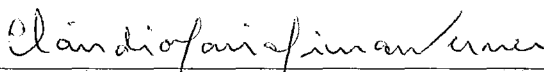
ODYSSEY-VCS: UMA ABORDAGEM DE CONTROLE DE VERSÕES PARA ELEMENTOS DA UML

Hamilton Luiz Rodrigues de Oliveira

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. Renata Pontin de Mattos Fortes, D.Sc.

Prof. Guilherme Horta Travassos, D.Sc.

RIO DE JANEIRO, RJ – BRASIL
MAIO DE 2005

OLIVEIRA, HAMILTON LUIZ RODRIGUES

Odyssey-VCS: Uma Abordagem de Controle de Versões para Elementos da UML [Rio de Janeiro] 2005

X, 94 p., 29,7 cm (COPPE/UFRJ, M.Sc., Engenharia de Sistemas e Computação, 2005)

Tese – Universidade Federal do Rio de Janeiro, COPPE

1. Sistemas de Controle de Versões
2. Artefatos de Análise e Projeto
3. *Unified Modeling Language* (UML)

I. COPPE/UFRJ II. Título (série)

A minha esposa, Ana Paula,

a meu filho, Arthur

e a meus pais

Agradecimentos

À Prof. Claudia Werner pela paciência, dedicação, preocupação, orientação, pelo apoio, pelos ensinamentos e pela ajuda antes mesmo de meu ingresso no mestrado na COPPE. Agradeço ainda pelo espírito de grupo e também pelas críticas, sempre construtivas, que me forçavam a uma reflexão sobre os rumos desta dissertação.

Aos professores da área de Engenharia de Software da COPPE, em especial ao Prof. Guilherme Horta Travassos, por participar da banca examinadora da minha tese. Agradeço também por me dar a oportunidade de participar das disciplinas de Engenharia de Software, Engenharia de Software Orientada a Objetos e Laboratório de Engenharia de Software. Estas disciplinas foram muito importantes na minha formação durante o mestrado e tiveram sua parcela de contribuição na elaboração desta dissertação.

À Prof. Renata Pontin Mattos Fortes por participar da banca examinadora da minha tese.

À Cristine, pela companhia e trocas de idéias que foram muito importantes durante as disciplinas e também na pesquisa da dissertação. Ao Leonardo, pela imensa ajuda na área de Gerência de Configuração de Software e por me mostrar que é possível se divertir durante e com a dissertação de mestrado.

Aos demais colegas do Projeto Odyssey: Aline Vasconcelos, Marco Aurélio Mangan, Alexandre Correa, Ana Paula, Marco Lopes, Alexandre Dantas, Luiz Gustavo, Regiane, Natanael, Isabella, Beto, Artur e Rafael pelo companheirismo.

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

ODYSSEY-VCS: UMA ABORDAGEM DE CONTROLE DE VERSÕES PARA ELEMENTOS DA UML

Hamilton Luiz Rodrigues de Oliveira

Maio / 2005

Orientadora: Cláudia Maria Lima Werner

Programa: Engenharia de Sistemas e Computação

Grande parte do esforço da comunidade de gerência de configuração de software foi direcionado para a pesquisa de técnicas e ferramentas para automatizar a evolução do código-fonte dos sistemas. Por outro lado, a área de desenvolvimento de software evoluiu para utilizar artefatos de análise e projeto, na tentativa de melhor gerenciar a complexidade inerente aos sistemas modernos.

Neste contexto, este trabalho propõe uma abordagem de controle de versões para elementos da *Unified Modeling Language* (UML). Para atender a esse objetivo, foram definidos seis requisitos: (i) granularidade fina; (ii) não-intrusão; (iii) compatibilidade com ferramentas *CASE*; (iv) flexibilidade; (v) acesso concorrente; e (vi) distribuição. Desta forma, este trabalho contribui para que organizações que adotam uma abordagem orientada a modelos, para o desenvolvimento de seus produtos, sejam capazes de evoluir seus artefatos de análise e projeto de forma automatizada.

As propostas desta dissertação foram realizadas no contexto do Projeto Odyssey, em desenvolvimento na COPPE/UFRJ.

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

ODYSSEY-VCS: AN APPROACH TO VERSION CONTROL FOR UML ELEMENTS

Hamilton Luiz Rodrigues de Oliveira

May / 2005

Advisor: Cláudia Maria Lima Werner

Department: Computer and System Engineering

A significant effort of the community of software configuration management was directed to the research of techniques and tools to automate the evolution of the source-code. On the other hand, the area of software development evolved to use analysis and design artifacts, to better manage the inherent complexity of modern systems.

In this context, this work proposes an approach for versioning Unified Modeling Language (UML) elements. To achieve this objective, six requirements were defined: (i) fine granularity; (ii) no-intrusion; (iii) compatibility with existing *CASE* tools; (iv) flexibility; (v) concurrent access; and (vi) distribution.

In this way, this work contributes so that organizations that adopt a model-driven approach for the development of their products, are able to evolve their analysis and design artifacts in an controlled way.

The proposals of this thesis were done in the context of the Odyssey Project, under development at COPPE/UFRJ.

Índice

Capítulo 1 - Introdução 1

1.1 Contexto 1

1.2 Motivação 2

1.3 Problema 3

1.4 Objetivo 4

1.4.1 Granularidade fina 5
1.4.2 Não-intrusão 5
1.4.3 Compatibilidade com ADSs ou ferramentas *CASE* existentes 5
1.4.4 Flexibilidade 5
1.4.5 Acesso concorrente 6
1.4.6 Distribuição 6

1.5 Organização 6

Capítulo 2 - Controle de Versões 7

2.1 Introdução 7

2.2 Conceitos básicos de controle de versões 8

2.2.1 Versão, item de configuração e configuração 8
2.2.2 Delta 8
2.2.3 Revisões e variantes 9
2.2.4 Espaço de trabalho, *check-out* e *check-in* 10
2.2.5 Acesso concorrente, políticas otimista e pessimista, e junção 10

2.3 Controle de versões para código-fonte 10

2.3.1 Modelo de dados baseado em sistema de arquivos 11
2.3.2 Outros modelos de dados 12
2.3.3 A computação de deltas 13
2.3.4 Versionamento orientado a mudança 14
2.3.5 Considerações finais sobre controle de versões para código-fonte 16

2.4 Controle de versões para hipertexto e linguagem de marcação 17

2.4.1 Versionamento de documentos disponíveis na *web* 17
2.4.2 Versionamento de documentos descritos em XML 19
2.4.3 Considerações finais sobre controle de versões para hipertexto e linguagem de marcação 20

2.5 Controle de versões para artefatos de análise e projeto 20

2.5.1 Abordagem de OHST e KELTER 20
2.5.2 *Adaptive/IBM* 21
2.5.3 MIMIX 22
2.5.4 Abordagem de LUCRÉDIO e PRADO 22
2.5.5 DVM 23
2.5.6 MOLHADO 23
2.5.7 Considerações finais sobre controle de versões para artefatos de análise e projeto 24

2.6 Considerações finais 24

Capítulo 3 - Odyssey-VCS: Controle de versões para elementos da UML 27

3.1 Introdução 27

3.2 Granularidade fina 28

3.3 Não-intrusão ... 31

3.4 Compatibilidade com ADSs e ferramentas *CASE* existentes ... 33

3.5 Flexibilidade ... 35
3.5.1 Flexibilidade na definição do grão de versionamento ... 35
3.5.2 Flexibilidade na identificação e notificação de conflitos ... 36

3.6 Acesso concorrente ... 39

3.7 Distribuição ... 44

3.8 Exemplo ... 45

3.9 Considerações finais ... 49

Capítulo 4 - O Protótipo do Odyssey-VCS ... 51

4.1 Introdução ... 51

4.2 Cenário de utilização ... 52
4.2.1 Ambiente Odyssey ... 52
4.2.2 Poseidon for UML ... 54

4.3 Detalhamento do Odyssey-VCS ... 56
4.3.1 Detalhamento do projeto ... 56
4.3.2 Detalhamento da granularidade ... 57

4.4 Camada de transporte ... 59
4.4.1 Protocolo de comunicação ... 59
4.4.2 Formato de representação dos dados ... 61

4.5 Exemplo de utilização ... 63
4.5.1 Utilização local ... 63
4.5.2 Utilização distribuída ... 68
4.5.3 Tratamento de conflitos ... 72

4.6 O Odyssey-VCS na Gerência de Configuração de Componentes ... 75
4.6.1 Detecção de rastros de modificação entre elementos ... 76
4.6.2 Exemplo de utilização do Odyssey-VCS na Gerência de Configuração de Componentes ... 77

4.7 Considerações finais ... 81

Capítulo 5 - Conclusão ... 83

5.1 Visão geral ... 83

5.2 Contribuições ... 83

Referências bibliográficas ... 88

Índice de Figuras

Figura 1: Uma árvore de versões (ESTUBLIER *et al.*, 2002) 9
Figura 2: Uma árvore de versões com delta para frente e para trás 13
Figura 3: Perspectivas de desenvolvimento de software e de controle de versões 28
Figura 4: Versionamento da UML nas abordagens atuais (adaptado de MURTA, 2004). 29
Figura 5: A interface "Tratador" e algumas classes que a implementam 31
Figura 6: Meta-modelo de versionamento 32
Figura 7: Meta-modelo de versionamento e o meta-modelo parcial da UML 33
Figura 8: Arquitetura em quatro camadas do MOF (adaptado de MATULA, 2005) 34
Figura 9: Grão de comparação aplicado a arquivos texto (MURTA, 2004) 37
Figura 10: Cenário de modificação envolvendo diferentes configurações 40
Figura 11: A classe "Hospede", com atributo "telefone" 45
Figura 12: *"Check-out"* sobre os elementos no repositório 46
Figura 13: Realização de junção no *"Check-in"* do João 47
Figura 14: Resultado da junção aplicada à classe "Hospede" 48
Figura 15: Visão geral do Odyssey-VCS 51
Figura 16: Seleção do OdysseyVCSPlugin para instalação no Odyssey 54
Figura 17: Visualização do Odyssey-VCS *Client* 55
Figura 18: Detalhamento do projeto do Odyssey-VCS 56
Figura 19: Tratamento para GV e GC 58
Figura 20: Visão geral do Odyssey-VCS com ênfase na camada de transporte 59
Figura 21: Arquivo em XML definindo o serviço ControleDeVersão 60
Figura 22: Elementos em XMI enviados ao Odyssey-VCS 62
Figura 23: Elementos em XMI com identificador em *TaggedValue* 63
Figura 24: Elementos da UML para o exemplo de utilização 63
Figura 25: Ambiente Odyssey expondo as funcionalidades de controle de versões 64
Figura 26: Visualização dos elementos que se tornaram ICs. 65
Figura 27: Resultado da ação da opção *"Check-out"* 66
Figura 28: Inserção do atributo "telefone", na classe "Hospede" 67
Figura 29: Visualização do repositório após a ação de *"Check-in"* 68
Figura 30: Desenvolvedor utilizando o Odyssey para modificar elementos 69
Figura 31: OdysseyVCS *Client* - intermediário entre o Poseidon e o OdysseyVCS 70
Figura 32: Tela de modelagem do Poseidon expondo os elementos do Odyssey-VCS 71
Figura 33: Visualização do repositório após o *"Chech-in"* de João 72
Figura 34: Relatório informando os conflitos 74
Figura 35: Resultado da junção aplicada a classe "Hospede" 74
Figura 36. Diagramas de caso de uso (a) e de classes (b) do domínio de hotelaria 77
Figura 37. Componentes do exemplo de hotelaria 78
Figura 38: Visualização dos elementos através do Odyssey-VCS Client 78
Figura 39: Visualização do repositório após a realização das modificações 80
Figura 40: Resultado da detecção dos rastros de modificação 80

Índice de Tabelas

Tabela 1: Quadro-resumo das abordagens descritas no capítulo. 25
Tabela 2: Procedimentos de junção em função do cenário de modificação. 41
Tabela 3: Quadro-resumo comparando o Odyssey-VCS com as demais abordagens. 50
Tabela 4. Descrição das modificações realizadas no exemplo hotelaria 79

Capítulo 1 - Introdução

1.1 Contexto

Desenvolver software não é tarefa trivial. As dificuldades têm aumentado porque os sistemas estão se tornando mais complexos em tamanho, sofisticação e tecnologias utilizadas (LEON, 2000). O aumento da complexidade é um obstáculo na tentativa de melhorar o grau de previsibilidade dos produtos a serem construídos, porque aumenta o número de incertezas nos projetos de desenvolvimento. Além da complexidade associada ao produto e às tecnologias utilizadas, a forma de organização das equipes, que podem estar geograficamente distribuídas, torna o desenvolvimento de software uma tarefa ainda mais difícil.

O paradigma de desenvolvimento utilizado também contribui para o aumento ou a diminuição do grau de complexidade. O Desenvolvimento Baseado em Componentes (DBC) surgiu com a proposta de melhorar a reutilização, na medida que artefatos de software são produzidos para atuarem em um domínio de aplicações, ao invés de serem destinados a uma aplicação específica (BRAGA, 2000). O DBC opõe-se a idéia de desenvolvimento de software a partir do "zero", além de propor que os componentes sejam facilmente substituíveis, do mesmo modo que ocorrem com dispositivos de hardware.

O DBC trouxe mudanças na forma como se desenvolve software, como, por exemplo, a criação de papéis para equipes de desenvolvimento. Existem as equipes produtoras, responsáveis pelo desenvolvimento de componente *para* reutilização, e as equipes consumidoras, encarregadas do desenvolvimento de componentes *com* reutilização.

Se considerados isoladamente, esses fatores são suficientes para tornar ainda mais complexa a atividade de desenvolvimento de software moderno. No entanto, a mudança é inerente a qualquer projeto de desenvolvimento, independentemente do paradigma adotado. Projetos de desenvolvimento que não controlam a mudança são rapidamente conduzidos ao caos.

O caos surge quando as mudanças não são analisadas antes de serem realizadas, registradas antes que sejam implementadas, reportadas para aqueles que precisam

conhecê-las, ou controladas de tal forma que a qualidade seja melhorada e os erros sejam reduzidos (PRESSMAN, 2001).

A Gerência de Configuração de Software (GCS) introduz uma série de atividades e procedimentos aos ambientes de desenvolvimento de software. TICHY (1988) define a GCS como uma disciplina para gerenciar o desenvolvimento e a evolução de sistemas grandes e complexos. O objetivo da GCS é maximizar a produtividade através da redução dos erros (PRESSMAN, 2001).

A IEEE (*Institute of Electrical and Electronic Engineers*) Std 828 (IEEE, 1998) divide as funções da GCS em quatro atividades principais: (1) identificação da configuração, (2) controle da configuração, (3) relato da situação e (4) avaliação da configuração. A função de *controle da configuração*, na qual esta dissertação se encontra inserida, consiste em avaliar, coordenar, aprovar, ou desaprovar, e implementar mudanças em um Item de Configuração[1] (IC).

Os Sistemas de Controle de Versões (SCV) combinam procedimentos e ferramentas para gerenciar as diferentes versões de artefatos que são criados e modificados durante o ciclo de vida do software (PRESSMAN, 2001). Por esse motivo, são considerados a parte mais importante da GCS (CONRADI e WESTFECHTEL, 1998) e, de longe, a área, neste contexto, que mais recebeu atenção da indústria e da academia (ESTUBLIER, 2000). O objetivo principal do controle de versões é auxiliar a função de controle da configuração.

1.2 Motivação

O aumento da complexidade dos sistemas dificulta o trabalho dos desenvolvedores, na medida em que os conceitos e relacionamentos envolvidos no problema em questão exigem maior esforço para que sejam entendidos e representados. Historicamente, a técnica da abstração tem sido a resposta da indústria e da academia para tratar a crescente complexidade dos sistemas (FRANKEL, 2005).

A abstração é uma técnica que ajuda a gerenciar a complexidade inerente a problemas grandes, reduzindo assim os impactos negativos da limitação dos seres

---

[1] O termo Item de Configuração pode significar, por exemplo, arquivos (usualmente textual), diretórios, objetos em banco de dados orientado a objetos, entidades, relacionamentos e atributos em banco de dados relacionais.

humanos em tratar grandes quantidades de informação. Através da abstração é possível identificar e manipular somente os aspectos do problema considerados relevantes em um dado momento.

A utilização de artefatos de análise e projeto para o desenvolvimento e a manutenção de um sistema pode ser entendida como uma evolução natural na tentativa de melhorar a capacidade dos desenvolvedores em gerenciar a complexidade inerente aos sistemas modernos. Os sistemas complexos demandam um esforço adicional para a sua compreensão, tornando necessário o uso de artefatos que descrevem os aspectos não representados pelo código-fonte, agregando informação em um nível de abstração mais elevado (BOOCH *et al.*, 2000).

No desenvolvimento de software orientado a modelos, o desenvolvedor cria um conjunto de artefatos que descreve apenas os aspectos conceituais do problema. Esses artefatos são modificados porque novos conceitos necessitam ser representados ou porque conceitos que foram incorretamente entendidos e modelados precisam ser corrigidos.

Para gerenciar essas modificações, os desenvolvedores, na maior parte das vezes, utilizam-se de sistemas de cópias (*backup*) ou simplesmente controlam os diretórios nos quais esses arquivos estão contidos (OHST e KELTER, 2002). Entretanto, este tipo de sistema é rudimentar para ser utilizado em ambientes de desenvolvimento de software modernos. Portanto, são necessários SCVs capazes de auxiliar os gerentes e desenvolvedores na evolução de artefatos de análise e projeto, considerando que a mudança é uma característica inerente a qualquer projeto de desenvolvimento de software (BERSOFF *et al.*, 1980).

1.3 Problema

Conforme dito anteriormente, a GCS é uma disciplina para gerenciar o desenvolvimento e a evolução de sistemas grandes e complexos. A GCS tem conseguido resultados satisfatórios e teve sua importância reconhecida, conforme demonstrado particularmente no *Capability Maturity Model Integration* (CMMI) (SEI, 2005). Neste modelo, a GCS é uma das áreas necessárias para uma organização passar do nível "inicial" (processo indefinido) para o "repetível" (gerência de projeto, GCS e avaliação

da qualidade). Além disso, a GCS realiza um papel importante para a obtenção do certificado ISO 9000 (CONRADI e WESTFECHTEL, 1998).

Atualmente, a GCS, mais especificamente, os SCVs, são essencialmente voltados para código-fonte e, para realizar esta tarefa, utilizam um modelo de dados baseado em sistemas de arquivos (OHST e KELTER, 2002). Este modelo de dados tem se mostrado suficiente porque o código-fonte dos sistemas é representado em arquivos no formato de texto.

No entanto, quando se versiona artefatos de alto nível de abstração (modelos UML, por exemplo) utilizando modelo de dados baseado em sistemas de arquivos, os resultados obtidos são pouco satisfatórios. O problema está relacionado à granularidade oferecida, porque nesse modelo de dados um arquivo é um IC indivisível. Portanto, um modelo persistido em um arquivo é versionado como um elemento único.

Utilizando esses sistemas, não é possível saber a história de evolução de um pacote, uma classe, um caso de uso, uma operação ou um atributo, por exemplo. A adoção de um modelo de dados que possibilite a manipulação de objetos complexos pelos SCVs é um dos desafios a serem enfrentados pela comunidade de GCS (ESTUBLIER, 2000).

1.4 Objetivo

Diante do problema apresentado na Seção 1.3, esta dissertação tem como objetivo propor uma abordagem de controle de versões para elementos da UML, que leva em conta o conhecimento sobre a estrutura desses elementos. No contexto dessa dissertação, um elemento da UML é uma instância de qualquer conceito descrito no meta-modelo da UML. Alguns exemplos desses conceitos são pacote, classe, operação, atributo, entre outros. Para alcançar esse objetivo, a abordagem deve atender aos seguintes requisitos: (i) granularidade fina; (ii) não-intrusão; (iii) compatibilidade com ambientes de desenvolvimento de software (ADS) ou ferramentas *CASE* existentes; (iv) flexibilidade; (v) acesso concorrente; e (vi) distribuição.

A seguir, são descritos os motivos que reforçam a necessidade de cada um desses requisitos.

1.4.1 Granularidade fina

Os SCVs atuais versionam todos os elementos contidos em um modelo como uma entidade única, o que se configura numa granularidade grossa. Por outro lado, os elementos da UML representam conceitos e relacionamentos complexos presentes no domínio do problema. A adoção da granularidade fina para versionar tais elementos equivale a dizer que serão registradas as modificações individuais de cada elemento, à medida que os mesmos evoluem.

1.4.2 Não-intrusão

No contexto desta dissertação, não-intrusão significa que os elementos da UML não conterão qualquer informação de versionamento (por exemplo, o número da versão), e representarão somente os conceitos e relacionamentos presentes no domínio do problema em análise. O problema tratado pelos desenvolvedores já traz uma certa complexidade, por isso uma abordagem de controle de versões não deve modificar a estrutura dos elementos versionados para adicionar informações que sejam específicas à tarefa de versionamento.

1.4.3 Compatibilidade com ADSs ou ferramentas *CASE* existentes

O foco principal da GCS é auxiliar no desenvolvimento e manutenção de sistemas grandes e complexos. Quando sistemas complexos estão sendo desenvolvidos, é normal a existência de um grande número de pessoas envolvidas. Essas pessoas podem estar utilizando diferentes ADSs ou ferramentas *CASE*. Portanto, uma abordagem de controle de versões moderna deve ser compatível com ADSs e ferramentas *CASE* existentes.

1.4.4 Flexibilidade

A flexibilidade constitui-se no reconhecimento de que dois projetos de desenvolvimento de software não são idênticos. Desta forma, devem ser fornecidos mecanismos para que os gerentes de configuração possam moldar, tanto quanto possível, a abordagem de controle de versões conforme suas necessidades.

1.4.5 Acesso concorrente

Seria irreal supor que todo trabalho em um sistema grande e complexo é realizado seqüencialmente (GULLA *et al.*, 1991). Restrições de cronograma são também um fator determinante para permitir o acesso concorrente, na medida em que o desenvolvimento linear inibe a produtividade, principalmente em equipes grandes e distribuídas. Por isso, uma abordagem moderna de controle de versões deve suportar modificações paralelas em um mesmo elemento.

1.4.6 Distribuição

Suporte à engenharia distribuída é um dos fatores mais preocupantes para os usuários de SCVs atuais. Isso ocorre porque equipes modernas de desenvolvimento de software podem estar geograficamente distribuídas. Diante disso, ESTUBLIER (2000) enfatiza a necessidade da abordagem de controle de versões estar disponível, preferencialmente, através da *web*.

1.5 Organização

Este documento está organizado em outros quatro capítulos, além desta introdução. O Capítulo 2 apresenta os conceitos básicos de controle de versões e fornece uma revisão da literatura sobre o assunto. O Capítulo 3 descreve a abordagem de controle de versões, objeto desta dissertação, organizada de acordo com os requisitos definidos na Seção 1.4. Algumas limitações da abordagem são também descritas. O Capítulo 4 apresenta o protótipo resultante da implementação da abordagem, um exemplo de utilização, além das tecnologias utilizadas. O Capítulo 5 apresenta as considerações finais, relatando as contribuições e possíveis trabalhos futuros.

Capítulo 2 - Controle de Versões

2.1 Introdução

As características dos projetos de desenvolvimento de software foram bastante modificadas ao longo de trinta anos. As modificações afetaram as equipes, as tecnologias, as metodologias e até mesmo o grau de exigência dos usuários. Os problemas nesses projetos tornam-se mais graves quando um novo elemento é adicionado: a mudança. BERSOFF (1980) declarou que, independentemente de onde se esteja no ciclo de vida, o software mudará e a mudança persistirá através de todo o ciclo de vida.

Portanto, em qualquer projeto de desenvolvimento de software moderno, o SCV é um elemento essencial. Várias soluções comerciais e livres estão disponíveis e oferecem características satisfatórias quando se deseja versionar artefatos simples, definidos através de arquivos do sistema operacional (MURTA, 2004).

No entanto, desenvolver software significa criar e modificar diferentes tipos de artefatos. O documento de especificação de requisitos, os modelos de análise e projeto, o código-fonte e os esquemas de banco de dados são exemplos desses artefatos. Apesar disso, conforme dito anteriormente, a grande maioria dos SCVs está voltada para o código-fonte.

Este capítulo apresenta uma visão geral sobre a área de controle de versões. Como esta dissertação tem como objetivo propor uma abordagem de controle de versões para artefatos de análise e projeto, descritos em UML, as abordagens foram categorizados de acordo com o tipo do artefato versionado, de modo a facilitar a análise e, posteriormente, a comparação entre elas.

Este capítulo está organizado da seguinte forma: a Seção 2.2 descreve os conceitos básicos da área de controle de versões; a Seção 2.3 descreve as abordagens de controle de versões para código-fonte; a Seção 2.4 trata as abordagens para hipertexto e linguagem de marcação; a Seção 2.5 descreve as abordagens de controle de versões para artefatos de análise e projeto; e na Seção 2.6 são apresentadas as considerações finais do capítulo.

2.2 Conceitos básicos de controle de versões

Esta seção apresenta alguns conceitos importantes, utilizados na área de controle de versões. Uma descrição detalhada desses conceitos pode ser encontrada em (CONRADI e WESTFECHTEL, 1998) e (ESTUBLIER *et al.*, 2002).

2.2.1 Versão, item de configuração e configuração

Uma versão *v* representa um estado de um *Item de Configuração (IC) i* evoluindo. O termo IC pode significar, por exemplo, arquivos (usualmente textual), diretórios, objetos em banco de dados orientado a objetos, entidades, relacionamentos e atributos em banco de dados relacionais. No contexto desta dissertação, um IC pode ser também um elemento da UML.

Uma versão *v* é caracterizada pelo par *v=(ep,ev)*, onde *ep* e *ev* denotam um estado no espaço do produto e um ponto no espaço de versão, respectivamente. O espaço do produto contém os ICs para serem versionados e o espaço de versão organiza suas versões em estruturas, como um grafo de versão ou em forma de grade *(grid)*. Por exemplo, *ep* pode representar o conteúdo do arquivo "Pessoa.java" e *ev* a versão 1 desse arquivo.

Uma *configuração* é uma versão de um objeto complexo. Ela é composta das versões das suas partes. Por exemplo, uma configuração de um sistema é composta de versões de documento de requisitos, da arquitetura do software, dos modelos de análise e projeto, do código-fonte, etc.

2.2.2 Delta

Nas décadas de 70 e 80, a escassez de recursos de memória influenciava fortemente as decisões de projeto no desenvolvimento dos SCVs. Para solucionar o problema, foi proposto o conceito de *delta*. Deltas são importantes porque duas versões sucessivas são usualmente bastante similares (98%, na média). Com essa abordagem, são armazenadas somente as modificações (os 2% diferentes) (ESTIBLIER, 2000).

Existem duas variações do conceito de delta. Com o *delta para frente*, a versão mais antiga é armazenada e, para montar as versões mais recentes, são processadas as

diferenças. Com o *delta para trás*, é armazenada integralmente a versão mais recente e as diferenças existentes até então.

2.2.3 Revisões e variantes

De acordo com o tipo de evolução, as versões são classificadas em *revisões* e *variantes*. As versões seqüenciais que evoluem ao longo do tempo são chamadas de revisões. Elas são criadas quando defeitos são corrigidos ou quando são adicionadas novas funcionalidades. As versões paralelas, ou alternativas, que coexistem são chamadas de variantes. Enquanto as novas versões sucedem as versões mais antigas, variantes não substituem umas as outras. Ao invés disso, elas são usadas concorrentemente em configurações alternativas.

Por exemplo, variantes de estruturas de dados podem diferir em relação à eficiência ou consumo de memória, ou ainda serem destinadas a diferentes sistemas operacionais (CONRADI e WESTFECHTEL, 1998).

