GERÊNCIA DE CONFIGURAÇÃO NO
DESENVOLVIMENTO BASEADO EM COMPONENTES

Leonardo Gresta Paulino Murta

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 DOUTOR EM CIÊNCIAS EM ENGENHARIA DE SISTEMAS E COMPUTAÇÃO.

Aprovada por:

Prof.ª Cláudia Maria Lima Werner, D.Sc.

Prof.ª Ana Regina Cavalcanti da Rocha, D.Sc.

Prof. Arndt Von Staa, Ph.D.

Prof. Márcio de Oliveira Barros, D.Sc.

Prof. José Carlos Maldonado, D.Sc.

RIO DE JANEIRO, RJ – BRASIL
OUTUBRO DE 2006

MURTA, LEONARDO GRESTA PAULINO

Gerência de Configuração no Desenvolvimento Baseado em Componentes [Rio de Janeiro] 2006

XVI, 213 p., 29,7 cm (COPPE/UFRJ, D.Sc., Engenharia de Sistemas e Computação, 2006)

Tese – Universidade Federal do Rio de Janeiro, COPPE

1. Gerência de Configuração de Software
2. Desenvolvimento Baseado em Componentes

I. COPPE/UFRJ II. Título (série)

Aos meus pais José Múcio e Renilde,
à minha noiva Vanessa e
ao querido amigo Antônio Carlos Motta (*in memorian*).

Agradecimentos

A Deus, pois sem ele nada seria possível.

Aos meus pais José Múcio e Renilde, e à minha noiva Vanessa, que estiveram comigo em cada passo, me dando forças para continuar e amor para enxergar o que realmente importa... Vocês são tudo para mim!

À professora Cláudia Werner, por ter se dedicado tanto e por tanto tempo à minha formação, por ter tido paciência comigo nos meus momentos de questionamento, por ter confiado em mim nos meus primeiros passos como professor... Enfim, considere as minhas vitórias como suas, pois sem dúvida você é a maior responsável pela minha formação profissional.

Ao professor André van der Hoek, por ter me orientado com tanta dedicação durante o estágio de doutorado na Universidade da Califórnia, em Irvine, e por ter me guiado posteriormente na escrita de artigos em inglês.

À professora Ana Regina Rocha, por ter aberto as portas de empresas para que eu pudesse confrontar as pesquisas com a realidade de Gerência de Configuração no Brasil. Sem o seu apoio e incentivo a minha formação sem dúvida não teria sido a mesma.

Ao professor Márcio Barros, que foi de extrema importância em toda a minha jornada desde a graduação. O seu exemplo, como pesquisador, professor e sobretudo amigo, tem sido muito valioso para me nortear nessa jornada.

Ao professor Arndt Von Staa, por ter participado do meu exame de qualificação e por ter aceitado participar desta banca. Ao professor José Carlos Maldonado, por ter aceitado participar desta banca.

Aos professores da UFRJ, que me guiaram nesses últimos 12 anos através dos caminhos da ciência, mostrando como agir e como pensar. Em especial ao professor

Guilherme Travassos e à professora Marta Mattoso, pelo carinho que sempre tiveram comigo e que também tiveram com a Vanessa.

À professora Heloisa e aos professores da Universidade da Pensilvânia, por terem me ajudado a aprimorar o inglês.

A todos os integrantes do Projeto Odyssey, que sempre contribuíram muito para a minha formação. Em especial a Cristine Dantas, Hamilton Oliveira, Luiz Gustavo Lopes e Anderson Marinho, por terem sido sempre tão dedicados nas atividades relacionadas ao Odyssey-SCM e por terem relevado as minhas diversas falhas.

A toda a equipe da Instituição Implementadora MPS.BR da COPPE, em especial ao Ahilton Barreto, pelas diversas discussões sobre a melhor forma de implementar Gerência de Configuração em empresas.

A Roberto Silva Filho, Brian, Xiwen Zhang, Mirella Moro, André Nacul, Márcio Dias, Leila Naslavsky, Cleidson Souza, Rogério de Paula e Wayne, por terem me tratado tão bem em Irvine.

A todos os meus parentes que, desde pequenino, estavam torcendo por mim. Em especial a Romolo, Sérvio e Adeliza, por sempre estarem por perto, adicionando muita alegria à jornada; aos meus afilhados Remo, Yuri e Gustavo, por compreenderem as minhas ausências; e a tia Clísia e Adriana, pela companhia nas viagens para conferências.

Aos meus amigos, que foram compreensíveis em relação às minhas ausências no decorrer do doutorado.

A Lucimar e Sônia, por cuidarem para que eu tivesse um ambiente propício para desenvolver a minha tese, e à Ifigênia, por ter ajudado na minha criação.

À CAPES, pelo apoio financeiro tanto no Brasil quanto nos Estados Unidos, durante o estágio de doutorado (BEX0323/04-7), e ao PESC e à ACM SIGSOFT, pelo apoio financeiro para apresentação de artigos em conferências.

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

GERÊNCIA DE CONFIGURAÇÃO NO DESENVOLVIMENTO BASEADO EM COMPONENTES

Leonardo Gresta Paulino Murta

Outubro/2006

Orientadora: Cláudia Maria Lima Werner

Programa: Engenharia de Sistemas e Computação

O paradigma de Desenvolvimento Baseado em Componentes (DBC) fornece mecanismos para lidar com a complexidade crescente do desenvolvimento de software. No DBC, componentes são elementos fundamentais, que servem como unidade de encapsulamento e podem ser reutilizados ou substituídos, alavancando a produtividade e a qualidade. Contudo, componentes evoluem com o passar do tempo.

A Gerência de Configuração de Software (GCS) é uma disciplina para controlar a evolução de sistemas de software. Os sistemas existentes de GCS foram intencionalmente projetados para serem genéricos, evitando apoio a linguagens específicas e atuando diretamente sobre arquivos. Entretanto, essa estratégia leva à falta de apoio a elementos arquiteturais e de projeto de alto nível, utilizados no DBC.

Este trabalho apresenta o Odyssey-SCM, uma infra-estrutura integrada de GCS para o DBC. Essa infra-estrutura é composta por um processo de GCS adaptado ao DBC; um sistema de controle de modificações customizável, fortemente integrado com um sistema de controle de versões flexível para elementos de modelo UML em granularidade fina; um mecanismo para estabelecer e evoluir ligações de rastreabilidade entre elementos arquiteturais de alto nível e código fonte; um mecanismo para implantar componentes por demanda; e um mecanismo que aplica mineração de dados sobre o repositório integrado de GCS para detectar rastros de modificação entre os elementos de modelo UML versionados.

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

CONFIGURATION MANAGEMENT APPLIED TO
COMPONENT BASED DEVELOPMENT

Leonardo Gresta Paulino Murta

October/2006

Advisor: Cláudia Maria Lima Werner

Department: Computer and Systems Engineering

The Component-based Development (CBD) paradigm provides ways for dealing with the increasing complexity of software development. In CBD, components are first class elements, which work as encapsulation units and can be reused or replaced, leveraging productivity and quality. However, components evolve overtime.

Software Configuration Management (SCM) is a discipline for controlling the evolution of software systems. The current SCM systems were intentionally designed to be generic, avoiding language-specific support and directly dealing with files. This strategy leads to lack of support to high level architectural and design elements, used in CBD.

This work presents Odyssey-SCM, an integrated SCM infrastructure for CBD. This infrastructure is composed by a SCM process tailored to CBD; a customizable change control system tightly integrated with a flexible version control system for fine-grained UML model elements; a mechanism for establishing and evolving traceability links among high level architectural elements and source code; a mechanism for deploying components on demand; and a mechanism that applies data-mining over the integrated SCM repository to discover change traces among versioned UML model elements.

Índice

Capítulo 1 – Introdução 1
1.1 Preâmbulo 1
1.2 Motivação 2
1.3 Problema 2
1.4 Contexto 4
1.5 Objetivo 5
1.6 Organização 6
Capítulo 2 – Gerência de Configuração de Software 8
2.1 Introdução 8
2.2 Processos de GCS 12
2.2.1 IEEE Std 1042 13
2.2.2 IEEE Std 828 15
2.2.3 ISO 10007 17
2.2.4 CMM e CMMI 18
2.2.5 MPS.BR 20
2.2.6 Considerações finais sobre processos de GCS 21
2.3 Sistemas de controle de modificações 21
2.3.1 GCCS 23
2.3.2 ConfigCASE e SoSoft 23
2.3.3 GConf 24
2.3.4 Análise de Impacto 26
2.3.5 Considerações finais sobre sistemas de controle de modificações 27
2.4 Sistemas de controle de versões 27
2.4.1 TVM 29
2.4.2 DVM 30
2.4.3 MIMIX 30
2.4.4 Versionamento de XML 31
2.4.5 MCCM 32
2.4.6 Políticas de Controle de Objetos 33
2.4.7 Considerações finais sobre sistemas de controle de versões 34
2.5 Sistemas de gerenciamento de construção 34

2.5.1 Tinderbox 36
2.5.2 Automação de testes 36
2.5.3 Considerações finais sobre sistemas de gerenciamento de construção 37
2.6 Integração dos espaços de trabalho de GCS 37
2.6.1 Mineração de informações 39
2.6.2 Bloof 40
2.6.3 Considerações finais sobre integração dos espaços de trabalho de GCS 41
2.7 Considerações finais 41
Capítulo 3 – Gerência de Configuração de Software no Desenvolvimento Baseado em Componentes 43
3.1 Introdução 43
3.2 Processos de GCS no DBC 44
3.2.1 MwR 45
3.2.2 KobrA 47
3.2.3 Considerações finais sobre processos de GCS no DBC 49
3.3 Sistemas de controle de modificações no DBC 49
3.3.1 TERRA 50
3.3.2 KobrA 52
3.3.3 Considerações finais sobre sistemas de controle de modificações no DBC 53
3.4 Sistemas de controle de versões no DBC 54
3.4.1 RCM 55
3.4.2 KobrA 56
3.4.3 xADL 57
3.4.4 Considerações finais sobre sistemas de controle de versões no DBC 59
3.5 Sistemas de gerenciamento de construção no DBC 59
3.5.1 Navegador de dependências 61
3.5.2 CMPro 62
3.5.3 JBCM 63
3.5.4 Variabilidade ao longo do ciclo de vida 64
3.5.5 Considerações finais sobre sistemas gerenciamento de construção no DBC 65
3.6 Integração dos espaços de trabalho de GCS no DBC 65
3.6.1 JBuilder 66
3.6.2 XDoclet 67
3.6.3 Ménage 68

3.6.4 Considerações finais sobre integração dos espaços de trabalho de GCS no DBC ... 69
3.7 Considerações finais ... 69
Capítulo 4 – O Projeto Odyssey ... 71
4.1 Introdução ... 71
4.2 As suas origens ... 71
4.3 O seu histórico até 2002 ... 72
4.4 O seu histórico de 2002 a 2006 ... 74
4.5 O seu estado atual ... 75
4.6 Gerência de configuração e o Projeto Odyssey ... 76
4.7 Considerações finais ... 79
Capítulo 5 – A Abordagem Odyssey-SCM ... 80
5.1 Introdução ... 80
5.2 Odyssey-SCMP ... 84
5.2.1 Detalhamento do problema ... 84
5.2.2 Abordagem proposta ... 89
5.2.2.1 Função de identificação da configuração ... 89
5.2.2.2 Função de controle da configuração ... 90
5.2.2.3 Função de contabilização da situação da configuração ... 91
5.2.2.4 Função de avaliação e revisão da configuração ... 92
5.2.2.5 Função de gerenciamento de liberação e entrega ... 93
5.3 Odyssey-CCS ... 93
5.3.1 Detalhamento do problema ... 93
5.3.2 Abordagem proposta ... 94
5.4 Odyssey-VCS ... 97
5.4.1 Detalhamento do problema ... 98
5.4.2 Abordagem proposta ... 101
5.5 Odyssey-BRD ... 104
5.5.1 Detalhamento do problema ... 104
5.5.2 Abordagem proposta ... 106
5.5.2.1 ArchTrace ... 106
5.5.2.2 Carga Dinâmica ... 109
5.6 Odyssey-WI ... 110
5.6.1 Detalhamento do problema ... 110

5.6.2 Abordagem proposta 111
5.7 Considerações finais 113
Capítulo 6 – O Protótipo Odyssey-SCM 116
6.1 Introdução 116
6.2 Exemplo 116
6.3 Odyssey-SCMP 118
6.3.1 Detalhamento da solução 118
6.3.2 Considerações finais sobre o Odyssey-SCMP 122
6.4 Odyssey-CCS 123
6.4.1 Detalhamento da solução 123
6.4.2 Considerações finais sobre o Odyssey-CCS 130
6.5 Odyssey-VCS 131
6.5.1 Detalhamento da Solução 131
6.5.2 Considerações finais sobre o Odyssey-VCS 138
6.6 Odyssey-BRD 139
6.6.1 Detalhamento da solução 140
6.6.1.1 ArchTrace 140
6.6.1.2 Carga Dinâmica 145
6.6.2 Considerações finais sobre o Odyssey-BRD 148
6.7 Odyssey-WI 149
6.7.1 Detalhamento da solução 149
6.7.2 Considerações finais sobre o Odyssey-WI 153
6.8 Considerações finais 154
Capítulo 7 – Avaliação da Abordagem Odyssey-SCM 156
7.1 Introdução 156
7.2 Avaliação do Odyssey-VCS 156
7.2.1 Planejamento do estudo 157
7.2.2 Preparação do ambiente 158
7.2.3 Coleta de estatísticas 160
7.2.4 Execução do estudo 161
7.2.5 Análise dos resultados 161
7.2.6 Considerações finais sobre a avaliação do Odyssey-VCS 165
7.3 Avaliação do Odyssey-BRD 166
7.3.1 Planejamento do estudo 167

7.3.2 Preparação do ambiente 168
7.3.3 Coleta de estatísticas 170
7.3.4 Execução do estudo 170
7.3.5 Análise dos resultados 172
7.3.6 Considerações finais sobre a avaliação do Odyssey-BRD 175
7.4 Avaliação do Odyssey-WI 176
7.4.1 Preparação do ambiente 177
7.4.2 Execução do estudo 178
7.4.3 Análise dos resultados 178
7.4.4 Considerações finais sobre a avaliação do Odyssey-WI 181
7.5 Considerações finais 182
Capítulo 8 – Conclusão 183
8.1 Epílogo 183
8.2 Contribuições 183
8.3 Limitações 186
8.3.1 Integração entre Odyssey-CCS e Odyssey-VCS 186
8.3.2 Integração entre Odyssey-BRD e Odyssey-VCS 187
8.3.3 Evolução da especificação UML 187
8.3.4 Deficiências do repositório MOF adotado 188
8.3.5 Restrições para a carga de componentes por demanda 189
8.4 Trabalhos Futuros 189
8.4.1 GCS embutida na metáfora de DBC 189
8.4.2 Verificação de componentes 189
8.4.3 Visualização das informações 190
8.4.4 Simulações de estratégias de GCS 190
8.4.5 Análise estatística de desempenho do processo de GCS 191
8.4.6 Avaliações adicionais da abordagem 191
Referências Bibliográficas 192

Índice de Figuras

Figura 2.1: Perspectivas de GCS. 11
Figura 3.1: Desenvolvimento de software em vários níveis. 50
Figura 3.2: Reutilização de conjuntos de modificações. 53
Figura 3.3: Dependências para diferentes versões de um mesmo componente. 56
Figura 4.1: Ferramenta Token apoiando a GCS do Projeto Odyssey. 77
Figura 4.2: Ferramenta LockEd apoiando o acesso concorrente a elementos de modelo no ambiente Odyssey. 78
Figura 5.1: Panorama estático do Odyssey-SCM. 82
Figura 5.2: Panorama dinâmico do Odyssey-SCM. 83
Figura 5.3: Processo de DBC convencional. 85
Figura 5.4: Processo de DBC contemplando as atividades de manutenção. 86
Figura 5.5: Propagação de solicitações de modificação do processo de GCS no DBC. 87
Figura 5.6: Esforço envolvido na reutilização de componentes em diferentes níveis de abstração. 88
Figura 5.7: Propagação de solicitações de modificação. 88
Figura 5.8: Modelo UML simplificado de controle de modificações. 95
Figura 5.9: Modelo UML simplificado do mapa de reutilização. 97
Figura 5.10: Unidade de comparação em documento texto e código fonte orientado a objetos. 99
Figura 5.11: Unidade de versionamento em documento texto e código fonte orientado a objetos. 100
Figura 5.12: Unidades de comparação e de versionamento em modelos XMI. 101
Figura 5.13: Relacionamento entre o modelo de versionamento e o modelo de dados. 103
Figura 5.14: Estratégia otimista para o controle de concorrência. 103
Figura 5.15: Visão geral da abordagem ArchTrace. 107
Figura 5.16: Detecção de conjuntos de modificações. 112
Figura 5.17: Rastros de modificação. 112
Figura 6.1: Casos de uso (a), componentes (b) e classes (c) do exemplo de hotelaria. 117
Figura 6.2: Processo Odyssey-SCMP modelado no Odyssey-CCS. 125
Figura 6.3: Formulário de solicitação de modificação modelado no Odyssey-CCS. 126
Figura 6.4: Visão geral da máquina de processos Charon. 127

