAN, J. R., 1990, "Dynamic Program Slicing". In: *Proceedings of the ACM SIGPLAN'90 Conference on Programming Language Design and Implementation*, v. 25, n. 6, pp. 246-256, New York, June.

AGRAWAL, R., SRIKANT, R., 1994, "Fast Algorithms for mining association rules". In: *Proceedings of the 20th Very Large Data Bases Conference (VLDB)*, pp. 487-499, Santiago, Chile, September.

ALFORD M., 1991, "Strengthening the systems engineering process". In: *Proceedings of International Council on System Engineering INCOSE*, San Jose, CA, March.

AMBLER, S., 1999, "Tracing your Design", *Software Development Magazine*, April.

ANT, 2005, "The Apache Ant Project". In: http://ant.apache.org/, Accessed in 21/02/2005.

ANTONIOL, G., CANFORA, G., DE LUCIA, A., 1999, "Maintaining Traceability during Object-Oriented Software Evolution: A Case-Study". In: *Proceedings of the IEEE International Conference on Software Maintenance*, pp. 211-219, Oxford, England, September.

ASF, 2004, "Apache Software Foundation". In: http://www.apache.org/, Accessed in 1712/2004.

BALL, T., KIM, J., PORTER, A. A., et al., 1997, "If your Version Control System Could Talk…". In: *ICSE '97 Workshop on Process Modeling and Empirical Studies of Software Engineering*, Boston, MA, May.

BERRY M.J.A, LINOFF G., 1997, *Data Mining Techniques*, John Wiley & Sons, Inc.

BOHNER S.A., 2002, "Software Change Impacts: An Evolving Perspective". In: *Proceedings International Conference on Software Maintenance*, pp. 263-272, Montreal, Quebec, Canada.

BRAGA, R. M. M., 2000, *Busca e Recuperação de Componentes em um Ambiente de Reuso*, Tese de D.Sc., COPPE/UFRJ, Rio de Janeiro, RJ, Brasil.

BRATTHALL, L., JOHANSSON, E., REGNELL, B., 2000, "Is a design rationale vital when predicting change impact? A controlled experiment on software architecture evolution". In: *Proceedings of the International Conference on Product Focused Software Process Improvement*, pp. 126-139, Oulu, Finland, January.

BRIAND, L. C., LABICHE, Y., O'SULLIVAN, L., 2003, "Impact Analysis and Change Management of UML Models". In: *Proceedings International Conference on Software Maintenance (ICSM'03)*, pp. 256-265, Amsterdam, the Netherlands, September.

CHEESMAN, J., DANIELS, J., 2000, *UML Components: A Simple Process for Specifying Component-Based Software*. 1st ed., Addison-Wesley.

CHEN, A., CHOU, E., WONG, J., et al., 2001, "CVSSearch: searching through source code using CVS comments". In: *Proceedings International Conference on Software Maintenance (ICSM 2001)*, pp. 364-374, Florence, Italy, November.

CHEN, M.-S., HAN, J., YU, P. S., 1996, "Data Mining: An Overview from Database Perspective", *IEEE Transactions on Knowledge and Data Engineering*, v. 8, n. 6, pp. 866-883.

CHEUNG D., LEE S.D., KAO B., 1997, "A general incremental technique for updating discovered association rules". In: *International Conference on Database Systems for Advanced Applications*, April.

CLELAND-HUANG, J., CHANG, C. K., CHRISTENSEN, M., 2003, "Event-Based Traceability for Managing Evolutionary Change", *IEEE Transactions on Software Engineering*, v. 29, n. 9, pp. 796-810.

CLEMENTS, P., NORTHROP, L. M., 2001, *Software Product Lines: Practices and Patterns*. 1st ed., Boston, MA, Addison-Wesley.

COHEN, W. W., 1995, "Inductive specification recovery: Understanding software by learning from example behaviors", *Automated Software Engineering*, v. 2, n. 2, pp. 107-129.

CONKLIN, J., BEGEMAN, M. L., 1988, "gIBIS: A hypertext tool for exploratory policy discussion", *ACM Transactions On Office Information Systems*, v. 6, n. 4, pp. 303-331, October.