ESTUBLIER *et al.* (2002), por outro lado, tratam revisões e variantes como relacionamentos sucessores e utilizam o termo *ramo* como sinônimo de variante, conforme a Figura 1.

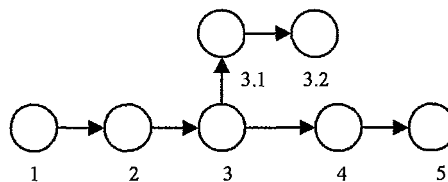


Figura 1: Uma árvore de versões (ESTUBLIER *et al.*, 2002)

Analisando a Figura 1, é possível dizer que a versão 4 é uma revisão da versão 3. Desta forma, a versão 4 substitui a versão 3. Por outro lado, a versão 3.1 é uma variante (ou ramo) da versão 3. A versão 3.1 foi criada para atender a algum requisito que foi considerado desnecessário na linha principal do desenvolvimento, e que justificou a criação de uma versão alternativa para ser utilizada paralelamente à versão 3. Os autores argumentam ainda que variantes (ou ramos) podem diferir em relação a algum aspecto funcional, de projeto ou de implementação.

2.2.4 Espaço de trabalho, *check-out* e *check-in*

Espaço de trabalho é uma área independente onde um desenvolvedor pode realizar seu trabalho, efetuando modificações sobre os ICs, isolado das tarefas realizadas pelos demais desenvolvedores.

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

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

2.2.5 Acesso concorrente, políticas otimista e pessimista, e junção

Acesso concorrente pressupõe que mais de um desenvolvedor poderá realizar modificações nos ICs existentes no repositório. Para gerenciar o acesso concorrente, duas políticas são normalmente utilizadas: a política pessimista e a política otimista.

A *política pessimista* enfatiza o uso do *Check-out* reservado, realizando bloqueio (*lock*) e inibindo o paralelismo do desenvolvimento sobre o mesmo IC (MURTA, 2004). Nestes cenários, não ocorrerão conflitos porque os demais desenvolvedores terão que aguardar até que o IC seja novamente liberado. Infelizmente, a realização de uma tarefa pode levar dias ou semanas, fazendo com que um IC se mantenha bloqueado por um tempo demasiadamente grande (ESTUBLIER *et al.*, 2002).

Considerando que o número de conflitos é normalmente baixo, a *política otimista* permite que os ICs sejam modificados ao mesmo tempo e propõe o tratamento individual dos conflitos, casos eles venham a ocorrer (ESTUBLIER *et al.*, 2003). Esta política usa um mecanismo de *junção*, que une os trabalhos efetuados em paralelo sobre um mesmo IC e produz uma nova versão que contém a soma desses trabalhos (MURTA, 2004).

2.3 Controle de versões para código-fonte

Esta seção trata de diversas abordagens para controle de versões para código-fonte disponíveis na literatura. Cabe dizer que os conceitos e técnicas propostos por esses sistemas estão presentes na maioria dos sistemas utilizados (ESTUBLIER, 2000). A Seção 2.3.1 trata do modelo de dados baseado em sistemas de arquivos, a Seção 2.3.2

descreve outros modelos de dados utilizados para versionar código-fonte, a Seção 2.3.3 trata da computação de deltas, a Seção 2.3.4 aborda o versionamento orientado a mudança, e a Seção 2.3.5 contém as considerações finais sobre controle de versões para código-fonte.

2.3.1 Modelo de dados baseado em sistema de arquivos

O *Source Code Control System* (SCCS) (ROCKIND, 1975) foi o primeiro SCV desenvolvido (ESTUBLIER, 2000). Alguns anos mais tarde, foi desenvolvido o *Revision Control System* (RCS) (TICKY, 1982), cujo objetivo era corrigir algumas limitações do SCCS. TICKY (1982) introduziu o conceito de nome simbólico, permitindo que um mesmo nome pudesse ser atribuído a diversas revisões, cada um com sua versão específica. Além disso, cada versão específica de uma revisão pode estar associada a mais de um nome simbólico.

Diferentemente, no SCCS, os arquivos são manipulados individualmente, como entidades independentes, sem um elemento global que os relacione logicamente. A partir do conceito de nome simbólico, pôde ser viabilizado o mecanismo de criação de configuração.

O *Concurrent Versions System* (CVS) (FOGEL e BAR, 2001) é um SCV de código-aberto bastante utilizado atualmente. Ele consiste numa variação do RCS/SCCS e um dos diferenciais é a possibilidade de realização de acesso concorrente. Desta forma, o CVS permite que dois ou mais desenvolvedores modifiquem o mesmo arquivo ao mesmo tempo, através do *Check-out* não-reservado.

Arquivos no formato texto são os ICs mais importantes para esses sistemas, pois possibilitam que operações comuns de controle de versões - como delta, por exemplo - sejam aplicadas. Por isso, esses sistemas são essencialmente voltados para manipular código-fonte.

O RCS/SCCS e suas variantes utilizam um modelo de dados baseado em sistemas de arquivos. Este é um modelo de dados fraco e constitui-se no maior problema dessas abordagens, porque impede que outros aspectos, mais notadamente o suporte a processo, atinjam um nível satisfatório (ESTUBLIER, 2000). Isso ocorre porque esses sistemas ignoram, quase que por completo, o conhecimento contido na estrutura dos elementos

que compõem uma linguagem de programação, ou qualquer outro tipo de artefato contido nos arquivos versionados.

2.3.2 Outros modelos de dados

Outros autores propuseram abordagens que utilizam modelos de dados mais sofisticados, como entidade-relacionamento, orientado a objetos, ou ainda modelo baseado na árvore sintática de uma linguagem específica. A seguir, são descritas algumas dessas abordagens.

HABERMANN e NOTKIN (1986) propuseram um sistema de controle de versões orientado a estrutura. A abordagem é baseada na árvore sintática da linguagem C, o que permite ao sistema gerenciar a consistência sintática do código-fonte versionado. Entretanto, somente pode ser utilizado quando o artefato controlado for o código-fonte escrito em linguagem C.

RENDER e CAMPBEL (1991) propuseram um sistema de controle de versões que utiliza um modelo de dados orientado a objetos. O sistema foi desenvolvido utilizando a linguagem Smaltalk e versiona código-fonte escrito em Pascal. A estrutura da linguagem Pascal é mapeada para o modelo de dados descrito pelos autores. Curiosamente, os autores continuaram versionando arquivos do sistema operacional, ao invés de manipular ICs menores, como, por exemplo, um procedimento ou uma função.

GOLDSTEIN e BOBROW (1980) propuseram uma abordagem que gerencia programas escritos em Smaltalk, representados por uma estrutura de dados em forma de grafo. No entanto, como toda abordagem demasiadamente específica, sofre o problema da aplicabilidade limitada.

SILVA *et al.* (2003) apresentam uma abordagem que faz uso de um Modelo Temporal Versionado (TVM) aplicado a uma perspectiva de controle de versões. Essa abordagem faz com que um IC passe a ser um objeto, ao invés de um arquivo do sistema operacional. No entanto, é empregado um meta-modelo proprietário, o que torna difícil adotá-la em ambientes onde a interoperabilidade é um requisito importante. Além disso, toda a informação de versão é armazenada no próprio artefato versionado, constituindo-se numa abordagem intrusiva.

2.3.3 A computação de deltas

Conforme dito anteriormente, deltas foram propostos para que as versões dos ICs ocupassem menos espaço em memória. O SCCS utiliza o delta para frente, enquanto que o RCS utiliza o delta para trás. TICKY (1982) argumenta que as versões mais recentes são mais freqüentemente acessadas, por isso, não computar delta sobre elas melhora o desempenho do sistema. Os ramos, no entanto, exigem um tratamento especial. Eles poderiam ser armazenados integralmente, o que era inaceitável para os padrões de utilização de memória da época. TICKY (1982) solucionou o problema utilizando uma combinação de delta para frente e delta para trás, conforme mostra a Figura 2. O triângulo invertido representa o delta para trás, enquanto o triângulo normal simboliza o delta para frente.

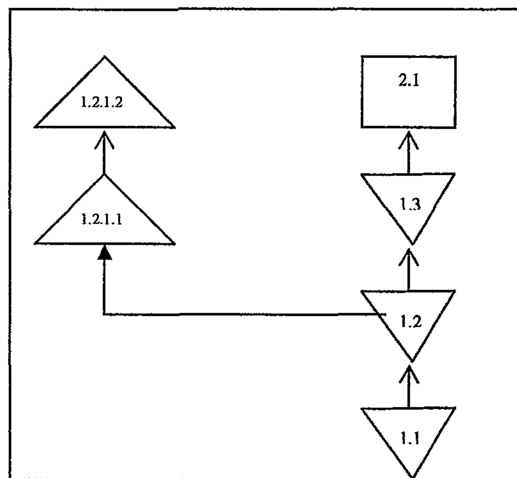


Figura 2: Uma árvore de versões com delta para frente e para trás

A última versão da linha principal (2.1) é armazenada integralmente. Para se obter qualquer versão dos ramos (1.2.1.1, 1.2.1.2), deve-se realizar o seguinte procedimento: realiza-se uma cópia da versão da linha principal (2.1) e aplica-se o delta para trás, até obter o antecedente do ramo na linha principal do desenvolvimento (1.2). A partir daí, aplica-se o delta para frente até obter a última versão do ramo (1.2.1.2).

Durante os anos 80, foram propostos diversos mecanismos de recuperação e armazenagem de delta. Entretanto, nos anos 90, objetos não-textuais tornaram-se mais comuns, o que obrigou o desenvolvimento de algoritmos totalmente novos. Além disso, algoritmos para delta são complexos. A complexidade em software significa algoritmos

mais difíceis para desenvolver e modificar, o que se traduz em um maior esforço e aumento do custo geral do sistema.

Atualmente, o barateamento dos recursos de memória, a melhora na capacidade de processamento e o aumento no número de objetos não-textuais diminuíram a importância na computação de deltas. Por isso, ESTUBLIER *et al.* (2002) sugerem que algoritmos de compressão como, por exemplo, o *zip*, podem ser utilizados no armazenamento dos ICs.

2.3.4 Versionamento orientado a mudança

A criação de novas configurações, através da utilização de ICs já existentes no repositório, é outro problema no RCS/SCCS e suas variantes. O problema está relacionado ao desempenho do sistema, e se torna particularmente crítico quando o número de variantes existentes no repositório é muito elevado. A técnica de versionamento orientado a mudança (VOM), em oposição ao clássico versionamento orientado a estado (VOE), foi proposta para amenizar esse problema (HARTER, 1981).

O VOM funciona da seguinte forma: quando é realizado o *Check-in*, todas as mudanças realizadas (materializadas na computação de deltas) são etiquetadas com o mesmo nome (*label*). ESTUBLIER *et al.* (2002) argumentam que, desta forma, torna-se possível reconstruir arquivos que nunca foram criados antes, através da combinação de mudanças realizadas independentemente, talvez concorrentemente. Por exemplo, a instrução *novaVersãoUnix = Unix ^ func35 ^ ~correcaoErro27* determina a criação de uma nova configuração do Unix, que se chamará *novaVersãoUnix*, adicionada da funcionalidade *35*, sem a modificação que corrigiu o erro de número 27 (GULLA *et al.*, 1991).

No VOM, uma *mudança* é a soma de modificações relacionadas aplicadas sobre muitos ICs. A motivação é que uma modificação local (o delta de um arquivo) é freqüentemente parte de uma modificação global (uma transação), envolvendo muitos outros ICs.

Uma importante contribuição dessa abordagem é a mudança de foco em relação ao repositório. No VOE, as informações que motivaram uma modificação constam apenas como comentários introduzidos pelos desenvolvedores, no momento da realização

do *Check-in*. No VOM, por outro lado, elas adquirem um papel mais relevante e são utilizadas como índices, a partir dos quais os ICs passam a ser identificados e recuperados.

Dessa forma, os usuários deixam de visualizar os estados de ICs e passam a visualizar as mudanças que são aplicadas sobre esses ICs. Nenhuma informação a respeito da forma como as versões são armazenadas é disponibilizada, e os usuários conhecem somente as mudanças lógicas (ESTUBLIER e CASALLAS, 1994).

No entanto, sistemas que utilizam VOM nem sempre trabalham bem na prática. Um dos motivos é que, algumas vezes, deltas sobrepõem-se e conflitam de tal forma que uma configuração construída pode nem mesmo ser compilável ou depurável (ESTUBLIER *et al.*, 2002).

O modelo de dados utilizado é baseado em sistema de arquivos, o que limita o artefato versionado ao código-fonte. Apesar disso, é possível trabalhar numa granularidade fina, onde a menor unidade pode ser uma linha (GULLA *et al*, 1991; ESTUBLIER e CASALLAS, 1994), ou um método em Smaltalk (GOLDSTEIN e BOBROW, 1980).

VOE e VOM, no entanto, não são mutuamente exclusivos e podem ser combinados, permitindo a construção de sistemas híbridos (WEBER, 1997). O próprio SCCS, embora originalmente suporte VOE, fornece alguns comandos de baixo nível para manipular deltas, de uma forma mais próxima a VOM (CONRADI e WESTFECHTEL, 1998).

Outro exemplo é o Projeto Subversion (TIGRIS, 2005a). Iniciado em maio de 2000, o Subversion tem como objetivo fornecer uma melhor implementação para o CVS, de tal forma que certas deficiências funcionais sejam solucionadas. Um dos diferenciais do Subversion é a possibilidade de versionar, não somente arquivos do sistema operacional, mas também diretórios. Diferentemente, o CVS não é capaz de versionar diretórios.

O Subversion utiliza a representação interna do VOE, mas adiciona características do VOM. Nesta abordagem, inicialmente proposta por REICHENBERGER (1994), os ICs são identificados utilizando-se um número de versão global. Esta numeração é

baseada no elemento que é a raiz na configuração, permitindo identificar a quantidade de *Check-ins* realizados.

Com essa abordagem, quando um desenvolvedor visualiza as diversas versões de um conjunto (uma configuração, por exemplo) consegue perceber claramente que elementos (arquivos ou diretórios) foram modificados e/ou removidos após cada operação de *Check-in*. Diferentemente, no RCS/SCCS são necessárias outras consultas, como, por exemplo, a descrição textual para que a mesma informação seja obtida. Esta descrição é inserida pelo desenvolvedor no momento da realização do *Check-in*. Cabe dizer que o desenvolvedor pode optar por não adicionar esta descrição, o que prejudica o conhecimento sobre as razões e ICs relacionados às modificações.

2.3.5 Considerações finais sobre controle de versões para código-fonte

Em um evento de GCS recente, foi perguntado aos participantes quais os aspectos dos SCVs atuais eram considerados mais críticos. A opinião geral é que as ferramentas atuais são boas e estáveis, mas deficientes em eficiência, escalabidade e interoperabilidade, o que confere um relativo grau de aceitação em relação à maioria dos mecanismos propostos pelo RCS /SCCS.

No entanto, o modelo de dados baseado em sistema de arquivos herdado do RCS/SCCS impacta de forma negativa na avaliação dos sistemas atuais, porque um dos desafios da área de controle de versões é poder representar objetos complexos e seus relacionamentos (ESTUBLIER *et al.*, 2002).

Além do modelo de dados, outro ponto questionável, no RCS/SCCS, é o trabalho dispendido para a computação de deltas. Da forma como esses sistemas trabalham, o desempenho é um ponto crítico, na medida em que quanto mais variantes existirem no repositório, pior será o desempenho. Por outro lado, houve evoluções nas técnicas de compactação de arquivos e na capacidade de processamento.

Finalmente, a própria estrutura dos ICs é outro ponto discutível. RCS/SCCS trabalham com ICs e seus estados (VOE). Em contraposição, foi proposta a técnica de VOM. Embora VOE apresente problemas de desempenho na construção de configurações, é uma técnica madura e eficiente (ESTUBLIER *et al*, 2002). VOM, por outro lado, não garante a construção de configurações consistentes e, em alguns casos,

adiciona complexidade à utilização do SCV, na medida em que a lógica de primeira ordem, ou técnica similar, deve ser utilizada para manipular as configurações. É provável que abordagens híbridas forneçam resultados mais satisfatórios.

2.4 Controle de versões para hipertexto e linguagem de marcação

Nesta seção, os SCVs para hipertexto foram organizados em duas categorias: (i) versionamento de documentos disponíveis na *web*, (ii) versionamento de documentos descritos em XML (*eXtensible Markup Language*) (W3C, 2004).

2.4.1 Versionamento de documentos disponíveis na *web*

Um documento hipertexto constitui-se numa rede de documentos interligados através de *links* que, na maior parte das vezes, encontram-se fisicamente distribuídos (WHITEHEAD, 2000). Esses documentos ficam disponíveis através da *web* e são modificados com muita freqüência, normalmente por um grande número de pessoas fisicamente distribuídas (SOARES *et al.*, 2000). Por isso, eles necessitam de mecanismos de controle de versões.

No entanto, existem algumas diferenças entre o desenvolvimento de software e a gerência de páginas *web* (DART, 1999), que enfatizam a necessidade da adoção de técnicas e ferramentas específicas para controlar as versões de documentos hipertexto (ESTUBLIER *et al.*, 2002).

A necessidade de manutenção da integridade referencial, por exemplo, é uma conseqüência exclusiva da estrutura em forma de rede dos documentos hipertexto. Se considerarmos uma modificação na localização de um documento, todos os demais documentos que o referenciam devem ser ajustados para referenciar a nova localização. Ajustes de referências danificadas podem ser realizados através da intervenção humana, que recupera a nova referência e atualiza o *link* (VITALI, 1999).

Pode ainda ser aplicada a heurística da melhor aposta (*best bet*), onde o *link* é ajustado por encontrar o conteúdo mais similar ao conteúdo antigo (VITALI, 1999). As dificuldades de automatização da análise semântica do conteúdo são um obstáculo para esta abordagem.

O protocolo *HTTP Extensions for Distributed Authoring* (WebDAV, 1999) propõe extensões ao protocolo *Hypertext Transfer Protocol* (HTTP) (W3C, 1999) para fornecer funcionalidades de autoria e versionamento dos documentos disponíveis na *web*. O protocolo, no entanto, trata apenas questões de autoria, como, por exemplo, a definição do modelo pessimista baseado em bloqueio para gerenciar o acesso concorrente. Suprir as lacunas existentes e oferecer funcionalidades apropriadas de controle de versões é o objetivo do protocolo Delta-V (DELTA-V, 2002).

Portanto, Delta-V é uma extensão do protocolo WebDAV. Delta-V apresenta uma solução para o problema da integridade referencial. O protocolo fornece uma operação – VERSION – que informa ao servidor que o recurso sobre o qual a operação está sendo aplicada passará a ser versionado.

Neste momento, uma cópia é criada e associada uma *Uniform Resource Locator* (URL) *estável* ao recurso. A URL recebe a denominação de estável porque mesmo que a localização do recurso seja modificada, a URL não se modificará, funcionando como um identificador único para o recurso (HNETYNKA e PLÁSIL, 2004).

Os SCVs existentes propõem protocolos proprietários para a comunicação cliente/servidor (HUNT e REUTER, 2001). Tais sistemas sofrem os problemas inerentes às soluções proprietárias, que é a dificuldade de utilização da abordagem em cenários diferentes daqueles para os quais foram propostos.

A definição de um protocolo para controle de versões pode solucionar o problema da limitação da aplicabilidade, mas deve ser analisada com cautela. A simplicidade é um requisito essencial para que um protocolo seja aceito e adotado pela indústria e pela academia. Em razão disso, deve-se primeiramente analisar se a área que o protocolo pretende padronizar pode prescindir de funcionalidades complexas. Por exemplo, a junção é uma funcionalidade complexa e imprescindível em qualquer SCV moderno. No entanto, conforme dito anteriormente, o WebDAV pretende gerenciar o acesso concorrente através da técnica de bloqueio.

VersionWeb (SOARES *et al.*, 2000), por outro lado, permite que seus usuários modifiquem as páginas *web* em seus respectivos espaços de trabalho, e depois as devolvam para o repositório central. A ferramenta realiza a operação de junção se a mesma página for modificada simultaneamente em mais de um espaço de trabalho.

É uma abordagem mais condizente com a realidade atual, onde cronogramas restritos pressionam por soluções que favoreçam a produtividade. No entanto, a operação de junção é sensivelmente prejudicada porque o modelo de dados utilizado é baseado em sistema de arquivos. Para a realização desta operação, um bom conhecimento da estrutura do produto se faz necessário.

Contudo, a quantidade de formatos disponíveis, a evolução contínua desses formatos e a velocidade com que novos formatos surgem na *web* tornam a tarefa de dotar o protocolo WebDAV com este conhecimento bastante complexa. Em pouco tempo, a especificação do protocolo se tornaria obsoleta (WHITEHEAD *et al.*, 1999).

A adoção de um protocolo tem a vantagem de melhorar a interoperabilidade na comunicação cliente/servidor. Todavia, o custo dessa abordagem pode ser a impossibilidade de se utilizar funcionalidades avançadas dos SCVs.

2.4.2 Versionamento de documentos descritos em XML

O padrão *XLink* (*XML Linking Language*) (W3C, 2001) é utilizado para associar documentos descritos em XML. Sua principal vantagem é que pode ser armazenado externamente aos documentos que relaciona. Os documentos hipertexto são descritos, em sua grande maioria, utilizando HTML. DYRESON *et al.* (2004) argumentam que a tendência é que XML substitua HTML como linguagem para descrição de documentos hipertexto.

Diversas abordagens têm sido propostas para versionar documentos XML. A comunidade de pesquisa em banco de dados tem sido a grande responsável pelos trabalhos publicados. CHIEN *et al.* (2001) ressaltam que, nos SCVs, dois níveis distintos podem ser identificados: (1) nível lógico, e (2) nível físico.

As principais preocupações no nível lógico são inserir um IC para ser versionado, conseguir uma cópia do IC para realizar modificações e, novamente, devolver o IC para o repositório. As abordagens oscilam em torno de quais funcionalidades serão fornecidas e como estas funcionalidades serão implementadas.

No nível físico, por outro lado, o desempenho e a otimização do espaço de armazenamento fazem parte dos principais problemas a serem solucionados. Essas são

preocupações autênticas da comunidade de banco de dados, e, portanto, foram estes os aspectos explorados nos trabalhos que tratam do versionamento de documentos XML.

Essa mudança de foco - do nível lógico para o nível físico - é a principal diferença entre as abordagens da comunidade de GCS e os trabalhos da comunidade de banco de dados, sobre o versionamento de documentos XML.

2.4.3 Considerações finais sobre controle de versões para hipertexto e linguagem de marcação

As diferenças entre o desenvolvimento de software e a gerência de documentos hipertexto fazem com que as soluções adotadas para software não sejam facilmente utilizadas para documentos hipertexto, e vice-versa (ESTUBLIER *et al.*, 2002).

O protocolo WebDAV constitui-se numa abordagem moderna para o versionamento de documentos disponíveis na *web*. Considerando que WebDAV propõe extensões ao protocolo HTTP, será possível ter funcionalidades de controle de versões distribuídas através da *web*. No entanto, um protocolo deve ser genérico o suficiente de modo que seja aplicável ao maior número de ICs possível. A funcionalidade de junção exige conhecimento da estrutura de cada IC para que seja realizada de modo satisfatório. Certamente o protocolo não será capaz de contemplar todos os ICs existentes, e menos ainda outros que irão surgir.

2.5 Controle de versões para artefatos de análise e projeto

Esta seção descreve as abordagens de controle de versões para artefatos de análise e projeto. Considerando que a seção está diretamente relacionada com o tema desta dissertação e dado o número reduzido de trabalhos disponíveis na literatura, é apresentada a seguir uma descrição individual de cada trabalho encontrado na literatura. Esta forma de descrição fornece uma base melhor para análise e comparação.

2.5.1 Abordagem de OHST e KELTER

OHST e KELTER (2002) propõem uma abordagem para versionar artefatos de análise e projeto utilizando uma granularidade fina. A abordagem assume que os artefatos estão descritos em XML e as operações são aplicadas sobre a árvore sintática descrita nesse formato. Como uma das principais contribuições da abordagem, os autores

destacam a possibilidade de detectar modificações estruturais nos artefatos. Isso é possível porque os artefatos são identificados unicamente e a navegabilidade é bi-direcional.

Segundo os autores, este é um diferenciador da abordagem em relação àquelas que trabalham com os modelos em XML, versionados através de sistemas que utilizam o modelo de dados baseado em arquivos. Nestes sistemas, se um atributo é deslocado de uma classe para outra, o sistema irá detectar a remoção de um conjunto de linhas, seguida de uma adição de um conjunto de linhas. Com isso, não será capaz de perceber que o mesmo elemento, no caso um atributo, foi somente deslocado.

A utilização de um modelo de dados orientado a objetos possibilitou que os autores conseguissem versionar utilizando uma granularidade fina. Por outro lado, duas deficiências da abordagem referem-se à adoção de um meta-modelo proprietário para representar um modelo de classes UML e a impossibilidade de realização da operação de junção. Esta limitação pressupõe que o desenvolvimento será realizado de forma linear, inibindo a produtividade e criando dificuldades em projetos com cronogramas restritos.

2.5.2 *Adaptive/IBM*

ADAPTIVE/IBM Initial MOF 2.0 Versioning and Development Lifecycle Submission (OMG, 2005a) consiste numa proposta para tratar questões referentes ao versionamento, no contexto da especificação *Meta-Object Facility* (MOF) (OMG, 2005b) versão 2.0. MOF é uma especificação do *Object Management Group* (OMG) que define uma linguagem abstrata para descrever outras linguagens.

Embora o documento descreva apenas uma proposta, os autores argumentam que uma implementação já está disponível no mercado há aproximadamente dois anos. A proposta permite registrar as informações de versionamento de qualquer elemento que seja baseado no MOF, incluindo a UML. São previstas funcionalidades de junção e tratamento de conflitos, sendo que as informações resultantes dessas operações são armazenadas em repositórios.

O armazenamento de informações provenientes de conflitos é interessante porque permite gerar um banco de dados com as informações para análises posteriores. Por exemplo, podem-se identificar períodos, desenvolvedores envolvidos e sobre quais

artefatos ocorre um maior número de conflitos. Essas informações podem ser úteis na definição de políticas para acesso ao sistema de controle de versões.

Por outro lado, a abordagem propõe o versionamento somente do elemento que é raiz. Dessa forma, se for criada uma árvore que contenha um modelo, uma classe e um atributo, todos os elementos serão versionados como uma entidade única.

2.5.3 MIMIX

O MIMIX (EL-JAICK, 2004) é uma abordagem para suporte ao trabalho cooperativo assíncrono. A abordagem tem como objetivo principal propiciar a integração entre ferramentas *CASE*. Para atingir seu objetivo, é utilizado o *XML Metadata Interchange* (XMI) (OMG, 2005c) como formato de dados entre as ferramentas. Em linhas gerais, XMI define regras que permitem transformar modelos baseados no MOF em arquivos no formato XML. A especificação define: (i) regras de produção de Definição de Tipos de Documentos (DTD) para a geração de DTD XML; e (ii) regras de produção de documentos XML para a geração de metadados num formato compatível com XML.

O MIMIX se propõe a atuar em cenários onde as equipes se encontram geograficamente distribuídas. Para tal, foi utilizado *Web Services* na comunicação entre as ferramentas *CASE* e o MIMIX. *Web Services* compreendem um conjunto de padrões e protocolos que permitem a comunicação entre aplicações através de uma rede, geralmente a Internet (BARISH, 2002).

O MIMIX disponibiliza procedimentos automáticos e semi-automáticos para a resolução de conflitos, quando for necessário reintegrar os elementos em um único modelo. O problema da abordagem reside na manipulação dos elementos contidos no arquivo em XMI, como uma entidade única, o que a torna similar a uma abordagem de controle de versões baseada em sistema de arquivos.