Figura 6.5: Atividades e decisões pendentes, sendo exibidas pelo Odyssey-CCS..... 128
Figura 6.6: Mapa de reutilização sendo consultado pelo Odyssey-CCS..... 129
Figura 6.7: Visão geral do Odyssey-VCS. 131
Figura 6.8: Exemplo de configuração do comportamento do Odyssey-VCS em termos de unidades de comparação e versionamento..... 133
Figura 6.9: Modificações feitas pelo João no Poseidon. 136
Figura 6.10: Modificações feitas pela Maria no Odyssey. 137
Figura 6.11: Cenário de detecção de conflito durante junção. 137
Figura 6.12: Odyssey (a) acessando o repositório via *plug-in* do Odyssey-VCS (b)... 138
Figura 6.13: Visão geral do ArchTrace. 140
Figura 6.14: Esquema xADL definido pelo ArchTrace. 141
Figura 6.15: API para definição de políticas no ArchTrace. 142
Figura 6.16: Configuração do ArchTrace. 143
Figura 6.17: ArchTrace evoluindo ligações de rastreabilidade do exemplo de hotelaria. 145
Figura 6.18: DTD para descrição de componentes da Carga Dinâmica. 146
Figura 6.19: Parte do descritor de componentes do exemplo de hotelaria. 147
Figura 6.20: Carga Dinâmica de componentes. 147
Figura 6.21: Configuração das medidas de mineração de dados (a) e da coleta de informações de contextualização dos rastros de modificação (b) no Odyssey-WI. 151
Figura 6.22: Odyssey-WI apresentando os rastros de modificação da classe “Reserva”. 152
Figura 7.1: Visão geral do estudo. 159
Figura 7.2: Resultados de *check-out* e *check-in* em relação ao sistema S1. 162
Figura 7.3: Resultados de *check-out* e *check-in* em relação ao sistema S2. 162
Figura 7.4: Número médio de ICs nos espaços de trabalho de S1 e S2. 164
Figura 7.5: Tamanho médio dos espaços de trabalho de S1 e S2. 164
Figura 7.6: Leitura do XMI durante o *check-in*. 165
Figura 7.7: Detecção de ligações de rastreabilidade no ArchTrace. 169
Figura 7.8: Reprodução incremental dos *check-ins*. 171
Figura 7.9: Evolução dos ICs. 172
Figura 7.10: Execuções de políticas. 173
Figura 7.11: Evolução das ligações de rastreabilidade. 173

Figura 7.12: Sumário da análise. 174
Figura 7.13: Curva de precisão x revocação (a) e histograma de precisão-R (b)..... 179
Figura 7.14: Média harmônica entre precisão e revocação em termos de suporte e confiança..... 180

Índice de Tabelas

Tabela 6.1: Modificações realizadas no exemplo de hotelaria. 117
Tabela 6.2. Artefatos modificados no exemplo de hotelaria. 118
Tabela 6.3: Algoritmo de junção do Odyssey-VCS. 135
Tabela 6.4: Políticas implementadas no ArchTrace. 143
Tabela 6.5: Terminologia de mineração de dados. 150
Tabela 6.6: Formulários e campos inspecionados no Odyssey-CCS para a contextualização dos rastros de modificação do exemplo de hotelaria. 152
Tabela 7.1: Sumário das políticas do ArchTrace. 168
Tabela 7.2: Estatísticas do ambiente Odyssey. 169
Tabela 7.3: Sumário das medidas coletadas. 181

Capítulo 1 – Introdução

1.1 Preâmbulo

Durante os primeiros anos de pesquisa nas áreas de Gerência de Configuração de Software[1] (GCS) e de Reutilização de Software[2], o foco era sobre código fonte, armazenado em arquivos do sistema operacional. Uma grande infra-estrutura foi criada nas últimas décadas para apoiar a evolução de código fonte utilizando técnicas de GCS. Durante esse mesmo período, a Reutilização de Software foi aplicada no nível de código fonte por meio de linguagens orientadas a objeto, idiomas, bibliotecas de ligação dinâmica, dentre outras.

Com o passar do tempo, novos paradigmas foram definidos para possibilitar a uniformização dos conceitos usados para representar as entidades do mundo real, passando pelos diversos níveis de abstração abordados pelas atividades de análise, projeto e codificação. Apesar da orientação a objetos atender, de certa forma, a essa demanda, a unidade de encapsulamento utilizada (i.e., classe) é muito fina, dificultando a sistematização da substituição ou reutilização das partes que constituem os sistemas.

O paradigma de Desenvolvimento Baseado em Componentes (DBC) contorna algumas das deficiências detectadas na orientação a objetos no que tange a reutilização de software (PAGE-JONES, 1999). A unidade de encapsulamento utilizada (i.e., componente) é mais grossa, acarretando em um menor acoplamento com os demais componentes e maior coesão dentro do próprio componente. Além disso, os acoplamentos existentes são tratados de forma especial, devido ao uso de interfaces e conectores, visando maximizar a capacidade de substituição das partes constituintes de sistemas. Mais ainda, os componentes, interfaces e conectores podem ser vistos como importantes ferramentas para o tratamento da complexidade crescente do desenvolvimento de software. Além desses aspectos estruturais, o DBC também envolve aspectos metodológicos (JACOBSON *et al.*, 1997), que colocam a reutilização em destaque no processo de desenvolvimento de sistemas.

---

[1] Gerência de Configuração de Software é uma disciplina para o controle da evolução de sistemas de software (DART, 1991).

[2] Reutilização de Software é o processo de criação de sistemas de software a partir de software preexistente (KRUEGER, 1992).

1.2 Motivação

Num cenário de DBC, podem existir, dentro de uma mesma organização, diversas equipes de desenvolvimento de componentes (construção de componentes) e diversas equipes de desenvolvimento com componentes (reutilização de componentes). Além disso, as equipes de desenvolvimento de componentes podem fazer uso de componentes providos por outras equipes de desenvolvimento de componentes, caracterizando equipes híbridas, que atuam concomitantemente nos dois papéis.

Essas diferentes equipes estão intimamente relacionadas no processo de reutilização, onde as equipes de desenvolvimento de componentes atuam como produtoras e as equipes de desenvolvimento com componentes atuam como consumidoras. Nesse processo, as equipes produtoras constroem e mantêm componentes reutilizáveis, enquanto as equipes consumidoras adaptam e reutilizam esses componentes para construir aplicações específicas.

Devido à necessidade intransponível de manter o software após a sua liberação, a reutilização deve ocorrer de forma controlada, possibilitando que a evolução dos componentes reutilizáveis não dificulte o trabalho já complexo de desenvolvimento de componentes e de desenvolvimento com componentes. Esse controle pode ser obtido pelo uso de técnicas de GCS.

Apesar de existir um forte apelo para o uso da GCS durante a etapa de manutenção, a sua aplicação não se restringe somente a essa etapa do ciclo de vida do software (IEEE, 1987; LEON, 2000; CHRISSIS *et al.*, 2003; SOFTEX, 2006b). Durante o desenvolvimento, os sistemas de GCS são fundamentais para prover controle sobre os artefatos produzidos e modificados por diferentes engenheiros de software. Além disso, esses sistemas possibilitam um acompanhamento minucioso do andamento das tarefas de desenvolvimento.

1.3 Problema

O problema tratado nesta tese está relacionado com a falta de apoio das técnicas de GCS existentes ao cenário de DBC. A evolução de artefatos reutilizáveis introduz complexidade extra ao problema tratado pela GCS convencional. Essa complexidade ocorre em virtude da necessidade freqüente de adaptar o artefato antes da sua efetiva reutilização. Contudo, as várias instâncias adaptadas de um artefato são profundamente

semelhantes, e, possivelmente, os mesmos defeitos ou necessidades de melhorias acontecerão em todas essas instâncias.

O retrabalho de manutenção e conseqüentes defeitos introduzidos por esse retrabalho poderiam ser minimizados com a aplicação de técnicas apropriadas de GCS para apoiar a evolução desses artefatos de forma controlada. No cenário de desenvolvimento convencional, usualmente de menor complexidade que o cenário de DBC, é possível detectar aumentos substanciais de qualidade e produtividade devido à adoção de GCS (LEON, 2000). Por essa razão, os benefícios da GCS poderão ser ainda maiores em cenários como o de DBC, onde a complexidade envolvida dificulta a atuação do engenheiro de software na sua tarefa propriamente dita, devido ao aumento de atividades de suporte não automatizadas, e nem sequer definidas em alguns casos.

Contudo, vale ressaltar que a GCS a ser aplicada ao DBC difere em vários aspectos da GCS convencional. Desta forma, este problema pode ser subdividido em cinco partes mais específicas, que são: (1) os processos que guiam a GCS num cenário de DBC; (2) o controle de solicitações de modificação sobre os artefatos, levando em consideração a cadeia de responsabilidades existente no DBC; (3) o controle de versões de artefatos reutilizáveis no nível de abstração do próprio artefato; (4) o gerenciamento de construção, que determina como artefatos se relacionam para estruturar sistemas; e (5) a integração dos espaços de trabalho de GCS e DBC.

Apesar da existência de diversas normas de GCS, intencionalmente, nenhuma delas se preocupa com aspectos de um paradigma específico. Por outro lado, devido ao DBC ser um paradigma relativamente novo, pouco ainda foi feito pela comunidade científica no que diz respeito ao processo de GCS nesse cenário. Contudo, as atividades de GCS, planejadas ou não, são executadas em todos os projetos de desenvolvimento de software (IEEE, 2005). A ausência desse planejamento pode acarretar em perda de eficiência na sua execução.

Uma das características que diferem o processo de GCS convencional do processo de GCS no DBC é a responsabilidade de implementação das solicitações de modificação. No DBC, essa responsabilidade não é tão óbvia quanto no desenvolvimento convencional. Devido a possíveis cadeias de produção de componentes, compostos por outros componentes, essa responsabilidade fica distribuída entre as diversas equipes existentes no ciclo produtivo do componente. Todavia, os sistemas de controle de modificações convencionais não estão aptos para tratar esse problema. Além disso, devido à própria imaturidade das poucas propostas existentes

para o processo de GCS no DBC, esse processo pode precisar ser modificado com freqüência, forçando alterações no próprio sistema de controle de solicitações.

No que tange ao controle de versões, os sistemas convencionais usualmente atuam sobre arquivos do sistema operacional. Entretanto, a maioria das normas de GCS (IEEE, 1987; ISO, 1995a) recomenda uma identificação seletiva dos Itens de Configuração[3] (ICs), que depende das características individuais dos projetos de desenvolvimento de software. Em algumas circunstâncias não é desejado, nem possível, mapear elementos de alto nível de análise e projeto em arquivos individuais, dificultando ou inviabilizando um controle de versões adequado utilizando os sistemas convencionais sobre esses ICs.

Mesmo quando é possível mapear elementos arquiteturais de alto nível para a sua implementação em arquivos do sistema operacional, a manutenção não automatizada desse mapeamento é custosa e suscetível a erros (SETTIMI *et al.*, 2004). Modificações tanto nos próprios elementos arquiteturais quanto na sua implementação podem levar a inconsistências. Os sistemas convencionais de gerenciamento de construção não se preocupam com a evolução do mapeamento entre a arquitetura e a sua implementação, tornando inviáveis diversos usos possíveis desse mapeamento.

Finalmente, um dos grandes desafios de manutenção de software em geral é identificar o impacto de modificações. Essa preocupação ganha maiores dimensões no DBC, pois a introdução de defeitos devido a modificações incompletas afeta não somente um sistema, mas possivelmente todos os sistemas que reutilizam o componente modificado.

1.4 Contexto

Este trabalho de pesquisa está contextualizado no Projeto Odyssey, que visa prover um ambiente de apoio à reutilização baseado em Engenharia de Domínio[4], Linha

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

[4] Engenharia de Domínio é uma abordagem baseada em reutilização para definir o escopo, especificar a estrutura e construir os ativos para uma classe de sistemas, subsistemas ou aplicações. (IEEE, 2004)

de Produtos[5] e DBC. A reutilização de software no ambiente Odyssey (WERNER *et al.*, 2003) ocorre por meio do processo de engenharia de domínio Odyssey-DE (BRAGA, 2000; BLOIS, 2006), que tem por objetivo construir componentes reutilizáveis voltados para domínios de conhecimento específicos, e do processo de engenharia da aplicação Odyssey-AE (MILER, 2000), que tem por objetivo construir aplicações em um determinado domínio, reutilizando os componentes existentes. O ambiente Odyssey utiliza modelos de características[6] e UML (*Unified Modeling Language*) (OMG, 2001) para representar o conhecimento de um domínio, e permite que esses modelos sejam reutilizados para facilitar a construção de aplicações.

1.5 Objetivo

O objetivo deste trabalho consiste em detectar os problemas existentes no DBC que podem ser apoiados por técnicas de GCS e elaborar uma solução, envolvendo processos e ferramental de apoio, para minimizar o efeito dos problemas detectados. Essa meta pode ser decomposta em cinco objetivos, em função dos problemas definidos na Seção 1.3, que são: (1) processos, (2) controle de modificações, (3) controle de versões, (4) gerenciamento de construção e (5) integração dos espaços de trabalho.

No que se refere aos processos, o principal objetivo consiste em estabelecer um processo de GCS voltado para as questões específicas de DBC, tendo como base as principais normas existentes em GCS. O objetivo referente ao controle de modificações é fornecer um sistema configurável que permita a modelagem de processos de GCS e a adaptação desses processos sempre que necessário, apoiando na identificação da responsabilidade de manutenção. Quanto ao controle de versões, o objetivo consiste em apoiar o versionamento de artefatos de alto nível de abstração, utilizando comportamentos configuráveis. Em relação ao gerenciamento de construção, o objetivo é mapear os conceitos existentes na GCS dentro da metáfora do DBC, possibilitando que as funções de GCS possam ser propagadas dos componentes para a sua implementação. Para que isso seja viável, é necessário evoluir consistentemente as

[5] Linha de produtos é um conjunto de sistemas compartilhando características gerenciáveis comuns, que satisfazem às necessidades específicas de um segmento de mercado em particular e que são desenvolvidos sistematicamente, a partir de um conjunto comum de artefatos (CLEMENTS e NORTHROP, 2001).

[6] Características (*features*) são os aspectos de um domínio perceptíveis ao usuário, que definem similaridades e diferenças entre os sistemas existentes nesse domínio (KANG *et al.*, 1990).

ligações de rastreabilidade entre a arquitetura e a sua implementação. Finalmente, o objetivo referente à integração dos espaços de trabalho consiste na interação semitransparente entre GCS e DBC, evitando modificar profundamente as atividades existentes de DBC, contudo, provendo a troca de conhecimento entre as duas áreas por meio da recuperação de informações existentes nos repositórios de GCS.

1.6 Organização

Esta tese está organizada em outros sete capítulos, além deste primeiro capítulo de introdução. Todos os demais capítulos, com exceção dos capítulos 4 e 8, estão organizados nos cinco objetivos apresentados na Seção 1.5. Desta forma, é possível fazer tanto uma leitura transversal, analisando aspectos específicos de cada objetivo, quanto uma leitura incremental, percorrendo os capítulos em seqüência.

O Capítulo 2 apresenta uma introdução à área de GCS, descrevendo as suas perspectivas, as suas funções e os sistemas que dão apoio à execução dessas funções. Além disso, também são apresentadas as principais abordagens e ferramentas existentes na literatura de GCS.

O Capítulo 3 faz uma breve descrição da área de DBC, juntamente com uma revisão da literatura da aplicação de GCS no DBC, discutindo as principais abordagens e ferramentas para esse fim e ressaltando os aspectos já tratados por elas e os aspectos ainda em aberto.

O Capítulo 4 descreve o contexto de desenvolvimento deste trabalho, que é o Projeto Odyssey. São descritos, inicialmente, a origem do projeto e o seu histórico até o início deste trabalho de pesquisa, permitindo a identificação das necessidades naquele momento. Além disso, também são descritos os demais trabalhos executados em paralelo a este trabalho de pesquisa e o estado atual do projeto.

O Capítulo 5 introduz a abordagem proposta por este trabalho de pesquisa, detalhando, para cada objetivo apresentados na Seção 1.5, os problemas específicos a serem tratados e os caminhos escolhidos para solucioná-los. Esse capítulo apresenta, também, uma visão geral da abordagem, que pode ser útil para a compreensão das relações entre as suas partes.

O Capítulo 6 discute os detalhes de projeto e implementação da abordagem proposta por esse trabalho de pesquisa. Para facilitar a compreensão, é definido um exemplo no início do capítulo, e utilizado durante todo o seu decorrer. Além disso, para

cada objetivo apresentado na Seção 1.5, são apresentados tanto os seus detalhes de implementação quanto uma breve comparação com os principais trabalhos relacionados.

O Capítulo 7 apresenta algumas avaliações executadas sobre a abordagem. Apesar da implementação apresentada no Capítulo 6 ser somente um protótipo acadêmico, essas avaliações permitem identificar deficiências e possibilidades de melhoria para uma futura transformação do protótipo em produto, principalmente no que diz respeito a aspectos relacionados ao desempenho do protótipo.

Finalmente, o Capítulo 8 conclui esta tese, apresentando as suas principais contribuições, relatando as limitações detectadas tanto na abordagem quanto no protótipo implementado e enumerando possíveis trabalhos futuros.

Capítulo 2 – Gerência de Configuração de Software

2.1 Introdução

A Gerência de Configuração (GC) surgiu nos anos 50 devido à necessidade da indústria aeroespacial norte-americana controlar as modificações na documentação referente à produção de aviões de guerra e naves espaciais (LEON, 2000; HASS, 2003; ESTUBLIER *et al.*, 2005). Posteriormente, nos anos 60 e 70, a GC passou a abranger artefatos de software, indo além dos artefatos de hardware já estabelecidos e desencadeando o surgimento da Gerência de Configuração de Software (GCS) (CHRISTENSEN e THAYER, 2002).