CONROW, E. H., SHISHIDO, P. S., 1997, "Implementation Risk Management on Software Intensive Projects", *IEEE Software*, v. 14, n. 3, pp. 83-89.

CUBRANIC, D., MURPHY, G. C., 2003, "Hipikat: Recommending pertinent software development artifacts". In: *Proceedings 25th International Conference on Software Engineering (ICSE)*, pp. 408-418, Portland, Oregon, May.

D'SOUZA, D., WILLIS, A., 1998, *Objects, components, and frameworks with UML: The catalysis approach*, Addison Wesley.

DANTAS, A. R., 2001, *Oráculo: Um Sistema de Críticas para a UML*, Projeto Final de Curso, DCC/IM/UFRJ, Rio de Janeiro, RJ, Brasil.

DANTAS, A. R., VERONESE, G. O., CORREA, A., et al., 2002, "Suporte a Padrões no Projeto de Software". In: *Caderno de Ferramentas do XVI Simpósio Brasileiro de Engenharia de Software (SBES'2002)*, pp. 450-455, Gramado, RS, Brasil, Outubro.

DANTAS, C. R., OLIVEIRA, H. L. R., MURTA, L. G. P., et al., 2003, "Um Estudo sobre Gerência de Configuração de Software aplicada ao Desenvolvimento Baseado em

Componentes". In: *Terceiro Workshop de Desenvolvimento Baseado em Componentes*, São Carlos, SP, Setembro.

DANTAS, C. R., MURTA, L. G. P., WERNER, C. M. L., 2004, "Mineração de Rastros de Modificação de Modelos em Repositórios Versionados". In: *Quarto Workshop de Desenvolvimento Baseado em Componentes*, João Pessoa, PB, Brasil, Setembro.

DART, S., 1991, "Concepts in Configuration Management Systems". In: *Proceedings of the 3rd international workshop on Software configuration management*, pp. 1-18, Trondheim, Norway.

DE LUCIA, A., FASANO, F., OLIVETO, R., et al., 2004, "Enhancing an Artefact Management System with Traceability Recovery Features". In: *20th IEEE International Conference on Software Maintenance (ICSM'04)*, pp. 306-315, Chicago, Illinois, September.

DIAS M.M, 2001, *Estudo e Análise de Técnicas e Ferramentas de Mineração de Dados*. Relatório de Projeto de Ensino, Universidade Estadual de Maringá - UEN.

DIRCKZE, R., 2002, *Java Metadata Interface (JMI) Specification - version 1.0.*, Unisys Corporation and Sun Microsystems.

DOD, 1988, *DoD-2167a Military Standard - Defense System Software Development*. US Departament of Defense.

DOURISH, P., BELLOTTI, V., 1992, "Awareness and Coordination in Shared Workspaces". In: *Conference on Computer Supported Cooperative Work*, pp. 107-114, Toronto, Canada, October.

DRAHEIM, D., PEKACKI, L., 2003, "Analytical Processing of Version Control Data: Towards a Process-Centric Viewpoint". In: *Proceedings International Workshop on Principles of Software Evolution (IWPSE 2003)*, Helsinki, Filand, September.

ECLIPSE FOUNDATION, 2004, "Eclipse.org Main Page". In: http://www.eclipse.org, Accessed in 03/02/2004.

EGYED, A., 2001, "A Scenario-Driven Approach to Traceability Problem". In: *Proceedings of the 23rd International Conference on Software Engineering (ICSE 2001)*, pp. 123-132, Toronto, Canada, May.

EICK, S. G., GRAVES, T. L., KARR, A. F., et al., 2001, "Does code decay? Assessing the evidence from change management data", *IEEE Transactions on Software Engineering*, v. 27, n. 1, pp. 1-12.

ESTUBLIER, J., LEBLANG, D., CLEMM, G., et al., 2002, "Impact of the Research Community on the field of Software Configuration Management", v. 27, pp. 31-39, Orlando, Florida, September.