2.5.4 Abordagem de LUCRÉDIO e PRADO

LUCRÉDIO e PRADO (2004) propõem extensões à ferramenta MVCASE (MVCASE, 2005) de modo a possibilitar armazenamento e busca remota de artefatos de software. A MVCASE auxilia o engenheiro de software nas atividades relativas à análise,

projeto, implementação e implantação de software orientado a objetos. A adoção da linguagem abstrata MOF nas camadas de armazenamento e busca possibilita a manipulação de artefatos de análise e projeto como, por exemplo, modelos UML.

No entanto, os autores representam os artefatos no formato XMI para persistência e utilizam o CVS como solução para controle de versões. Considerando que CVS utiliza o modelo de dados baseado em sistema de arquivos, todos os artefatos contidos no arquivo em XMI (modelos, pacotes, classes,..) são versionados como uma entidade única.

2.5.5 DVM

O *Distributed Versioning Model for MOF* (DVM) (HNETYNKA e PLÁSIL, 2004) é uma abordagem de versionamento capaz de atuar em cenários onde existam mais que um repositório para armazenar os ICs. Os autores apontam um problema em *ADAPTIVE/IBM Initial MOF 2.0 Versioning and Development Lifecycle Submission* (OMG, 2005a), que se refere à utilização do CVS, como repositório de ICs.

O problema está relacionado à identificação única dos ICs, quando se trabalha com diferentes repositórios. Utilizando o CVS, quando um artefato for copiado entre repositórios, um novo identificador será criado, acarretando a duplicação não controlada dos ICs (HNETYNKA e PLÁSIL, 2004).

Para solucionar este problema, os autores propõem um esquema onde o nome do repositório faça parte da identificação do artefato. Desta forma, quando o elemento for copiado de um repositório para outro, será possível identificar a origem do artefato, e assim obter sua história. A evolução dos artefatos é permitida em ambos os repositórios. No entanto, no repositório destino, o artefato evoluirá como um ramo. Nenhuma implementação, no entanto, foi apresentada pelos autores.

2.5.6 MOLHADO

MOLHADO (NGUYEN *et al.*, 2004) é capaz de versionar elementos da UML, entre outros artefatos. Os autores permitem o versionamento numa granularidade fina, no nível de classes, métodos e atributos. Para viabilizar essa abordagem, os autores utilizam um meta-modelo proprietário. Todos os ICs manipulados pelo ambiente devem ser descritos para esse meta-modelo.

Com essa abordagem, toda vez que se desejar modificar a estrutura de um artefato, um grande esforço deve ser necessário para realizar a operação. Por exemplo, atualizar o meta-modelo da UML da versão 1.4, para adicionar as características da UML 2.0.

2.5.7 Considerações finais sobre controle de versões para artefatos de análise e projeto

Vem aumentando continuamente o interesse em versionar artefatos de análise e projeto. Para tal, algumas abordagens utilizam um meta-modelo proprietário (TEIXEIRA *et al.*, 2001; SILVA *et al.*, 2003; NGUYEN *et al.*, 2005). Estas abordagens sofrem do problema da aplicabilidade limitada. Em cenários de desenvolvimento de software complexos, que normalmente comportam um grande número de desenvolvedores, é difícil restringir a equipe a utilizar somente um ADS ou ferramenta *CASE*.

OHST e KELTER (2002) versionam UML utilizando XML. Considerando que manipulam a árvore sintática de XML, os autores conseguem trabalhar numa granularidade fina. No entanto, foi necessário definir um meta-modelo da UML para transformar a metáfora de documentos XML (elementos, atributos e textos) em metáfora de elementos UML (modelo, pacotes, classes,...). Embora estejam utilizando XML, que é um padrão aceito na indústria e na academia, esta abordagem restringe o cenário de utilização da ferramenta, porque seus algoritmos, técnicas e estruturas estão atrelados ao meta-modelo definido pelos autores.

Outros autores utilizam o MOF como linguagem abstrata para descrição dos artefatos versionados (*ADAPTIVE/IBM*, 2005; LUCRÉDIO e PRADO, 2005; HNETYNKA e PLÁSIL, 2002). Estas abordagens atingem maior abrangência, se comparadas as soluções proprietárias, porque o MOF consiste numa especificação da OMG. No entanto, os autores não utilizam o potencial do modelo de dados orientado a objetos da linguagem porque versionam todos os elementos como uma entidade única.

2.6 Considerações finais

Este capítulo apresentou uma visão geral da área de controle de versões, procurando focar nos aspectos pertinentes à proposta a ser apresentada nessa dissertação.

As abordagens foram categorizadas de acordo com o artefato versionado, e descritas as vantagens e deficiências ao final de cada subseção.

A Tabela 1, a seguir, apresenta um quadro-resumo, enfatizando as características de cada abordagem, com base nos requisitos definidos no Capítulo 1.

Tabela 1: Quadro-resumo das abordagens descritas no capítulo.

Abordagens	Granularidade fina	Não-Intrusivo	Compatibilidade com ADSs/*CASE*	Flexível	Acesso concorrente	Distribuído
OHST/KELTER	Sim	Sim	Sim	Não	Não	Não
ADAPTIVE/ IBM	Não	Sim	Sim	Não	Sim	Sim
MIMIX	Não	Sim	Sim	Não	Sim	Sim
LUCRÉDIO/PRADO	Não	Não	Sim	Não	Sim	Sim
DVM	Não	Sim	Sim	Não	Sim	Sim
Molhado	Sim	Não	Não	Não	Não	Não

As abordagens descritas na Seção 2.3 (código-fonte) e na Seção 2.4 (hipertexto e linguagem de marcação) não foram incluídas na Tabela 1. Somente as abordagens descritas na Seção 2.5 foram incluídas, pois versionam artefatos de análise e projeto, tema de interesse desta dissertação.

O requisito *granularidade fina* permite avaliar se as abordagens levam em conta o conhecimento contido nos artefatos versionados. Portanto, abordagens que versionam todos os elementos contidos em um modelo, como uma entidade única, foram avaliadas negativamente nesse requisito.

Não-intrusivo permite avaliar se as abordagens modificaram a estrutura dos artefatos de análise e projeto para adicionar informações que são específicas a tarefa de versionamento. Por exemplo, abordagens que incluíram o número da versão ou qualquer outra informação no próprio artefato versionado são classificadas como intrusivas.

Compatibilidade com ADSs e ferramentas CASE verifica se os autores tomaram medidas que possibilitassem que sua abordagem mantivesse algum grau de interoperabilidade. SCVs atrelados a um ADSs ou ferramenta *CASE* específicos foram negativamente avaliados nesse requisito.

Flexibilidade permite verificar se foi concedido algum mecanismo de configuração, de modo que o gerente de configuração pudesse tornar o SCV mais próximo das características diferenciadoras de cada projeto.

Acesso concorrente verifica se o SCV permite que mais de um desenvolvedor possa acessar os ICs, seja através da política otimista ou da política pessimista.

Finalmente, *distribuição* possibilita verificar se o SCV pode ser utilizado num cenário onde as equipes se encontram geograficamente distribuídas.

Capítulo 3 - Odyssey-VCS: Controle de versões para elementos da UML

3.1 Introdução

As organizações estão utilizando artefatos de mais alto nível de abstração para desenvolver seus sistemas porque existem limites para a capacidade humana em lidar com a complexidade. Conforme dito anteriormente, a utilização de modelos de análise e projeto possibilita a divisão de um problema em partes menores para que possa ser mais facilmente tratado. Parte dessa mudança de foco (do código-fonte para modelos de análise e projeto) pode ser atribuída ao paradigma Orientado a Objetos (OO) que diminuiu a distância (*gap*) entre as fases de análise, projeto e implementação.

No paradigma OO, uma classe que é identificada e definida na análise, permanece na fase de projeto e se torna um artefato de implementação, através de uma linguagem de programação. Esta abordagem aproxima o código-fonte dos modelos de análise e projeto, tornando interessante para os desenvolvedores gerenciar a evolução, não somente do código-fonte, mas também dos artefatos de análise e projeto.

Este capítulo tem como objetivo apresentar uma abordagem de controle de versões para elementos da UML. A abordagem leva em conta o conhecimento sobre a estrutura dos elementos. Para alcançar esse objetivo, foram definidos seis requisitos no Capítulo 1: (i) granularidade fina; (ii) não-intrusão; (iii) compatibilidade com ADSs ou ferramentas *CASE* existentes; (iv) flexibilidade; (v) acesso concorrente; e (vi) distribuição.

Para facilitar a leitura e a compreensão da abordagem, a estrutura do capítulo foi definida com base em cada um desses requisitos. A Seção 3.2 trata o problema da granularidade fina. A Seção 3.3 descreve como é possível versionar os elementos da UML de forma não-intrusiva. A Seção 3.4 aborda a utilização de uma linguagem abstrata para a descrição de modelos, para atender ao requisito da compatibilidade com ADSs e ferramentas *CASE* existentes. A Seção 3.5 trata da flexibilidade. A Seção 3.6 descreve o tratamento dado para o acesso concorrente. Na Seção 3.7, é discutida a distribuição e como ela é tratada nessa abordagem. A Seção 3.8 fornece, através de um exemplo, uma

visão menos abstrata da abordagem. Finalmente, na Seção 3.9 são apresentadas as considerações finais deste capítulo e algumas limitações da abordagem.

3.2 Granularidade fina

Para desenvolver e manter os elementos da UML, os desenvolvedores dispõem de Ambientes de Desenvolvimento de Software (ADS) que utilizam o modelo de dados OO. No entanto, quando esses elementos são enviados para os SCVs, estes utilizam o modelo de dados baseado em sistema de arquivos. O resultado deste cenário está descrito na Figura 3.

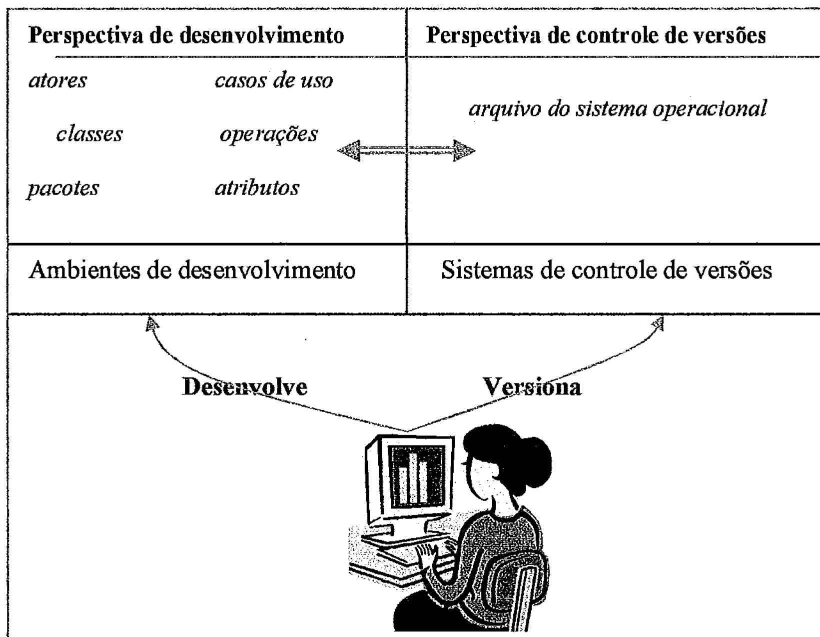


Figura 3: Perspectivas de desenvolvimento de software e de controle de versões

A Figura 3 mostra que quando a desenvolvedora está utilizando um ADS, ela trabalha com elementos de granularidade fina. Desta forma, é possível manipular atores, classes, casos de uso, pacotes, operações e atributos, como entidades próprias. No entanto, quando esses elementos são enviados para os SCVs atuais, os elementos deixam de ser entidades próprias e passam a fazer parte de uma entidade única, que é um arquivo do sistema operacional. Essa forma de tratamento faz surgir uma distância entre a perspectiva de desenvolvimento de software e a perspectiva de controle de versões.

Uma tarefa comum quando se trabalha com SCV é tomar conhecimento sobre os elementos que foram modificados a partir de uma determinada versão. Isso ocorre

quando um desenvolvedor passa algum tempo sem manipular os elementos que, no entanto, continuaram sendo modificados por outros desenvolvedores. Se o desenvolvedor desejar saber quais as classes que foram modificadas, isso não será possível, porque o SCV não reconhece os elementos que são classes, conforme mostra a Figura 4.

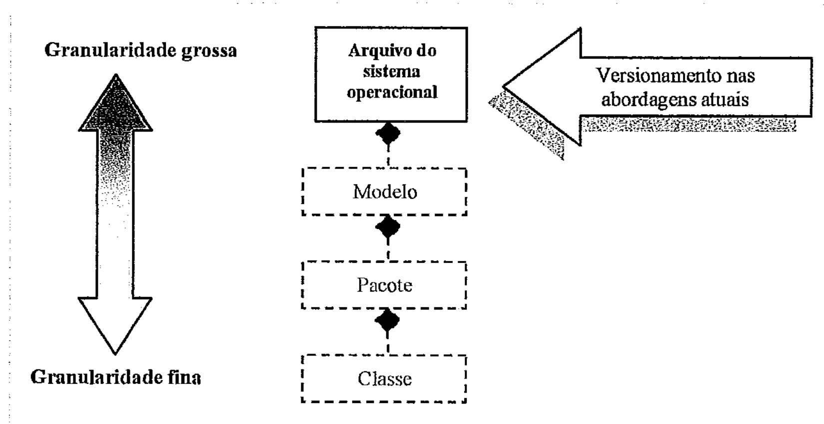


Figura 4: Versionamento da UML nas abordagens atuais (adaptado de MURTA, 2004)

No contexto da abordagem descrita nesta dissertação, é utilizado o modelo de dados OO para permitir versionar elementos da UML, numa granularidade fina. A utilização desse modelo de dados, em SCVs, permite o registro da história de evolução de elementos da UML, individualmente.

A título de exemplificação, suponhamos um sistema no domínio de hotelaria que contenha as classes "Hospede" e "Reserva", descritas em UML. A classe "Hospede" possui ainda um atributo, "nome", e uma operação, "getNome". Em nossa abordagem, que trabalha numa granularidade fina, as seguintes consultas devem ser respondidas:

- ✓ Visualizar todas as versões da classe "Hospede";
- ✓ Quem modificou a classe "Reserva" versão 1 e gerou a versão 2?
- ✓ O que mudou da versão 2 para a versão 3, na operação "getNome"?
- ✓ Quem realizou a última modificação no tipo do atributo "nome"?

Essas consultas não são possíveis de serem realizadas em SCVs que utilizam um modelo de dados baseado em sistema de arquivos.

Trabalhando com uma granularidade fina, tanto gerentes de projeto quanto desenvolvedores são beneficiados. Da perspectiva do gerente, os benefícios ficam por conta da maior facilidade na condução dos projetos de desenvolvimento. Ele pode, por exemplo, realizar a divisão do trabalho com base nos casos de uso definidos na fase de análise.

Dessa forma, cada equipe fica encarregada de trabalhar sobre um ou mais casos de uso, além das classes que realizam os serviços definidos em cada caso de uso. Durante o andamento do projeto, o gerente tem melhores condições de acompanhar a evolução das atividades de cada equipe individualmente. Isso é possível porque a abordagem registrará a história de evolução de cada elemento. Da perspectiva do desenvolvedor, o tratamento num grão mais fino também traz benefícios quando se utiliza o modelo otimista, principalmente na realização da junção, conforme será apresentado na Seção 3.8.

O tratamento individual, para cada elemento versionado, entretanto, introduz a necessidade de conhecimentos sobre a estrutura dos elementos. Por exemplo, um pacote possui características diferentes de uma classe, e ambos apresentam diferenças em relação a uma operação.

Ou seja, um pacote pode conter classes e outros pacotes, enquanto que uma classe pode conter atributos e operações. No entanto, uma operação não pode conter um pacote. Esse conhecimento sobre a estrutura dos elementos é importante porque o SCV precisa navegar sobre ela, para que a tarefa de versionamento possa ser realizada. No contexto desta dissertação, essa questão é contornada através da definição da interface "Tratador", conforme descrito na Figura 5.

Desta forma, todos os elementos da UML devem possuir classes tratadoras específicas, ou seja, classes que implementam a interface "Tratador". Para um modelo, existe a classe "TratadorModelo", para um pacote, existe a classe "TratadorPacote", e assim por diante. Essa solução melhora a qualidade da tarefa de versionamento, na medida em que as características e a estrutura de cada elemento são levadas em conta.

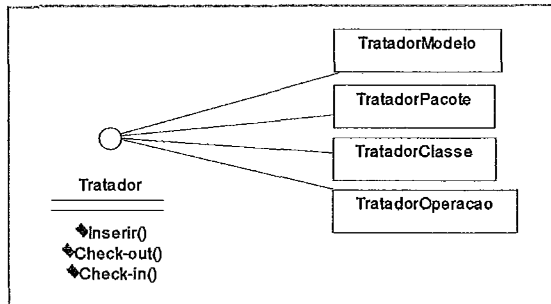


Figura 5: A interface "Tratador" e algumas classes que a implementam

3.3 Não-intrusão

Os SCVs introduzem novos elementos no cotidiano dos desenvolvedores. Termos como: item de configuração, revisão, versão, ramo e configuração passam a fazer parte do vocabulário das equipes de desenvolvimento de software. Ou seja, os SCVs possuem seus próprios dados, além dos elementos que pretendem versionar. No contexto desta dissertação, esses dados fazem parte do meta-modelo de versionamento proposto, apresentado na Figura 6.

Conforme se observa, um projeto pode ter vários ICs. Além disso, cada IC pode possuir diversas versões ao longo de sua história. "Elemento" representa a classe que é a raiz no meta-modelo da UML, permitindo que, através do polimorfismo, qualquer elemento da UML possa ter suas informações de versionamento registradas.

A operação de modificação que resulta na criação/modificação de uma nova instância de versão é registrada na classe "Transação". As operações podem ser: "Inserir", "*Check-in*", "*Check-out*" ou "Listar".

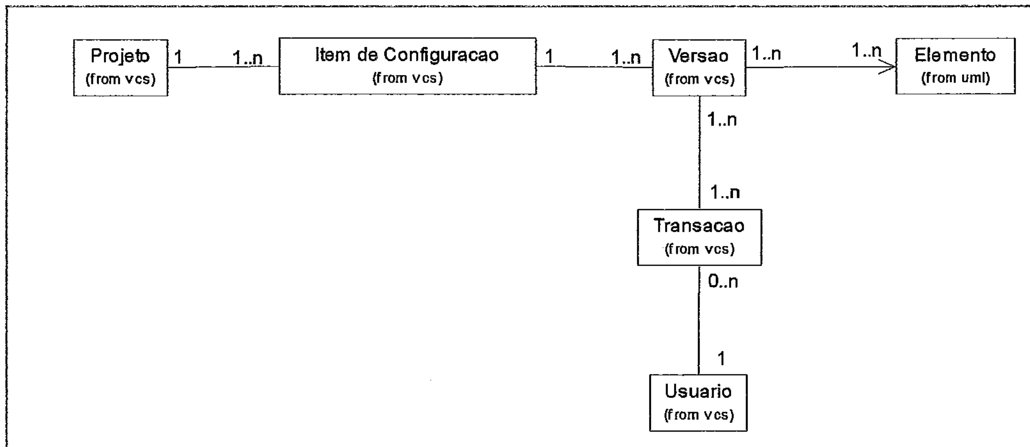


Figura 6: Meta-modelo de versionamento

A operação "Inserir" possibilita que o cliente envie um elemento para ser inserido no repositório. A operação "*Check-out*" permite enviar uma cópia do elemento do repositório para o cliente, e a operação "*Check-in*" é utilizada para devolver o elemento do espaço de trabalho do desenvolvedor para o repositório. Finalmente, a operação "Listar" possibilita visualizar todos os elementos constantes no repositório que são ICs. O meta-modelo prevê, ainda, o registro do usuário que realizou a transação sobre uma dada instância de versão.

A Figura 7 apresenta a ligação entre o meta-modelo de versionamento (Figura 6) e o meta-modelo parcial da UML. Observando a figura, pode-se notar que a associação entre "Versao" e "Elemento" possui uma navegabilidade unidirecional. Ou seja, considerando que temos como requisito não modificar a estrutura dos elementos versionados (abordagem não-intrusiva), não é indicada a inserção de informações de versionamento no meta-modelo da UML.

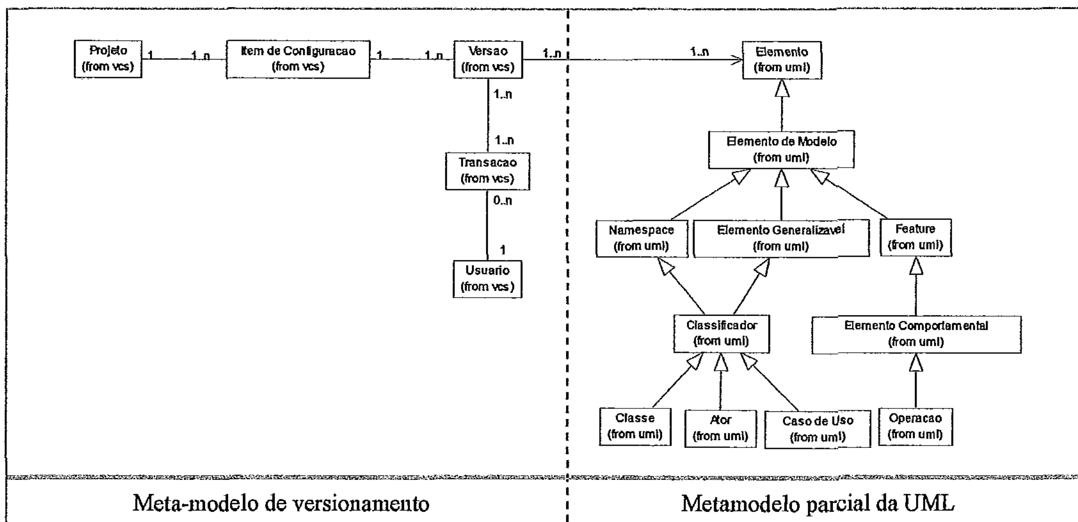


Figura 7: Meta-modelo de versionamento e o meta-modelo parcial da UML

3.4 Compatibilidade com ADSs e ferramentas *CASE* existentes

Conforme dito anteriormente, esta abordagem permite versionar artefatos de análise e projeto, descritos em UML. No entanto, considerando o número de pessoas envolvidas em sistemas grandes e complexos, que é o foco principal da GCS, é possível que, num mesmo projeto, mais de um ADS ou ferramenta *CASE* seja utilizado.

Diversos fatores podem influenciar na escolha de um ADS ou ferramenta *CASE*, como a disponibilidade orçamentária, que impede a aquisição de um ADS ou ferramenta *CASE* comercial, ou simplesmente a opção do desenvolvedor por alguma funcionalidade disponível.

Este cenário nos conduz a propor uma abordagem que não fique restrita a um único ADS ou ferramenta *CASE*. Utilizar uma linguagem proprietária, portanto, não seria uma solução aceitável. A adoção de uma linguagem não-proprietária traz maiores benefícios porque o esforço na realização de correções ou atualizações não fica concentrado em uma única organização.

Para atender a esse requisito, no contexto desta dissertação, optou-se pela utilização do MOF. MOF não é usado para descrever uma gramática para uma linguagem, ele é usado para descrever a estrutura de objetos que podem ser representados em uma linguagem (MATULA, 2005). O OMG, normalmente, se refere ao MOF como uma arquitetura em quatro camadas, conforme mostra a Figura 8.

No quarto nível (M3), encontra-se o MOF. No terceiro nível (M2), estão as linguagens descritas a partir do MOF. A UML é uma dessas linguagens descritas a partir do MOF. Outras linguagens são *Common Warehouse Metamodel* (CWM) (OMG, 2005d), Java (NETBEANS, 2005), entre outros. O CWM, embora tenha sido originalmente definido para atuar na área de banco de dados, inclui também um metamodelo de XML baseado no MOF. No nível M1, encontram-se os modelos e, no nível M0, as instâncias desses modelos.

Nível	Descrição
M3	MOF
M2	metamodelo UML (por exemplo, o elemento "Classe")
M1	elementos UML (por exemplo, a classe "Hospede")
M0	instâncias de elementos UML (por exemplo, o hospede "João")

Figura 8: Arquitetura em quatro camadas do MOF (adaptado de MATULA, 2005)

Uma vez que o MOF foi selecionado como elemento central nesta abordagem, o XMI representa a escolha natural como formato de representação dos dados, de modo a tornar esta abordagem compatível com ADSs e ferramentas *CASE* existentes. Contudo, a utilização de XMI não garante a ausência de problemas na comunicação com os ADSs e as ferramentas *CASE*. Um dos obstáculos é a falta de compatibilidade entre diferentes versões da especificação. Por exemplo, uma ferramenta que utiliza a especificação XMI versão 1.0 não será compatível com a especificação XMI versão 1.1.

No entanto, esse problema pode ser resolvido com o uso de abordagens existentes na literatura para integração de ferramentas. Por exemplo, SPINOLA *et al.* (2004) propõem uma abordagem que permite a realização de um mapeamento entre dois documentos descritos em XML. A partir desse mapeamento, um conversor, que realiza a transformação entre esses documentos, é automaticamente gerado.

3.5 Flexibilidade

Esta seção descreve os mecanismos propostos no sentido de fornecer maior flexibilidade à equipe de desenvolvimento. O objetivo é permitir que a abordagem possa se moldar, tanto quanto possível, às características de cada projeto de desenvolvimento de software. Dois tipos de configuração são possíveis: (1) configuração do Grão de Versionamento (GV); e (2) configuração do Grão de Comparação (GC).

No contexto dessa abordagem, um GV corresponde a um elemento definido para ser versionado (MURTA, 2004). Esse elemento terá sua história de evolução registrada, através do meta-modelo de versionamento. Por outro lado, um GC é o menor elemento que tem coesão e semântica próprias e é usado para identificar e notificar conflitos (MURTA, 2004).

O nível do meta-modelo da UML (M2), mostrado na Figura 8, será utilizado para a definição dos grãos (GV e GC). No nível dos modelos (M1), situam-se os elementos que serão versionados. Ou seja, as configurações no nível do meta-modelo (M2) irão customizar o tratamento a ser dispensado para cada elemento versionado (M1). A configuração dos GVs é apresentada na Seção 3.5.1 e a configuração dos GCs é descrita na Seção 3.5.2.

3.5.1 Flexibilidade na definição do grão de versionamento

Conforme discutido na Seção 3.2, deve ser possível controlar a evolução dos elementos individualmente. Além de permitir versionar os elementos num grão mais fino, nossa abordagem deve permitir que o gerente de configuração defina quais os tipos de elementos devem ser versionados. Desta forma, um gerente de configuração pode definir que, em um dado projeto, classes, pacotes e casos de uso sejam GVs. Para outro projeto, no entanto, versionar operações e atributos pode ser tão importante quanto versionar classes, pacotes e casos de uso.

Os requisitos próprios de auditoria também são motivadores para se versionar ou não atributos e operações. A auditoria de configuração é uma importante função da GCS e consiste no meio pelo qual as organizações podem assegurar que o desenvolvimento do software foi realizado de forma correta, em conformidade com as diretrizes e padrões de

desenvolvimento (LEON, 2000). As regras definidas em contrato entre as partes envolvidas também podem ser asseguradas através de auditoria.

As auditorias, no contexto de GCS, variam em formalidade, mas todas têm a mesma função: garantir a completude e corretude do software que está sendo entregue ao usuário. Qualquer anomalia encontrada durante uma auditoria não somente deve ser corrigida, mas a causa deve ser identificada para assegurar que o problema não irá ocorrer novamente (LEON, 2000).

Portanto, encontrar o momento exato, na história de evolução, em que um atributo ou uma operação (ou qualquer outro elemento) deixaram de atender aos requisitos definidos na fase da análise, contribui para identificar as causas e apresentar uma solução definitiva para o problema.