Apesar do surgimento da GCS nos anos 70, o seu foco era muito restrito às aplicações militares e aeroespaciais, e somente no início dos anos 80, com o surgimento da primeira versão da norma IEEE Std 828 (IEEE, 2005), a GCS foi finalmente assimilada no processo de desenvolvimento de software de organizações não militares (LEON, 2000).

Como toda área de pesquisa, existem diversas definições para GCS. Contudo, a definição mais aceita e utilizada caracteriza a GCS como "uma disciplina que aplica procedimentos técnicos e administrativos para identificar e documentar as características físicas e funcionais de um Item de Configuração (IC), controlar as alterações nessas características, armazenar e relatar o processamento das modificações e o estágio da implementação e verificar a compatibilidade com os requisitos especificados" (IEEE, 1990). Desta forma, a GCS não se propõe a definir quando e como devem ser executadas as modificações nos artefatos de software, papel este reservado ao próprio processo de desenvolvimento de software. A sua atuação ocorre como processo auxiliar de controle e acompanhamento.

A GCS pode ser tratada sob diferentes perspectivas em função do papel exercido pelo participante do processo de desenvolvimento de software (ASKLUND e BENDIX, 2002). Na perspectiva gerencial, a GCS é dividida em cinco funções, que são (IEEE, 2005): identificação da configuração, controle da configuração, contabilização da

situação da configuração, avaliação e revisão da configuração e gerenciamento de liberação[7] e entrega.

A **função de identificação da configuração** tem por objetivo possibilitar: (1) a seleção dos ICs que são os elementos passíveis de GCS; (2) a definição do esquema de nomes e números, que possibilite a identificação inequívoca dos ICs no grafo de versões[8] e variantes[9]; e (3) a descrição dos ICs, tanto física quanto funcionalmente.

A **função de controle da configuração** é designada para o acompanhamento da evolução dos ICs selecionados e descritos pela função de identificação. Para que os ICs possam evoluir de forma controlada, esta função estabelece as seguintes atividades: (1) solicitação de modificação, iniciando um ciclo da função de controle para uma dada manutenção, que pode ser corretiva, evolutiva, adaptativa ou preventiva (PRESSMAN, 2005); (2) classificação da modificação, que estabelece a prioridade da solicitação em relação às demais solicitações efetuadas anteriormente; (3) análise de impacto, que visa relatar os impactos em esforço, cronograma e custo e definir uma proposta de implementação da manutenção; (4) avaliação da modificação pelo Comitê de Controle da Configuração[10] (CCC), que estabelece se a modificação será implementada, rejeitada ou postergada, em função do laudo fornecido pela análise de impacto da modificação; (5) implementação da modificação, caso a solicitação tenha sido aprovada pela avaliação da modificação; (6) verificação da modificação com relação à proposta de implementação levantada na análise de impacto; e (7) atualização da linha base[11], que pode ou não ser liberada para o cliente em função da sua importância e questões de marketing associadas.

---

[7] O termo liberação (*release*) representa a notificação formal e distribuição de uma versão aprovada do software (IEEE, 2005).

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

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

[10] O termo comitê de controle da configuração (*configuration control board*) representa o grupo de pessoas responsável por avaliar e aprovar ou reprovar modificações propostas para ICs e por assegurar a implementação das modificações aprovadas (IEEE, 1990).

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

A **função de contabilização da situação da configuração** visa: (1) armazenar as informações geradas pelas demais funções; e (2) permitir que essas informações possam ser acessadas em função de necessidades específicas. Essas necessidades específicas abrangem o uso de medições para a melhoria do processo, a estimativa de custos futuros e a geração de relatórios gerenciais.

A **função de avaliação e revisão da configuração** ocorre quando a linha base, gerada na função de controle da configuração, é selecionada para ser liberada para o cliente. Suas atividades compreendem: (1) auditoria funcional da linha base, via revisão dos planos, dados, metodologia e resultados dos testes, assegurando que a mesma cumpra corretamente o que foi especificado; e (2) auditoria física da linha base, com o objetivo de certificar que a mesma é completa em relação ao que foi acertado em cláusulas contratuais.

A **função de gerenciamento de liberação e entrega** descreve o processo formal de: (1) construção[12], produzindo ICs derivados a partir de ICs fonte, (2) liberação, identificando as versões particulares de cada IC que serão disponibilizadas, e (3) entrega, implantando os produtos de software no ambiente final de execução.

Contudo, sob a perspectiva de desenvolvimento, a GCS é dividida em três sistemas principais, que são: controle de modificações, controle de versões e gerenciamento de construção.

O **sistema de controle de modificações** é encarregado de executar a função de controle da configuração de forma sistemática, armazenando todas as informações geradas durante o andamento das solicitações de modificação e relatando essas informações aos participantes interessados e autorizados, assim como estabelecido pela função de contabilização da situação da configuração.

O **sistema de controle de versões** permite que os ICs sejam identificados, segundo estabelecido pela função de identificação da configuração, e que eles evoluam de forma distribuída e concorrente, porém disciplinada. Essa característica é necessária para que diversas solicitações de modificação efetuadas na função de controle da configuração possam ser tratadas em paralelo, sem corromper o sistema de GCS como um todo.

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

O **sistema de gerenciamento de construção** automatiza o complexo processo de transformação dos diversos artefatos de software que compõem um projeto no sistema executável propriamente dito, de forma aderente à função de gerenciamento de liberação e entrega. Além disso, esse sistema estrutura as linhas bases selecionadas para liberação, conforme necessário para a execução da função de avaliação e revisão da configuração.

Apesar de existirem essas diferentes perspectivas para abordar a GCS, elas não se relacionam de forma complementar, mas sim, de forma sobreposta. As cinco funções descritas na perspectiva gerencial podem ser implementadas pelos três sistemas descritos na perspectiva de desenvolvimento, acrescidos de alguns procedimentos manuais. Além disso, para que esses recursos sejam efetivamente utilizados, alguns serviços especializados devem ser definidos na integração dos espaços de trabalho de GCS e Ambientes de Desenvolvimento de Software (ADS). A Figura 2.1 exibe essa interação entre as perspectivas.

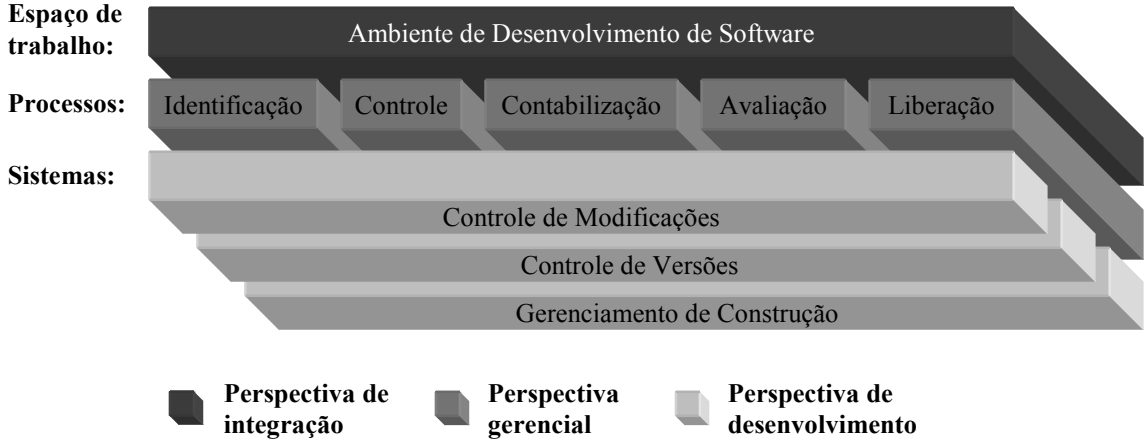


Figura 2.1: Perspectivas de GCS.

Cada sistema descrito na perspectiva de desenvolvimento deve fazer uso de procedimentos próprios para atender às funções descritas na perspectiva gerencial. Por exemplo: solicitações de modificação podem seguir fluxos díspares em diferentes projetos, numerações e rotulação de versões podem ocorrer de diversas formas em função das necessidades específicas de cada organização e a liberação de versões de produção pode depender de fatores como decisões de marketing e grau de qualidade desejada (LEON, 2000). Esses procedimentos são especificados por meio de processos, definidos no âmbito da organização como um todo (processo padrão) ou no âmbito de projetos específicos (processos definidos, adaptados do processo padrão).

Desta forma, uma taxonomia baseada em cinco elementos será utilizada para caracterizar a literatura de GCS, que cobre as diferentes perspectivas levantadas na Figura 2.1. Esses elementos são:

- Processos;
- Sistemas de controle de modificação;
- Sistemas de controle de versões;
- Sistemas de gerenciamento de construção; e
- Integração dos espaços de trabalho.

Na Seção 2.2, são descritos processos que regem a aplicação da GCS nos diversos contextos da engenharia de software. Posteriormente, as abordagens descritas nas seções 2.3, 2.4 e 2.5 preocupam-se com os problemas específicos dos subsistemas de controle de modificações, controle de versões e gerenciamento de construção, respectivamente, mesmo que esses problemas envolvam mais de uma das funções da GCS segundo a perspectiva gerencial. Cada subsistema atua em partes ortogonais das funções de GCS, segundo a perspectiva gerencial, e fornece funcionalidades independentes dos demais subsistemas. Por esta razão, as abordagens descritas nessas seções, apesar de não serem abrangentes para toda a GCS, são úteis individualmente dentro do subsistema a que se propõem atuar. Na Seção 2.6, são apresentadas abordagens para a integração dos espaços de trabalho de GCS e ADS. Finalmente, a Seção 2.7 conclui o capítulo com algumas considerações finais sobre GCS.

2.2 Processos de GCS

A GCS, por ser uma área fortemente calcada em controle, é celeiro de uma vasta gama de normas. Inicialmente, em 1962, a Força Aérea dos Estados Unidos publicou a primeira norma relacionada com a gerência de configuração, a AFSCM 375-1 (DOD, 1962). Apesar dessa norma separar claramente o processo de engenharia do processo de controle sobre a evolução do produto, apenas produtos físicos (*hardware*) eram levados em consideração até então.

Somente em 1971 foi publicada, também pela Força Aérea dos Estados Unidos, uma norma que relacionava software como possível IC, a MIL Std 483 (DOD, 1971). A partir daí, um grande esforço foi feito para consolidar as diversas iniciativas de estruturação da área de GCS (OLIVEIRA *et al.*, 2001), culminando na norma DOD Std 2167A (DOD, 1985).

Em 1988, o governo dos Estados Unidos indicou que estaria se afastando da responsabilidade de criação de normas, passando essa tarefa para organizações como EIA (*Electronics Industries Association*), IEEE (*Institute of Electrical and Electronics Engineers*), SAE (*Society of Automotive Engineers*), ANSI (*American National Standards Institute*) e ISO (*International Organization for Standardization*).

A partir desta indicação de afastamento do governo dos Estados Unidos da tarefa de produzir normas, as principais referências internacionais no contexto de GCS passaram a ser a ANSI/IEEE e a ISO. Dentre as diversas normas produzidas desde então, que de alguma forma tocam no tema de GCS, as mais importantes são:

- IEEE Std 1042 (IEEE, 1987), que é considerada a norma internacional mais completa sobre GCS e serve como um guia para a aplicação da IEEE Std 828. Apesar de reconfirmada em 1993, foi descontinuada em 2000;
- IEEE Std 828 (IEEE, 2005), que trata da confecção de planos de GCS e é a terceira reedição da versão inicial que data de 1983;
- ISO 10007 (ISO, 1995a), que fornece diretrizes para a utilização de GCS na indústria e define a interface da GCS com as demais áreas de gerência;
- CMM (JALOTE, 1999) e CMMI (CHRISSIS *et al.*, 2003), que fornecem modelos multi-níveis para a classificação de maturidade de empresas de desenvolvimento de software; e
- MPS.BR (SOFTEX, 2006b), que consiste em um programa para melhoria de processos de software, voltado para a realidade da micro, pequenas e médias empresas de software brasileiras.

Além dessas normas estabelecidas por órgãos internacionais, existem várias publicações relacionadas ao estabelecimento de procedimentos para uso de sistemas específicos de GCS, como, por exemplo, CVS (FOGEL e BAR, 2001; VENUGOPALAN, 2002) e Bugzilla (GOLDBERG, 2002).

No restante desta seção, são apresentadas, em maior detalhe, algumas abordagens referentes a processos de GCS (IEEE, 1987; ISO, 1995a; JALOTE, 1999; CHRISSIS *et al.*, 2003; IEEE, 2005; SOFTEX, 2006b).

2.2.1 IEEE Std 1042

A norma IEEE Std 1042 tem como principal objetivo descrever a aplicação da disciplina de GC ao desenvolvimento de software. Essa aplicação é dividida em duas partes: planejamento e implementação. Para apoiar o planejamento, são apresentadas

sugestões referentes aos vários aspectos da GCS. Para apoiar a implementação, são descritos quatro exemplos de planos de GCS elaborados segundo a versão de 1983 da norma IEEE Std 828. Esses exemplos consistem em: (1) projeto complexo e crítico; (2) projeto pequeno; (3) projeto somente de manutenção; e (4) projeto de software embutido em hardware.

Nas seções iniciais da norma, são apresentadas algumas siglas comumente usadas na área, como, por exemplo, as de CCC e de IC. São também definidos alguns termos, como, por exemplo, o de linha base, como sendo uma configuração da especificação ou do produto, revisada e aprovada, que serve como base para desenvolvimento futuro. Segundo a norma, o formalismo aplicado às linhas bases pode variar em função da flexibilidade que determinados processos de desenvolvimento de software necessitam. A linha base pode ser promovida internamente pelos níveis de análise (linha base funcional), de projeto (linha base alocada) e implementação (linha base de produto). Quando a linha base passa por um processo de auditoria e é entregue ao cliente, recebe o nome de liberação. O mecanismo de etiqueta (*tag*) nas versões de um conjunto de artefatos, utilizado em diversos sistemas de controle de versões, é sugerido para implementar o conceito de linha base.

Nessa norma, os termos versão e revisão, que são rotineiramente utilizados na literatura como sinônimos, recebem conotações diferentes. As versões de ICs representam evoluções funcionais, enquanto as revisões em ICs representam correções de erros ou melhorias que não modificam as funcionalidades dos ICs.

A importância da GCS como processo de apoio ao processo de desenvolvimento de software é enfatizada. A sua efetividade nessa função só pode ser obtida caso ela esteja incorporada ao dia-a-dia do projeto, desde o desenvolvimento até a manutenção. As etapas iniciais de modelagem, que normalmente recebem menor apoio de GCS, devem ser tratadas com maior atenção, por causa da dificuldade de divisão das atividades em tarefas mais granulares.

Uma característica importante levantada pela norma é o fato da GCS diferenciar da GC de Hardware pela forma que o produto é tratado. No caso do hardware, somente a documentação reside nos repositórios[13]. Contudo, no caso de software, o próprio produto, além da documentação, reside nos repositórios de controle de versão. Essa

[13] O termo repositório representa o local de acesso controlado onde as versões dos ICs são armazenadas (LEON, 2000).

característica possibilita a introdução de uma maior automação na função de controle da configuração.

O processo de controle de modificações descrito na norma consiste na identificação das linhas bases e no acompanhamento das modificações aplicadas a essas linhas bases. Dependendo da etapa do processo de desenvolvimento, determinados artefatos terão maior importância e constituirão diferentes linhas bases. Por exemplo, no momento da codificação, os artefatos de projeto, que constituem a linha base alocada, são os de maior importância e devem receber maior atenção por parte da GCS. Isso ocorre porque o código fonte que está sendo construído em função do projeto pode ficar inconsistente, caso alguma modificação não relatada ocorra nesses ICs de projeto.

2.2.2 IEEE Std 828

A norma IEEE Std 828 se preocupa com a atividade de planejamento da GCS. Ela é compatível com a norma ISO 12207 e foi criada em 1983, sofrendo revisões em 1990, 1998 e 2005. A sua aplicabilidade não está relacionada com o tipo do sistema a ser construído, podendo ser utilizada desde sistemas de informação a sistemas de tempo-real, críticos ou não. O público alvo dessa norma engloba os responsáveis pela confecção do plano de GCS ou pelas auditorias de GCS. A norma ressalta que a GCS é sempre utilizada em projetos de software, seja de forma planejada ou *ad-hoc*. Contudo, o planejamento dessas atividades contribui para o aumento da eficiência das mesmas.

A norma determina que os planos de GCS devem conter, ao menos, as seções de introdução, gerenciamento, atividades, cronograma, recursos e manutenção do plano. O plano deve existir de forma individual ou estar dentro de outro documento do projeto. Além disso, o plano deve conter todas as informações de GCS, seja explicitamente ou via referência externa (e.g. ferramenta). Caso o formato do plano não seja aderente ao formato sugerido pela norma, esse formato deve ser devidamente explicitado na seção de introdução.

A **seção de introdução** deve fornecer uma visão geral do projeto, identificar os artefatos de software e hardware que participarão direta ou indiretamente da GCS, determinar o grau de formalismo e controle em que a GCS será aplicada, estipular sobre quais atividades do ciclo de vida do software a GCS será adotada e descrever quais aspectos geram restrições em custo, cronograma ou na própria possibilidade de execução da GCS.