FISCHER, M., PINZGER, M., GALL, H., 2003, "Populating a release history database from version control and bug tracking systems". In: *International Conference on*

Software Maintenance (ICSM 2003), pp. 23-32, Amsterdam, The Netherlands, September.

FOGEL, K., BAR, M., 2001, *Open Source Development with CVS*. 2 ed., Scottsdale, Arizona, The Coriolis Group.

GALL, H., HAJEK, K., JAZAYERI, M., 1998, "Detection of logical coupling based on product release history". In: *Proceedings International Conference on Software Maintenance (ICSM 1998),* pp. 190-198, Washington D.C, USA, November.

GAMMA, E., HELM, R., JOHNSON, R., et al., 1995, *Design Patterns: Elements of Reusable Object-Oriented Software*. 1st ed., Addison-Wesley.

GOEBEL, M., GRUENWALD, L., 1999, "A Survey of Data Mining and Knowledge Discovery Software Tools", *SIGKDD Explorations*, v. 1, n. 1, pp. 20-33.

GOLDBERG, E., 2002, *Bug Writing Guidelines*. The Mozilla Organization.

GOTEL, O., FINKELSTEIN, A., 1994, "An Analysis of the Requirements Traceability Problem". In: *Proceedings of IEEE International Conference on Requirements Engineering, IEEE Computer Society Press*, pp. 94-101, Colorado, Colorado Springs.

GOTEL, O., FINKELSTEIN, A., 1995, "Contribution structures". In: *Proceedings of the 2nd International Symosium on Requirements Engineering, IEEE Computer Society Press*, pp. 100-107, York, UK, March.

GRAVES, T. L., MOCKUS, A., 1998, "Inferring Change Effort from Configuration Management data". In: *Metrics 98: Fifth International Symposium on Software Metrics*, pp. 267-273, Bethesda, Maryland, November.

GRAVES, T. L., KARR, A. F., MARRON, J. S., et al., 2000, "Predicting fault incidence using software change history", *IEEE Transactions on Software Engineering*, v. 26, n. 7, pp. 653-661.

GUTWIN, C., GREENBERG, S., 2001, "A Descriptive Framework of Workspace Awareness for Real-Time Groupware", *Journal of Computer Supported Cooperative Work*, v. 11, n. 3, pp. 411-446.

HAN, J., 1996, *Supporting Impact Analysis and Change Propagation in Software Engineering Environments*. Technical Report 96-09. Peninsula School of Computing & Information Technology, Monash University, Australia.

HARRISON T.H., 1998, *Intranet data warehouse*, Berkeley.

HASSAN, A. E., HOLT, R. C., 2003, "ADG: Annotated Dependency Graphs for Software Understanding". In: *Workshop VISSOFT 2003: 2nd Annual "DESIGNFEST" On Visualizing Software For Understanding And Analysis (co-located with ICSM'2003*, Amsterdam, Netherlands, September.

HAUMER, P., POHL, K., WEIDENHAUPT, K., et al., 1999, "Improving Reviews by Extending Traceability". In: *Proceedings of the 32nd Annual Hawaii International Conference on System Sciences (HICSS)*, v. 3, pp. 3052-3061, Maui, Hawaii, January.

HAYES, H., DEKHTYAR, A., OSBORNE, J., 2003, "Improving Requirements Tracing via Information Retrieval". In: *11th IEEE International Requirements Engineering Conference, IEEE Computer Society Press*, pp. 138-147, Monterey, CA, USA, September.

IEEE, 1990, *Std 610.12 - IEEE Standard Glossary of Software Engineering Terminology*.

IEEE, 1987, *Std 1042 - IEEE Guide to Software Configuration Management*.

ISO, 1995, *ISO 10007, Quality Management - Guidelines for Configuration Management*.

KANG, K., COHEN, S., HESS, J., et al., 1990, *Feature-Oriented Domain Analysis (FODA) Feasibility Study SEI Technical Report CMU/SEI - 90-TR-21*. Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA.