Além disso, para cada elemento versionado, todo um conjunto de instâncias do meta-modelo de versionamento deve ser criado e registrado. Para um gerente que não deseja versionar, por exemplo, o elemento ator, a abordagem deve garantir que informações de versionamento não sejam geradas para esse elemento. Caso contrário, tais informações não terão a menor utilidade para esses gerentes.

Outro fator a ser considerado é que, a longo prazo, o armazenamento desse volume elevado de informações teria como conseqüência a degradação do desempenho do sistema. Quando um elemento é devolvido para o repositório, através da operação de "*Check-in*", uma quantidade considerável de processamento é necessária para identificar e registrar as modificações. É razoável providenciar que esse processamento seja realizado somente sobre os elementos cujas informações de evolução sejam de interesse dos gerentes e desenvolvedores.

3.5.2 Flexibilidade na identificação e notificação de conflitos

Na Seção 3.2, discutimos os problemas de se utilizar o modelo de dados baseado em sistema de arquivos para versionar elementos da UML. No entanto, os problemas com essa abordagem tornam-se ainda mais críticos quando dois ou mais desenvolvedores acessam o mesmo elemento concorrentemente. Vale dizer que esta proposta aborda a política otimista apenas, apesar da complexidade gerada na realização da junção,

considerando suas vantagens em relação à política pessimista, no sentido de favorecer a produtividade.

Conforme descrito anteriormente, os sistemas atuais não possuem conhecimento sobre a estrutura dos elementos porque trabalham somente com arquivos texto. Desta forma, eles consideram uma linha como GC. Um conflito ocorre quando dois ou mais desenvolvedores modificam o mesmo GC concorrentemente.

Em arquivos texto, um parágrafo se comporta como uma única linha, porque, em seu interior, não existe o caracter *linefeed*. Este caracter é utilizado para determinar uma mudança de linha. Conseqüentemente, um parágrafo é definido como GC, conforme se observa na Figura 9. Entretanto, utilizar esse mecanismo para o tratamento de conflitos em elementos da UML não é uma abordagem apropriada.

Se a divisão do trabalho for realizada com base nos casos de uso, bem como no conjunto de classes que os realizam, é possível que uma mesma classe esteja presente na realização de mais de um caso de uso. Seria interessante, ou mesmo necessário, que o sistema emitisse uma notificação toda vez que dois desenvolvedores modificassem essa classe concorrentemente. Neste cenário, uma classe seria um GC aceitável para modelos UML.

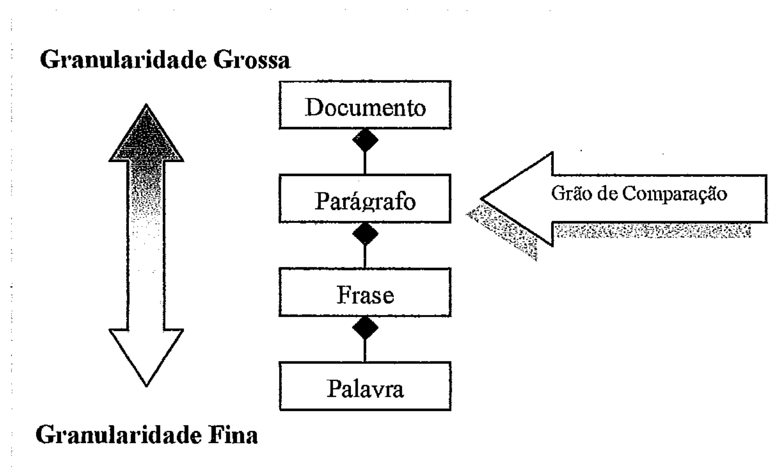


Figura 9: Grão de comparação aplicado a arquivos texto (MURTA, 2004)

Contudo, quando utilizamos os SCVs atuais para versionar elementos da UML, descritos no formato XML, por exemplo, uma classe não pode ser definida como GC. Nesse formato, uma classe é descrita por muitas linhas. Se dois desenvolvedores

modificarem linhas diferentes, mesmo que estas linhas descrevam uma mesma classe, não será detectado um conflito, podendo até levar a um estado inconsistente.

Isso ocorre porque esses sistemas utilizam uma linha como GC, a qual não tem nenhum significado estrutural para uma classe da UML. Uma classe possui propriedades como nome, visibilidade, além de possuir atributos e operações. São esses elementos que devem ser levados em consideração para a identificação ou não de um conflito em uma classe.

Por exemplo, se dois desenvolvedores modificam o nome da classe concorrentemente, então uma notificação de conflito deve ser disparada. Da mesma forma, se um desenvolvedor modifica um atributo e outro desenvolvedor modifica uma operação na mesma classe, também teremos uma notificação de conflito, caso a classe seja GC.

Nossa abordagem conduz a três tipos distintos de conflitos: (i) conflito baseado em propriedades primitivas; (ii) conflito baseado na estrutura; e (iii) conflito misto.

Conflito baseado em propriedades primitivas é quando a sobreposição de tarefas ocorre sobre as propriedades primitivas da classe. Se dois desenvolvedores modificam o nome da classe; ou um desenvolvedor modifica o nome e outro modifica a visibilidade, tem-se um conflito baseado nas propriedades primitivas da classe.

Conflito baseado na estrutura é quando a sobreposição ocorre nos elementos internos da classe, como atributos e operações, por exemplo. Se dois desenvolvedores modificam o mesmo atributo; ou se um desenvolvedor modifica um atributo e outro desenvolvedor modifica uma operação, na mesma classe, surge um conflito baseado na estrutura da classe.

Conflito misto é quando ocorre a modificação de uma propriedade primitiva combinada com uma remoção da classe. Por exemplo, um desenvolvedor modifica o nome de uma classe, enquanto outro desenvolvedor remove a mesma classe. Nesse caso, será detectado um conflito misto, no contexto da classe.

É possível ainda que ocorra uma combinação de conflitos. Por exemplo, um desenvolvedor remove uma classe e, conseqüentemente, todos os elementos nela contidos. Outro desenvolvedor modifica o nome de uma operação contida na mesma classe. No contexto da operação, ocorre um conflito misto porque houve modificação e

remoção sobre o mesmo elemento. Por outro lado, ocorre um conflito baseado em estrutura no contexto da classe, porque dois desenvolvedores editaram/removeram elementos contidos em sua estrutura, no caso uma operação.

Nos parágrafos anteriores, uma classe foi utilizada como GC para exemplificação. No entanto, nada impede que um atributo, uma operação ou um pacote também pudessem ser definidos como GC. Além disso, mais de um elemento podem ser definidos como GC. No caso descrito acima, tanto uma classe quanto uma operação poderiam ser definidas como GC.

Isso conduz a um comportamento recursivo na identificação e notificação de conflitos. Por exemplo, um desenvolvedor modifica a visibilidade de uma operação enquanto outro desenvolvedor modifica o tipo de retorno da mesma operação. Um conflito será detectado na operação, e, por recursividade, um conflito será detectado também no âmbito da classe, porque a operação encontra-se no escopo da classe.

3.6 Acesso concorrente

Projetos constituídos por apenas um desenvolvedor são cada vez mais difíceis de serem encontrados nos dias atuais. Por outro lado, projetos com equipes médias (entre 10 e 20 desenvolvedores) e grandes (mais que 20 desenvolvedores) estão mais condizentes com a realidade (TEIXEIRA *et al.*, 2000). Como esta proposta adota a política otimista para gerenciar o acesso concorrente, uma atenção especial deve ser dada à realização da junção.

Para um melhor entendimento das situações de acesso concorrente, a Figura 10 apresenta um cenário de realização de junção. A figura mostra que um desenvolvedor realiza "*Check-out*" sobre uma configuração que se encontra no repositório, a "Configuração **original**". Essa denominação é utilizada porque a configuração serve como base para o "*Check-out*". Este procedimento faz com que uma cópia da configuração seja enviada para seu espaço de trabalho. Esta cópia é denominada "Configuração do **usuário**".

No entanto, enquanto o desenvolvedor realiza suas modificações, outros desenvolvedores também solicitam "*Check-out*", transferindo cópias para seus respectivos espaços de trabalho. Estes desenvolvedores aplicam modificações às suas

configurações e as devolvem para o repositório, através de uma operação de "*Check-in*", fazendo surgir a "Configuração **atual**". Esta denominação é utilizada porque ela é a última configuração que se encontra na linha principal de desenvolvimento.

No entanto, um problema surge quando o desenvolvedor, que detém a "Configuração do **usuário**", devolve sua cópia para o repositório. O "*Ckeck-in*" não pode ser realizado sobrepondo a configuração mais atual no repositório, sob pena de se perder o trabalho realizado pelos demais desenvolvedores.

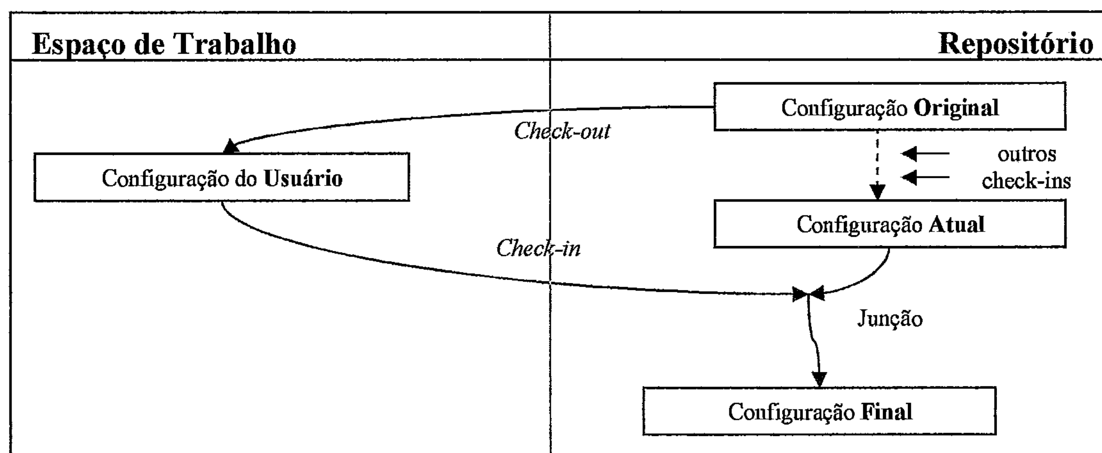


Figura 10: Cenário de modificação envolvendo diferentes configurações.

O procedimento de junção deve ser realizado neste caso, o que determina o surgimento da "Configuração **final**". As situações de conflito devem ser tratadas no momento da realização da junção. É importante dizer, entretanto, que nem todas as operações de junção resultam em conflitos. A Tabela 2 descreve todas as possíveis situações de junção e indica o procedimento a ser adotado, em cada uma. As linhas não consistentes ou redundantes foram eliminadas. Os seguintes conjuntos foram definidos, com o intuito de descrever as possíveis configurações relacionadas com o processo de junção:

O: Conjunto que representa a "Configuração **original**" do repositório, sobre a qual o "*Check-out*" foi efetuado.

A: Conjunto que representa a "Configuração **atual**" do repositório.

U: Conjunto que representa a "Configuração do **usuário**".

F: Conjunto que representa a "Configuração **final**", após o processo de junção.

As configurações Original, Atual, do Usuário e Final foram apresentadas na Figura 10. Além disso, as seguintes funções foram definidas para possibilitar as operações entre os conjuntos:

e_C: Fornece o elemento "e" pertencente à configuração "C"

$e_{C1} \equiv e_{C2}$: Verifica se o elemento "e", pertencente à configuração "C1", possui os mesmos valores para suas propriedades primitivas, se comparado ao elemento "e", pertencente à configuração "C2". Por exemplo, para uma classe, seriam comparados os valores das propriedades nome, visibilidade, entre outros.

Tabela 2: Procedimentos de junção em função do cenário de modificação.

	Verifica a existência do elemento			Compara		Define a ação
Situação	**$e \in O$**	**$e \in A$**	**$e \in U$**	$e_O \equiv e_A$	$e_O \equiv e_U$	**Procedimento**
1	V	V	V	V	V	Adicionar e_A (ou e_u) em F
2	V	V	V	V	F	Adicionar e_U em F
3	V	V	V	F	V	Adicionar e_A em F
4	V	V	V	F	F	Notificar conflito baseado em propriedades primitivas*
5	V	V	F	V	N/A**	Não adicionar "e" em F
6	V	V	F	F	N/A**	Notificar conflito misto
7	V	F	V	N/A**	V	Não adicionar "e" em F
8	V	F	V	N/A**	F	Notificar conflito misto
9	V	F	F	N/A**	N/A**	Não adicionar "e" em F
10	F	V	V	N/A**	N/A**	N/A***
11	F	V	F	N/A**	N/A**	Adicionar e_A em F
12	F	F	V	N/A**	N/A**	Adicionar e_U em F
13	F	F	F	N/A**	N/A**	N/A****

* É possível que os desenvolvedores tenham realizado as mesmas modificações no elemento. Este cenário será melhor descrito na análise da situação a seguir.
** Se o elemento não existe, não é possível definir valores para expressões que envolvam o elemento.
*** Não é possível que um elemento inexistente no repositório seja criado em paralelo em espaços de trabalho distintos.
**** Nenhum procedimento é necessário, caso o elemento não exista em nenhuma configuração.

Os procedimentos de junção descritos nas situações 4, 6 e 8 resultam na notificação de conflitos. Na situação 4, o conflito é baseado em propriedades primitivas, enquanto que nas situações 6 e 8, ocorre conflito misto. É possível notar que nenhuma das situações descritas na Tabela 2 conduziu à notificação de conflito baseado em estrutura. Isto ocorre porque este conflito é avaliado com base nas modificações aplicadas aos sub-elementos do elemento sob análise. Por outro lado, cada situação descrita na

Tabela 2 avalia individualmente o elemento modificado, e nenhuma ação leva em conta os sub-elementos. O conflito baseado em estrutura será melhor descrito na Seção 3.8.

Os valores das colunas 2 a 6 determinam um cenário que será tratado conforme procedimento descrito na coluna 7. A partir da análise dos valores, três categorias de procedimentos são possíveis: (1) o elemento aparecerá na "Configuração **final**"; (2) o elemento não aparecerá na "Configuração **final**"; ou (3) será notificado um conflito.

Na análise das Situações 1 a 4, o elemento aparece nas três configurações, porque as 2ª, 3ª e 4ª colunas aparecem com valor verdadeiro (V). Portanto, as considerações para essas situações são realizadas apenas na comparação das propriedades dos elementos (5ª e 6ª colunas) e no procedimento adotado (7ª coluna).

Na Situação 1, o elemento não foi modificado na "Configuração **atual**" ($e_O \equiv e_A$,com valor V) nem na "Configuração do **usuário**" ($e_O \equiv e_u$,com valor V). Nesse caso, o elemento pertencente a qualquer uma das duas configurações pode ser adicionado à "Configuração **final**".

Na Situação 2, o elemento não foi modificado na "Configuração **atual**" ($e_O \equiv e_A$,com valor V). No entanto, foi modificado na "Configuração do **usuário**" ($e_O \equiv e_u$,com valor F). Portanto, o elemento pertencente à "Configuração do **usuário**" será adicionado à "Configuração **final**".

Na Situação 3, o elemento foi modificado na "Configuração **atual**" ($e_O \equiv e_A$,com valor F), mas não foi modificado na "Configuração do **usuário**" ($e_O \equiv e_u$,com valor V). O elemento pertencente à "Configuração **atual**" deve ser adicionado à "Configuração **final**".

Na Situação 4, o elemento foi modificado nas duas configurações, na "Configuração **atual**" ($e_O \equiv e_A$,com valor F) e na "Configuração do **usuário**" ($e_O \equiv e_u$,com valor F). Duas situações podem ocorrer nesse caso:

a. O elemento pertencente à "Configuração do **usuário**" possui valores diferentes do elemento pertencente à "Configuração **atual**". O procedimento a ser adotado consiste na notificação de um conflito; ou
b. As mesmas modificações foram realizadas aos elementos pertencentes às duas configurações. Portanto, os elementos são iguais, e não faz diferença em adicionar um ou outro à "Configuração **final**".

A partir da Situação 5, os valores das 2ª, 3ª e 4ª colunas são também levados em consideração, porque algumas linhas aparecem com valor falso (F), o que significa que descrevem situações nas quais o elemento não pertence a uma ou mais configurações.

Na Situação 5, o elemento não foi modificado na "Configuração **atual**" ($e_O \equiv e_A$,com valor V) e foi removido da "Configuração do **usuário**" (**e** $\in$ **U,** com valor F). O elemento não aparecerá na "Configuração **final**" porque um desenvolvedor removeu o elemento, enquanto que para o outro desenvolvedor o elemento é indiferente, já que ele não realizou qualquer modificação.

Na Situação 6, o elemento foi removido da "Configuração do **usuário**" (**e** $\in$ **U,** com valor F) e modificado na "Configuração **atual**" ($e_O \equiv e_A$, com valor F). O procedimento consistirá na notificação de um conflito, porque ocorreu modificação e remoção sobre um mesmo elemento.

Na Situação 7, o elemento não foi modificado na "Configuração do **usuário**" ($e_O \equiv e_u$,com valor V) e foi removido na "Configuração **atual**" (**e** $\in$ **A,** com valor F). Portanto, não será adicionado à "Configuração **final**".

Na Situação 8, o elemento foi removido da "Configuração **atual**" (**e** $\in$ **A,** com valor F) e modificado na "Configuração do **usuário**" ($e_O \equiv e_u$, com valor F). Será notificado um conflito, porque ocorreu modificação e remoção sobre um mesmo elemento.

Na Situação 9, o elemento foi removido nas duas configurações, na "Configuração **atual**" (**e** $\in$ **A,** com valor F) e na "Configuração do **usuário**" (**e** $\in$ **U,** com valor F). Ele não aparecerá na "Configuração **final**".

Na Situação 10, apareceu pela primeira vez um valor falso (F) para a 2ª coluna. Isto significa que o elemento não pertence à "Configuração **original**" (**e** $\in$ **O,** com valor F) e não constava no repositório, quando os desenvolvedores realizaram "*Check-out*". No entanto, as colunas seguintes demonstram que o elemento pertence às duas configurações, à "Configuração do **usuário**" (**e** $\in$ **U,** com valor V) e à "Configuração **atual**" (**e** $\in$ **A,** com valor V). Esta situação não é possível de ocorrer porque um elemento não pode ser criado em paralelo em espaços de trabalho distintos.

É importante ressaltar que é possível que dois desenvolvedores criem uma classe (ou qualquer outro elemento) com o mesmo nome em seus respectivos espaços de trabalho, e depois as enviem para o respositório. As classes, no entanto, serão tratadas

como dois elementos distintos, na medida em que o nome não é utilizado como identificador, no contexto desta abordagem.

Na Situação 11, o elemento foi adicionado à "Configuração **atual**" (**e** ∈ **A,** com valor V). Portanto, será também adicionado à "Configuração **final**".

Na Situação 12, o elemento foi adicionado à "Configuração do **usuário**" (**e** ∈ **U,** com valor V). Portanto, será também adicionado à "Configuração **final**".

Na Situação 13, o elemento não existe em nenhuma das configurações e nenhum procedimento cabe nesse caso.

3.7 Distribuição

Um segmento significativo de empresas de desenvolvimento de software está adotando um modelo de organização de projeto que envolve o uso de múltiplos desenvolvedores em múltiplos lugares trabalhando em um único projeto (HOEK *et al.*, 1996). Considerando que os SCVs são os responsáveis pela manutenção dos repositórios de ICs e disponibilização destes nos espaços de trabalhos dos desenvolvedores, a distribuição geográfica da equipe deve ser devidamente tratada.

Existem duas formas distintas de atender ao requisito da distribuição no contexto dos SCVs: (i) os repositórios podem ser fisicamente distribuídos; ou (ii) pode ser utilizada uma arquitetura cliente/servidor, que possibilita o acesso remoto a um único repositório.

A manutenção de *repositórios distribuídos* significa que o SCV deverá ter consciência de que um mesmo IC pode estar replicado em mais de um repositório e que, em algum momento do ciclo de desenvolvimento, estas cópias poderão ser reintegradas a um único IC. É possível, no entanto, que elas nunca sejam reintegradas e coexistam em diferentes repositórios.

Um problema surge quando se deseja levantar a história de um IC replicado em diferentes repositórios. Da perspectiva do desenvolvedor, é irrelevante em que repositório o IC foi criado e modificado. Todas as modificações relacionadas ao IC, bem como os desenvolvedores responsáveis pelas modificações, devem ser disponibilizadas.

Uma solução possível, nesse caso, é tratar o IC replicado como uma variante do IC original (HNETYNKA e PLÁSIL, 2002). Essa solução permite que o IC replicado

evolua naturalmente, como se estivesse em seu repositório de origem. No entanto, em algum momento do ciclo de desenvolvimento deve ser possível reintegrar os ICs (original e replicado) a um só IC. Os problemas relacionados à manutenção de repositórios distribuídos são bem conhecidos pela comunidade de banco de dados, sendo que a complexidade desta arquitetura, bem como seu custo elevado, são fatores que impactam sua efetiva adoção (NAVATHE, 2000).

A manutenção de uma arquitetura cliente/servidor com acesso remoto a um único repositório, por outro lado, é uma solução mais simples, se comparada à arquitetura de repositórios distribuídos. No contexto desta dissertação, o requisito de distribuição foi solucionado através da arquitetura cliente/servidor, devido ao menor grau de complexidade que esta apresenta. A complexidade é uma característica inerente à tarefa de desenvolvimento dos SCVs, portanto, qualquer medida que vise reduzir essa complexidade deve ser considerada.

A comunicação cliente/servidor é realizada com a utilização de *Web Service.* A possibilidade de publicação e consumo de serviços entre aplicações desenvolvidas para plataformas diferentes é a principal característica dos *Web Service.* Outro fator determinante na escolha de *Web Service* é a possibilidade de utilização da *web* como meio de comunicação, já que a distribuição através da *web* é uma importante característica dos SCVs modernos (ESTUBLIER, 2000).

3.8 Exemplo

O objetivo desta seção é apresentar, através de um exemplo, uma explicação menos abstrata de alguns dos requisitos tratados ao longo do texto. Granularidade fina, flexibilidade e acesso concorrente serão tratados de uma maneira mais detalhada. A Figura 11 apresenta os elementos da UML que serão utilizados neste exemplo, que são a classe "Hospede" e o atributo "telefone". Para esse exemplo, deve-se considerar classe como GC.

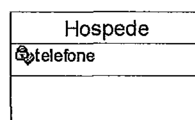


Figura 11: A classe "Hospede", com atributo "telefone"

Analisando a Figura 12, pode-se ver que os elementos foram inseridos no repositório e ambos se encontram em sua versão 1. A seguir, dois desenvolvedores aplicaram *"Check-out"* sobre os elementos, fazendo com que uma cópia fosse enviada para seus respectivos espaços de trabalho. No caso, basta aplicar *"Check-out"* sobre o elemento "Hospede" que o atributo "telefone" também é enviado para o espaço de trabalho, considerando que o mesmo é parte da estrutura da classe "Hospede". Esse é um comportamento recursivo realizado com base no conceito de "*namespace*" da UML.

A Figura 12 demonstra a existência de três configurações, utilizando a terminologia apresentada na Figura 10. A configuração que se encontra no repositório acumula os papéis de "Configuração **original**" e "Configuração **atual**". As configurações que se encontram no espaço de trabalho dos desenvolvedores, neste momento, são denominadas "Configurações do **usuário**".

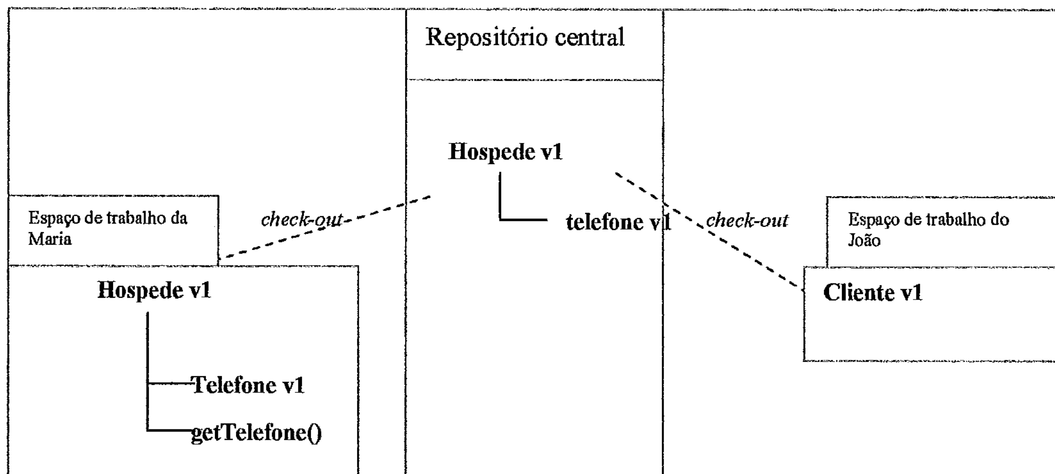


Figura 12: *"Check-out"* sobre os elementos no repositório

Depois que os elementos foram transferidos, os desenvolvedores já aplicaram suas modificações. João fez duas modificações: trocou o nome de "Hospede" para "Cliente", e removeu o atributo "telefone". Maria apenas adicionou a operação "getTelefone()" à classe "Hospede". Após realizadas as modificações, é João quem devolve o elemento primeiro, fazendo com que o cenário assuma a forma descrita na Figura 13.

Após o *"Check-in"* do João, é criada uma nova versão da classe "Hospede", agora com o nome de "Cliente", que aparece na versão 2. Pode-se notar que o atributo

"telefone" não existe mais, na versão 2. As configurações existentes agora são: "Hospede", v1 continua sendo a "Configuração **original**"; "Cliente", v2 agora passa a ser a "Configuração **atual**"; e a configuração que se encontra no espaço de trabalho de Maria continua sendo a "Configuração do **usuário**". Esta é a situação utilizada para a análise a seguir.

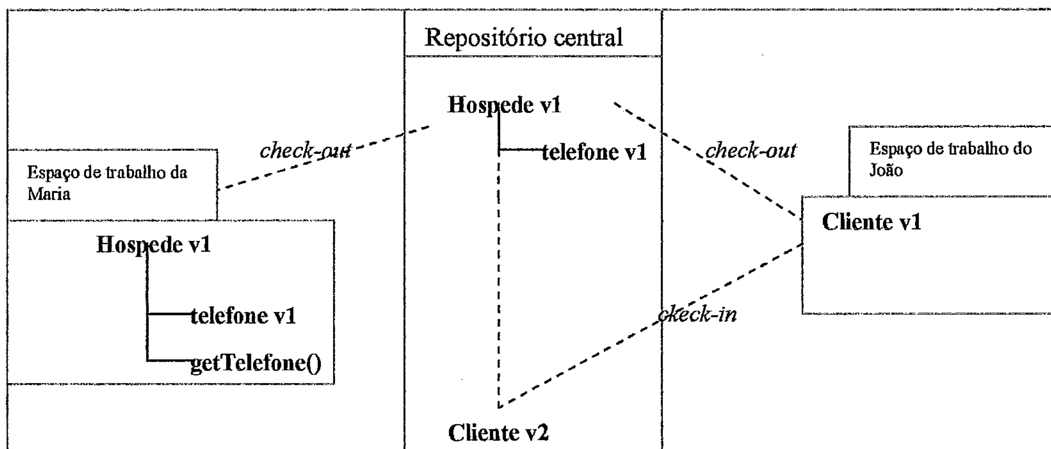


Figura 13: Realização de junção no *"Check-in"* do João

O cenário fica interessante quando Maria solicita *"Check-in"* e envia suas modificações para o repositório. O *"Check-in"* realizado pelo João transcorreu sem maiores problemas porque a configuração que ele utilizou como base para o *"Check-out"*, a "Configuração **original**", também exercia o papel de "Configuração **atual**", conforme descrito na Figura 13.