A **seção de gerenciamento** deve determinar quais unidades organizacionais participarão das atividades de GCS do projeto, estipulando seus papéis e relacionamentos. Além disso, devem ser claramente descritos, para cada comitê criado pelas unidades organizacionais, o seu propósito, membros, período de vigência, escopo de autoridade e procedimentos operacionais.

A **seção de atividades**, que usualmente é a mais extensa de todas, é subdividida em outras sete subseções: identificação da configuração, controle da configuração, contabilização da situação da configuração, avaliação e revisão da configuração, controle de interfaces, controle de terceiros e gerenciamento de liberação e entrega. As quatro primeiras subseções e a última, que são equivalentes às funções de GCS, devem estabelecer como essas funções devem ser executadas no contexto do projeto em questão. As demais subseções devem estabelecer como artefatos e equipes fora do escopo do sistema em questão devem ser tratados.

A **seção de cronograma** deve estipular a seqüência e a coordenação das atividades identificadas no plano. O cronograma pode ser descrito usando datas absolutas ou relativas às demais atividades do projeto. O uso de representação gráfica é indicado para esse tipo de informação.

A **seção de recursos** deve determinar as ferramentas, técnicas, equipamentos, pessoal e treinamento necessários para implementar o plano de GCS. Além disso, deve ser estabelecido como cada recurso será obtido.

A **seção de manutenção** deve estipular quem é responsável pela manutenção do plano, com qual freqüência o plano será atualizado e como as modificações no plano serão verificadas, aprovadas e relatadas.

Como todo artefato que evolui durante o ciclo de vida do software, os planos gerenciais do projeto, incluindo o plano de GCS, devem ser tratados como ICs e pertencer a uma linha base. É desejável a criação de uma linha base gerencial, para permitir o controle individualizado sobre esses artefatos, minimizando o impacto que a evolução desses artefatos pode ter sobre a atividade de análise. Contudo, a primeira linha base definida na norma IEEE Std 1042 é a funcional. Neste caso, esses artefatos gerenciais serão controlados pela GCS somente depois do término da atividade de análise.

Um maior detalhamento sobre as partes que compõem planos de GCS segundo a norma IEEE Std 828, juntamente com exemplos de suas aplicações, pode ser obtido na seção 3 e nos apêndices da norma IEEE Std 1042.

2.2.3 ISO 10007

A norma ISO 10007, que fornece diretrizes para a GC, tem como propósito aumentar o entendimento comum sobre o assunto e incentivar o uso de GC, alinhando a abordagem na indústria. Essa norma visa atender os requisitos de GC apresentados nas normas da família ISO 9000. Uma característica interessante da norma é sua abrangência. Ela não foca somente na GCS, mas em GC de qualquer produto. Contudo, em algumas partes do seu corpo são levantadas questões especificamente relacionadas a software.

Nas suas seções iniciais, são definidas terminologias. As maiores diferenças de terminologia em relação aos demais trabalhos da literatura são: o uso do termo configuração básica no lugar do termo linha base, o uso do termo conselho de controle de configuração no lugar do termo comitê de controle de configuração e o uso do termo gestão de configuração no lugar de gerência de configuração.

Nas seções seguintes da norma, é apresentada a visão geral sobre o assunto, onde questões semelhantes às levantadas pelas normas IEEE Std 1042 e IEEE Std 828 são relatadas. Em especial, é discutida a necessidade de auditoria sobre os próprios planos e procedimentos de GC, a seleção de IC usando técnicas de decomposição e o uso de rastros entre as alterações e suas linhas bases.

Assim como apontado pela norma IEEE Std 1042, essa norma salienta que a GC pode e deve ser aplicada sobre o próprio produto quando o mesmo é constituído de software, além do controle sobre a documentação do produto. Uma discussão introduzida nessa norma, que não é tocada nas demais, se refere ao grão de seleção de ICs. A norma indica que caso o grão seja fino, o número de ICs será grande, e isso poderá afetar a visibilidade do produto, dificultar o gerenciamento e aumentar o custo de operação. Por outro lado, se o grão for grosso, o número de ICs será pequeno, e isso poderá gerar dificuldades de logística e manutenção, limitando as possibilidades de gerência. Esse problema é contornado pelas ferramentas atuais por meio do uso de granularidade dinâmica de ICs em função das solicitações de modificação.

São também descritos na norma os critérios para seleção dos ICs, a função do CCC, as informações necessárias para solicitações de modificação, os critérios para a avaliação das solicitações de modificação e os tipos de relatório mais comuns da função de contabilização da situação da configuração. Um fato colocado explicitamente nessa norma é a existência de dependência entre a função de contabilização da situação da

configuração e as funções de identificação da configuração e de controle da configuração. A função de contabilização da situação da configuração só será executada de forma correta se as funções de identificação da configuração e de controle da configuração forem capazes de coletar as informações apropriadas.

No anexo A dessa norma, a estrutura desejada de um plano de GC é apresentada. Essa estrutura diverge da estrutura definida na norma IEEE Std 828. Contudo, o conteúdo é basicamente o mesmo, organizado de forma diferente. No anexo B, é apresentada uma tabela de referência cruzada entre as seções dessa norma e as seções das normas ISO 9001, ISO 9002, ISO 9003 e ISO 9004-1.

2.2.4 CMM e CMMI

A versão 1.1 do CMM, lançada em 1993, é considerada o modelo de maturidade relacionado a software mais utilizado mundialmente (HASS, 2003). Esse modelo define cinco níveis de maturidade: (1) Inicial; (2) Repetível; (3) Definido; (4) Gerenciável; e (5) Otimizado. Para cada nível de maturidade, são estipuladas áreas chave de processo que devem ser atendidas, possibilitando que esse nível seja alcançado.

A GCS pode contribuir indiretamente para atender a áreas chave de processo existentes em vários níveis de maturidade, como, por exemplo, prevenção de defeitos e gerenciamento quantitativo do processo. Contudo, sua maior contribuição está no nível 2 de maturidade, em uma área chave de processo batizada com o seu nome.

Os objetivos da área chave de processo de GCS são: (1) planejamento das atividades de GCS; (2) identificação e controle dos ICs; (3) controle das solicitações de modificação; e (4) propagação das informações aos indivíduos afetados. Para que esses objetivos possam ser atendidos, um conjunto de atividades é definido: (1) preparação do plano de GCS para cada projeto; (2) aprovação do plano de GCS para guiar as demais atividades; (3) estabelecimento de um repositório para as linhas bases; (4) identificação dos ICs; (5) acompanhamento das solicitações de modificação segundo processo previamente acordado; (6) controle sobre modificações em linhas bases; (7) controle sobre liberação das linhas bases; (8) armazenamento das informações necessárias; (9) relato do andamento das atividades de GCS e do estado atual das linhas bases; e (10) auditoria sobre as linhas bases.

Um fato que contribuiu com a propagação da GCS nas grandes empresas de software foi a determinação, feita pelo departamento de defesa dos Estados Unidos, onde os principais contratos governamentais de software obrigavam as empresas

interessadas a estarem ao menos no nível 3 de maturidade do CMM (CHRISTENSEN e THAYER, 2002). Como a GCS é área chave de processo para o nível 2, todas essas empresas de desenvolvimento de software para o governo foram obrigadas indiretamente a adotar GCS.

O modelo CMMI-SE/SW/IPPD/SS teve a sua versão 1.1 lançada em 2002. Essa versão se baseia nos rascunhos da versão 2.0 do CMM. O CMMI define duas representações de modelos de maturidade: em estágios (SEI, 2002b) e contínua (SEI, 2002a). A representação em estágios é similar ao modelo de maturidade definido pelo CMM. A representação contínua introduz o conceito de nível de capacidade, que difere do conceito de nível de maturidade usado pela representação em estágios por focar de forma individual nas áreas de processo, ao invés de lidar com o processo organizacional como um todo.

Dentro do CMMI, a GCS é área do grupo de processos de suporte. Essa área é composta por três objetivos específicos: (1) estabelecimento de linhas bases; (2) acompanhamento e controle de modificações; e (3) manutenção da integridade das linhas bases. Para cada objetivo da área de GCS, são identificadas práticas específicas: (1.1) identificação dos ICs; (1.2) estabelecimento de sistema de GCS; (1.3) criação e liberação de linhas bases; (2.1) acompanhamento de modificações; (2.2) controle das modificações; (3.1) registro das informações; e (3.2) auditoria da configuração.

A representação contínua do CMMI é composta por seis níveis de capacidade: (0) Incompleto; (1) Executável; (2) Gerenciável; (3) Definido; (4) Quantitativamente Gerenciável; e (5) Otimizado. Cada um desses níveis de capacidade, a partir do nível 1, tem objetivos genéricos associados. Para cada objetivo genérico, são definidas práticas genéricas que devem ser atendidas para que a área de processo cumpra o objetivo genérico e possa ser classificada no nível de capacidade que esse objetivo genérico está relacionado. A GCS é a prática genérica número 6 do objetivo genérico número 2, relacionado ao nível de capacidade Gerenciável. Desta forma, a GCS é requisito para que qualquer área de processo possa atingir o nível 2 de capacidade.

Além disso, a área de GCS está intimamente relacionada com outras áreass de processo no CMMI. Por exemplo, o Planejamento de Projetos pode apoiar na elaboração do plano de GCS; a Análise e Resolução de Causas pode apoiar na atividade de análise de impacto de GCS; e a Análise de Decisão e Resolução pode apoiar na atividade de avaliação de solicitações de modificação de GCS. Por outro lado, a GCS pode apoiar a Gerência de Requisitos, no que diz respeito ao controle de modificações

sobre os requisitos, e a Integração do Produto, no que diz respeito ao controle da evolução de interfaces.

HASS (2003) fornece um guia incremental detalhado para a adoção da GCS nos níveis de capacidade CMMI. Esse guia estabelece, no nível 1, os passos necessários para atender a todos os objetivos de GCS definidos no CMMI. No nível 2, os passos necessários para planejar, executar, monitorar e controlar a GCS em projetos individuais. No nível 3, os passos necessários para instanciar processos definidos a partir de adaptações do processo padrão da organização. No nível 4, os passos necessários para fazer uso de técnicas quantitativas para medir o processo de GCS. Finalmente, no nível 5, os passos necessários para melhorar o processo em função da interpretação das estatísticas obtidas.

2.2.5 MPS.BR

O MPS.BR (SOFTEX, 2006b), iniciado em dezembro de 2003, visa prover uma alternativa para as micro, pequenas e médias empresas de software brasileiras na melhoria da qualidade dos seus processos. Para isso, o MPS.BR é aderente à norma ISO/IEC 15504 (ISO, 2004) e compatível com o modelo CMMI.

No nível G, parcialmente gerenciado, encontram-se os processos básicos para a execução de projetos de software. O nível F, gerenciado, introduz alguns processos de suporte, incluindo Gerência de Configuração. O nível E, parcialmente definido, adiciona processos necessários para padronizar os demais processos na unidade organizacional como um todo. Em seqüência, o nível D, largamente definido, introduz processos referentes às atividades de engenharia. No nível C, definido, são introduzidos processos que viabilizam a redução da variância nos projetos. O nível B, gerenciado quantitativamente, insere processos que possibilitam um acompanhamento baseado em fatos do projeto. Finalmente, no nível A, em otimização, são acrescentados processos que viabilizam a melhoria contínua dos demais processos da unidade organizacional. Os níveis F, C, B e A equivalem aos níveis 2, 3, 4 e 5 do CMMI, respectivamente.

De acordo com o MPS.BR, o propósito do processo de GCS, situado no nível F, é estabelecer e manter a integridade de todos os produtos de trabalho de um processo ou projeto e disponibilizá-los a todos os envolvidos. Esse processo é dado como satisfeito se produzir os seguintes resultados esperados: (1) Os ICs são identificados; (2) Os ICs gerados pelo projeto são definidos e colocados sob uma linha base; (3) É estabelecido e mantido um Sistema de GCS; (4) As modificações e liberações dos ICs são controladas;

(5) As modificações e liberações são disponibilizadas para todos os envolvidos; (6) A situação dos ICs e as solicitações de mudanças são registradas, relatadas e o seu impacto é analisado; (7) A completeza e a consistência dos ICs são asseguradas; (8) O armazenamento, o manuseio e a entrega dos produtos de trabalho são controlados; (9) A integridade das linhas bases é estabelecida e mantida, via auditoria da configuração e de registros da GCS.

Os principais diferenciais do MPS.BR em relação ao CMMI são: mais níveis de maturidade, possibilitando uma maior visibilidade da evolução da organização; redução dos custos de avaliação, viabilizando a aplicação do modelo em micro, pequenas e médias empresas; avaliação sobre resultados esperados, e não sobre as práticas que produzem esses resultados; e uso de validade para as avaliações, incentivando avaliações recorrentes como motor para a melhoria dos processos.

2.2.6 Considerações finais sobre processos de GCS

A norma IEEE Std 1042, que é considerada a norma mais completa de GCS, fornece exemplos de utilização da norma IEEE Std 828, que enfatiza a construção do plano de GCS. Além disso, apresenta um modelo evolutivo para a classificação das ferramentas de GCS. Contudo, a primeira linha base considerada é a funcional, o que resulta na falta de controle sobre ICs gerenciais antes do término da atividade de análise.

As normas ISO 10007 e IEEE Std 1042 levantam considerações especiais referentes ao contexto de GCS. Quando o IC é *hardware*, somente a documentação do IC pode ser controlada. Entretanto, no caso de software, o IC propriamente dito pode ficar sob controle dos sistemas de GCS. Além disso, apesar das seções propostas pela ISO 10007 para o plano de GCS serem diferentes das seções propostas pelo IEEE Std 828, ambos os planos contém as mesmas informações, só que organizadas de forma diferente. Diferentemente das normas, os modelos CMM e CMMI apresentam um mecanismo evolutivo para adoção das funções de GCS.

2.3 Sistemas de controle de modificações

A GCS é fortemente calcada em processos e normas (ISO, 1995a; IEEE, 2005), que definem os procedimentos necessários para permitir uma evolução controlada do desenvolvimento de software. Para automatizar a execução desses processos, são utilizados sistemas de controle de modificações.

Inicialmente, os sistemas de controle de modificações tinham seu foco principal em modificações corretivas, mas, com o amadurecimento da área, esses sistemas passaram a ser utilizados para qualquer tipo de modificação. Em alguns casos, quando o processo de desenvolvimento prioriza a geração precoce de liberações, os sistemas de controle de modificações passam a ser utilizados desde o início do desenvolvimento, exercendo a função de máquina de processo e recebendo o nome de sistema de acompanhamento de solicitações (*issue tracking system*).

Um dos sistemas de controle de modificações mais conhecidos e utilizados em projetos de software livre é o Bugzilla (BARNSON *et al.*, 2003). O Bugzilla provê apoio à busca detalhada de modificações, criação de vínculos entre modificações, controle do estado das modificações e relacionamento entre modificações e os artefatos alterados propriamente ditos. Além disso, uma preocupação constante no projeto é a necessidade de um alto desempenho da ferramenta. Para atingir esse objetivo, o Bugzilla faz uso de banco de dados relacional e interfaces HTML (*Hypertext Markup Language*) (W3C, 1999).

Contudo, existe atualmente um número relativamente grande de sistemas com características semelhantes ao Bugzilla (CM CROSSROADS, 2006), como, por exemplo, o Scarab (TIGRIS, 2006), que apresenta como principal diferencial o uso da linguagem Java, e o ClearQuest (WHITE, 2000), que faz parte da iniciativa da IBM Rational em prover um ambiente integrado de GCS, conhecido como UCM (*Unified Change Management*).

Apesar dessa diversidade de sistemas de controle de modificações, essa área ainda é carente de pesquisa que faça uso da infra-estrutura existente com o objetivo de atingir maior qualidade e produtividade no desenvolvimento de software. Por exemplo, a capacidade de configuração do sistema de controle de modificações em relação aos processos de GCS é uma característica desejável.

No restante desta seção, são apresentadas, com maior detalhamento, algumas abordagens referentes a sistemas de controle de modificações (SCHNEIDER, 2001; BRIAND *et al.*, 2003; ESTRADA, 2003; KNETHEN e GRUND, 2003; FIGUEIREDO, 2004). Essas abordagens foram escolhidas visando apresentar diferentes perspectivas de pesquisas realizadas em controle de modificações.

2.3.1 GCCS

Com o intuito de aumentar o grau de colaboração durante a GCS, foram criados um método e um ferramental de apoio à Gerência Cooperativa de Configuração de Software (GCCS) (SCHNEIDER, 2001). Esse método tem como pontos chave o controle de versões, a gerência de modificações, o controle sobre o processo e a parametrização do nível de colaboração desejado.

O modelo de controle de modificações adotado pelo GCCS herda as atividades definidas no contexto do ClearQuest, que são: (1) solicitação; (2) análise; (3) avaliação; (4) implementação; (5) verificação; e (6) fechamento. Para armazenar as informações geradas durante a execução do processo de controle de modificações, foi utilizado o sistema de gerenciamento de banco de dados InterBase (BORLAND, 2006a). Para prover controle de versões à atividade de implementação, foi utilizado o CVS (CEDERQVIST, 2003).

O GCCS pode ser acessado, via um *plug-in*, diretamente do ambiente de modelagem ou de programação que o usuário está habituado a utilizar. Além disso, um agente, situado na barra de ferramenta do sistema operacional, permite a execução automática de tarefas rotineiras relacionadas à coordenação e comunicação entre os membros da equipe e a base de dados. As principais tarefas desse agente são: sincronizar os dados dos usuários, registrar usuários ativos nos ambientes dos demais usuários, procurar por usuários ativos, permitir a comunicação síncrona e assíncrona entre os usuários, controlar as seções de trabalho ou de revisão, de forma síncrona ou assíncrona, e avisar a ocorrência de eventos de reuniões agendadas.