KELLER, R., SCHAUER, R., ROBITAILLE, S., et al., 1999, "Pattern-based reverse-engineering of design components". In: *Proceedings of the 21stInternational Conference on Software Engineering (ICSE 1999)*, pp. 226-235, Los Angeles, USA, May.

KNETHEN, A., GRUND, M., 2003, "QuaTrace: A Tool Environment for (Semi-) Automated Impact Analysis Based on Traces". In: *International Conference on Software Maintenance (ICSM) 2003*, pp. 246-255, Amsterdam, The Netherlands, September.

KNETHEN, A. V., 2001, "A Trace Model for System Requirements Changes on Embedded Systems". In: *IWPSE'2001 International Workshop on the Principles of Software Evolution*, Vienna, Austria, September.

KOTONYA, G., SOMMERVILE, I., 1996, "Requirements Engineering with Viewpoints", *Software Engineering Journal*, v. 11, n.1, pp. 5-18, January.

KOWALCZYKIEWICZ, K., WEISS, D., 2002, "Traceability: Taming uncontrolled change in software development". In: *Proceedings of IV National Software Engineering Conference*, Tarnowo Podgorne, Poland.

LEBSACK, C. S., MROCZEK, A. J., MUELLER, C. J., 2001, "Controlling Configuration Items in Component Based Software Development". In: *Tenth International Workshop on Software Configuration Management (SCM-10)*, Toronto, Canada, May.

LEHMAN, M. M., PERRY, D. E., RAMIL J.F., et al., 1997, "Metrics and Laws of Software Evolution: The Nineties View". In: *4th International Symposium on Software Metrics (Metrics 97)*, pp. 20-32, Albuquerque, NM, November.

LEON, A., 2000, *A Guide to Software Configuration Management*, Norwood, MA, Artech House Publishers.

LETELIER, P., 2002, "A Framework for Requirements Traceability in UML-based Projects". In: *1st International Workshop on Traceability in Emerging Forms of Software Engineering*, Edinburgh, U.K, September.

LINDVALL M., SANDAHL, K., 1996, "Practical Implications of Traceability", *Software Practice and Experience*, v. 26, n. 10, pp. 1161-1180.

LOPES, L. G. B., *Odyssey-CCS: Um Sistema de Controle de Modificações Para Desenvolvimento Baseado em Componentes*. COPPE/UFRJ Proposta de Tese de Mestrado.

MATULA, M., 2003, *NetBeans Metadata Repository*. NetBeans Community.

MICHAIL, A., 2000, "Data Mining library reuse patterns using generalized association rules". In: *Proceedings of the 22nd International Conference on Software Engineering*, pp. 167-176, Limerick, Ireland, June.

MICROSOFT, 2005, "Microsoft COM Technologies - Information and Resources for the Component Object Model-based technologies". In: http://www.microsoft.com, Accessed in 21/02/2005.

MILLER, M., 2000, *A Engenharia de Aplicações no Contexto da Reutilização Baseada em Modelos de Domínio*, Tese de M.Sc., COPPE/UFRJ, Rio de Janeiro, RJ, Brasil.

MOCKUS, A., VOTTA, L. G., 2000, "Identifying reasons for software changes using histories databases". In: *Proceedings International Conference on Software Maintenance (ICSM 2000)*, pp. 120-130, San Jose, California, USA, October.

MOCKUS, A., WEISS, D. M., ZHANG, P., 2003, "Understanding and Predicting effort in software projects". In: *Proceedings 25th International Conference on Software Engineering (ICSE)*, pp. 274-284, Portland, Oregon, May.

MONTES DE OCA, C., CARVER, L. D., 1998, "Identification of data cohesive subsystems using data mining techniques". In: *Proceedings of the International Conference on Software Maintenance ICSM'98*, pp. 16-23, Bethesda, Maryland, USA, November.

MURPHY, G. C., NOTKIN, D., SULLIVAN, K., 1995, "Software Reflexion Models: Bridging the Gap between Source and High-Level Models". In: *Proceedings of the Third ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pp. 18-28, New York, NY, October.