O mesmo não ocorreu com Maria. A configuração que ela utilizou como base para o "*Check-out*" não é mais a "Configuração **atual**", o que significa que outro desenvolvedor aplicou "*Check-in*", no caso o João. Tem-se uma situação de realização de junção.

Neste momento, é possível retornar à Tabela 2 e relacioná-la com os cenários descritos na Figura 13. Considerando que temos três elementos ("Hospede", "telefone" e "getTelefone"), a análise tem que ser realizada individualmente, pois é dessa forma que essa abordagem trata os elementos.

A junção da classe "Hospede" refere-se à Situação 3 da Tabela 2. Ou seja, o elemento pertence às três configurações, "Configuração **original**", "Configuração **atual**" e "Configuração do **usuário**". Ele foi modificado na "Configuração **atual**", mas não foi

modificado na "Configuração do **usuário**". O elemento pertencente à "Configuração **atual**" será adicionado à "Configuração **final**".

Não houve conflito baseado em propriedades primitivas e também não ocorreu conflito misto, porque não houve edição e remoção sobre o elemento concorrentemente. No entanto, ainda não é possível saber se ocorreu conflito baseado em estrutura, porque esse tipo de conflito surge em decorrência da análise dos subelementos.

Com relação ao atributo "telefone", o elemento pertence à "Configuração **original**", não pertence à "Configuração **atual**" e pertence à "Configuração do **usuário**". Portanto, é relatado na Situação 7 da Tabela 2. O elemento não será adicionado à "Configuração **final**" e também não ocorre situação de conflito.

Finalmente, a situação descrita pela operação "getTelefone()", criada pela Maria, refere-se à Situação 12 da Tabela 2. O elemento não pertence à "Configuração **original**", não pertence à "Configuração **atual**", mas pertence à "Configuração do **usuário**". O elemento da "Configuração do **usuário**" será adicionado à "Configuração **final**". Não há conflito nesse caso também.

Agora já é possível identificar se houve conflito baseado em estrutura na classe "Hospede", na medida em que todos os seus subelementos já foram tratados. "Hospede" teve duas modificações em seus subelementos. João removeu o atributo "telefone", enquanto Maria adicionou uma operação "getTelefone()".

Considerando que classe é GC, então tem-se um conflito baseado em estrutura, no escopo da classe "Hospede", porque a classe foi modificada concorrentemente. Portanto, o *"Check-in"* realizado pela Maria não será efetivado, em virtude do conflito ocorrido. Basta que um conflito ocorra para que a operação de *"Check-in"* deixe de surtir efeito. Ao final da operação, Maria receberá um relatório indicando o conflito e nenhuma alteração será aplicada sobre os elementos no repositório.

Se a classe não fosse definida como GC, a junção seria realizada e a classe "Hospede" estaria com a aparência descrita na Figura 14.

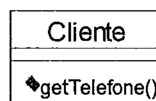


Figura 14: Resultado da junção aplicada à classe "Hospede"

De acordo com a especificação da UML, a classe "Hospede" estaria correta. O problema é que, após a junção, ela deixou de representar o desejo de Maria quando criou a operação "getTelefone()". A operação foi criada para manipular o atributo "telefone", que não existe mais.

De posse de um relatório de conflito, Maria tem a oportunidade de analisar a situação e fornecer uma solução apropriada, que pode incluir um diálogo com João, responsável pelo último *"Check-in"*, já que essa informação fica registrada no meta-modelo de versionamento.

3.9 Considerações finais

Este capítulo descreveu uma abordagem de controle de versões para os elementos da UML, onde o conhecimento sobre a estrutura desses elementos foi fundamental para a realização das tarefas. Para atender a esse objetivo, seis requisitos foram definidos: (i) granularidade fina; (ii); não-intrusão (iii) compatibilidade com ADSs e ferramentas *CASE* existentes; (iv) flexibilidade; (v) acesso concorrente; e (vi) distribuição. Esses requisitos foram selecionados porque o foco da GCS é o desenvolvimento e a manutenção de sistemas grandes e complexos.

Os ambientes de desenvolvimento de software foram bastante modificados, sendo que os SCVs atuais ainda utilizam técnicas e ferramentas propostas há aproximadamente trinta anos. Um dos problemas mais graves das abordagens atuais é o modelo de dados utilizado, que não permite o tratamento adequado de artefatos de análise e projeto.

Mesmo as abordagens modernas direcionadas para artefatos de análise e projeto, que utilizam modelos de dados orientado a objetos, não dão a devida atenção para às informações contidas em cada elemento individualmente, e versionam todo o modelo como uma entidade única.

Utilizando uma granularidade fina, os desenvolvedores e gerentes obterão melhores resultados quando interagirem com o SCV, seja realizando uma simples consulta para obter informações da evolução das versões, ou para realizar operações complexas, como a junção, por exemplo.

A Tabela 3, a seguir, apresenta um quadro-resumo comparando o Odyssey-VCS com as abordagens descritas no Capítulo 2.

Tabela 3: Quadro-resumo comparando o Odyssey-VCS com as demais abordagens.

Abordagens	Granularidade fina	Não-Intrusivo	Compatibilidade com *CASE*	Flexível	Acesso concorrente	Distribuído
OHST/KELTER	Sim	Sim	Sim	Não	Não	Não
ADAPTIVE/ IBM	Não	Sim	Sim	Não	Sim	Sim
MIMIX	Não	Sim	Sim	Não	Sim	Sim
LUCRÉDIO/PRADO	Não	Não	Sim	Não	Sim	Sim
DVM	Não	Sim	Sim	Não	Sim	Sim
Molhado	Sim	Não	Não	Não	Não	Não
Odyssey-VCS	Sim	Sim	Sim	Sim	Sim	Sim

Capítulo 4 - O Protótipo do Odyssey-VCS

4.1 Introdução

O desenvolvimento do protótipo Odyssey-VCS teve como finalidade demonstrar a viabilidade de implementação das idéias discutidas no Capítulo 3. Sendo assim, o objetivo deste capítulo é descrever as decisões de projeto e tecnologias que foram utilizadas no desenvolvimento do protótipo.

Além de tratar do Odyssey-VCS, este capítulo descreve um ADS e uma ferramenta *CASE*, que são capazes de interagir com o Odyssey-VCS. É também apresentado um exemplo de utilização do protótipo.

Para facilitar o entendimento do capítulo, as seções foram definidas com base na estrutura apresentada na Figura 15. A figura possui três divisões: (i) cenário de utilização; (ii) camada de transporte; e (iii) o servidor Odyssey-VCS. Cada divisão será detalhada por uma seção do capítulo.

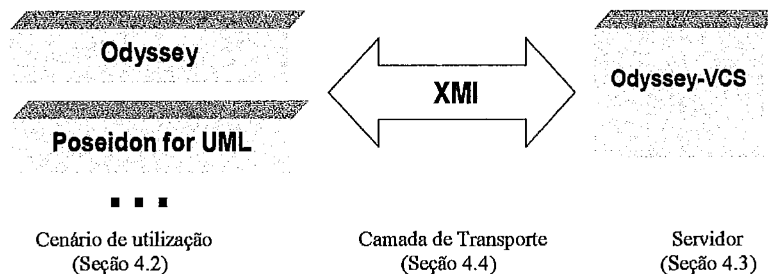


Figura 15: Visão geral do Odyssey-VCS

Este capítulo está organizado da seguinte forma: a Seção 4.2 descreve um cenário de utilização, além dos procedimentos que devem ser tomados para que cada ADS ou ferramenta *CASE* seja capaz de interagir com o Odyssey-VCS. A Seção 4.3 detalha o servidor Odyssey-VCS, os elementos principais de sua estrutura e uma breve explicação da função de cada um. Detalhes sobre o tratamento dado à granularidade também são fornecidos. A Seção 4.4 descreve a camada de transporte, detalhando todos os elementos que a integram. A Seção 4.5 destina-se à apresentação do exemplo de utilização do

protótipo e aborda o tratamento de conflitos a partir desse exemplo. A Seção 4.6 contém as considerações finais do capítulo.

4.2 Cenário de utilização

Esta seção tem como objetivo apresentar um cenário de utilização do Odyssey-VCS. Este cenário é composto pelos ADSs e ferramentas *CASE* que interagem com o servidor Odyssey-VCS, enviando e recebendo elementos da UML. A seção descreve ainda os procedimentos necessários para que estes ADSs e ferramentas *CASE* sejam capazes de interagir com o Odyssey-VCS.

O ambiente de desenvolvimento Odyssey é um desses ADSs e está descrito na Seção 4.2.1 e uma das ferramentas *CASE* é o Poseidon, descrita na Seção 4.2.2. É importante dizer que, considerando que o formato de representação dos dados utilizado é XMI, qualquer outro ADS ou ferramenta *CASE* poderia ser utilizado.

4.2.1 Ambiente Odyssey

No contexto da engenharia de software orientada à reutilização, o ambiente Odyssey tem sido desenvolvido na COPPE/UFRJ desde 1997, sendo concebido como um ambiente de reutilização baseado em modelos de domínio, que permite a construção de componentes reutilizáveis para uma determinada família de aplicações e sua posterior utilização no desenvolvimento de aplicações específicas (BRAGA, 2000).

Com a finalidade de disciplinar o conjunto de atividades necessárias à consecução dos objetivos propostos pelo ambiente, foram definidos dois processos genéricos: (i) Odyssey-DE (BRAGA, 2000), que descreve as atividades a serem realizadas na produção de artefatos reutilizáveis, caracterizando o desenvolvimento *para* reutilização; e (ii) Odyssey-EA (MILER, 2000), que descreve as atividades realizadas na construção de aplicações a partir dos artefatos previamente construídos pelo Odyssey-DE, caracterizando o desenvolvimento *com* reutilização.

Um conjunto de ferramentas foi desenvolvido com o objetivo de automatizar as diversas etapas definidas pelos processos Odyssey-AE e Odyssey-DE. Podemos citar ferramentas como documentação de componentes (MURTA, 1999), 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 acompanhamento de processos (MURTA, 2002), ferramentas para controle de alterações (TEIXEIRA *et al.*, 2001), entre outras.

O ambiente Odyssey utiliza um modelo de dados orientado a objetos, o que lhe permite manipular elementos complexos de análise e projeto, necessários em ambientes orientados à reutilização. No entanto, esse modelo de dados proprietário é o principal obstáculo quando se deseja realizar a integração do ambiente Odyssey com outros ambientes de desenvolvimento e mesmo com o Odyssey-VCS.

Para possibilitar que o ambiente Odyssey pudesse interagir com o servidor Odyssey-VCS, foi necessário desenvolver, primeiramente, o Odyssey-XMI. Esse módulo é necessário porque o Odyssey-VCS recebe os elementos no formato XMI. O Odyssey-XMI é capaz de importar/exportar os elementos do Odyssey no formato XMI, e pode ser carregado dinamicamente, de acordo com as necessidades dos desenvolvedores (MURTA *et al.*, 2004).

Esta abordagem configurável é conseqüência da nova arquitetura do ambiente Odyssey, que passou por um processo de reestruturação no último ano. A reestruturação teve como objetivo tornar o ambiente mais "leve", de modo que somente as funcionalidades classificadas como essenciais permanecessem fixas. O módulo que contém as funcionalidades essenciais é denominado o núcleo (*kernel*) do Odyssey e os módulos carregados dinamicamente são os *plug-ins* (MURTA *et al.*, 2004).

O *plug-in*, denominado OdysseyVCSPlugin, foi desenvolvido com as mesmas características de carga dinâmica do Odyssey-XMI. Desta forma, o OdysseyVCSPlugin utiliza os serviços do Odyssey-XMI, possibilitando que os elementos sejam enviados/recebidos do repositório no formato XMI.

O OdysseyVCSPlugin fornece as funcionalidades de "Inserir", *"Check-out"*, *"Check-in"* e "Listar" no lado do cliente. A Figura 16 apresenta o momento em que o OdysseyVCSPlugin está sendo selecionado para instalação no ambiente Odyssey. É possível ver que o Odyssey-XMI aparece na lista de módulos dos quais o OdysseyVCSPlugin depende. Desta forma, o Odyssey se encarrega de garantir as

condições mínimas necessárias para que o *plug-in* funcione adequadamente, de forma transparente para o usuário. Essas informações estão descritas em um arquivo em XML que é lido pelo Odyssey, para realizar a carga dinâmica.

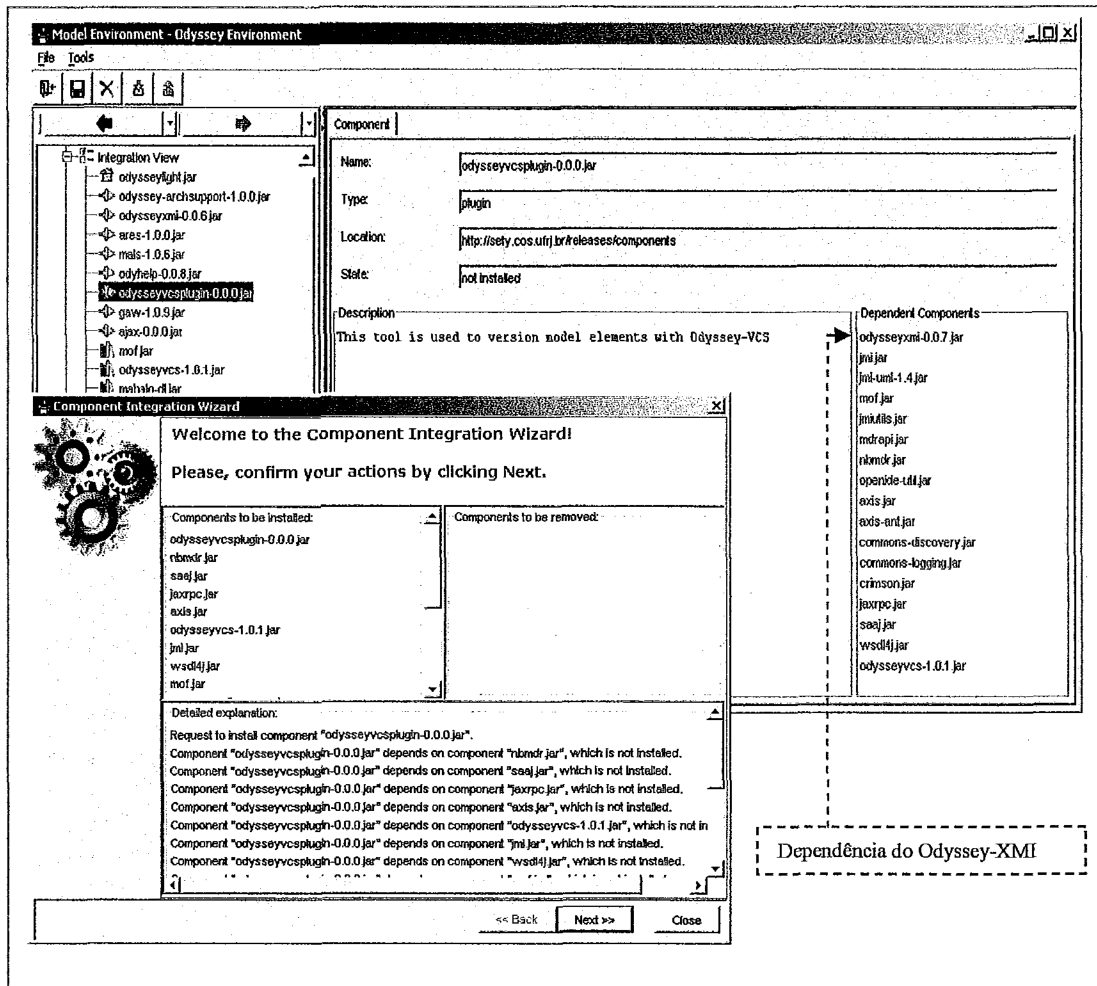


Figura 16: Seleção do OdysseyVCSPlugin para instalação no Odyssey

4.2.2 Poseidon for UML

O Poseidon é uma ferramenta *CASE* comercial que consiste numa evolução do ArgoUML (TIGRIS, 2005b), um projeto código-aberto e livre. Embora seja uma ferramenta comercial, o Poseidon possui uma versão gratuita, o *Poseidon for UML Community Edition,* utilizada nesta dissertação.

Um problema que surge quando se utiliza o Poseidon, nesta abordagem, é como permitir que o mesmo possa interagir com o Odyssey-VCS, enviando e recebendo

elementos. No contexto do ambiente Odyssey, esse problema foi solucionado com o desenvolvimento do OdysseyVCSPlugin. O Poseidon não possui uma arquitetura extensível semelhante à do ambiente Odyssey. No entanto, possui as funcionalidades de importação/exportação de elementos no formato XMI. Precisa-se apenas de uma aplicação capaz de ler/escrever esse arquivo XMI gerado pelo Poseidon.

A Figura 17 apresenta o Odyssey-VCS *Client*, desenvolvido para atuar como intermediário entre o Odyssey-VCS e o Poseidon. Esta aplicação interage com o Odyssey-VCS, invocando as operações de "Inserir", *"Check-out"*, *"Check-in"* e "Listar". Desta forma, o desenvolvedor pode se beneficiar das funcionalidades providas pelo Odyssey-VCS, utilizando o Poseidon como ferramenta de modelagem.

O lado esquerdo da figura (*Repository*) permite visualizar todos os ICs existentes no repositório. É importante dizer que os ICs são armazenados integralmente no repositório gerenciado pelo Odyssey-VCS, sem a computação de deltas. O lado direito (*Workspace*) apresenta os elementos que se encontram na área de trabalho do desenvolvedor. A opção "*Open*" é utilizada para ler um arquivo do sistema operacional contendo elementos da UML, descritos no formato XMI. A opção "*Save*" realiza o procedimento inverso e gera elementos da UML no formato XMI, de modo que o mesmo possa ser lido pelo Poseidon.

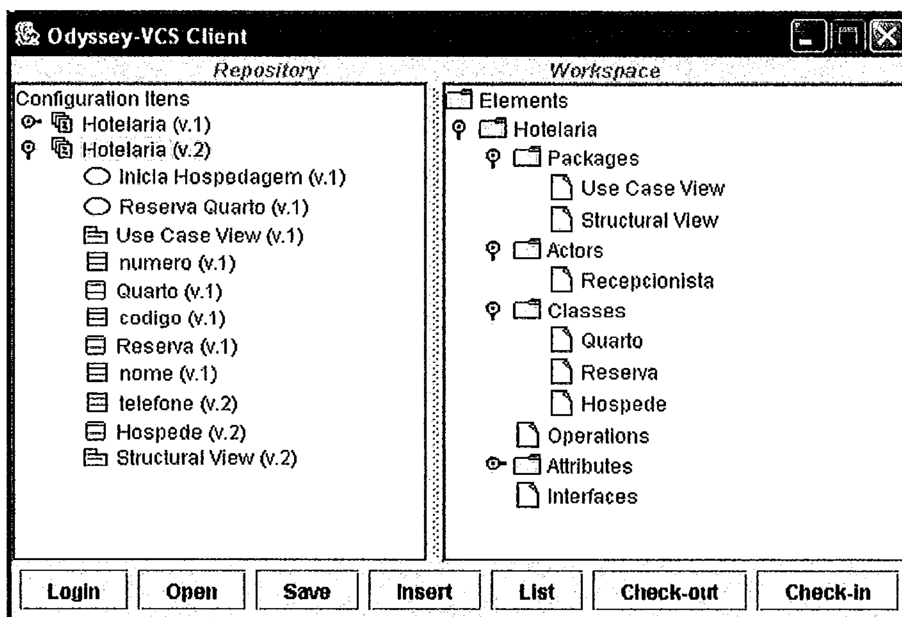


Figura 17: Visualização do Odyssey-VCS *Client*

4.3 Detalhamento do Odyssey-VCS

Esta seção é dedicada a uma explicação da estrutura e do funcionamento do servidor Odyssey-VCS. A Seção 4.3.1 descreve o projeto do Odyssey-VCS, enfatizando os elementos mais importantes de sua estrutura e uma breve explicação da função de cada um, enquanto que a Seção 4.3.2 aborda assuntos referentes à granularidade, descrevendo a forma como é possível configurar o Grão de Versão e (GV) e o Grão de Configuração (GC), no servidor Odyssey-VCS.

4.3.1 Detalhamento do projeto

A Figura 18 apresenta o projeto de alto nível do servidor Odyssey-VCS. A classe "VCSFacade" representa a fronteira entre o Odyssey-VCS e os ADSs ou ferramentas *CASE*. Somente através dessa classe é que as operações disponibilizadas pelo Odyssey-VCS podem ser solicitadas.

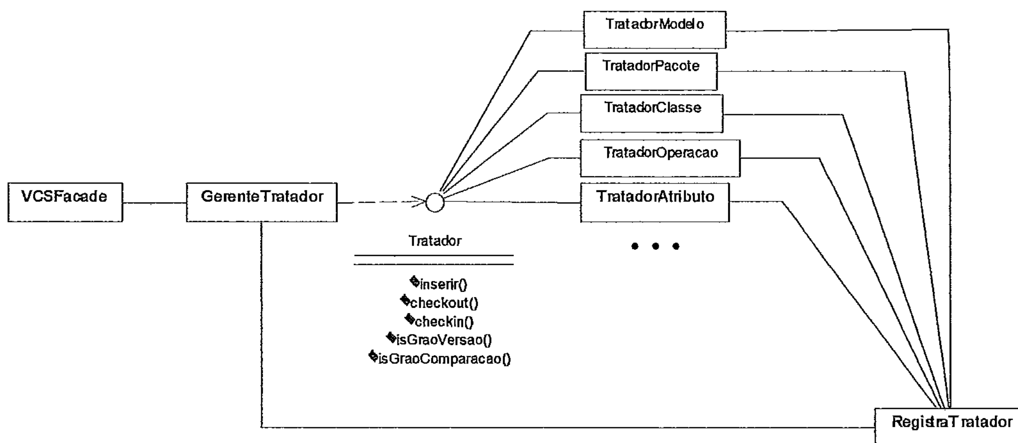


Figura 18: Detalhamento do projeto do Odyssey-VCS

A interface "Tratador" foi criada para solucionar o problema da individualidade de cada elemento. Desta forma, as classes "TratadorModelo", "TratadorPacote", "TratadorClasse", "TratadorOperação" e "TratadorAtributo" implementam a interface "Tratador", proporcionando um versionamento apropriado, que leva em conta o conhecimento contido na estrutura de cada elemento.

A classe "RegistraTratador" encapsula o conhecimento referente à associação entre o elemento da UML e seu tratador específico. Finalmente, a classe "GerenteTratador" consiste no núcleo do Odyssey-VCS. Esta classe manipula o meta-modelo de versionamento, permitindo que se registre e recupere todas as informações de versão dos elementos.

Quando um elemento é enviado para o Odyssey-VCS, através do "*Check-in*", os seguintes passos são realizados: "VCSFacade" recebe os elementos no formato XMI, os transforma para objetos Java e, a seguir, aciona "GerenteTratador". De posse dos elementos, "GerenteTratador" repassa-os para a classe "RegistraTratador", que retorna o tratador específico de cada elemento. "GerenteTratador", então, interage com os tratadores específicos para versionar os elementos.

O próximo passo consiste em verificar as modificações realizadas pelo desenvolvedor. Para cada elemento, é necessário saber se: (1) suas propriedades primitivas (nome, por exemplo) se modificaram; e (2) algum subelemento contido no elemento se modificou.

Esse entendimento é importante porque uma nova versão de um elemento será criada se: (1) suas propriedades primitivas se modificaram; e/ou (2) algum de seus subelementos se modificaram. Os elementos que são GVs e foram modificados terão suas informações registradas com base no meta-modelo de versionamento, manipulado por "GerenteTratador". Essa tarefa é realizada de forma idêntica, através da recursividade, para todos os elementos que são GVs.

4.3.2 Detalhamento da granularidade

O Odyssey-VCS permite que os elementos que irão se tornar GV e/ou GC sejam definidos pelos gerentes de configuração. Esse tratamento é realizado através de um arquivo em XML, conforme mostra a Figura 19.

A estrutura do arquivo XML é composta por cinco elementos, que são: "graos", "grao", "tipo", "graoVersao" e "graoComparacao". O elemento "graos" (Linha 1) é a raiz na estrutura do arquivo XML. O elemento "grao" (Linha 2) tem a finalidade de definir as propriedades no escopo de cada elemento. Por exemplo, um modelo é definido no escopo

do elemento “grao”, da mesma forma, existe um elemento “grao” para definir as propriedades para pacote, classe, caso de uso, operação e atributo.

A valor do elemento “tipo” (Linha 3) define o elemento que está sendo tratado, e para o qual os valores dos elementos “graoVersao” e “graoComparacao” serão atribuídos. O valor do elemento “graoVersao” (Linha 4) define se o elemento será versionado. E o valor do elemento “graoComparacao” (Linha 5) define se o elemento irá se comportar como GC.

Resumidamente, a partir de uma análise nas Linhas 2 a 6, do arquivo XML, conclui-se que o modelo, da UML, será versionado (graoVersao= “true”) e não será utilizado para tratamento de conflitos (graoComparacao= “false”). As Linhas 7 a 11 tratam de pacote e têm os mesmos valores de modelo.

```
1 <graos>
2   <grao>
3        <tipo>org.omg.uml.modelmanagement.Model</tipo>
4        <graoVersao>true</graoVersao>
5        <graoComparacao>false</graoComparacao>
6   </grao>
7   <grao>
8        <tipo>org.omg.uml.modelmanagement.UmlPackage</tipo>
9        <graoVersao>true</graoVersao>
10       <graoComparacao>false</graoComparacao>
11  </grao>
12  <grao>
13       <tipo>org.omg.uml.foundation.core.UmlClass</tipo>
14       <graoVersao>true</graoVersao>
15       <graoComparacao>true</graoComparacao>
16  </grao>
17  <grao>
18       <tipo>org.omg.uml.behavioralelements.UseCases</tipo>
19       <graoVersao>true</graoVersao>
20       <graoComparacao>true</graoComparacao>
21  </grao>
22  <grao>
23       <tipo>org.omg.uml.behavioralelements.Actor</tipo>
24       <graoVersao>false</graoVersao>
25       <graoComparacao>false</graoComparacao>
26  </grao>
27  <grao>
28       <tipo>org.omg.uml.foundation.core.Operation</tipo>
29       <graoVersao>true</graoVersao>
30       <graoComparacao>false</graoComparacao>
31  </grao>
32  <grao>
33       <tipo>org.omg.uml.foundation.core.Attribute</tipo>
34       <graoVersao>true</graoVersao>
35       <graoComparacao>false</graoComparacao>
36  </grao>
32 </graos>
```

Figura 19: Tratamento para GV e GC

Por outro lado, as Linhas 12 a 16 definem que as classes da UML serão versionadas e que também serão tratadas como GC. Os casos de uso têm os mesmos

valores de classes, conforme se observa nas Linhas 17 a 21. O ator não será versionado e também não se comportará como GC, conforme descrito nas Linhas 22 a 26. E, finalmente, as operações (Linhas 27 a 31) e os atributos (Linhas 32 a 36) serão versionados e não serão GCs.

4.4 Camada de transporte

A camada de transporte viabiliza a comunicação entre os ADSs, as ferramentas *CASE* e o servidor Odyssey-VCS. A Figura 20, que consiste num refinamento da Figura 15, apresenta uma visão mais detalhada da camada de transporte do Odyssey-VCS.