Dentre os parâmetros configuráveis do GCCS estão: a periodicidade de controle, que determina de quanto em quanto tempo cada usuário deve relatar o andamento das suas tarefas, a data de execução de controle, que estabelece a execução periódica de controle, e o tipo do projeto, que determina quais artefatos devem ser armazenados no repositório.

2.3.2 ConfigCASE e SoSoft

ESTRADA (2003) detectou a ineficiência dos processos e normas de GCS existentes no contexto de pequenas e médias empresas. O seu principal argumento é que esses processos e normas foram criados pensando em cenários mais complexos, existentes em grandes empresas, o que torna sua aplicação inviável em empresas que não podem arcar com o nível de burocracia associada. Um exemplo é o questionamento

da efetividade de um CCC neste contexto. Como resultado desse cenário, é apresentada uma estatística que aponta 73% do universo das empresas de software como não utilizadoras de GCS.

Para fornecer uma solução que atendesse à pequena e média empresa, com até 25 empregados, representando 8% do total das empresas de software, foram definidos cinco processos, que abrangem a identificação de ICs, o controle de modificações, o controle de versões, o planejamento e a geração de informações referentes ao estado da configuração.

Juntamente com os processos, foram definidas 17 medidas, sendo que 8 delas são uni-processos e 9 são bi-processos. Uma medida uni-processo é definida pela dependência dos dados gerados por um, e somente um, dos cinco processos definidos. Já as medidas bi-processo correlacionam dados produzidos por dois dos cinco processos definidos.

Como ferramental, foram construídos os sistemas ConfigCASE e SoSoft para a automatização dos processos e da medição. Além disso, é proposta a utilização do Visual SourceSafe (ROCHE e WHIPPLE, 2001) para apoiar o processo de controle de versões.

A abordagem, apesar de diferenciar pequenas e médias empresas das grandes empresas, e ressaltar a necessidade de utilização de medições durante os processos de GCS, não emprega adequadamente essas medições para obter benefícios tangíveis no desenvolvimento de software das pequenas e médias empresas. Questões como a utilização dos resultados obtidos pela aplicação das medições para a construção de rastros, aprendizado com projetos similares e análise de impacto das modificações são colocadas como trabalhos futuros.

2.3.3 GConf

FIGUEIREDO (2004) apresenta um processo de GCS baseado na norma ISO/IEC 12207 (ISO, 1995b), no processo de GCS do SWEBOK (SCOTT e NISSE, 2001) e no CMMI, juntamente com uma ferramenta de apoio voltada para ambientes de desenvolvimento de software orientados à organização (ADSOrg). Os ADSOrgs enfatizam a necessidade da gestão do conhecimento relacionado com a produção de software de uma determinada organização. Esse conhecimento, obtido em projetos anteriores, é utilizado como diferencial no apoio à execução das atividades de engenharia de software, dentre elas as relacionadas com GCS.

A ferramenta proposta faz uso da infra-estrutura existente, provida pela estação TABA (ROCHA *et al.*, 1990; TRAVASSOS, 1994), que permite a propagação de conhecimento entre os participantes das atividades de GCS. Esses participantes, segundo o GConf, são agrupados em dois papéis: gerentes de projeto e engenheiros de software. Esses papéis colaboram na execução de três atividades principais, que são: planejar GC, controlar liberação e distribuição e controlar configuração.

A **atividade de planejamento da GC** abrange as atividades de definição e implementação do processo, definida nos três processos em que esse processo se baseia, e de identificação da configuração, descrita tanto na ISO/IEC 12207 quanto no SWEBOK. Essa atividade é restrita aos gerentes de projeto e tem por objetivo preparar a infra-estrutura necessária, tanto no que diz respeito a processos quanto no que diz respeito à definição de produtos que irão compor os ICs, para que o desenvolvimento de software possa ser iniciado.

A **atividade de controle de liberação e distribuição** permite que determinadas versões de ICs sejam obtidas pelos engenheiros de software. Segundo o autor, essa atividade está provendo serviços semelhantes às atividades de relato da situação da configuração, definida nos três processos base, e de gerenciamento de liberação e entrega, definida nos processos ISO/IEC 12207 e SWEBOK. Contudo, a semântica para liberação descrita nos processos base está fortemente relacionada com a entrega de versão fechada ao usuário final, e não com a entrega de versão em desenvolvimento aos engenheiros de software.

Finalmente, a **atividade de controle da configuração** define um processo de controle de solicitações de modificação composto pelas subatividades de solicitação da modificação, análise da solicitação, implementação da modificação e verificação dos ICs alterados. Devido à simplicidade do processo sugerido, que exclui a atividade de classificação e junta a atividade de análise com a atividade de avaliação em uma única atividade de análise exercida pelo gerente do projeto, o papel de gerente de projeto fica sobrecarregado com atribuições referentes ao desenvolvimento. A criação de um novo papel, referente ao analista de impacto, poderia aliviar a carga de trabalho sobre o gerente do projeto. Além disso, seria interessante possibilitar que o próprio cliente pudesse solicitar modificações diretamente no sistema, como determinado pela norma ISO 10007. Da forma que foi concebida, essa atividade abrange a atividade de controle da configuração, definida nos três processos base, e avaliação e revisão da configuração, definida nos processos ISO/IEC 12207 e SWEBOK.

2.3.4 Análise de Impacto

Uma das atividades mais importantes do processo de GCS é a análise de impacto. É a partir dos laudos elaborados durante essa atividade que o CCC pode tomar as decisões referentes à aprovação ou não das solicitações de modificação. Devido a essa importância, a atividade de análise de impacto é responsável por grande parte do tempo gasto durante o ciclo de vida de uma modificação.

Podem ser detectadas diferentes linhas de trabalho para automatizar, ao menos de forma parcial, a atividade de Análise de Impacto. BRIAND et al. (2003) definem uma abordagem, implementada na ferramenta iACMTool, para a análise da propagação do impacto de modificações já efetuadas. Para atingir esse objetivo, o modelo UML construído para contemplar a modificação é comparado com o modelo UML original, fazendo uso de regras descritas em OCL (*Object Constraint Language*) (OMG, 2006a).

As regras OCL para auxílio à análise de impacto são agrupadas em três categorias: regras de consistência, regras de classificação e regras de impacto. As 120 regras de consistência têm por objetivo verificar se os modelos estão consistentes com a especificação da UML. Sem essa avaliação não seria possível confiar na qualidade da análise de impacto gerada. As 97 regras de classificação têm por objetivo detectar o tipo da modificação em questão. E, finalmente, outras 97 regras de impacto têm por objetivo identificar, para cada tipo possível de modificação, os impactos existentes.

Com objetivos ligeiramente diferentes, KNETHEN et al. (2003) apresentam uma ferramenta denominada QuaTrace que fornece apoio semi-automático para a análise de impacto, fazendo uso dos rastros entre artefatos. A principal diferença de objetivos entre a iACMTool e a QuaTrace é que a QuaTrace atua apoiando a análise de impacto referente à própria modificação e às modificações que são propagadas em função dessa modificação principal. Já a iACMTool atua somente na análise de impacto referente às modificações propagadas, sem apoiar a análise de impacto da modificação que originou as propagações.

A abordagem QuaTrace faz uso de um papel adicional no desenvolvimento de software, o de gerente de requisitos. O gerente de requisitos é responsável por preparar as atividades necessárias para a coleta de rastros no contexto da organização. Além desse papel, os papéis já existentes de engenheiro de requisitos, analista de impacto e mantenedor também são afetados pela abordagem. O engenheiro de requisitos faz uso da QuaTrace durante a construção dos rastros entre os diversos artefatos em

desenvolvimento. O analista de impacto faz uso da QuaTrace para analisar, de forma semi-automática, os impactos de uma modificação. Essa análise ocorre via navegação pelos rastros e possibilita uma estimativa mais precisa dos custos. O mantenedor faz uso da QuaTrace para detectar possíveis propagações da modificação original.

2.3.5 Considerações finais sobre sistemas de controle de modificações

As abordagens de controle de modificações existentes têm, em sua maioria, um processo predefinido, implementado diretamente no seu código fonte. Essa limitação pode ser encontrada, por exemplo, nas abordagens GCCS, ConfigCASE e SoSoft, e GConf. Entretanto, essas abordagens apresentam inovações úteis para a GCS. O GCCS introduz novas funcionalidades referentes ao estímulo à colaboração entre os participantes do processo. As abordagens ConfigCASE e SoSoft enfatizam o uso de métricas configuráveis sobre esse processo. Já a GConf relaciona características do modelo CMMI com características existentes na norma ISO/IEC 12207 e com o SWEBOK. O ClearQuest, diferentemente da maioria das abordagens, possibilita alguma configuração sobre o processo e sobre as informações necessárias a cada atividade.

Outras abordagens mais pontuais foram apresentadas. Essas abordagens dão apoio a atividades específicas, sem tratar o processo como um todo. Por essa razão, elas poderiam trabalhar em conjunto com as demais abordagens. Dentre essas abordagens específicas, a iACMTool e a QuaTrace apóiam a análise de impacto utilizando técnicas diferentes, mas com efeitos semelhantes para o usuário.

2.4 Sistemas de controle de versões

O subsistema de GCS que mais recebeu contribuições, tanto de pesquisa quanto comercialmente, é o de controle de versões. Várias soluções comerciais, como, por exemplo, ClearCase (WHITE, 2000) e Visual SourceSafe, fornecem características satisfatórias para o problema quando os ICs são artefatos simples, definidos por meio de arquivos do sistema operacional. Além disso, outras tantas soluções livres estão disponíveis, como, por exemplo, CVS, Subversion (COLLINS-SUSSMAN *et al.*, 2004) e RCS (TICHY, 1985).

Geralmente, essas soluções provêem infra-estrutura para a definição de quais artefatos necessitam de controle de versões e permitem que esses artefatos sejam

obtidos, por meio de um processo conhecido como *check-out*[14], modificados dentro do espaço de trabalho do engenheiro de software e retornados ao repositório, por meio de um processo conhecido como *check-in*[15]. Além disso, essas soluções normalmente suportam a definição de diferentes políticas de trabalho. Dentre essas políticas, podemos citar a política pessimista, que enfatiza o uso de *check-out* reservado, fazendo bloqueio (*lock*) e inibindo o paralelismo do desenvolvimento sobre o mesmo artefato. Outra política amplamente utilizada é a otimista, que assume que a quantidade de conflitos é naturalmente baixa e que é mais fácil tratar cada conflito individualmente, caso eles venham a ocorrer. A política otimista usa um mecanismo de tratamento de conflito conhecido como junção (*merge*), que une os trabalhos efetuados em paralelo sobre um mesmo IC e produz uma nova versão do IC que contém a soma desses trabalhos.

Existem situações onde uma determinada política é mais indicada do que as demais. Nos casos onde a junção dos trabalhos tende a ser complexa, quando, por exemplo, os ICs não são textuais e a ferramenta que conhece a representação binária dos ICs não dá apoio automatizado para junções, é mais indicado trabalhar usando políticas pessimistas. Contudo, na grande maioria das situações referentes ao desenvolvimento de software, as políticas otimistas atendem satisfatoriamente (ESTUBLIER, 2001).

Além desses recursos fundamentais de sistemas de controle de versão, outros recursos um pouco mais elaborados também são encontrados com freqüência. Dentre eles, os mais difundidos são a automação no acesso a linhas bases, que são representadas por determinadas versões de ICs e identificadas na grande maioria das vezes pelo uso de etiquetas, e o controle de desenvolvimento paralelo com isolamento obtido pelo uso de ramos[16], que podem ser unidos novamente à linha de desenvolvimento principal por meio do processo de junção.

Apesar da variedade de soluções disponíveis, um conjunto de novas características é desejável e novas pesquisas estão sendo feitas com o intuito de atender a essa demanda (ESTUBLIER, 2000). Dentre as características desejadas, podemos citar:

---

[14] O termo *check-out* representa o processo de solicitação, aprovação e cópia de ICs do repositório para o espaço de trabalho do engenheiro de software (LEON, 2000).

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

[16] O termo ramo (*branch*) representa versões temporárias que não seguem a linha principal do desenvolvimento (LEON, 2000).

- Metáfora uniforme para o versionamento de ICs primitivos e compostos;
- Mecanismo e versionamento não intrusivo nas tarefas já complexas de desenvolvimento;
- Necessidade de um conceito de IC que não traga as heranças indesejáveis existentes em arquivos do sistema operacional;
- Apoio aos diferentes níveis de abstração existentes no desenvolvimento de software, contemplando de modelos a código;
- Mecanismos de armazenamento eficientes e confiáveis, fazendo uso de banco de dados dentro do paradigma adotado;
- Escalabilidade e disponibilidade compatível com a importância do problema em questão.

Para minimizar problemas referentes a desempenho, foram criados sistemas de controle de versões distribuídos. O ClearCase tem uma versão denominada ClearCase Multisite, que permite a configuração de repositórios descentralizados. Acesso ponto a ponto também pode ser encontrado em sistemas modernos de controle de versões, como, por exemplo, o BitKeeper (BITMOVER, 2006). Também existem sistemas livres com essas características, como, por exemplo, GNU Arch (LORD, 2006), GIT (TORVALDS e HAMANO, 2006) e SVK (KAO, 2006).

Além dessas características técnicas, fatores como custo reduzido de aquisição e manutenção, e facilidade na adição de novas funcionalidades também devem ser levados em conta.

No restante desta seção, são apresentadas, com maior detalhamento, algumas abordagens referentes a sistemas de controle de versões (CHIEN *et al.*, 2001; ESTUBLIER, 2001; MORO *et al.*, 2001; COBENA *et al.*, 2002; WANG *et al.*, 2003; EL-JAICK, 2004; HNETYNKA e PLASIL, 2004; LINGEN e VAN DER HOEK, 2004). Essas abordagens foram escolhidas visando apresentar diferentes perspectivas de pesquisas realizadas em controle de versões.

2.4.1 TVM

Com o objetivo de desvincular o controle de versões do conceito de arquivos de sistema operacional, SILVA et al. (2003) fizeram uso do modelo temporal versionado TVM (MORO *et al.*, 2001) aplicado a uma perspectiva de controle de versões. Nesse modelo, o IC passa a ser um objeto, em contraposição às soluções do estado da prática, que fazem uso de arquivos do sistema operacional. Apesar dessa prática ser positiva

(ESTUBLIER, 2000), a implementação do modelo ocorre sobre um banco de dados relacional. Para acessar as versões de forma transparente, é utilizada a linguagem TVQL (MORO *et al.*, 2002), que é uma extensão da sintaxe de SQL. Para armazenar os objetos, o modelo faz uso de versionamento baseado em estados, onde cada versão armazenada contém toda a informação necessária para representar a versão. Essa modalidade de versionamento oferece menores tempos de acesso às versões se comparada com a de versionamento baseado em diferenças[17], mas consome mais espaço em disco e banda de rede.

2.4.2 DVM

HNETYNKA et al. (2004) argumentam a necessidade de versionamento para modelos armazenados em repositórios MOF (*Meta Object Facility*) (OMG, 2002a) e a insuficiência das soluções tradicionais para esse fim, como, por exemplo, CVS. Mais ainda, é ressaltado que as respostas enviadas para a solicitação de propostas (RFP – *Request for Proposals*) de versionamento MOF, publicada pela OMG, não são suficientes para tratar as questões de distribuição envolvidas no problema.

Para contornar essas deficiências, foi definido um conjunto de regras que fazem uso de identificadores de localidade na construção do identificador de versão dos elementos armazenados em repositórios MOF distribuídos. Essas regras permitem a criação de versões na linha corrente de desenvolvimento somente no mesmo repositório, mas possibilitam a criação de ramos em repositórios distribuídos desde que esses ramos tenham identificadores de versão que encadeiem os nomes dos seus nós originais. A implementação de um repositório MOF fazendo uso dessas regras está em andamento.

2.4.3 MIMIX

Com a adoção do MOF como meta-modelo da UML, e da especificação XMI (*XML Metadata Interchange*) (OMG, 2002b) como mecanismo para externar modelos (M1) criados no contexto de meta-modelos MOF (M2), tornou-se possível compartilhar modelos UML criados em diferentes ferramentas CASE.

---

[17] O termo diferença (*delta*) representa o que mudou entre duas versões consecutivas (LEON, 2000). A técnica de diferença para frente (*forward delta*) armazena a versão mais antiga do IC e as diferenças para as versões posteriores. Já a técnica de diferença para trás (*reverse delta*) armazena a versão mais recente do IC e as diferenças para as versões anteriores.

O MIMIX (EL-JAICK, 2004) foi criado com o intuito de controlar a evolução desses modelos. As ferramentas CASE podem acessar o MIMIX, internamente ou externamente ao seu ambiente, usando *Web Services* (BOOTH *et al.*, 2005). O MIMIX é capaz de armazenar diversas versões de modelos UML contidas em documentos XMI e executar a junção entre essas versões, caso necessário.

Apesar do versionamento de modelos UML ser extremamente importante, a solução adotada pelo MIMIX é deficiente no que diz respeito à unidade de versionamento utilizada. Como o modelo é versionado como um todo, não é possível acompanhar a evolução individual de cada elemento de modelo da UML, como, por exemplo, classes e casos de uso. Em sistemas grandes, compostos por centenas de casos de uso e milhares de classes, esse requisito é fundamental para possibilitar uma evolução controlada do modelo e facilitar a medição sobre essa evolução.