MURTA, L. G. P., BARROS, M. O., WERNER, C. M. L., 2001, "FrameDoc: Um Framework para a Documentação de Componentes Reutilizáveis". In: *IV International Symposium on Knowledge Management/Document Management (ISKM/DM'2001)*, pp. 241-259, Curitiba, Brasil.

MURTA, L. G. P., BARROS, M. O., WERNER, C. M. L., 2002, "Charon: Uma Ferramenta para a Modelagem, Simulação, Execução e Acompanhamento de Processos de Software". In: *XVI Simpósio Brasileiro de Engenharia de Software*, pp. 366-371, Gramado, RS, Brasil, Outubro.

MURTA, L. G. P., VASCONCELOS, A. P. V., BLOIS, A. P. T. B., et al., 2004, "Run-time Variability through Component Dynamic Loading". In: *XVIII Simpósio Brasileiro de Engenharia de Software, Seção de Ferramentas*, Brasília, DF, Brasil, Outubro.

MURTA, L. G. P., OLIVEIRA, H., DANTAS, C. R., et al., 2004, "Towards Component-based Software Maintenance via Software Configuration Management Techniques". In: *XVIII Simpósio Brasileiro de Engenharia de Software, Workshop de Manutenção de Software*, Brasília, DF, Brasil, Outubro.

MURTA, L. G. P., 2004, *Odyssey-SCM: Uma Abordagem de Gerência de Configuração de Software para o Desenvolvimento Baseado em Componentes*, Exame de Qualificação, COPPE/UFRJ, Rio de Janeiro, RJ, Brasil.

NETBEANS COMMUNITY, 2005, "Welcome to NetBeans". In: http: www.netbeans.org, Accessed in 21/02/2005.

OLIVEIRA, H., MURTA, L. G. P., WERNER, C. M. L., 2004, "Odyssey-VCS: Um Sistema de Controle de Versões Para Modelos Baseados no MOF". In: *XVIII Simpósio Brasileiro de Engenharia de Software, Sessão de Ferramentas*, Brasília, DF, Brasil, Outubro.

OMG, 2002b, *UML 2.0 Sperstructure: Final Adopted Specification*. Object Management Group.

OMG, 2003a, *Common Warehouse Metamodel (CWM) Specification, version 1.1*. Object Management Group.

OMG, 2002a, *Meta Object Facility (MOF) Specification, version 1.3*. Object Management Group.

OMG, 2003b, *Unified Modeling Language (UML) Specification, version 1.4*. Object Management Group.

OMG, 2003c, *XML Metadata Interchange (XMI) Specification, version 1.1*. Object Management Group.

PAGE-JONES, M., 1999, *Fundamentals of Object-Oriented Design in UML*. 1st. ed., Addison-Wesley.

PINHEIRO, F. A. C., GOGUEN, J. A., 1996, "An Object-Oriented Tool for Tracing Requirements", *IEEE Software*, v. 13, n. 2, pp. 52-64.

PINHEIRO, F. A. C., 2000, "Formal and Informal Aspects of Requirements Tracing". In: *III Workshop em Engenharia de Requisitos*, Rio de Janeiro, Brasil, Junho.

POHL, K., 1996, "PRO-ART: Enabling Requirements Pre-Traceability". In: *Proceedings of the 2nd IEEE International Conference on Requirements Engineering (ICRE'96), IEEE Computer Society Press*, pp. 76-84, Colorado Springs, Colorado, USA, April.

PRESSMAN, 2001, *Software Engineering: A Practitioner´s Approach*. 5 ed., McGraw-Hill.

RAMESH, B., JARKE, M., 2000, "Towards Reference Models for Requirements Traceability", *IEEE Transactions of Software Engineering*, v. 27, n. 1, pp. 58-93.

RAMESH, B., 2002, "Process Knowledge Management with Traceability", *IEEE Software*, v. 19, n. 3, pp. 50-52.

SANTOS, H. L., BARROS, R., FONSECA, D., 2003, "Uma Proposta para Gerenciamento de Metadados nos Padrões XML e DTD em Repositórios MOF". In: *XVIII Simpósio Brasileiro de Banco de Dados (SBBD)*, Manaus, AM, Brasil, Outubro.