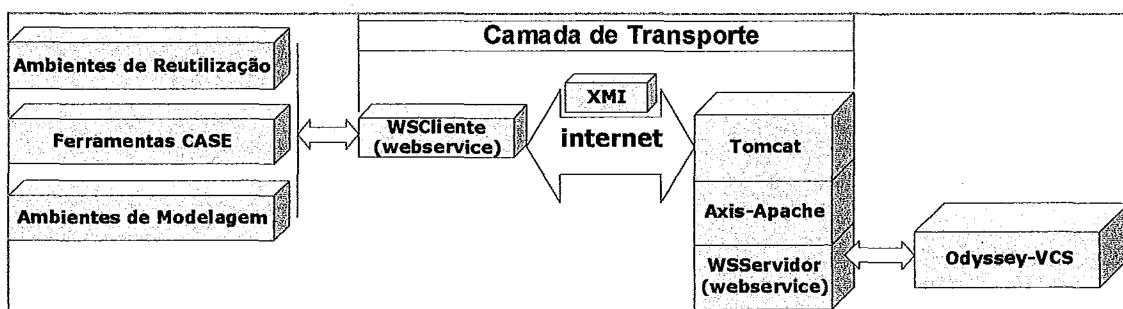


Figura 20: Visão geral do Odyssey-VCS com ênfase na camada de transporte

A Seção 4.4.1 descreve os elementos que constituem a camada de transporte e que tornam possível efetuar a comunicação utilizando *Web Service* como protocolo de comunicação. A Seção 4.4.2 aborda as questões referentes ao formato de representação dos dados na comunicação cliente/servidor.

4.4.1 Protocolo de comunicação

Conforme descrito na Seção 3.7, para atender ao requisito da *distribuição*, no contexto dessa abordagem, foi utilizado *Web Service*. Os elementos que viabilizam essa comunicação são descritos a seguir.

O *Tomcat*, descrito na Figura 20, é um *container servlet* desenvolvido pelo projeto Jakarta (JAKARTA, 2005). *Servlet* é uma tecnologia baseada em Java para desenvolver aplicações para a *web* com conteúdo dinâmico (SUN, 2005). Por implementar o protocolo HTTP, o *Tomcat* pode ser utilizado como servidor de aplicações para a *web*. No entanto, se comparado a outros servidores *web* disponíveis, o *Tomcat*

apresenta problemas de desempenho na manipulação de arquivos em HTML convencionais, o que torna difícil sua utilização em um ambiente de produção real.

Uma configuração apropriada é instalá-lo como uma extensão de um servidor *web*, por exemplo, o Apache HTTP *Server* (APACHE, 2005a). Desta forma, toda vez que o Apache receber uma requisição para um *servlet*, esta será redirecionada para o *Tomcat*, que, juntamente com o *servlet* responsável pela requisição, realizará o processamento necessário para que uma resposta seja gerada. Apesar das limitações, o *Tomcat* foi utilizado como servidor HTTP na implementação da camada de transporte, na medida em que tais restrições não constituem obstáculos para utilizá-lo no protótipo do Odyssey-VCS.

O *Axis-Apache* (APACHE, 2005b) é uma implementação *Web Service*, disponibilizada pelo *Apache Software Foundation*. O *Axis-Apache* foi concebido para ser uma aplicação inserida no contexto do *Tomcat*, o que lhe permite receber requisições enviadas via HTTP. De maneira simplificada, para utilizar o *Axis-Apache*, é necessário desenvolver duas classes principais. No contexto do Odyssey-VCS, essas classes são: WSCliente e WSServidor (Figura 20).

O WSCliente, como o próprio nome sugere, é o elemento cliente que realiza as chamadas remotas. Estas chamadas são recebidas pelo *Tomcat*, que as repassa para o *Axis-Apache*. O *Axis-Apache*, por outro lado, identifica a classe responsável por tratar o serviço solicitado que deve ser previamente publicado. Esta publicação é realizada em um arquivo de configuração do *Axis-Apache*, no formato XML. O arquivo está descrito na Figura 21.

```
1    <service name="ControleDeVersao" provider="java:RPC">
2        <parameter name="allowedMethods" value="*"/>
3        <parameter name="className" value="WSServidor"/>
4    </service>
```

Figura 21: Arquivo em XML definindo o serviço ControleDeVersão

"WSServidor", cujo nome aparece na linha 3 da Figura 21, é uma classe implementada em Java que é responsável pelo tratamento do serviço "ControleDeVersao" (linha 1). O serviço é composto de quatro operações implementadas

na classe "WSServidor". As operações são "Inserir", "Listar", "*Check-out*" e "*Check-in*". A propriedade "allowedMethods" (linha 2) permite definir todas as operações contidas na classe "WSServidor" que serão disponibilizadas para acesso remoto. Como o valor para essa propriedade é um asterístico (*), isto significa que todas as operações da classe "WSServidor", com modificador de acesso *public*, serão disponibilizadas como parte do serviço "ControleDeVersão".

Além de garantir acesso remoto, a utilização de *Web Service* oferece melhores condições de interoperabilidade já que a comunicação cliente/servidor pode ser realizada através de diferentes plataformas de desenvolvimento.

Por exemplo, a classe "WSServidor" foi implementada utilizando a linguagem Java. No entanto, como esta é apenas uma implementação *Web Service*, o elemento WSCliente pode ser implementado em qualquer linguagem de programação, como, por exemplo, C++, Delphi ou Visual Basic.

4.4.2 Formato de representação dos dados

Durante o desenvolvimento da camada de transporte surgiu o problema de identificar unicamente o elemento tanto nos ADSs quanto no Odyssey-VCS. Esse problema pode ser descrito da seguinte forma: na realização do *"Check-out"*, é enviada uma cópia do elemento para o cliente. O desenvolvedor realiza as modificações e devolve o elemento para o servidor, através do *"Check-in"*. Neste momento, o Odyssey-VCS precisa relacionar a cópia que está sendo devolvida pelo desenvolvedor com o elemento original que reside no repositório. Esse relacionamento é necessário por causa das informações que devem ser registradas no meta-modelo de versionamento.

Considerando que optamos por uma abordagem não-intrusiva, a modificação do meta-modelo da UML não pôde ser considerada uma solução viável. Ou seja, não poderíamos simplesmente inserir um meta-atributo *identificadorÚnico* em um modelo, um pacote ou uma classe da UML.

No contexto do Odyssey-VCS, quando um elemento é inserido no repositório, um identificador único lhe é atribuído. Uma solução possível seria fazer com que esse identificador fosse registrado no atributo *xmi.id*, na geração do arquivo XMI. Desta

forma, o identificador único seria transportado para o cliente em conjunto com o elemento que ele identifica.

No entanto, não há como garantir que o valor do *xmi.id*, que foi enviado no arquivo XMI, é o mesmo valor do *xmi.id* que será devolvido pelo ambiente ou ferramenta que será utilizada pelo desenvolvedor para modificar o elemento. Esta limitação tornou a solução inviável.

O problema foi solucionado através da utilização do elemento Etiqueta (*tag*), que faz parte do mecanismo de extensão da UML. Uma Etiqueta especifica novos tipos de propriedades que podem ser anexadas aos elementos. As propriedades individuais de cada elemento são especificadas usando valores etiquetados. Eles podem ser tipos de dados simples ou referências a outros elementos. Uma Etiqueta pode ser usada para representar propriedades tais como informações de gerenciamento (autor, data, situação) e informação de geração de código (nível de otimização).

A Figura 22 e a Figura 23 demonstram a solução proposta. A Figura 22 descreve um arquivo no formato XMI, contendo uma classe, de nome "Hospede". Esse arquivo foi enviado pelo cliente para ser inserido no Odyssey-VCS, através da operação "Inserir".

```
1	<UML:Class xmi.id = '7ff9' name = 'Hospede' visibility = 'public'
2		isSpecification = 'false' isRoot = 'false' isLeaf = 'false' isAbstract = 'false'
3		isActive = 'false'>
4	</UML:Class>
```

Figura 22: Elementos em XMI enviados ao Odyssey-VCS

A Figura 23 apresenta o mesmo elemento descrito na Figura 22. A diferença reside na existência do elemento *taggedValue* (linhas 3 a 5). Este elemento, bem como seus valores, foram criados pelo Odyssey-VCS e representam o identificador único (linha 4) da classe "Hospede", quando persistido no repositório gerenciado pelo Odyssey-VCS. É através desse valor que o Odyssey-VCS será capaz de identificar a "Configuração **original**", a qual ele pertence, quando ele é devolvido através do *"Check-in"*.

```
1   <UML:Class xmi.id = 'a3' name = 'Hospede' visibility = 'public' isSpecification = 'false'
2       isRoot = 'false' isLeaf = 'false' isAbstract = 'false' isActive = 'false'>
3       <UML:TaggedValue xmi.id = 'a4' isSpecification = 'false'>
4           <UML:TaggedValue.dataValue>E3DFFD2EAA77:0000000000000ECA</UML:TaggedValue.dataValue>
5       </UML:TaggedValue>
6   </UML:Class>
```

Figura 23: Elementos em XMI com identificador em *TaggedValue*

Embora satisfatória em relação às demais abordagens, essa solução não é totalmente livre de falhas. Muitos ambientes utilizam o mecanismo da Etiqueta para anexar informações de customização aos elementos. O problema é que dois ambientes distintos, que estejam manipulando o elemento podem utilizar o mesmo nome de Etiqueta, o que ocasionaria um conflito.

4.5 Exemplo de utilização

Esta seção descreve o versionamento de elementos utilizando o protótipo do Odyssey-VCS. A Figura 24 apresenta os elementos que serão utilizados ao longo da seção. A Seção 4.5.1 apresenta um exemplo de utilização local, onde somente um desenvolvedor, através do ambiente Odyssey, interage com o servidor Odyssey-VCS. A Seção 4.5.2 descreve um cenário de acesso distribuído, onde dois desenvolvedores, um utilizando o ambiente Odyssey e outro utilizando o Poseidon, acessam o servidor Odyssey-VCS. Na Seção 4.5.3 são descritos os detalhes sobre o tratamento de conflitos.

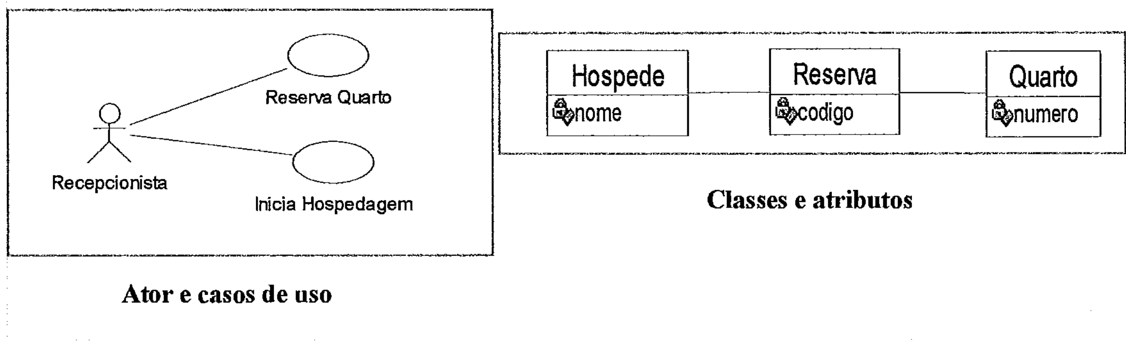


Figura 24: Elementos da UML para o exemplo de utilização

4.5.1 Utilização local

A Figura 25 apresenta o ambiente Odyssey, expondo as funcionalidades de controle de versões, fornecidas pelo OdysseyVCSPlugin. É possível notar que os

elementos apresentados na Figura 25 estão representados na árvore de elementos do ambiente Odyssey.

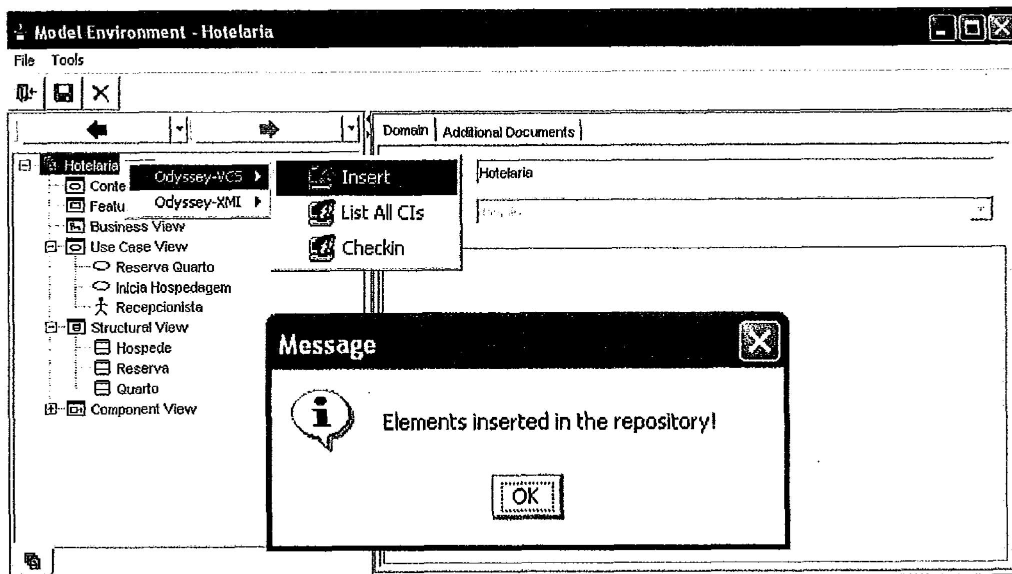


Figura 25: Ambiente Odyssey expondo as funcionalidades de controle de versões

A Figura 25 demonstra ainda que a opção "Inserir" foi selecionada, o que fez com que a árvore de elementos do Odyssey fosse enviada para o Odyssey-VCS. O Odyssey-VCS retornou uma mensagem, "Elements inserted in the repository", informando que a operação de inserção foi bem sucedida. A mensagem é mostrada pelo OdysseyVCSPlugin para que o desenvolvedor fique ciente do resultado da operação. A opção "Listar" permite visualizar o repositório para se certificar que os elementos foram persistidos. Essa opção permite ainda verificar quais elementos efetivamente se tornaram ICs. A Figura 26 apresenta o resultado da seleção da opção "Listar".

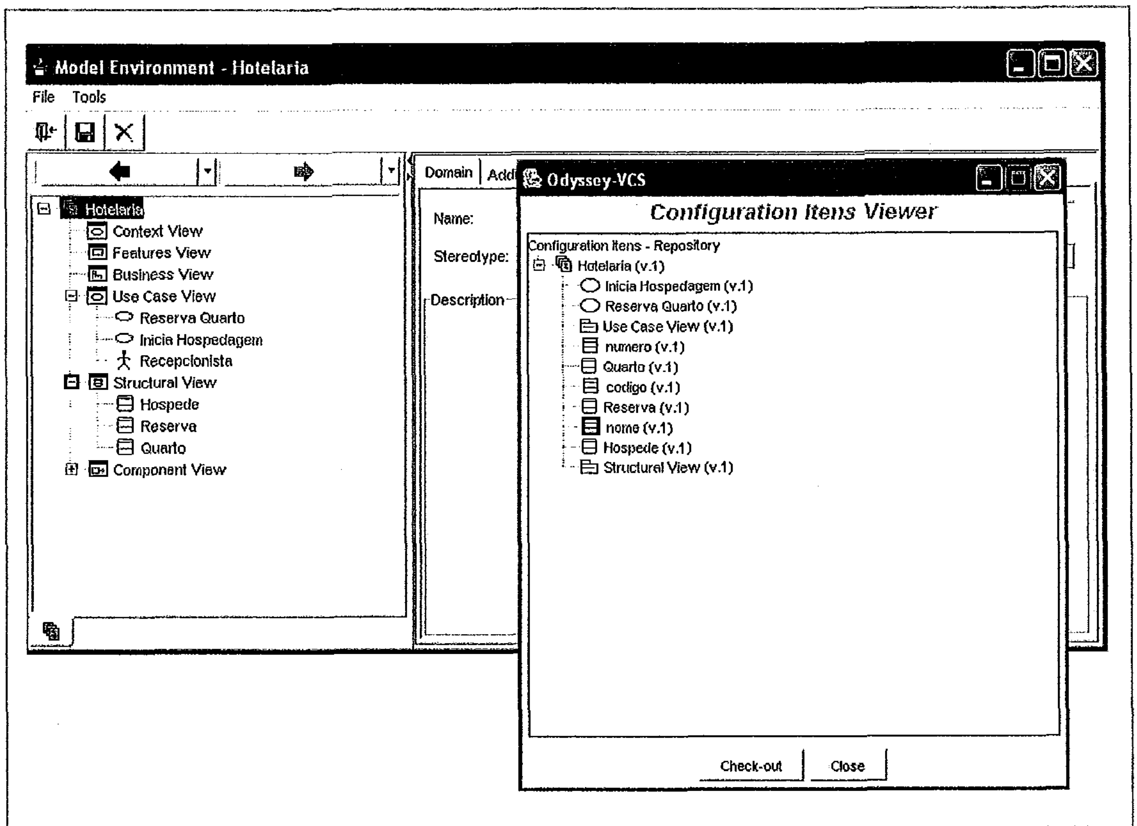


Figura 26: Visualização dos elementos que se tornaram ICs.

Para entender as informações da Figura 26, são necessárias algumas considerações sobre a proposta do ambiente Odyssey e o mapeamento realizado pelo Odyssey-XMI, para transformar a árvore de elementos do Odyssey em arquivo no formato XMI.

Conforme dito anteriormente, o Odyssey é um ambiente de reutilização que oferece apoio ao desenvolvimento *para* reutilização e *com* reutilização. Para esse propósito, a árvore do Odyssey é dividida em seis visões: Contexto, *Features*, Negócio, Caso de Uso, Estrutural e de Componente. Na visão Caso de Uso são colocados os atores e os casos de uso do domínio e a visão Estrutural é utilizada para agrupar os elementos comuns da UML, como pacotes, classes e interfaces.

No entanto, um mapeamento da árvore do Odyssey para a UML precisava ser realizado para que os elementos do Odyssey fossem descritos no formato XMI. O mapeamento foi feito da seguinte forma: (1) um domínio, no Odyssey, passou a

corresponder a um modelo na UML, já que ambos exercem o papel de elementos raiz; e (2) cada visão do Odyssey foi mapeada para um pacote estereotipado, na UML.

Ou seja, a visão Estrutural foi mapeada para um pacote com estereótipo <<Visão Estrutural>>, e a visão Caso de Uso, para um pacote com estereótipo <<Visão Caso de Uso>>. Dessa forma, outras ferramentas *CASE*, como Poseidon, por exemplo, são capazes de ler o XMI gerado pelo Odyssey, conforme será visto na Seção 4.5.2.

Uma vez descrito o mapeamento utilizado pelo Odyssey-XMI, pode-se analisar novamente a Figura 26. "Hotelaria" aparece como modelo. A "Visão Estrutural" e a "Visão Caso de Uso" aparecem na forma de pacotes. Os demais elementos, que são as classes, casos de uso e os atributos aparecem na sua forma original. É possível notar que foram criados onze ICs. O elemento "Recepcionista", por ser um ator, não se tornou IC. O resultado está compatível com a configuração dos graos, definida no arquivo em XML descrito na Figura 19.

A parte inferior da Figura 26 apresenta a opção *"Check-out"*. A Figura 27 apresenta o resultado da ação de *"Check-out"*, onde aparece uma mensagem informando que os elementos foram enviados do repositório para o espaço de trabalho do desenvolvedor. O resultado da operação *"Check-out"* é a criação de uma nova árvore no ambiente Odyssey, independentemente da existência de elementos no espaço de trabalho do desenvolvedor.

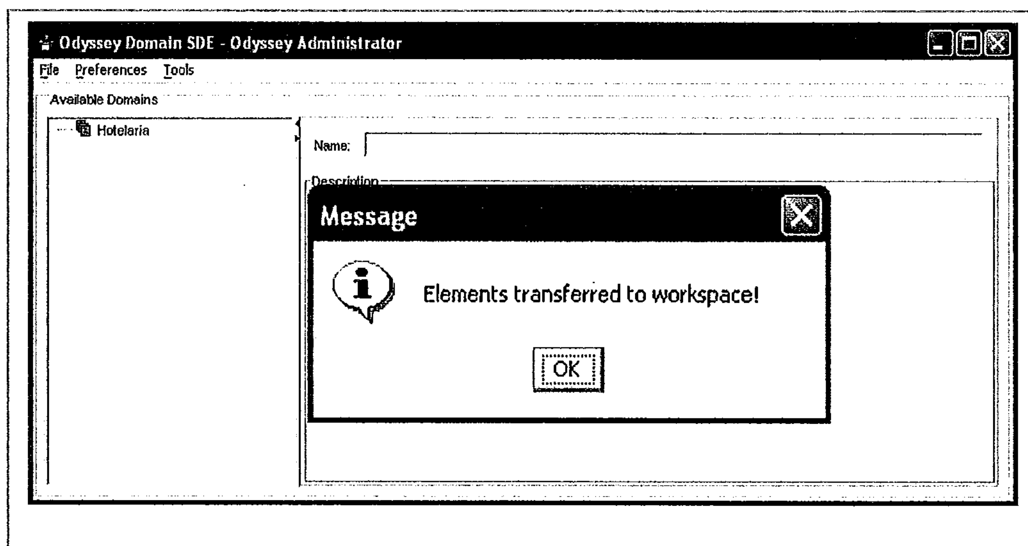


Figura 27: Resultado da ação da opção *"Check-out"*

A Figura 28 mostra a área de modelagem do Odyssey, com uma nova árvore já criada. Mostra também que um novo atributo, "telefone", foi inserido na classe "Hospede". O objetivo é enviar novamente os elementos para o OdysseyVCS para que as informações de versão sejam registradas. Essa ação é disparada com a opção *"Check-in"*.

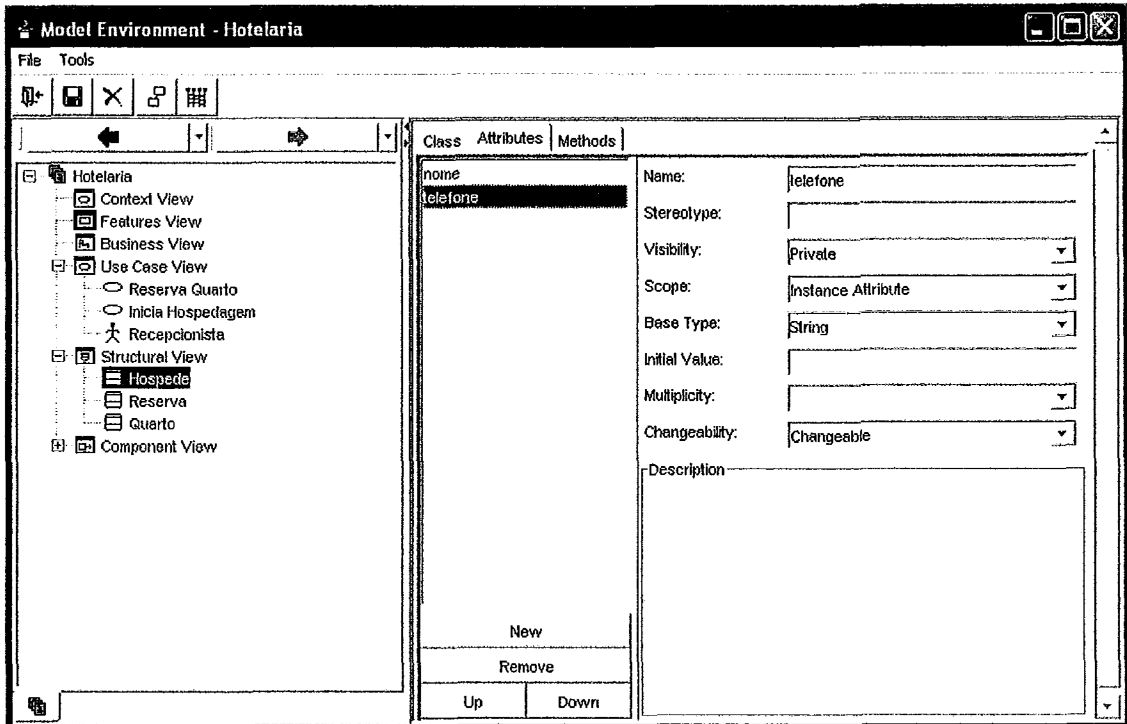


Figura 28: Inserção do atributo "telefone", na classe "Hospede"

Finalmente, a Figura 29 permite visualizar o repositório e verificar o resultado da ação de *"Check-in"*. Na medida em que um novo IC foi criado (i.e., "telefone"), uma nova versão da classe "Hospede", do pacote "Visão Estrutural" e do modelo "Hotelaria" também é criada. Ou seja, a modificação de um elemento determina a criação de uma nova versão do elemento no qual ele está contido. Nenhum dos outros elementos foi modificado e, por isso, permanecem na versão 1.

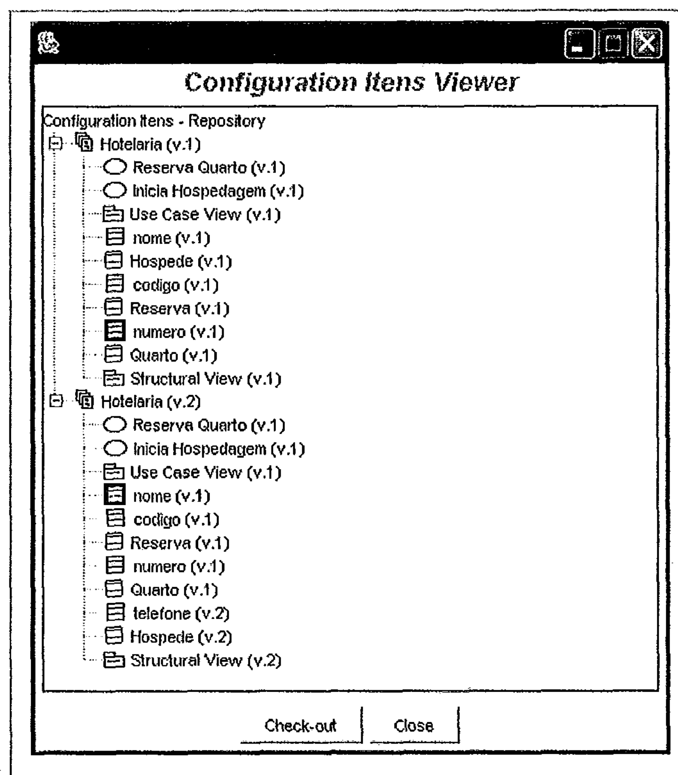


Figura 29: Visualização do repositório após a ação de *"Check-in"*

4.5.2 Utilização distribuída

Na Seção 4.5.1, consideramos o cenário onde apenas um desenvolvedor modifica os elementos. O foco desta seção é quando dois ou mais desenvolvedores modificam os mesmos elementos.

A Figura 30 apresenta novamente a tela de modelagem do ambiente Odyssey. É possível notar que os elementos gerados na última solicitação de *"Check-in"* foram novamente solicitados para o espaço de trabalho, através do *"Check-out"*.

A desenvolvedora, daqui para frente denominada Maria, realizou três modificações nos elementos. São elas:

- ✓ modificou o nome do caso de uso "Reserva Quarto" para "Reservar Quarto" - a modificação consistiu simplesmente em passar o verbo para o infinitivo;
- ✓ adicionou duas operações à classe "Hospede". São eles: "getTelefone" e "setTelefone"; e
- ✓ removeu o atributo "codigo" da classe "Reserva".