2.4.4 Versionamento de XML

Os sistemas de controle de versões necessitam de algoritmos para a comparação de artefatos e detecção dos pontos em que esses artefatos foram modificados. O algoritmo LCS (*Longest Common Subsequence*) (MYERS, 1986) foi criado com o intuito de detectar diferenças entre documentos de texto. Esse algoritmo está implementado na ferramenta GNU diff (FSF, 2002) e é usado pela maioria dos sistemas de controle de versões atuais, como, por exemplo, o RCS e o CVS.

Contudo, quando o artefato que está sendo manipulado contém uma estruturação própria, apesar de ser um documento de texto, esse tipo de algoritmo passa a gerar resultados não desejados. No caso dos documentos XML (*Extensible Markup Language*) (W3C, 2004), o conceito de linha como unidade de comparação não é o ideal, já que a XML define estruturas próprias que seriam recomendadas para esse fim, como, por exemplo, atributos e elementos. Na literatura existem diversas propostas para a comparação e versionamento de documentos XML (CHIEN *et al.*, 2001; COBENA *et al.*, 2002; WANG *et al.*, 2003). Dentre os sistemas comerciais de controle de versões, alguns tratam de forma integrada alguns tipos específicos de arquivos ou permitem o acoplamento de tratadores externos. O ClearCase é um exemplo de sistema de controle de versões com tratamento especial para XML, DOC (MICROSOFT, 2006b) e MDL (IBM RATIONAL, 2006).

2.4.5 MCCM

O MCCM (LINGEN e VAN DER HOEK, 2004), continuação de um trabalho desenvolvido por VAN DER HOEK et al. (2002) intitulado NUCM, é um sistema de controle de versões baseado na composição de políticas. A motivação desse trabalho está na complexidade de se construir do zero um sistema de controle de versões, juntamente com a necessidade constante de criar novos sistemas de controle de versões que atendam a políticas particulares de desenvolvimento. Assim sendo, com o uso do MCCM, é possível instanciar conjuntos de políticas que definam de que forma um sistema de controle de versões deve se portar e implementar em poucas linhas de código a interação entre essas políticas.

Dois tipos de políticas são definidos: políticas de restrição e políticas de ação. As políticas de restrição são definidas no contexto do servidor do sistema de controle de versões e as políticas de ação são definidas no contexto dos clientes. O efeito do uso das políticas de restrição é diminuir o espaço de opções do usuário, e o efeito do uso das políticas de ação é definir quais caminhos seguir dentro desse espaço de opções já imposto pelas políticas de restrição. Desta forma, este trabalho permite a combinação de políticas específicas para a construção da política geral do sistema de controle de versões. Além disso, novas políticas podem ser especificadas e implementadas para atender a requisitos específicos de sistemas de controle de versão futuros.

Cinco políticas de restrição já foram especificadas e implementadas. Essas políticas atendem às seguintes necessidades: (1) armazenamento, que possibilita a definição das situações onde é necessário persistir o IC completo ou somente a diferença entre versões consecutivas do IC; (2) composição, que permite a definição de como os ICs se compõem e como as modificações em ICs compostos podem ser representadas via modificações em ICs primitivos; (3) concorrência, que define se os ICs poderão ser modificados de forma otimista ou pessimista; (4) distribuição, que estipula se os ICs estarão em somente um servidor ou em múltiplos servidores para contemplar requisitos de alta disponibilidade e desempenho; e (5) seleção, que permite a definição das possibilidades de construção da área de trabalho sem tornar a mesma inconsistente.

Além dessas políticas localizadas no servidor, outras cinco políticas, localizadas no cliente, foram definidas: (1) evolução, que estabelece como serão representadas as diversas versões dos artefatos e como serão os relacionamentos de dependência entre

essas versões; (2) hierarquia, que especifica de que forma devem ser propagadas as operações como, por exemplo, a de bloqueio, dentro de uma estrutura de composição de ICs; (3) bloqueio, que determina quais outros elementos, como, por exemplo, ICs e linhas bases, devem ser bloqueados juntamente com os ICs selecionados; (4) colocação, que estabelece quando um IC deve ou não ser replicado em outros servidores; e (5) população, que possibilita a seleção consistente de dependências entre ICs.

Experimentalmente, essas políticas foram compostas para construir comportamentos semelhantes aos sistemas RCS, CVS, Subversion e Aide de Camp (SMDS, 1994). Como resultado, foi constatado que, com menos de 700 linhas de código extras, essas políticas puderam ser combinadas para simular cada um desses sistemas. Apesar de nenhum desses sistemas simulados ser completamente funcional, os indícios obtidos levantam a possibilidade de reutilização efetiva na construção de sistemas de controle de versão pelo uso de meta-sistemas baseados em políticas.

Outra característica interessante do MCCM é a sua arquitetura distribuída em servidores ligados por conexões ponto a ponto (*peer to peer*), obtidas via tecnologia de chamadas remotas de métodos - RMI (*Remote Method Invocation*).

2.4.6 Políticas de Controle de Objetos

Em um trabalho prático, ESTUBLIER (ESTUBLIER, 2001) discute a falta de escalabilidade das soluções existentes para controle de versões e acrescenta que nenhuma solução baseada em repositório central pode suportar grandes projetos. O exemplo citado consiste no desenvolvimento de um software de quatro milhões de linhas de código por uma equipe de 800 engenheiros de software. Neste contexto, nos horários de pico, que ocorrem no início do dia e no final do dia, as atualizações do ambiente local de cada engenheiro de software demandavam da rede e do servidor uma carga que não era possível de ser atendida a contento.

Ainda neste contexto, o uso de modelos de concorrência pessimistas torna-se impraticável, pois em média três engenheiros de software atuam sobre o mesmo artefato em paralelo, com picos de até trinta engenheiros de software. Como as transações de desenvolvimento são longas, políticas de bloqueio, como as adotadas em sistemas de gerenciamento de banco de dados, se tornam inviáveis.

Para contornar o problema de escalabilidade, é proposta a criação de grupos onde um ambiente de trabalho serve como integrador para os demais ambientes de trabalho do grupo. Em um segundo nível, todos os ambientes de trabalho integradores

dos grupos são servidos por um outro ambiente integrador, e assim sucessivamente. Esse modelo em níveis, que pode ser encontrado em outros trabalhos da literatura (LINGEN e VAN DER HOEK, 2004), diminui a sobrecarga sobre um único servidor central.

Para estabelecer a mecânica da colaboração entre os ambientes de trabalho dentro de um grupo, foram definidas seis ações atômicas, que são: modificar, propor, integrar, sincronizar, reservar e liberar. Além disso, foram definidas três políticas, que são: exclusão, reserva tardia e reserva tardia com integração. Cada política é descrita por ações atômicas. Por exemplo, a política de exclusão é descrita por: (1) sincronizar, (2) reservar, (3) modificar, (4) propor, (5) integrar e (6) liberar.

2.4.7 Considerações finais sobre sistemas de controle de versões

Das abordagens apresentadas, o TVM, o DVM e o MIMIX têm como principal contribuição o apoio ao versionamento de objetos, rompendo com a metáfora de sistema de arquivos. Contudo, o TVM faz uso de sistemas de banco de dados relacional para armazenar os objetos, além de se basear em um meta-modelo proprietário. O DVM, por sua vez, introduz características de distribuição sobre ICs descritos via meta-modelo MOF, mas não tem nenhuma versão implementada até o momento.

O MIMIX, apesar de se propor a versionar elementos MOF transportados via XMI, se baseia em XML para fazer comparações entre documentos e não possibilita que a versão individual de cada elemento MOF possa ser controlada. Esta solução apresenta similaridade com as demais soluções de versionamento de XML apresentadas nesta seção.

Finalmente, as abordagens apresentadas por LINGEN e VAN DER HOEK (2004) e ESTUBLIER (2001) possibilitam a configuração de políticas de GCS. Todavia, essas abordagens se baseiam na metáfora de sistema de arquivos e os sistemas de versionamento obtidos pela aplicação das políticas do MCCM não são completamente funcionais.

2.5 Sistemas de gerenciamento de construção

Os procedimentos para a construção e liberação de sistemas complexos podem tomar grande parte do tempo de desenvolvimento, especialmente quando essas atividades se repetem com freqüência.

Em processos ágeis de desenvolvimento, como, por exemplo, o eXtreme Programming (XP) (BECK, 1999), é aplicado o conceito de integração contínua (FOWLER, 2006), que visa compilar e testar o sistema várias vezes durante o dia para evitar surpresas futuras. Esse tipo de procedimento só se torna viável com apoio de ferramental apropriado, automatizando a tarefa.

A construção do sistema, que consiste na geração de ICs derivados[18] para uma configuração alvo a partir de um conjunto de ICs fonte[19], já é apoiada em parte por ferramentas de compilação e link-edição. Contudo, em sistemas grandes, esse procedimento não consiste somente nessas tarefas. Funcionalidades como a definição de variáveis de ambiente, seleção de módulos que devem fazer parte do programa gerado segundo as variáveis de ambiente definidas e a ordenação da compilação dos ICs fonte são exemplos de aplicações para sistemas tradicionais de gerenciamento de construção, como, por exemplo, o Make (FELDMAN, 1979).

Os sistemas de gerenciamento de construção evoluíram muito desde suas primeiras implementações. Novas funcionalidades, que incluem a execução independente de plataforma, o acesso ao repositório de versões para obter os ICs fonte, a automação da execução de testes e publicação dos resultados, o empacotamento de liberações temporárias, conhecidas como *nightly build*, e a cópia das liberações que passaram nos testes para servidores visíveis aos responsáveis de testes beta passaram a ser requisitos necessários aos sistemas de gerenciamento de construção modernos, como, por exemplo, o Ant (HATCHER e LOUGHRAN, 2004).

No restante desta seção, são apresentadas, com maior detalhamento, algumas abordagens referentes a sistemas de gerenciamento de construção (HATCHER, 2001; FOWLER, 2006; MOZILLA, 2006c). Essas abordagens foram escolhidas visando apresentar diferentes perspectivas de pesquisas realizadas em gerenciamento de construção.

[18] O termo IC derivado representa um IC que pode ser obtido a partir de outros ICs pela aplicação de um procedimento de construção, não sendo necessário o seu armazenamento no repositório.

[19] O termo IC fonte representa um IC que serve como origem para a geração de ICs derivados via um procedimento de construção.

2.5.1 Tinderbox

O Tinderbox (MOZILLA, 2006c) é uma ferramenta de controle da qualidade que visa evitar que sistemas em desenvolvimento fiquem corrompidos por um longo período de tempo. O seu funcionamento na versão 1.0 depende do Bonsai (MOZILLA, 2006a), que é um sistema que possibilita a execução de uma grande variedade de consultas relacionando as informações coletadas pelo CVS. Todavia, a versão 2.0 do Tinderbox não necessita mais do Bonsai.

A atuação do Tinderbox consiste na verificação constante da compilação dos sistemas em desenvolvimento. Caso algum dos sistemas não compile, é feito um conjunto de consultas ao Bonsai para detectar quais são os engenheiros de software suspeitos da sua quebra[20]. Essas consultas procuram por engenheiros de software que fizeram modificações desde a última vez que o sistema foi compilado com sucesso. Os engenheiros de software suspeitos da quebra são notificados de imediato, para que o código seja revisto e o problema que originou a quebra seja sanado.

2.5.2 Automação de testes

Após o processo de construção, normalmente ocorre a execução de baterias de testes para assegurar que o produto a ser liberado está correto. Esses testes são capazes de detectar a ocorrência de quebras lógicas[21]. Diversas técnicas são utilizadas para a automação da execução dos testes em ambientes de integração contínua (HATCHER, 2001; FOWLER, 2006). Geralmente, essas técnicas fazem uso da combinação de sistemas de gerenciamento de construção e *frameworks* de testes.

O sistema definido por FOWLER et al. (2006) roda em uma máquina dedicada, com quatro processadores, que compila continuamente o código fonte dos sistemas em desenvolvimento, sempre que algum engenheiro de software faz *check-in*. Após a

[20] O termo quebra representa a situação onde uma falha de compilação ocorre após a implementação de alguma modificação no sistema. As quebras são nocivas ao desenvolvimento em equipe, pois outros integrantes da equipe estarão impedidos de efetuar testes sobre as suas modificações devido à impossibilidade de compilar o código fonte.

[21] O termo quebra lógica representa a situação onde o código fonte compila de forma correta, mas o sistema não funciona devido a defeitos na lógica de execução. As quebras lógicas são mais difíceis de serem detectadas, pois passam despercebidas pela ferramenta de análise sintática do processo de compilação.

construção, que é apoiada pela ferramenta Ant, o sistema é implantado[22] em um ambiente de desenvolvimento para a execução dos testes. Caso os testes rodem de forma correta, a versão que foi testada é rotulada no repositório de versões. Um e-mail é enviado, logo após a execução dos testes, para todos os engenheiros de software que fizeram *check-in* desde a última verificação. Esse e-mail notifica os resultados dos testes, transferindo para os engenheiros de software suspeitos de quebra, ou quebra lógica, a responsabilidade de consertá-las.

Os ambientes que fazem uso desse tipo de procedimento normalmente impõem novos requisitos aos engenheiros de software. Nesse contexto, um requisito novo é verificar a caixa de e-mail, após fazer *check-in*, para ver se o seu trabalho introduziu quebra no sistema. Há alguns anos esse tipo de requisito seria inviável, devido à demora dos procedimentos de compilação. Contudo, atualmente, a compilação e a execução de testes sobre um sistema de 200.000 linhas de código pode durar menos de quinze minutos, dependendo da máquina a ser utilizada para essa tarefa, o que torna os procedimentos de teste automático viáveis.

2.5.3 Considerações finais sobre sistemas de gerenciamento de construção

A ferramenta Tinderbox, apresentada nesta seção, fornece avanços consideráveis para o apoio ao desenvolvimento distribuído de sistemas em comparação à utilização de ferramentas de controle de versões isoladamente. Com a sua característica de integração contínua, é possível detectar, além dos conflitos, as quebras de compilação do sistema.

Para complementar esse cenário de construção contínua de sistemas, são apresentadas técnicas de automação de testes. Essas técnicas, que podem ser utilizadas para a detecção de quebras lógicas, hoje são viáveis devido aos avanços tecnológicos de hardware e de ferramentas de compilação.

2.6 Integração dos espaços de trabalho de GCS

As atividades de Engenharia de Software lidam com problemas complexos que podem ser agravados se acrescentadas preocupações referentes ao controle de modificações, controle de versões e gerenciamento de construção. Para lidar com essa

[22] O termo implantação (*deploy*) representa a arrumação estratégica do sistema no ambiente de desenvolvimento ou produção (TECHTARGET, 2006).

complexidade, diferentes técnicas de integração dos espaços de trabalho são utilizadas. Essas técnicas visam prover ao engenheiro de software um ambiente que forneça recursos de GCS sem afetar de forma profunda a rotina de trabalho.

Alguns ambientes de programação (IDE – *Integrated Development Environment*), como, por exemplo, o Eclipse (ECLIPSE FOUNDATION, 2006a), o NetBeans (NETBEANS COMMUNITY, 2006) e o JBuilder (BORLAND, 2006b), fornecem recursos de controle de versões integrados. Esses recursos permitem que o programador acesse as versões compatíveis de arquivos de código fonte sem ter que sair da IDE. Usualmente, o acesso a essas versões compatíveis é feito usando o conceito de projeto, que agrupa esses artefatos, facilitando a manutenção da consistência dos mesmos. Contudo, existem iniciativas para a introdução desse conceito na camada de GCS, como, por exemplo, o TCCS (*Trivial Configuration Control System*) (BOLINGER e BRONSON, 1995), que consiste em uma camada implementada sobre o SCCS (ROCHKIND, 1975) ou sobre o RCS, com o objetivo de explorar o conceito de projeto para o versionamento de código fonte.

Além de controle de versões, essas IDEs também permitem o acesso integrado ao gerenciamento de construção. Em alguns casos, o gerenciamento de construção é definido e gerenciado por mecanismos proprietários do IDE. Contudo, o uso de descritores de construção não proprietários está ficando cada vez mais comum. O Ant é um exemplo desses mecanismos de construção não proprietários amplamente adotados nas IDEs voltadas para a programação Java.

Com a adoção dessas infra-estruturas integradas, porém não proprietárias, é possível permitir que diferentes engenheiros de software de um mesmo projeto façam uso de diferentes IDEs para a programação, sem interferir no procedimento de compilação e execução de testes. Por exemplo, cada IDE tem um comando específico para iniciar a compilação, contudo, a execução real desse comando ocorre por meio da interpretação de um descritor de construção. Desta forma, o usuário do JBuilder usará o seu comando habitual de compilação, que é diferente do comando habitual de compilação do Eclipse, mas o efeito será o mesmo, pois ambos os comandos estarão acionando o Ant para a leitura do descritor de construção anexado ao projeto. Nesse cenário, o conhecimento referente à organização dos diretórios do projeto deixa de pertencer às IDEs e passa a pertencer ao descritor de construção, que é um IC do projeto que não depende de IDEs específicas.

No caso de controle de modificações, a integração mais comum ocorre diretamente no produto. O *browser* Firefox (MOZILLA, 2006b) e o sistema operacional Windows XP (MICROSOFT, 2006c) são exemplos dessa integração. Relatórios de erros são automaticamente enviados ao fabricante caso algo aconteça fora do previsto. Além disso, existem ambientes de desenvolvimento de software que fornecem esse recurso de solicitação de modificações integrado, possibilitando que o engenheiro de software faça o requerimento (FIGUEIREDO, 2004).