SAYYAD-SHIRABAD, J., 2003, *Supporting Software Maintenance by Mining Software Update Record*, Ph.D. Thesis, School of Information Technology and Engineering, University of Ottawa, May.

SMITH, M., WEISS, D., WILCOX, P., et al., 2003, "The Ophelia Traceability Layer". In: *2nd Workshop on Cooperative Supports for Distributed Software Engineering Processes (CSSE)*, Benevento, Italy, March.

SOHLENKAMP, M., 1998, *Supporting Group Awareness in Multi-User Environments through Perceptualization*, M.Sc. Dissertation, Fachbereich Mathematic, Informatik der Universität, Gesamthochschule, Paderborn.

SRIKANT, R., AGRAWAL, R., 1995, "Mining generalized association rules". In: *Proceedings of the 21st Very Large Data Bases Conference (VLDB)*, pp. 407-419, Zurich, Switzerland, September.

SRIKANT, R., AGRAWAL, R., 1995, "Mining generalized association rules". In: *Proceedings of the 21st Very Large Data Bases Conference (VLBD)*, pp. 407-419, Zurich, Switzerland, September.

STANDISH, T. A., 1984, "An Essay on Software Reuse", *IEEE Transactions on Software Engineering*, v. 10, n. 5, pp. 494-497.

SUN, 2004, "Java Technology". In: http://www.java.sun.com, Accessed in 16/11/2004.

SUN, 2005, "Enterprise Java Beans Technology". In: http://java.sun.com/products/ejb, Accessed in 21/02/2005.

SZYPERSKY, C., 2002, *Component Software: Beyond object-oriented programming*. 2nd. ed., Addison-Wesley.

TEIXEIRA, H. V., 2003, *Geração de Componentes de Negócio a Partir de Modelos de Análise*, Tese de M.Sc., COPPE/UFRJ, Rio de Janeiro, RJ, Brasil.

VERONESE, G. O., NETTO, F. J., 2001, *Uma Ferramenta de Apoio a Recuperação de Projetos no Ambiente Odyssey*, Projeto Final de Curso, DCC/IM/UFRJ, Rio de Janeiro, RJ, Brasil.

VIEIRA, V., 2003, *Ariane: Um Mecanismo de Apoio a Percepção em Bases de Dados Compartilhadas*, Tese de M.Sc., COPPE/UFRJ, Rio de Janeiro, RJ, Brasil.

W3C, 2004, *Extensible Markup Language (XML) 1.0 (Third Edition)*. World Wide Web Consortium.

W3C, 2005, "Web services Activity". In: http://www.w3.org/2002/ws/, Accessed in 21/02/2005.

WEISER, M., 1984, "Program Slicing", *IEEE Transactions on Software Engineering*, v. 10, n. 7, pp. 352-357.

WERNER, C. M. L., MANGAN, M. A. S., MURTA, L. G. P., et al., 2003, "OdysseyShare: an Environment for Collaborative Component-Based Development". In: *International Conference on Information Reuse and Integration (IRI)*, Las Vegas, Nevada, USA, October.

XAVIER, J. R., 2001, *Criação e Instanciação de Arquiteturas de Software Específicas de Domínio no Contexto de uma Infra-Estrutura de Reutilização*, Tese de M.Sc., COPPE/UFRJ, Rio de Janeiro, RJ, Brasil.

YING, A. T., 2003, *Predicting Source Code Changes by Mining Revision History*, M.Sc. Dissertation, University of British Columbia, October.

ZIMMERMANN, T., WEISGERBER, P., DIEHL, S., et al., 2003a, "Mining version histories to guide software changes". In: *Proceeding International Conference on Software Engineering (ICSE 2004)*, pp. 23-28, Scotland, UK, May.

ZIMMERMANN, T., DIEHL, S., ZELLER, A., 2003b, "How history justifies system architecture (or not)". In: *Proceedings International Workshop on Principles of Software Evolution (IWPSE 2003)*, pp. 73-83, Helsinki, Finland, September.