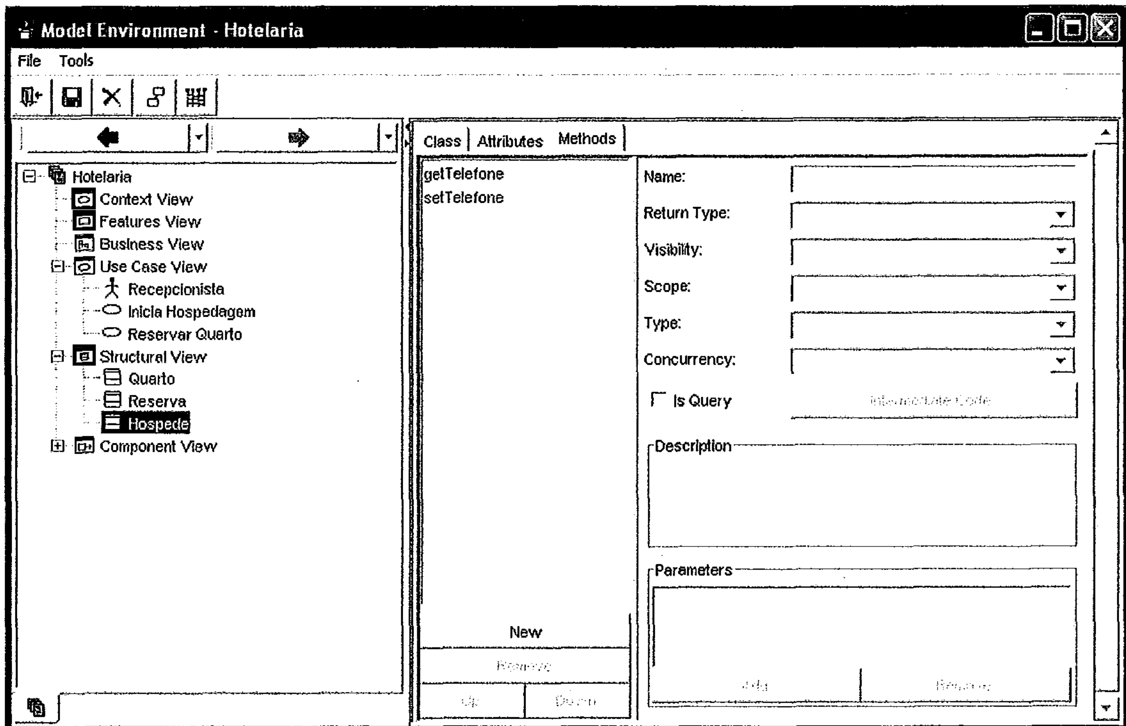


Figura 30: Desenvolvedor utilizando o Odyssey para modificar elementos

Antes que Maria solicitasse o *"Check-in"*, e enviasse os elementos para o repositório, João, utilizando o Poseidon, solicita *"Check-out"* e obtém uma cópia dos elementos. Desta forma, os dois desenvolvedores realizaram *"Check-out"* sobre a mesma versão do elemento no repositório, o que conduzirá a uma operação de junção, quando o segundo desenvolvedor, que pode ser tanto João quanto Maria, aplicar *"Check-in"*. É importante ressaltar que João necessita da aplicação Odyssey-VCS *Client*, descrita na Seção 4.2.2, para interagir com o Odyssey-VCS através do Poseidon. A Figura 31 apresenta esta utilização.

É possível notar que a opção *"List"* foi selecionada, fazendo com que o lado esquerdo de Odyssey-VCS *Client* mostrasse os ICs existentes no repositório. A execução de *"Check-out"* sobre o elemento "Hotelaria" permitiu que os elementos fossem transferidos para o espaço de trabalho do João, conforme se observa no lado direito da figura. Um caixa de diálogo aberta, sobreposta à tela da aplicação Odyssey-VCS *Client*, demonstra que a opção *"Save"* foi selecionada. Essa opção faz com que um arquivo,

denominado *output.xmi*, seja gerado com os elementos transferidos do Odyssey-VCS, descritos no formato XMI.

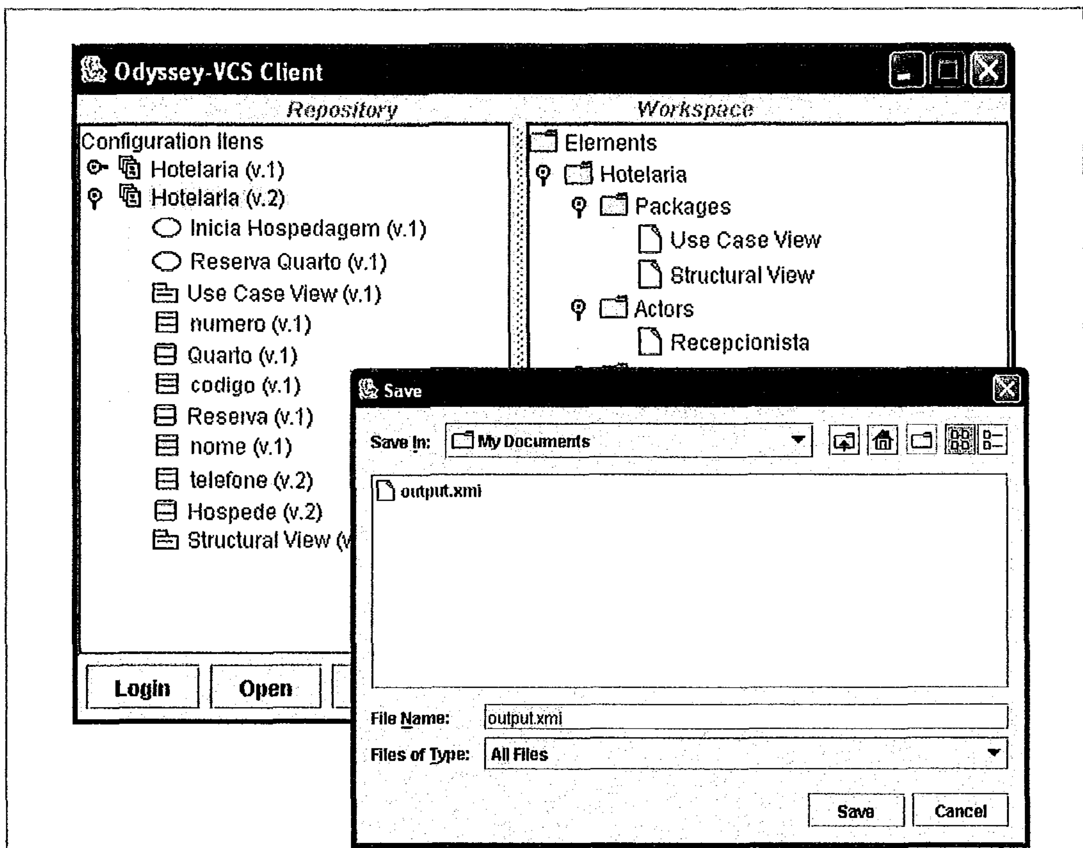


Figura 31: OdysseyVCS *Client* - intermediário entre o Poseidon e o OdysseyVCS

Resta somente importar esse arquivo no Poseidon, para que João possa realizar as modificações. A Figura 32 demonstra que os elementos agora se encontram na árvore de elementos do Poseidon, construída no lado esquerdo da figura. João realizou as seguintes modificações nos elementos:

- ✓ modificou o nome do atributo "telefone", da classe "Hospede", para "email";
- ✓ modificou o nome do atributo "codigo", da classe "Reserva", abreviando-o para "codReserva"; e
- ✓ modificou o nome do caso de uso "Reserva Quarto" para "Reservar"

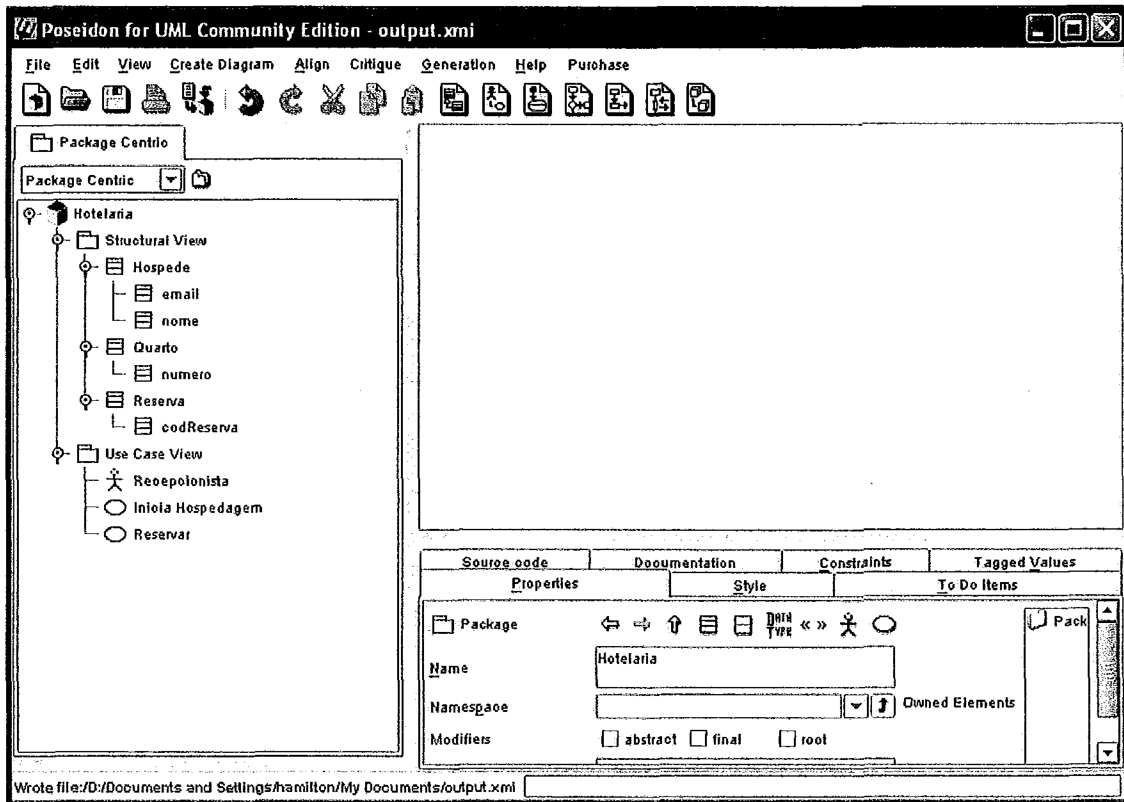


Figura 32: Tela de modelagem do Poseidon expondo os elementos do Odyssey-VCS

Neste momento, João envia os elementos para o Odyssey-VCS, através da aplicação Odyssey-VCS *Client*. Após solicitar *"Check-in"*, João deseja visualizar se suas modificações foram aceitas pelo Odyssey-VCS. Em análise à arvore exposta no lado esquerda da Figura 33, pode-se notar que as modificações foram efetivadas.

Cabem aqui algumas considerações sobre a forma como o Odyssey-VCS atribui números às versões dos elementos. Oito elementos foram modificados com o *"Check-in"* do João. Isto pode ser visto através do número da versão dos elementos: todos os que possuem o número de versão 3 foram modificados. Mais ainda, o atributo "codReserva", a classe "Reserva", e o caso de uso "Reservar" não possuem versão 2, o que pode ser visto confrontando-se a Figura 33 com a Figura 31. Importante dizer que o atributo "CodReserva", na versão 3 chamava-se "codigo" na versão 1 e o caso de uso "Reservar" na versão 3 chamava-se "Reserva Quarto" na versão 1.

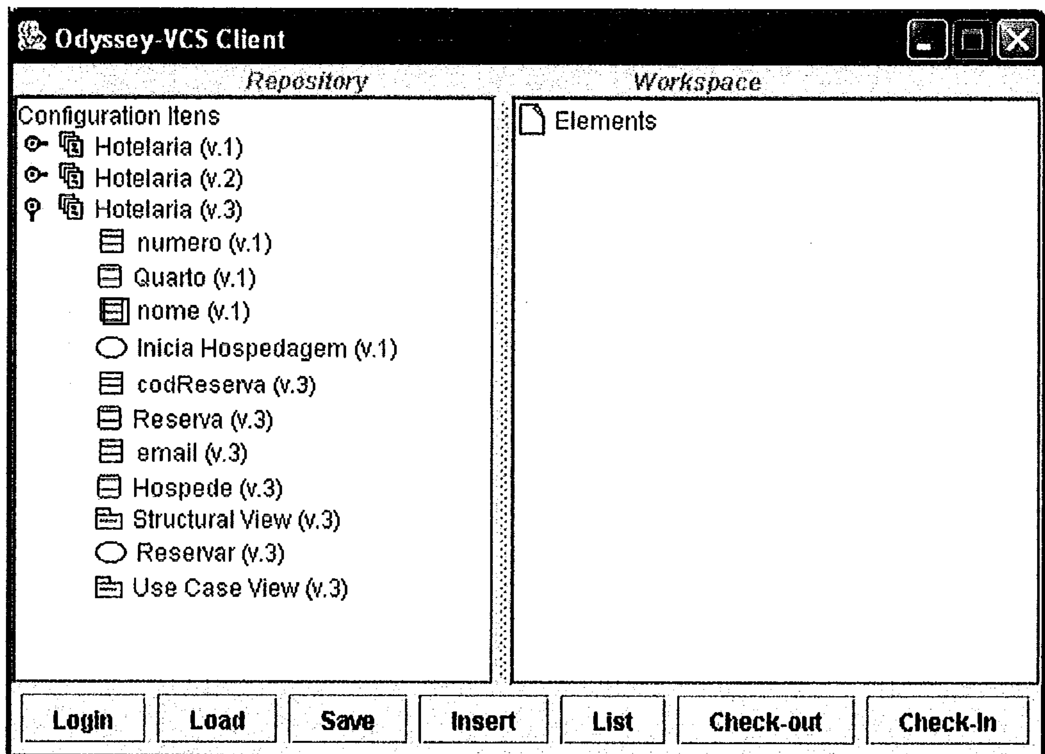


Figura 33: Visualização do repositório após o *"Chech-in"* de João

Uma das vantagens dessa abordagem é que se um desenvolvedor desejar tomar conhecimento das modificações aplicadas à versão 3, do elemento raiz "Hotelaria", pode facilmente identificar quais elementos foram modificados, visualizando apenas a árvore de ICs, sem precisar consultar qualquer outra fonte. O esquema de numeração de versões é atribuído com base no número da versão do elemento raiz e recebe a denominação de *versionamento global*. É o suporte ao Versionamento Orientado à Mudança (VOM), similar ao utilizado pelo Subversion, descrito no Capítulo 2.

4.5.3 Tratamento de conflitos

Depois das modificações de João, Maria solicitou *"Check-in"* e enviou os elementos para o Odyssey-VCS. Este identificou a necessidade do procedimento de junção porque João já realizou um *"Check-in"*, o que fez surgir uma nova "Configuração **atual**". Realizado o procedimento de junção, quatro conflitos são identificados (Figura 34):

1. um conflito baseado em propriedades primitivas, no caso de uso "Reservar";
2. um conflito misto, no atributo "codigo", da classe "Reserva";

3. um conflito baseado em estrutura, na classe “Reserva”. O Odyssey-VCS informa ainda o elemento que foi modificado. No caso, o atributo “codigo”; e
4. um conflito baseado em estrutura na classe “Hospede”. Os subelementos modificados foram: “telefone”, referenciado como “email”, e as operações “getTelefone” e “setTelefone”.

O conflito baseado em propriedades primitivas, no elemento “Reservar” ocorreu porque Maria modificou o nome caso de uso “Reserva Quarto” para “Reservar Quarto”, enquanto João modificou o nome do mesmo caso de uso para “Reserva”. É importante notar que essa notificação de conflito é realizada independentemente do caso de uso ter sido definido como GC.

O conflito misto no elemento “codigo” ocorreu porque Maria removeu o atributo e João renomeou-o para “codReserva”;

O conflito baseado em estrutura na classe “Reserva” ocorreu como consequência do conflito anterior. Os dois desenvolvedores modificaram concorrentemente o atributo “codigo”, da classe “Reserva”, e o tipo classe foi definido como GC.

Finalmente, no conflito baseado em estrutura, no elemento “Hospede”, os subelementos modificados foram “telefone”, “getTelefone” e “setTelefone”. Nesse caso, a notificação de conflito somente ocorreu porque o tipo classe foi definido como GC.

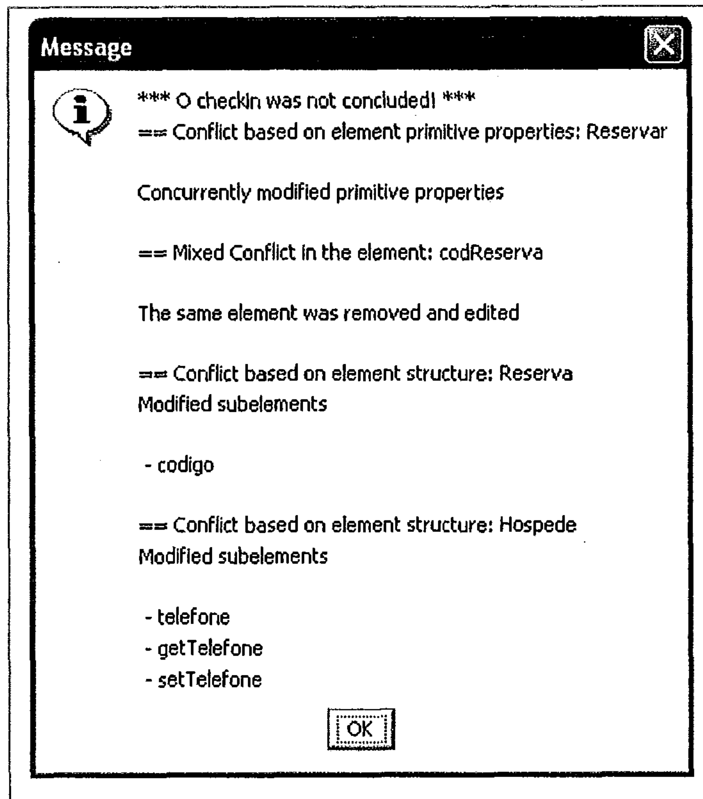


Figura 34: Relatório informando os conflitos

Se o tipo classe não fosse definido como GC, o sistema realizaria a junção sem nenhum problema, e a classe "Hospede" ficaria conforme descrito na Figura 35.

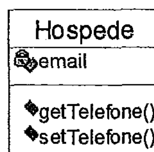


Figura 35: Resultado da junção aplicada a classe "Hospede"

De acordo com a especificação da UML, a classe "Hospede" estaria correta. O problema é que, após a junção, ela deixaria de representar o desejo de Maria quando criou as operações "getTelefone" e "setTelefone". As operações foram criadas para manipular o atributo "telefone", que, embora não tenha deixado de existir, aparece na classe (o que seria a versão 4) agora com o nome de "email". Isso poderia, ainda, confundir outros desenvolvedores que viessem a manipular essa classe.

De posse do relatório de conflito gerado pelo Odyssey-VCS, Maria tem a oportunidade de analisar a situação e fornecer uma solução apropriada, que pode incluir um diálogo com o desenvolvedor responsável pelo último *"Check-in"*, considerando que essa informação fica registrada no Odyssey-VCS. Os desenvolvedores poderiam, juntos, encontrar uma solução que atendesse aos objetivos de ambos.

4.6 O Odyssey-VCS na Gerência de Configuração de Componentes

Esta seção apresenta como o Odyssey-VCS pode ser utilizado para apoiar o DBC. BRAGA (2000) argumenta que o DBC é fundamentalmente uma evolução do paradigma OO, sugerindo que semelhanças podem ser identificadas quando as duas abordagens são analisadas.

Uma classe, desde que bem projetada, deve possuir alta coesão, baixo acoplamento, restringir o acesso a suas propriedades primitivas, ocultar a forma como seus métodos são implementados, e, principalmente, interagir com outras classes através de interfaces bem definidas, expressas na assinatura de suas operações. Com algumas extensões e adaptações, estas características podem ser atribuídas a um componente.

O paradigma OO foi proposto para, entre outras vantagens, oferecer melhores condições para a reutilização de software. É neste contexto, no entanto, que as diferenças entre uma classe e um componente aparecem de forma mais acentuada. Uma classe é desenvolvida de modo que agrupe propriedades e comportamentos relativos a um conceito que está contido nos limites de uma aplicação específica. Diferentemente, um componente é projetado para atuar em um domínio de aplicações, e, para realizar essa tarefa, agrega um conjunto de outros artefatos, como classes, casos de uso, pacotes, código-fonte e código-binário.

Em virtude dessa característica agregadora, um componente deve ser tratado como um elemento único, principalmente nas atividades relacionadas à manutenção, onde a modificação de um elemento deve refletir no componente que o contém. Neste contexto, a abordagem proposta nesta dissertação é bastante apropriada ao DBC, porque trata cada elemento como uma entidade autônoma.

Diante disso, a Seção 4.6.2 descreve como o Odyssey-VCS foi utilizado em combinação com a abordagem para detecção de rastros de modificação, proposta por DANTAS (2005), para auxiliar as atividades relativas ao DBC. O auxílio é essencialmente voltado para as atividades de manutenção de artefatos de análise e projeto, porque é grande o número de abordagens e ferramentas disponíveis direcionadas para o código-fonte. No entanto, antes de descrever a forma como as abordagens atuam em conjunto, é necessário entender a abordagem de DANTAS (2005), que é apresentada na próxima seção.

4.6.1 Detecção de rastros de modificação entre elementos

Quanto uma nova funcionalidade deve ser implementada, uma das primeiras dúvidas que surge é: "Quais artefatos devem ser modificados para que a funcionalidade seja implementada?" Consultas aos diagramas de caso de uso, de classes e de seqüência podem ajudar o desenvolvedor a identificar os elementos a serem modificados.

Uma vez que um grupo de elementos foi preliminarmente identificado e o desenvolvedor inicia sua tarefa, surge uma nova dúvida durante a implementação da funcionalidade: "Quando um grupo de elementos é modificado que outros elementos também são modificados?".

Esse conhecimento existe, de maneira implícita, no repositório de controle de versões criado e gerenciado pelo Odyssey-VCS. A explicitação desse conhecimento pode ser útil para: (i) sugerir modificações futuras; (ii) prevenir erros oriundos de modificações incompletas; (iii) detectar erros colaterais de modificações; (iv) detectar falhas de projeto, devido ao alto acoplamento entre artefatos não correlatos.

DANTAS (2005), utilizando técnicas de mineração, propôs uma abordagem de detecção de elementos que devem ser modificados em conjunto. Dentre os algoritmos de mineração de dados, DANTAS (2005) utilizou a técnica de regra de associação e o algoritmo *Apriori* (AGRAWAL e SRIKANT, 1994).

O algoritmo *Apriori* retorna todos os elementos associados ao artefato que será modificado, calculando, para cada associação, o suporte e a confiança. A medida de suporte indica a co-ocorrência de elementos. Ou seja, a medida indica o percentual de modificações realizadas nos elementos associados, considerando o universo de todas as

modificações existentes no repositório. Por exemplo, de um total de cem (100) modificações já realizadas, vinte (20) foram realizadas sobre os elementos analisados. Neste caso, a medida de suporte é equivalente a vinte por cento (20%).

A medida de confiança indica o percentual de vezes que os elementos são modificados em conjunto. Supondo que um elemento tenha sofrido cem (100) modificações, no entanto, em apenas dez (10) delas, o outro elemento foi modificado em conjunto. Neste caso, a medida de confiança é equivalente a dez por cento (10%).

O tratamento realizado pelo Odyssey-VCS, levando em conta o conhecimento contido na estrutura dos elementos, melhora a atividade de detecção de rastros. Por exemplo, suponha que a classe B seja modificada em oitenta por cento (80%) das vezes que a classe A também é modificada. Este regra pode ser gerada utilizando o repositório do Odyssey-VCS. Ao contrário, utilizando os SCVs atuais, esse conhecimento não pode ser obtido porque o GV é um arquivo do sistema, e somente estes podem aparecer nas relações.

4.6.2 Exemplo de utilização do Odyssey-VCS na Gerência de Configuração de Componentes

A principal diferença entre este exemplo e o descrito na Seção 4.5 é que, nesta seção, o foco da discussão é o DBC, enquanto que naquela seção, o foco é o paradigma OO. O exemplo, extraído de CHESSMAN e DANIELS (2001), está contido no domínio de hotelaria, onde aparecem os elementos principais, como hotel, hóspede, quarto e tipo de quarto. A Figura 36 apresenta os casos de uso, atores e classes extraídos do problema.

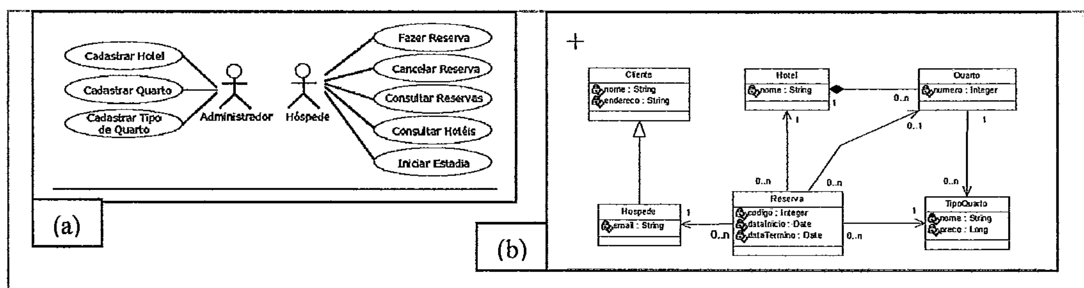


Figura 36. Diagramas de caso de uso (a) e de classes (b) do domínio de hotelaria

O próximo passo consiste em identificar e representar os componentes para este domínio. Utilizando as regras de geração de componentes propostas por TEIXEIRA

(2003), quatro componentes foram identificados: "Gerente de Reserva", "Gerente de Tipos de Quarto", "Gerente de Hotéis" e "Gerente de Hóspedes", Figura 37.

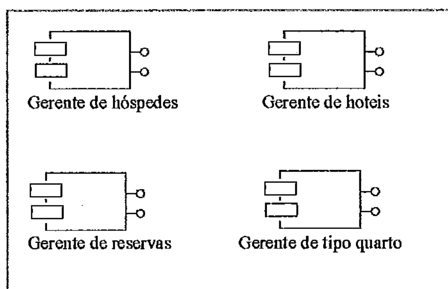


Figura 37. Componentes do exemplo de hotelaria

Uma vez identificados todos os elementos contidos no domínio, deve-se providenciar sua inclusão no repositório de controle de versões. Considerando que estes elementos podem ser representados no formato XMI, qualquer SCV baseado em sistema de arquivos pode ser utilizado. No entanto, a atividade de detecção de rastros, no contexto de DBC, necessita de SCV com modelos de dados mais sofisticados, como o utilizado pelo Odyssey-VCS. A **Figura 38** apresenta os elementos inseridos no repositório do Odyssey-VCS, com o auxílio do Odyssey-VCS *Client*.

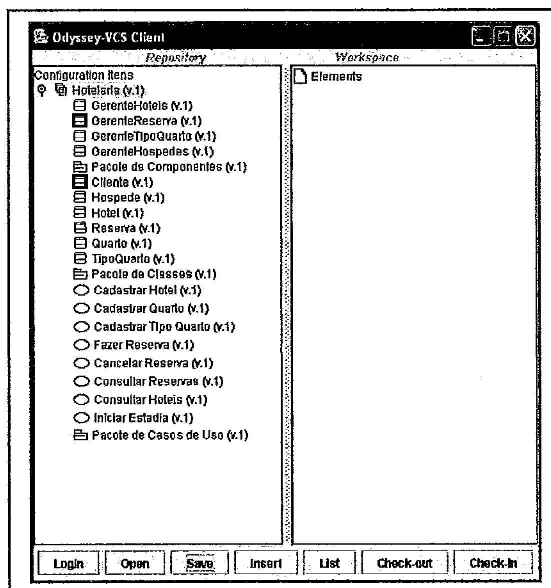


Figura 38: Visualização dos elementos através do Odyssey-VCS Client

A UML 1.4 não possui uma representação para um componente, o que somente ocorrerá com a UML 2.0. Em virtude disso, o Odyssey-VCS trata um componente como uma classe estereotipada, conforme se observa na **Figura 38**. A idéia central da abordagem para detecção de rastros é fornecer ao desenvolvedor informações relativas aos elementos que devem ser modificados em conjunto. Para que isto seja possível, no entanto, é necessário ter um repositório com várias versões dos elementos, sobre as quais serão calculadas as medidas de suporte e confiança. Supondo a necessidade de realizar as modificações a serem aplicadas aos elementos descritas na **Tabela 4**, durante a atividade de "Implementação", vários desenvolvedores devem modificar os elementos em paralelo para atendê-las.