Outros casos mais avançados de integração se referem à análise das informações referentes à GCS, com o intuito de dar subsídios para a melhoria das atividades de Engenharia de Software. Para atingir esse objetivo, as informações coletadas pelos diferentes sistemas de GCS devem ser organizadas e relacionadas, provendo conhecimento indireto referente ao desenvolvimento de software.

No restante desta seção, são apresentadas, com maior detalhamento, algumas abordagens referentes à integração dos espaços de trabalho de GCS (BALL *et al.*, 1997; DRAHEIM e PEKACKI, 2003).

2.6.1 Mineração de informações

BALL et al. (1997) propuseram que elementos que sofrem modificações em conjunto podem estar relacionados semanticamente. A partir da análise em um repositório existente, foi possível determinar a afinidade entre os arquivos (classes em C++). Inicialmente, cada uma das classes foi classificada em função do seu papel dentro da arquitetura utilizada. A partir dessa classificação, surgiram diferentes tipos de classe. Cada tipo de classe recebeu uma representação gráfica simbolizada por uma forma geométrica e uma cor. Um dos resultados da aplicação da técnica proposta gerou um grafo, onde o tamanho de cada nó, que simboliza uma classe em C++, indica a quantidade de modificações que a classe sofreu.

Um outro resultado ainda mais interessante, também explicitado via um grafo, usa a distância entre os nós para indicar o grau de afinidade existente entre classes em C++. Para construir esse grafo, foi necessária a utilização de uma fórmula que determinasse o tamanho de cada aresta existente entre os nós. Essa fórmula, $Distância\ (C, D) = CD_{mr} \div \sqrt{C_{mr} \times D_{mr}}$, define que a distância entre duas classes C e D é igual à quantidade de solicitações de modificação que afetarem C e D em conjunto dividido pela raiz quadrada da multiplicação da quantidade de modificações que atingiram C e D isoladamente.

Assim sendo, é possível averiguar se a arquitetura proposta para a aplicação distribui as responsabilidades nas classes de forma a proporcionar maior coesão e menor acoplamento durante a manutenção. Esse tipo de resultado também pode ser útil tanto durante a análise de impacto, etapa do processo de controle de modificações, quanto na definição de um novo projeto para a aplicação de manutenção preventiva.

2.6.2 Bloof

Usualmente, os sistemas de controle de versões guardam uma grande gama de informações. Contudo, o apoio para a análise dessas informações não é provido a contento pelo próprio sistema. Com o intuito de contornar esse problema, foi desenvolvido o Bloof (DRAHEIM e PEKACKI, 2003), que é uma interface para a análise dos dados coletados pelo CVS.

O Bloof se conecta a um repositório CVS e extrai as informações necessárias para possibilitar a análise. Essas informações passam a povoar um banco de dados cujo modelo inclui os nomes dos engenheiros de software, os arquivos e as revisões, além dos relacionamentos entre essas entidades. Todas as informações relacionadas com as revisões também são persistidas nesse repositório. Dentre essas informações estão o comentário que o engenheiro de software forneceu ao fazer a revisão, o número de linhas adicionadas e removidas, a versão gerada e o momento em que a revisão ocorreu.

Utilizando a interface de usuário Bloof Browser, é possível executar consultas pré-definidas sobre esse modelo, que são denominadas de medidas. Essas medidas, relatadas na forma de gráficos, mostram a evolução do desenvolvimento referente ao número de linhas de código, número de arquivos, contribuição dos engenheiros de software, impacto das modificações, etc.

Atualmente, o Bloof atua sobre somente um projeto de cada vez, correlacionando as informações referentes ao desenvolvimento desse projeto. Contudo, uma aplicação futura identificada por DRAHEIM e PEKACKI (2003) englobaria o acesso a um grande conjunto de projetos e a aplicação de medições sobre todos esses projetos, permitindo uma posterior análise dos resultados com o objetivo de detectar semelhanças na evolução dos seus desenvolvimentos. Essa detecção de semelhanças em projetos de software já foi tratada por LEHMAN et al. (1997), em um trabalho que constatou que ao menos 5 das 8 leis de Lehman criadas nos anos 70 ainda eram válidas nos anos 90.

2.6.3 Considerações finais sobre integração dos espaços de trabalho de GCS

Foram apresentadas duas abordagens para a extração e análise de informações existentes nos repositórios de versão. Enquanto o Bloof permite que medidas sejam definidas e coletadas sobre o histórico de versões do projeto, a abordagem proposta por BALL et al. (1997) visa detectar, automaticamente, relações entre artefatos por meio da análise das modificações. O uso de mineração de dados neste contexto aparenta ser mais promissor, pois permite ir além das informações explícitas existentes no repositório.

2.7 Considerações finais

Neste capítulo, foram apresentadas várias abordagens de GCS do estado da prática e da arte. Essas abordagens foram agrupadas segundo o tipo principal de contribuição para a GCS. Os cinco grupos definidos foram (1) processos; (2) controle de modificações; (3) controle de versões; (4) gerenciamento de construção; e (5) integração dos espaços de trabalho.

O uso de cada uma dessas abordagens e o formalismo adotado nesse uso podem variar em função do tipo de aplicação a ser desenvolvida. No caso de projetos de software livre, por exemplo, não existe nem interesse e nem recurso para a adoção de processos e sistemas que aumentem a burocracia com a introdução de um maior formalismo (FOGEL e BAR, 2001; ASKLUND e BENDIX, 2002).

Apesar da existência de processos e ferramental de GCS adequado para todos os níveis de formalismo desejados, ainda existem projetos de desenvolvimento de software que ignoram completamente a automação da GCS. Um exemplo clássico dessa situação é o Linux (TORVALDS, 2006), que até pouco tempo não fazia uso de sistema algum de controle de versões. O seu código fonte era organizado em diretórios em função da versão e somente os moderadores podiam aplicar remendos (*patches*) nessas versões. Atualmente, o Linux tem suas versões controladas pelo GIT (TORVALDS e HAMANO, 2006), que é um sistema de controle de versões distribuído, feito pelo próprio Linus Torvalds, autor do Linux.

No cenário brasileiro, de acordo com a última pesquisa sobre Qualidade e Produtividade no Setor de Software Brasileiro (MCT, 2002), referente a 2001, somente 10,4% das empresas entrevistadas (45 empresas de um espaço amostral de 431) praticavam gestão de mudanças. Mais ainda, somente 20,1% das empresas entrevistadas

(85 empresas de um total de 423) faziam uso de ferramentas de gerência de configuração, que é um elemento crítico para o processo de manutenção de software (IEEE, 1998).

No início deste milênio, o mercado mundial de GCS estava movimentando anualmente em torno de um bilhão de dólares, e cada vez mais as empresas têm interesse em introduzir processos de GCS em algum nível nos seus projetos (ESTUBLIER *et al.*, 2005). Contudo, a introdução gradativa no caso de projetos sem nenhum apoio de GCS é vista como a melhor solução (LEON, 2000). Caso uma equipe que não esteja habituada a trabalhar usando GCS passe a fazer uso de todos os sistemas e funções concomitantemente, o nível de burocracia associado pode se elevar rapidamente, sem que haja tempo suficiente para avanços significativos em produtividade e qualidade.

Das abordagens existentes para GCS, a grande maioria tem enfoque em código fonte. Por essa razão, o apoio às etapas iniciais do desenvolvimento de software, que englobam análise e projeto, e às etapas finais do desenvolvimento de software, que englobam implantação e evolução dinâmica, é visto como um campo promissor de pesquisas em GCS (ESTUBLIER *et al.*, 2005).

Independentemente do grau de formalismo adotado, a GCS deve ser vista como uma disciplina útil tanto para clientes quanto para gerentes e engenheiros de software. A GCS possibilita o aumento da transparência na relação entre o cliente e o fornecedor de software ao permitir o acompanhamento detalhado do andamento das solicitações de modificação. Além disso, a GCS fornece os subsídios necessários para que gerentes possam ter conhecimento sobre o andamento do projeto, facilitando a tomada de decisões. Do ponto de vista de desenvolvimento, a GCS é uma disciplina indispensável para controlar a complexidade existente quando equipes numerosas manipulam, concomitantemente, um conjunto de artefatos, evitando a introdução de defeitos e retrabalho e, conseqüentemente, aumentando a qualidade e a produtividade (LEON, 2000).

Capítulo 3 – Gerência de Configuração de Software no Desenvolvimento Baseado em Componentes

3.1 Introdução

No Capítulo 2 foram apresentados sistemas da GCS para o desenvolvimento convencional de software. Contudo, no contexto de DBC, é necessária a adoção de novos processos para reger a interação entre os sistemas de GCS das equipes produtoras e consumidoras de artefatos reutilizáveis (LARSSON, 2000). Além disso, novos problemas referentes à GCS surgem com a mudança do foco de desenvolvimento (LEBSACK *et al.*, 2001), tornando necessária a criação de novos sistemas ou a customização dos sistemas existentes ao novo cenário.

O DBC faz uso de componentes, interfaces e conectores como elementos de estruturação de sistemas. Um componente é visto como parte não trivial, independente e substituível de sistemas (KRUCHTEN, 2001). Componentes, que são partes reutilizáveis de software (D'SOUZA e WILLS, 1998), fazem uso de interfaces descritas de forma contratual para interagir com os demais elementos de software (PAGE-JONES, 1999; SZYPERSKI, 2002). A ligação propriamente dita entre os componentes, que pode ser simples ou complexa dependendo de questões como distribuição, adaptação e coordenação, é obtida pelos conectores (SHAW e GARLAN, 1996; LARSSON, 2000; OMG, 2005c).

Existem diferentes métodos para apoiar o DBC. Dentre os mais conhecidos estão Catalysis (D'SOUZA e WILLS, 1998), UML Components (CHEESMAN e DANIELS, 2000) e KobrA (ATKINSON *et al.*, 2000). Um maior detalhamento sobre os conceitos ou os métodos referentes ao DBC pode ser obtido em diversos trabalhos existentes na literatura (HERZUM e SIMS, 1999; BROWN, 2000; HEINEMAN e COUNCILL, 2001; WALLNAU *et al.*, 2001; GIMENES e HUZITA, 2005), ou em alguns trabalhos realizados na própria COPPE/UFRJ (BRAGA, 2000; TEIXEIRA, 2003; BLOIS, 2006).

Apesar da existência desses métodos de DBC, a carência de processos e ferramental de apoio ainda é grande. Com o uso de DBC, a quantidade de artefatos produzidos durante o processo de desenvolvimento aumenta substancialmente em comparação ao desenvolvimento convencional. Além disso, esses artefatos, que se situam em diferentes níveis de abstração, estão intimamente relacionados por meio do

conceito de componente. A reutilização desses componentes, precedida de adaptações, resulta em novas dimensões de relacionamento entre a versão reutilizável e a versão reutilizada do componente. Essas novas dimensões de relacionamento devem ser rastreadas pelos sistemas de GCS. Contudo, os componentes originais também evoluem com o tempo, e essa evolução deve ser propagada para as suas diversas instâncias reutilizadas de forma consistente, via os rastros existentes (ATKINSON *et al.*, 2001).

Tanto a GCS quanto o DBC têm como principais objetivos o aumento da produtividade, o aumento da qualidade e a redução de custos (KWON *et al.*, 1999). Entretanto, para atingir esses objetivos, é necessário que ambas as técnicas sejam adotadas de forma consistente e integrada.

Neste capítulo, são apresentadas abordagens de GCS voltadas para esse contexto específico de desenvolvimento de componentes reutilizáveis e as suas adoções posteriores em diferentes aplicações. Neste cenário, onde a GCS se torna ainda mais importante, faltam soluções e sobram problemas (ZHANG *et al.*, 2001). Desta forma, são apresentadas, na Seção 3.2, as abordagens que lidam com processos para a adoção efetiva de GCS em organizações calcadas no DBC. Posteriormente, são apresentadas, nas seções 3.3, 3.4 e 3.5, as abordagens focadas nos subsistemas específicos de GCS, que são, respectivamente, controle de modificações, controle de versões e gerenciamento de construção. Na Seção 3.6, são apresentadas abordagens para a integração dos espaços de trabalho de DBC e GCS. Finalmente, a Seção 3.7 conclui o capítulo com algumas considerações finais sobre GCS no DBC.

3.2 Processos de GCS no DBC

O processo de DBC difere do processo de desenvolvimento convencional devido à adição de atividades relacionadas à reutilização e substituição de componentes existentes. Enquanto um ciclo de desenvolvimento convencional é composto pelas macro-atividades de (1) análise, (2) projeto, (3) codificação, (4) testes e (5) implantação, um ciclo de desenvolvimento com reutilização de componentes é composto pelas macro-atividades de (1) busca por componentes existentes, (2) seleção dos componentes encontrados, (3) criação de componentes, caso necessário, (4) adaptação dos componentes, (5) implantação dos componentes em ambiente de produção e (6) substituição dos componentes (LARSSON, 2000).

Devido a essas modificações no processo de engenharia, o processo de evolução também deve ser adaptado à nova realidade. Até então, toda solicitação de modificação

era avaliada e tratada por completo pela equipe de desenvolvimento. Contudo, nesse novo cenário, pode ser necessário delegar parte da avaliação à equipe que implementou determinados componentes utilizados na aplicação. Além disso, decisões referentes a continuar utilizando esses componentes, trocar de fornecedor ou implementar as funcionalidades dentro da própria organização passam a fazer parte das atribuições do CCC.

No restante desta seção, são apresentadas, com maior detalhamento, algumas abordagens referentes a processos de GCS no DBC (KWON *et al.*, 1999; ATKINSON *et al.*, 2001).

3.2.1 MwR

KWON et al. (1999) fornecem processos integrados de GCS, de reutilização e de manutenção para apoiar a evolução de sistemas legados e de bibliotecas de software reutilizáveis. Esses processos são divididos em duas principais perspectivas: (1) DwR, que é um processo de desenvolvimento de aplicações utilizando componentes reutilizáveis, e (2) MwR, que é um processo de manutenção dos componentes reutilizáveis.

No MwR, são discutidas questões referentes à decisão de adaptar os componentes reutilizados por meio de técnicas de caixa-preta ou modificar esses componentes por meio de técnicas de caixa-branca. Segundo KWON et al. (1999), é preferível a adoção de reutilização caixa-preta, a não ser que o esforço para a reutilização caixa-branca seja inferior. Além disso, é realçada a importância dos testes de integração e de regressão dos componentes no contexto das aplicações, principalmente quando esses componentes tiverem sido modificados durante o processo de reutilização.

O MwR assume que as solicitações de modificação ocorrerão somente na etapa de manutenção, não considerando a etapa de desenvolvimento. O processo de manutenção descrito faz uso das atividades previstas nas normas IEEE Std 1042 (IEEE, 1987) e ISO 10007 (ISO, 1995a) para a função de controle da configuração.

Para apoiar o processo MwR, são definidos dois papéis: (1) o reutilizador e (2) o mantenedor. O reutilizador é responsável por povoar o repositório de componentes. Para povoar o repositório de componentes, pode ser necessário adquirir esses componentes de terceiros ou solicitar a construção de novos componentes à equipe de desenvolvimento de componentes, que fará uso, por sua vez, do processo DwR. Já o

mantenedor é responsável por aprovar, implementar e propagar as modificações nos componentes existentes no repositório. O mantenedor também é responsável por coletar informações sobre as modificações. Essas informações são necessárias para responder a perguntas como, por exemplo, quem implementou uma determinada modificação, quais modificações já foram implementadas, quando essas modificações foram implementadas e por que elas foram implementadas.

No MwR, as solicitações de modificação são propagadas tanto para o reutilizador quanto para o mantenedor. Em paralelo, o reutilizador procura por componentes que possam atender às necessidades da solicitação de modificação e o mantenedor verifica quais modificações seriam necessárias nos componentes já existentes. Após a apresentação dos dois laudos, o CCC decide se serão utilizados novos componentes ou se serão modificados os componentes já em uso.

A estratégia proposta pelo MwR faz uso das técnicas de reutilização como recurso no apoio à manutenção. Essa postura difere das demais, que tratam a reutilização como processo principal e acoplam processos auxiliares de GCS com o objetivo de aumentar o nível de controle sobre esse processo principal.

Outra característica do MwR é explicitar a possibilidade de construção de componentes a partir de partes de uma aplicação. Existem situações onde é decidido, inicialmente, não construir um componente para atender as funcionalidades requeridas. Por esse motivo, o desenvolvimento acontece no âmbito da aplicação específica que necessita das funcionalidades. Todavia, com o passar do tempo, pode ser constatado que essas funcionalidades também são interessantes para outras aplicações, iniciando um processo de fatoração para transformar as funcionalidades em um componente e reutilizar esse componente em todas as aplicações, inclusive na inicial.

O tratamento de testes descrito no MwR não contempla testes de integração e de regressão para os componentes no ambiente de desenvolvimento de componentes, antes das suas liberações. Esses testes são executados somente no ambiente de desenvolvimento com componentes, depois que os componentes foram inseridos no contexto de uma determinada aplicação. Entretanto, com o uso de rastros entre os dois ambientes de desenvolvimento e com a especificação de interfaces na forma de contratos (MEYER, 1992), seria possível executar testes de integração e regressão sobre os componentes antes de efetuar a liberação propriamente dita.

3.2.2 KobrA

O método KobrA (ATKINSON *et al.*, 2001) define procedimentos e processos para permitir a evolução de componentes reutilizáveis e reutilizados. Diferentemente do desenvolvimento convencional, é detectada a existência de dois contextos para a solicitação de modificação: (1) nos clientes, referente às aplicações, e (2) nos engenheiros de software de aplicações, referente aos componentes reutilizáveis.