Tabela 4. Descrição das modificações realizadas no exemplo hotelaria

No.	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 realização 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.

Cada modificação, quando implementada, gera um conjunto de versões de elementos. O Odyssey-VCS permite que estes elementos 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*.

Após a aplicação de todas as modificações listadas na Tabela 4, o repositório gerenciado pelo Odyssey-VCS ficará conforme apresentado na Figura 39.

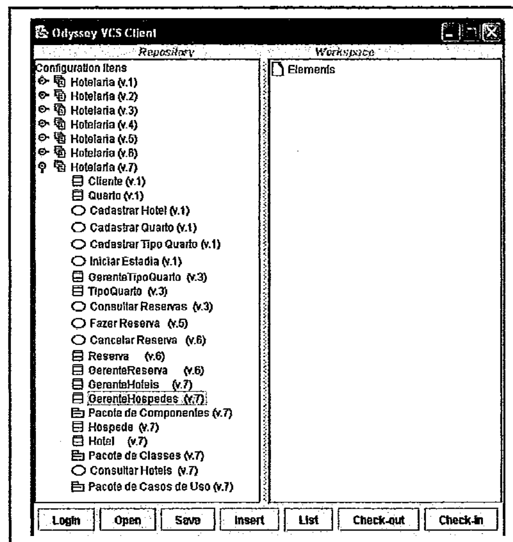


Figura 39: Visualização do repositório após a realização das modificações

Suponha que, neste momento, um desenvolvedor deseje realizar uma modificação na classe “Reserva”. No entanto, antes de iniciar suas atividades, ativou a função de detecção de rastros, para obter as informações relativas aos demais elementos que podem ser modificados em conjunto. A Figura 40 ilustra o resultado da mineração de dados sobre a classe “Reserva” com base no repositório descrito na Figura 39.

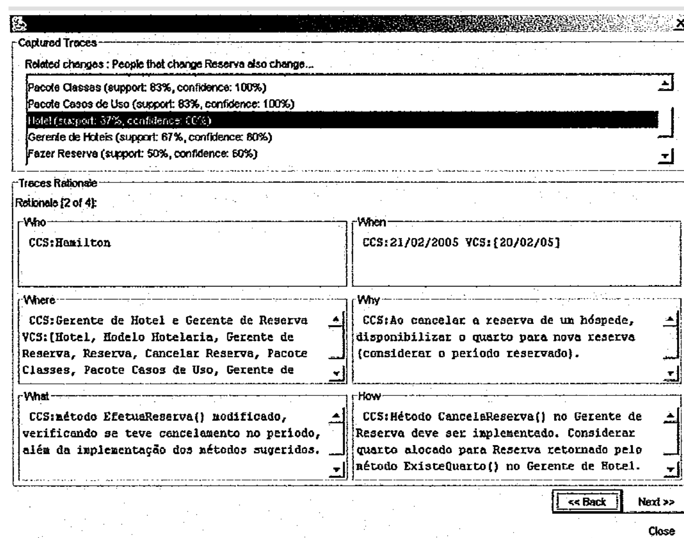


Figura 40: Resultado da detecção dos rastros de modificação

Na parte superior da janela são apresentados os artefatos que devem ser analisados quando a classe “Reserva” é modificada. Juntamente com esses artefatos, são fornecidos

os valores de suporte e confiança, para que o desenvolvedor possa priorizar a análise de forma criteriosa. Analisando a parte inferior da Figura 40, pode-se notar que aparecem informações adicionais relativas ao elemento selecionado, que no caso é a classe "Hotel".

As informações são: "Quem", "Quando", "Onde", "Porque", "O que" e "Como". Elas podem fornecer indícios de como proceder em modificações futuras. Neste exemplo, o desenvolvedor "Hamilton" poderia ser consultado antes que a modificação fosse implementada, visto que ele já modificou esses artefatos no passado e pode dar sugestões importantes para a modificação atual. Essas informações são inseridas no repositório a partir de um sistema de controle de modificações, que consiste de uma dissertação de mestrado em andamento na linha de pesquisa de Reutilização de Software da COPPE/UFRJ.

A qualidade das informações obtidas na detecção dos rastros é fortemente dependente do modelo de dados utilizado pelo SCV. Desta forma, a utilização de um modelo de dados mais sofisticado, como o utilizado pelo Odyssey-VCS, por exemplo, torna-se indispensável quando se trabalha com o DBC, devido à complexidade inerente aos elementos manipulados por este paradigma.

4.7 Considerações finais

Este capítulo descreveu o protótipo do Odyssey-VCS, sua camada de transporte, e um exemplo de utilização através do ambiente Odyssey e da ferramenta *CASE* Poseidon. Este capítulo foi importante para demonstrar que as idéias discutidas no Capítulo 3 são viáveis de serem implementadas, através de tecnologias disponíveis. Dessa forma, é possível dizer que o protótipo atendeu satisfatoriamente aos requisitos definidos no Capítulo 1, que são: (i) granularidade fina; (ii) não-intrusão; (iii) compatibilidade com ADSs e ferramentas *CASE* existentes; (iv) flexibilidade; (v) acesso concorrente; e (vi) distribuição.

Entretanto, algumas limitações do protótipo foram identificadas, dentre elas:

1. A forma de visualização dos ICs, através da opção "Listar", não considera a estrutura dos elementos da UML. Por exemplo, o IC "nome", que é um atributo, está contido no IC "Hospede", que é uma classe. Seria interessante que esse conhecimento da estrutura fosse observado na criação da árvore. No entanto, esta árvore é criada com base

no meta-modelo de versionamento. Uma solução possível seria consultar os elementos da UML, no momento da construção da árvore de ICs, de modo que a mesma pudesse ser construída corretamente.

2. O Odyssey-VCS não trata questões referentes a quedas na rede, quando a operação de "*Check-in*" está sendo realizada. Neste cenário, o conceito de transação, similar ao utilizado pela comunidade da banco de dados, poderia ser adotado. Ou seja, uma vez iniciada a operação, que é interrompida por uma queda na rede, todas as modificações aplicadas sobre o repositório por conta dessa transação teriam que ser desfeitas.

3. Quando um desenvolvedor solicita "*Check-in*", no ambiente Odyssey, o Odyssey-VCS envia os elementos, os quais são recebidos pelo Odyssey-VCSPlugin. Este constrói uma nova árvore de elementos do Odyssey, sem considerar a possibilidade da existência de uma árvore já criada. Aqui um procedimento possível seria aquele utilizado pela ferramenta LockED, descrita no Capítulo 2. O LockED realiza o procedimento de junção no ambiente Odyssey, ao invés de criar uma nova árvore.

Capítulo 5 - Conclusão

5.1 Visão geral

Grande parte dos erros encontrados nos produtos de software entregues aos clientes são frutos de análise e projeto mal conduzidos. Para piorar o cenário, o custo de correção de uma falha no código-fonte é 60 a 100 vezes superior, caso a mesma falha fosse identificada e sanada na fase de análise (PRESSMAN, 2001). Portanto, uma abordagem de desenvolvimento orientada a modelos contribui para a redução do custo do software.

Como qualquer outro artefato de software, os modelos de análise e projeto são constantemente modificados, seja porque melhorou o entendimento dos desenvolvedores em relação ao problema, ou porque os usuários solicitaram a adição de novas funcionalidades. Por isso, são necessários SCVs para automatizar a evolução dos artefatos de análise e projeto.

Para atender a esse propósito, essa dissertação descreveu uma abordagem de controle de versões para elementos da UML. A abordagem foi pautada por seis requisitos, que são: (i) granularidade fina; (ii) não-intrusão; (iii) compatibilidade com ADSs e ferramentas *CASE* existentes; (iv) flexibilidade; (v) acesso concorrente; e (vi) distribuição.

5.2 Contribuições

Como contribuições desse trabalho, pode-se destacar:

1. uma proposta de abordagem que permite versionar elementos da UML, numa granularidade fina, de modo que seja possível definir o GV e o GC, de acordo com as necessidades da equipe de desenvolvimento. Esse tipo de flexibilidade, permitindo selecionar os ICs com base no tipo do elemento da UML, não é encontrado na literatura.

2. A abordagem de DANTAS (2005), descrita na Seção 4.6.1, somente foi possível a partir da existência do repositório versionado de elementos, criado e mantido pelo Odyssey-VCS. É importante ressaltar que a manutenção do repositório, utilizando uma granularidade fina, melhora a atividade de detecção.

3. Conforme descrito na Seção 4.6, é possível identificar os componentes a serem modificados em conjunto. Quando um componente deve ser modificado, um passo importante diz respeito à análise de impacto. Desta forma, é possível verificar que outros componentes devem ser analisados, colaborando para uma definição menos subjetiva de prazos e orçamentos. Ou seja, o controle individual da evolução dos componentes, descrito nesta dissertação, contribui para a melhora na atividade de análise de impacto.

4. Nos SCVs atuais, a manipulação de um IC é dependente de sua estrutura. Por exemplo, as operações disponibilizadas para manipular uma configuração (um conjunto de ICs) são diferentes das operações utilizadas para manipular um arquivo do sistema operacional, o que se traduz em aumento da complexidade na utilização dos SCVs, por parte dos usuários. No Odyssey-VCS, todos os ICs, que pode ser um componente, um modelo, uma classe, uma operação, ou um atributo, recebem o mesmo tratamento, o que contribui para a obtenção da metáfora uniforme no contexto dos SCVs.

5. A implementação de uma ferramenta de controle de versões, no contexto do Ambiente Odyssey (WERNER *et al.*, 2003), capaz de manipular elementos de modelo da UML. Esta ferramenta foi desenvolvida utilizando tecnologias novas e promissoras, como *Meta-Object Facility* (MOF) e *Java Metadata Interface* (JMI).

Apesar de entender que a abordagem descrita nesta dissertação torna o SCV mais próximo das necessidades de ambientes de desenvolvimento modernos, algumas limitações podem ser destacadas:

1. Um SCV que atua no cenário de DBC deve ser capaz de diferenciar modificações realizadas por equipes produtoras daquelas realizadas por equipes consumidoras. Esta diferenciação é importante porque modificações realizadas por equipes produtoras estão relacionadas à evolução do domínio e podem ser interessantes para todos os consumidores dos componentes. Por outro lado, modificações realizadas por equipes consumidoras podem estar mais relacionadas à necessidade específica de uma aplicação e não serem relevantes para os demais consumidores do componente em questão. Atualmente, o Odyssey-VCS não é capaz de fazer essa diferenciação.

2. No DBC, é interessante que o controle das modificações seja direcionado para as interfaces dos componentes. Desta forma, seria possível identificar uma versão

específica de uma interface e então obter todos os componentes que implementam aquela versão. No entanto, a abordagem descrita nesta dissertação atua no nível dos elementos que compõem os componentes, ignorando a importância das interfaces no DBC.

3. Quando se atua em cenários de equipes grandes, a escalabilidade é um ponto importante para qualquer sistema, e isto não é diferente para os SCVs (ESTUBLIER, 2000). Para solucionar o problema da distribuição, foi utilizada uma arquitetura cliente/servidor com acesso remoto ao repositório centralizado. Essa solução foi adotada, em oposição à arquitetura de repositórios distribuídos, em função da facilidade de implementação. HOEK *et al.* (1996) e ESTUBLIER *et al.* (2002), no entanto, argumentam que a arquitetura cliente/servidor não é adequada para atuar em cenários distribuídos porque esta solução sofre problemas de escalabilidade.

4. Uma outra limitação da abordagem refere-se à gerência da consistência dos artefatos após a realização da junção. Se a operação de junção resultar num estado inconsistente, os artefatos serão armazenados dessa forma. Como inconsistência, entende-se que um elemento da UML desrespeita alguma regra de boa formação, definida no âmbito da linguagem.

Além do desenvolvimento de possíveis soluções para tratar essas limitações, os seguintes trabalhos futuros foram identificados:

1. Considerando a existência de dois componentes (Provedor e Consumidor), onde o Componente Consumidor utiliza os serviços do Componente Provedor, existe a necessidade de identificar se uma nova versão do Componente Consumidor deve ser criada quando modificações forem aplicadas ao Componente Provedor. Para isso, deve ser possível verificar se as modificações aplicadas ao componente implicaram também na modificação dos serviços fornecidos pelo mesmo. Em caso afirmativo, uma nova versão do Componente Consumidor desses serviços deve ser gerada, caso contrário, as modificações afetaram apenas os artefatos internos do componente e nenhuma versão do Componente Consumidor deve ser gerada. A dificuldade aqui encontra-se na verificação dos aspectos sintáticos e semânticos relativos aos serviços fornecidos. Neste cenário, as interfaces adquirem um papel mais relevante na evolução dos componentes.

2. O mecanismo de *release* (liberação) é utilizado para representar um conjunto de ICs, bem como suas versões específicas, que foram entregues a um ou mais

clientes. Este recurso é interessante quando se deseja recuperar o conjunto de todas as versões dos ICs entregues a um cliente específico, seja para adição de uma funcionalidade ou correção de *bugs*. Quando se trabalha com componentes, seria interessante poder criar *releases* de componentes e, desta forma, identificar todos os componentes, bem como suas versões, que estão sendo utilizados por uma aplicação específica. Além disso, torna-se possível identificar todas as aplicações contidas no domínio que estão fazendo uso de uma versão específica do componente. Isto seria útil na realização da análise de impacto, na operacionalização de notificação e substituição de componentes antigos pelas novas versões disponibilizadas.

3. Existe um consenso na comunidade de GCS de que é extremamente difícil projetar e implementar um sistema de controle de versões do "zero" (LINGEN e HOEK, 2004; CONRADI e WESTFECHTEL, 1998). Neste cenário, o Odyssey-VCS poderia ser estendido para versionar outros artefatos como documentos em XML e código-fonte Java, além da UML. A estrutura de tratadores adotada pela abordagem poderia ser utilizada para versionar artefatos tão distintos como XML e código-fonte Java, com a mesma habilidade. Uma vantagem dessa abordagem é que um único SCV e, conseqüentemente, um único repositório seria utilizado para gerenciar todo o ciclo de vida do software, desde a especificação de requisitos até o código-fonte, o que eleminaria problemas de integração de dados localizados em diferentes repositórios. Com isto, é possível identificar rastros de modificação a partir do requisito funcional para o código-fonte.

4. Implementar consultas aos ICs constantes no repositório utilizando *Object Constraint Language* (OCL) (OMG, 2005e). A utilização desta linguagem é favorecida pela adoção do MOF como elemento central desta abordagem, já que OCL também é baseada no MOF. Uma outra utilização possível para OCL, no contexto dessa abordagem, poderia ser na atividade de gerência da consistência, que é atualmente uma limitação da abordagem.

5. Finalmente, a realização de um estudo de caso controlado, visando identificar as reais melhorias que uma abordagem como esta proporciona para a equipe de desenvolvimento. Por exemplo, um estudo poderia ser destinado a identificar qual o GV e GC mais adequado para cada tipo de projeto, ou mesmo se os desenvolvedores

realmente iriam utilizar e se beneficiar de uma abordagem flexível, conforme proposto nesta dissertação. Perda ou ganho de desempenho em virtude da não utilização de deltas poderia também ser avaliado através de um estudo mais detalhado.

Referências bibliográficas

AGRAWAL, R., SRIKANT, R., 1994, "Fast Algorithms for Mining Association Rules in Large Databases". In: *International Conference on Very Large Data Bases*, pp. 487-499, Santiago de Chile, Chile, September.

APACHE, 2005a, *The Apache HTTP Server Project*. In: http://httpd.apache.org/, acessado em 06/04/2005.

APACHE, 2005b, *The Web Service Project Apache*. In: http://ws.apache.org/, acessado em 06/04/2005.

BARISH, G., 2002, *Building Scalable a High Performance Java Web Applications Using J2EE Technology*, 2a. ed., Addison-Wesley

BERSOFF, E.H., HENDERSON, V., SIEGEL, S., 1980, *Software Configuration Management*. 1 ed., Prentice-Hall.

BOOCH, G., RUMBAUGH, J., JACOBSON, J., 2000, *UML – Guia do Usuário*. 9 ed. Rio de Janeiro, Campus.

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

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

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

CHIEN, S., TSOTRAS, V. J., ZANIOLO, C., 2001, "Efficient Management of Multiversion Documents by Object Referencing". In: *XXVII International Conference on Very Large Data Bases*, pp. 291-300, Roma, Italy, September.

CLELAND H. J., CHANG, C.K., 2003, "Event-Based Traceability for Managing Evolutionary Change". In: *IEEE Transactions on Software Engineering*, pp. 796-810, September.

CONRADI, R., WESTFECHTEL, B., 1998, "Version Models for Software Configuration Management". In: *ACM Computing Surveys*, vol. 30, pp. 232-282, June.

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

DANTAS, C. R., 2005, *Odyssey-WI:Uma Abordagem para Mineração de Rastros de Modificações em Repositório Versionados*. M.Sc., COPPE/UFRJ, Rio de Janeiro, Brasil.

DANTAS, A. R., VERONESE, G. H., CORREA, A., XAVIER, J. R., WERNER, C. M. L., 2002, *Suporte a Padrões no Projeto de Software*. Caderno de Ferramentas do XVI Simpósio Brasileiro de Engenharia de Software, Gramado, Rio Grande do Sul, Brasil, Outubro.

DART, S., 1999, "Content Change Management: Problems for Web Systems". In: *Proceedings IX International Workshop on Software Configuration Management*, pp. 1-17, Toulouse, France, September.

DELTA-V, 2002, *Versioning Extensions to WebDAV*. Web Distributed Authoring and Versioning.

DYRESON, C., Lin, H., Wang, Y., 2004, "Managing Versions of Web Documents in a Transaction-time Web Server". In: *XXX International World Web Web Conference*, New York, USA, May.

EL-JAICK, D., 2004, "MIMIX: Sistema de Apoio à Modelagem Cooperativa de Software Utilizando Ferramentas *CASE* Heterogêneas", M.Sc., COPPE/UFRJ, Rio de Janeiro, Brasil.

ESTUBLIER, J., CASALLAS, R., 1994, *Configuration Management*. J. Wiley and Sons, England.

ESTUBLIER, J., 2000, "Software Configuration Management: a Roadmap". In: *Proceedings of 22nd International Conference on Software Engineering, The Future of Software Engineering*, pp. 279-289, Limerick, Ireland, June.

ESTUBLIER, J., GARCIA, S., VEGA, G., 2003, "Defining and Supporting Cuncurrent Engineering Policies in SCM". In: *Proceedings of the XI International Workshop on Software Configuration Management (SCM-11)*, Portland, Oregon, USA, May.

ESTUBLIER, J., LEBLANG, D., CLEMM, G., et. al., 2002, "Impact of the Research Community on The Field of Software Configuration Management". In: *Software Configuration Notes (ACM)*, v. 27, pp. 31-39, Orlando, Florida, September.

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

FRANKEL, D. S., 2005, "BPM and MDA: A Rise of Model-Driven Enterprise Systems". *Whitepaper*, June, http://www.bptrends.com/publicationfiles/Frankel11.pdf, acessado em 06/04/2005.

GULLA, B., KARLSSON, E., YEH, D., 1991, "Change-oriented Version Descriptions in EPOS". In: *Software Engineering Journal,* v. 6, pp. 378-386, November.

GENTLEWARE, 2005, *Poseidon for UML,* In: http://www.gentleware.com/index.php, acessado em 06/04/2005.

GOLDSTEIN, I. P., BOBROW, D. G., 1980, *A Layered Approach to Software Design.* In: Tech. Rep. CSL.-80-5, XEROX PARC, Palo Alto, CA.

HABERMANN A. N., NOTKIN D., 1986, "Gandalf: Software Development Environments". In: *Transactions on Software Engineering,* pp. 1117-1127, December.

HASSAN, A.E., HOLT, R.C., 2003, "ADG: Annotated Dependency Graphs for Software Understanding". In: *Visualizing Software For Understanding And Analysis (VISSOFT),* Amsterdam, Netherlands, September.

HARTER, R., 1981, "Some Thoughts on Source Update as a Software Maintenance Tool". In: *IEEE Conference on Trends and Applications,* pp. 163-171, May.

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

HNETYNKA, P., PLÁSIL, F., 2004, "Distributed Versioning Model for MOF" In: *Proceedings of the Winter International Symposium on Information and Communication Technologies (WISICT 2004)",* pp. 489-494, Dublin, Ireland, January.

HOEK, A., HEIMBIGNER, D., WOLF, A. L., 1996, "A Generic, Peer-to-Peer Repository for Distributed Configuration Management". In: *Proceedings of the 18th International Conference on Software Engineering,* Berlin, Germany, March.

HUNT, J. J., REUTER, J., 2001, "Using the Web for Document Versioning: An Implementation Report for DeltaV". In: *XXIII International Conference on Software Engineering,* p. 507, Toronto, Canada, May.

IEEE, 1998, *Std 828 – IEEE Standard for Software Configuration Management Plans.*

JAKARTA, 2005, *Apache Jakarta Tomcat.* In: http://jakarta.apache.org/tomcat/index, acessado em 06/04/2005.

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

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

LINGEN, R., HOEK, A., 2004, "An Experimental, Pluggable Infrastructure for Modular Configuration Management Policy Composition". In: *Proceedings of the Twenty-Sixth International Conference on Software Engineering,* pp. 573-582, Edinburgh, Scotland, United Kingdom, May.

LUCRÉDIO, D., PRADO, A. F., 2004, "Extensão da Ferramenta MVCASE com Serviços Remotos de Armazenamento e Busca de Artefatos de Software". In: *Workshop Cooperação UFSCar – FSA em Ciência da Computação,* pp. 234-241, Santo André, São Paulo, Brasil, Editora Gráfica Suprema.

MATULA, M., 2005, "NetBeans Metadata Repository". *Whitepaper.* In: http://mdr.netbeans.org/ , acessado em 06/04/2005.

MURTA, L. G. P., 1999, *FrameDoc: Um Framework para a Documentação de Componentes Reutilizáveis*", Projeto Final de Curso, DCC/IM, UFRJ, Rio de Janeiro, Brasil.

MURTA, L. G. P., 2002, *Charon: Uma Máquina de Processos Extensível Baseada em Agentes Inteligentes.* M.Sc., COPPE/UFRJ, Rio de Janeiro, Brasil.

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, Maio.

MILER, N., 2000, *A Engenharia de Aplicações no Contexto da Reutilização baseada em Modelos de Domínio,* M.Sc., COPPE/UFRJ, Rio de Janeiro, Brasil.

MVCASE, 2005, "MVCASE". In: http://www.recope.dc.ufscar.br/mvcase/top.html, acessado em 06/04/2005

NAVATHE, E. R., 2000, *Fundamentals of Database Systems.* 3a ed. Addison-Wesley.

NETBEANS, 2004, "Metamodel for Java Language". In: http://java.netbeans.org/models/java/java-model.html, acessado em 06/04/2005.

NGUYEN, T. N., MUNSON, E. V., BOYLAND, J. T., 2004, "The molhado hypertext versioning system". In: *Proceedings of the fifteenth ACM conference on Hypertext & hypermedia,* pp. 185-194, Santa Cruz, California, USA, August.

OHST, D., KELTER, U., 2002, "A Fine-grained Version and Configuration Model in Analysis and Design". In: *Proceedings of the International Conference on Software Maintenance (ICSM'02)",* p. 521, Montreal, Quebec, Canada, October.

OMG, 2005a, *ADAPTIVE/IBM Initial MOF 2.0 Versioning and Development Lifecycle Submission, version 1.0.* Object Management Group

OMG, 2005b, *Meta-Object Facility (MOF) Specification, version 1.3.* Object Management Group.

OMG, 2005c, *XML Metadata Interchange (XMI) Specification, version 1.2.* Object Management Group

OMG, 2005d, *Common Warehouse Metamodel (CWM) Specification, version 1.1.* Object Management Group

OMG, 2005e, *UML 2.0 OCL: Final Adopted Specification.* Object Management Group.

PRESSMAN, R. S., 2001, *Software Engineering, A Practitioner's Approach.* 5 ed., McGraw-Hill.

REICHENBERGER, C., 1994, "Concepts and Techniques for Software Version Control". In: *Software-Concepts and Tools,* v. 15, No. 3, pp. 97-104, July.

RENDER, H., CAMPBEL, R., 1991, "An Object-oriented Model of Software Configuration Management". In: *Proceedings of the 3rd International Workshop on Software Configuration Management,* pp. 127-139, Trondheim, Norway, June.

ROCKIND M. J., 1975, "The Source Code Control System". *IEEE Transactions on Software Engineering,* v. 1, n.4, pp. 364-370.

SEI, *The Software Engineering Institute*, In: http://www.sei.cmu.edu/cmmi/cmmi.html)

SILVA, F. A., COSTA, R. V. C., EDELWEISS, N., SANTOS, C. S., 2003, "Using the Temporal Versions Model in a Software Configuration Management Environment". In: *XVII Brazilian Symposium on Software Engineering,* Manaus, Amazonas, Brazil, October.

SOARES, M. D., FORTES, R. P. M., MOREIRA, D. A., 2000, "VersionWeb: A Tool for Helping Web Pages Version Control". In: *International Conference on Internet and Multimedia Systems and Applications (IMSA),* pp. 275-280, Las Vegas, Nevada, USA, November.

SPINOLA, R. O., KALINOWSKI, M., TRAVASSOS, G. H., 2004, "Uma Infra-estrutura para Integração de Ferramentas *CASE*". In: *XVIII Simpósio Brasileiro de Engenharia de Software*, pp. 147-162, Brasília, Brasil, Outubro.

SUN, 2005, *Java Servlet Technology.* In: http://java.sun.com/products/servlet/, acessado em 06/04/2005.

TEIXEIRA, H. V., MURTA, L. G. P., WERNER, C. M. L., 2001, "LockED: Uma Ferramenta para o Controle de Alterações de Artefatos de Software". In: *IV Workshop Ibero-americano de Engenharia de Requisitos e Ambientes de Software (IDEAS'01)*, pp. 348-359, San José, Costa Rica, April.

TICHY, W. F, 1982, "Design, Implementation, and Evaluation of a Revision Control System". In: *Proceedings of the 6th international conference on Software Engineering*, pp. 58-67, Tokyo, Japan, September.

TICHY, W. F, 1988, "Tools for Software Configuration Management". In: *Proceedings of the International Workshop on Software Version and Configuration Control*, pp. 1-20, Grassau, Germany, January.

TIGRIS.org, 2005a, *Subversion*, In: http://subversion.tigris.org, acessado em 19/04/2005

TIGRIS.org, 2005b, *ArgoUML*, In: http://argouml.tigris.org, acessado em 19/04/2005

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, Brasil.

VITALI, F., 1999, "Versioning Hypermedia". In: *ACM Computing Surveys (CSUR)*, v. 31, No. 24, ISSN:0360-0300, December.

W3C, 1999, *Hypertext Transfer Protocol (HTTP) version 1.1*. World Wide Web Consortium

W3C, 2001, *XML Linking Language (XLink) version 1.0*. World Wide Web Consortium

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

WebDAV, 1999, *HTTP Extensions for Distributed Authoring*. Web Distributed Authoring and Versioning.

WEBER, D. W., 1997, "Change Sets Versus Change Packages: Comparing Implementations of Change-Based SCM". In: *VII International Workshop on Software Configuration Management*, LNCS 1235, Boston, USA, May.

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

WHITEHEAD, E. J., 1999, "A Network Protocol for Remote Collaborative Authoring on the Web". In: *Proceedings of the Sixth European Conference on Computer Supported Cooperative Work,* Copenhagen, Denmark, September.

WHITEHEAD, E. J., 2000, "An Analysis of the Hypertext Versioning Domain". D.Sc., Irvine, Califórnia, USA.

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