Para apoiar a integração de modificações durante o processo de GCS, são definidas quatro estratégias: (1) no contexto de desenvolvimento de componentes, sempre que um artefato é modificado, essa modificação é propagada para os demais artefatos para manter a consistência; do mesmo modo, (2) no contexto de desenvolvimento com componentes, sempre que um artefato é modificado, essa modificação também é propagada para os demais artefatos; além disso, (3) sempre que um artefato reutilizável é modificado, essa modificação é propagada para os artefatos reutilizados; e, da mesma forma, (4) sempre que um artefato reutilizado é modificado, essa modificação também é propagada para os artefatos reutilizáveis. Por exemplo, nos dois primeiros casos, se o modelo UML de análise pertencente a um componente for modificado, os modelos de projeto e o código fonte desse componente também deverão refletir essa modificação. Por outro lado, nos dois últimos casos, se um erro for corrigido em uma ocorrência de um componente, todas as demais ocorrências desse componente devem refletir essa correção de erro.

Essas estratégias representam os quatro processos principais de manutenção no DBC (ATKINSON *et al.*, 2001): (1) modificação nos componentes reutilizáveis, (2) modificação nos produtos baseados em componentes, (3) propagação das modificações dos componentes reutilizáveis para os produtos e (4) propagação das modificações dos produtos para os componentes reutilizáveis. Cada um desses processos é detalhado, segundo a perspectiva de GCS, por três atividades genéricas: (1) identificação da modificação, responsável por (1.1) estabelecer quais artefatos devem ser modificados, (1.2) determinar o tipo da modificação, e (1.3) armazenar a modificação dentro do conjunto de modificações (*Change Set*) pertinente; (2) análise de impacto, responsável por (2.1) estabelecer quão profundo será o impacto em cada artefato afetado e (2.2) determinar se outras modificações serão necessárias; e (3) propagação da modificação, responsável por alastrar a modificação entre o desenvolvimento de componentes e o

desenvolvimento com componentes, visando obter a consistência mútua dos artefatos reutilizáveis e reutilizados.

Esse processo de reutilização de componentes sugere que quando os artefatos reutilizáveis sofrem modificação, essas modificações devem ser propagadas para os artefatos reutilizados contidos em aplicações previamente geradas. Contudo, segundo o método KobrA, caso os artefatos reutilizados tenham sido modificados desde a sua reutilização, será necessário gerar uma aplicação temporária a partir das novas versões dos artefatos reutilizados e efetuar procedimentos de junção entre a aplicação original e a temporária. Um procedimento análogo é sugerido para o processo de fatoração de componentes.

Entretanto, enquanto as rotinas de construção de aplicações não forem totalmente automatizadas, esse tipo de procedimento não será razoável, tendo em vista a complexidade envolvida na geração de aplicações a partir de artefatos reutilizáveis. Já que algumas modificações nunca são propagadas das aplicações para os artefatos reutilizáveis, a geração de aplicações temporárias deve levar em conta essas modificações, tornando o processo complexo o suficiente para resultar no abandono dos procedimentos de integração por parte dos engenheiros de software e na posterior inconsistência entre os ambientes de desenvolvimento de componentes e desenvolvimento de aplicações. Uma abordagem com maior possibilidade de sucesso é sugerida por VENUGOPALAN (2002) para o desenvolvimento convencional, onde correções de erros são efetuadas em ramos de desenvolvimento auxiliares e posteriormente integrados no ramo principal.

Outra característica questionável da abordagem KobrA se refere ao procedimento de sempre incorporar as modificações específicas de aplicações na linha de produtos (CLEMENTS e NORTHROP, 2001), mesmo quando elas não são de interesse de nenhuma outra aplicação. É sugerido que essas características específicas sejam incorporadas como elementos opcionais, mas o tamanho e a complexidade da linha de produtos aumentariam em função do número de aplicações que fossem geradas a partir dela. Esse cenário se opõe ao cenário em que a linha de produtos tende à estabilidade com o aumento do número de aplicações geradas. Essas aplicações geradas deveriam agregar conhecimento referente aos elementos que são realmente variantes e opcionais, obtidos a partir de processos de fatoração que levam em consideração as suas ocorrências em várias aplicações existentes no domínio (NEIGHBORS, 1980; ARANGO, 1988; PRIETO-DIAZ, 1990).

3.2.3 Considerações finais sobre processos de GCS no DBC

Os processos apresentados nesta seção fornecem características específicas para a GCS aplicada ao DBC. Contudo, o MWR, apesar de fazer uso das atividades descritas nas normas IEEE Std 1042 e ISO 10007, não tem ênfase na reutilização. A reutilização é somente uma técnica utilizada para a manutenção no contexto de DBC. O KobrA, por sua vez, é concebido especialmente para a reutilização de componentes. Todavia, a abordagem faz uso de propagação imediata e obrigatória de modificações, e assume como passo necessário para a incorporação das modificações a regeração da aplicação. Esse tipo de solução somente será viável quando existirem rotinas totalmente automatizadas para o gerenciamento de construção no DBC.

3.3 Sistemas de controle de modificações no DBC

Os sistemas de controle de modificações existentes para o desenvolvimento de software convencional geralmente implementam um processo específico de GCS, não possibilitando a customização desse processo para novas situações. Por esse motivo, o uso desses sistemas se torna impróprio ao contexto de DBC, que é regido por novos processos, a não ser que sejam introduzidas adaptações.

Mesmo que adaptados para atender aos processos de DBC, é importante que os sistemas possibilitem customizações futuras, devido à imaturidade desses processos. Como não existem normas que definam processos estáveis e amplamente utilizados para a GCS aplicada ao DBC, é provável que esses processos evoluam até atingir uma maturidade satisfatória. Os sistemas de controle de modificações que implementam esses processos devem estar preparados para possibilitar que essas adaptações ocorram sem prejudicar os processos em execução.

Além disso, o DBC introduz uma série de características até então não exploradas no desenvolvimento convencional. Várias dessas características afetam a forma em que a GCS deve ser utilizada. Dentre elas está o desenvolvimento de software em vários níveis, onde a equipe de desenvolvimento de componentes situada no nível N fornece componentes para equipes de desenvolvimento com componentes situadas no nível N-1, mas que faz uso de componentes providos por equipes situadas no nível N+1, como exibido na Figura 3.1.

Nesse cenário, é fundamental que o sistema de controle de modificações de cada equipe de desenvolvimento seja capaz de apoiar a evolução não somente dos ICs que

estão sendo construídos ou mantidos, mas também dos componentes adquiridos de terceiros. Para representar esses componentes externos dentro do sistema de controle de modificações, pode ser necessária a utilização de documentos específicos, que possibilitem o armazenamento e posterior consulta sobre os dados de contato do fabricante do componente (KWON *et al.*, 1999; LARSSON, 2000).

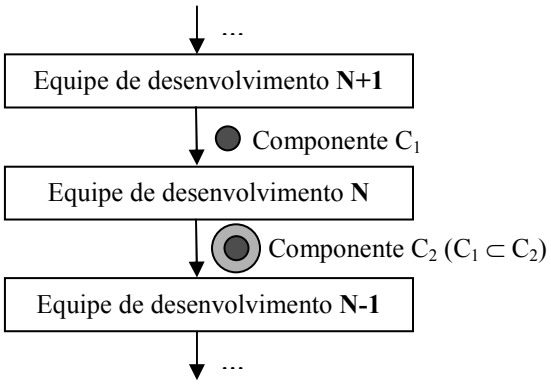


Figura 3.1: Desenvolvimento de software em vários níveis.

No restante desta seção, são apresentadas, em maior detalhe, algumas abordagens referentes a sistemas de controle de modificações no DBC (KWON *et al.*, 1999; ATKINSON *et al.*, 2001).

3.3.1 TERRA

O sistema TERRA (KWON *et al.*, 1999) é um protótipo que implementa o processo MwR de manutenção de componentes e sistemas legados, descrito na Seção 3.2.1. Ele possibilita o registro de novos componentes, a solicitação de modificações sobre os componentes e o acesso e reutilização dos componentes. Todas essas atividades são efetuadas pela Internet.

O registro de componente ocorre por meio do preenchimento de um formulário com as seguintes informações obrigatórias: identificador do componente, nome do componente, nome do autor, data de criação, data de registro, nome do mantenedor, sistemas operacionais compatíveis, linguagem de programação utilizada, formato (modelo, código, binário, etc.), domínios relacionados, métodos e técnicas relacionados e palavras-chave.

A etapa de solicitação de modificações, conforme implementada no sistema TERRA, engloba a solicitação, a classificação, o armazenamento e a recuperação das solicitações de modificação. O formulário de solicitação de modificações necessita obrigatoriamente das seguintes informações: identificador da solicitação de

modificação, nome do requerente, data da solicitação, tipo da solicitação (manutenção em componentes em produção, manutenção em componentes em desenvolvimento, etc.) e situação da modificação. As seguintes informações são opcionais: sistemas legados relacionados, identificadores dos componentes relacionados, versão dos componentes relacionados, tipo da manutenção (corretiva, evolutiva, preventiva ou adaptativa), descrição da modificação e razão da modificação.

O CCC faz uso de um terceiro formulário para expedir pareceres sobre a aprovação das solicitações de modificação. Segundo KWON et al. (1999), uma aprovação pode estar associada a várias solicitações de modificação e pode afetar diferentes versões de diversos componentes. As informações contidas nesse formulário são: identificador da aprovação, nome do expedidor, data da aprovação, identificadores das solicitações de modificação relacionadas, sistemas legados relacionados, identificadores dos componentes relacionados, versão dos componentes relacionados, identificadores das aplicações relacionadas, linhas de produtos relacionadas, tipo da manutenção, situação das solicitações de modificação, especificação da modificação, data prevista para a implementação e estimativas de custo e esforço.

Estranhamente, o registro do componente não possibilita a entrada de informações referentes a versões, nem a descrição dos rastros entre as diversas versões de um componente e as aplicações que reutilizam esse componente. Além disso, as informações de estimativas de custo e esforço, conforme proposto, são fornecidas pelo CCC. Porém, essa responsabilidade não é adequada, pois o CCC consiste de um comitê essencialmente gerencial, que faz uso de laudos técnicos para avaliar a viabilidade de uma modificação. Esses laudos técnicos deveriam ser de responsabilidade dos analistas de impacto, segundo as normas IEEE Std 1042 (IEEE, 1987) e ISO 10007 (ISO, 1995a).

Da forma em que o TERRA está implementado, o processo MwR é definido diretamente no código fonte em Perl (WALL *et al.*, 2000), comprometendo a sua evolução. Por esse motivo, seria necessária a modificação do código fonte do TERRA para que melhorias no processo pudessem ser incorporadas, corrompendo todas as instâncias de processos em execução. Mais ainda, o TERRA não fornece uma integração satisfatória entre os diversos formulários, deixando a cargo da pessoa responsável pelo preenchimento a responsabilidade de localizar as informações necessárias. Várias dessas informações poderiam ser preenchidas automaticamente pelo sistema. Além disso, o TERRA atua de forma semelhante ao Bugzilla, fazendo uso da

edição direta de um campo de situação para designar o novo estado das solicitações de modificação.

3.3.2 KobrA

A abordagem sugerida pelo método KobrA (ATKINSON *et al.*, 2001) para apoiar o sistema de controle de modificações se baseia no uso da técnica de conjuntos de modificações (*change sets*) (SMDS, 1994). A técnica de conjuntos de modificações consiste na definição explícita do conceito de modificação e na associação desse conceito a todas as diferenças geradas pela comparação das versões dos artefatos antes e depois da modificação ser implementada. Desta forma, é possível aplicar uma determinada modificação sobre qualquer linha base. A aplicação da modificação é feita via junção de cada uma das diferenças relacionadas com os respectivos artefatos da linha base.

Essas modificações são agrupadas, formando então um conjunto de modificações. Cada conjunto de modificações representa uma evolução que agrega valor suficiente para ser, por exemplo, considerada a geração de novas liberações. Esses conjuntos de modificações podem ser aplicados sobre configurações que evoluíram separadamente, permitindo que essas incorporem as novas funcionalidades. Por exemplo: a configuração C_1 foi entregue a um cliente, incluindo os modelos e código fonte. Esse cliente efetuou adaptações para customizar o software para o seu problema, transformando a configuração em C_1'. Contudo, em paralelo, a equipe de desenvolvimento corrigiu defeitos e incluiu funcionalidades importantes em C_1, gerando a configuração C_2. O cliente, que tem muito interesse nessas melhorias, pode solicitar o conjunto de modificações $M_{C1\text{-}C2}$ que represente a diferença entre as duas configurações C_1 e C_2 ($M_{C1\text{-}C2} = C_2 - C_1$). Ao aplicar $M_{C1\text{-}C2}$ sobre C_1', e efetuar as verificações de consistência necessárias, será obtida a configuração C_2', que inclui tanto as correções de defeito e novas funcionalidades quanto as customizações feitas pelo próprio cliente, como exibido na Figura 3.2.

O principal argumento para a utilização desse tipo de abordagem no DBC é a necessidade de transferir modificações entre os contextos de desenvolvimento de componentes e desenvolvimento com componentes. Assim, adaptações feitas pela equipe de desenvolvimento de um produto específico podem ser transferidas para a linha de produtos associada, caso seja útil para os demais produtos. Além disso, o uso de descrições de modificação em conjunto com a identificação dos artefatos afetados

permite que questões referentes a quem, quando, como, onde, o quê e por quê possam ser respondidas, por meio de consultas sobre o sistema de controle de modificações e sobre o sistema de controle de versões. Uma outra sugestão importante é o uso de associações de causa no modelo de modificações, viabilizando a adoção de técnicas de análise causal no caso de modificações corretivas (LEON, 2000).

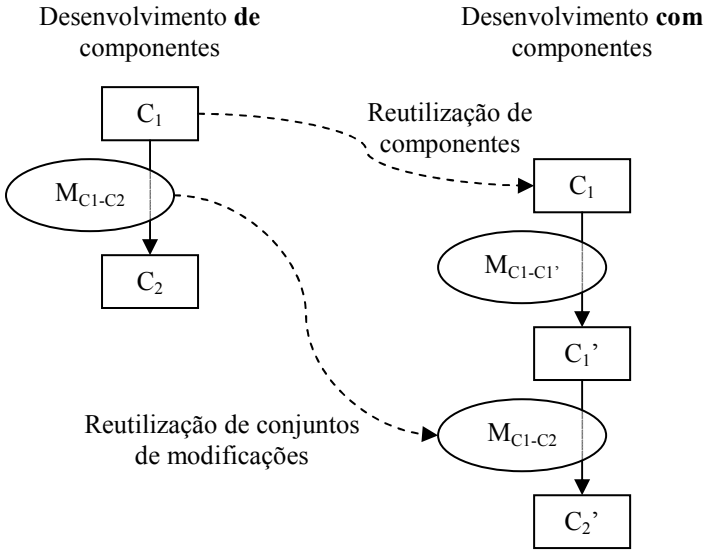


Figura 3.2: Reutilização de conjuntos de modificações.

Para que a técnica de conjuntos de modificações atinja o seu objetivo, é importante fazer uso de mecanismos de detecção de diferenças que possibilite o armazenamento relativo das diferenças, para que a posterior junção dessas diferenças faça sentido. Por exemplo, quando um código fonte CF é modificado na configuração C_1, não é desejável registrar que a linha 123 foi editada, pois a linha 123 do código fonte CF na configuração C_2 pode ser outra ou, em algumas situações, nem existir. O correto é detectar a posição relativa da modificação, de forma que seja possível encontrar a linha editada mesmo que ela esteja em outra posição. Só assim a técnica de conjuntos de modificações pode ser utilizada em todo o seu potencial.

3.3.3 Considerações finais sobre sistemas de controle de modificações no DBC

A maior deficiência dos sistemas de controle de modificações no contexto de DBC é referente à dificuldade de adaptação do processo de GCS sem afetar os projetos em execução. Além disso, existe uma necessidade especial relativa à manutenção das informações sobre fornecedores de componentes para a propagação das solicitações de

modificação, caso necessário. O sistema TERRA, apesar de não contribuir para a questão da customização e evolução do processo, atua no problema de registro de componentes, fazendo uso de um conjunto fixo de informações.

O KobrA, apesar de não atuar diretamente em nenhum desses dois problemas, apresenta outras duas características importantes para sistemas de controle de modificações no contexto de DBC: conjunto de modificações e análise de causa. Essas características viabilizam, respectivamente, a transferência de modificações entre os ambientes de desenvolvimento de componentes e de desenvolvimento com componentes e a detecção do real motivo da ocorrência de defeitos.

3.4 Sistemas de controle de versões no DBC

Com a adoção do DBC, se torna necessária a utilização de novos tipos de artefato para permitir a representação de componentes, interfaces e conectores em diferentes níveis de abstração. Esses novos tipos de artefato, que englobam modelos de componentes e descritores de implantação, necessitam de mecanismos especializados para o controle de versões.

Apesar dos mecanismos convencionais de controle de versões poderem ser utilizados para apoiar a evolução desses novos tipos de artefato, a perda semântica é significativa, pois esses sistemas convencionais não têm conhecimento referente aos conceitos existentes no domínio de DBC. Em algumas situações onde o conhecimento referente aos conceitos de DCB existe, as soluções para o problema de evolução não são satisfatórias, como, por exemplo, o acúmulo de interfaces utilizado pelo COM (MICROSOFT, 2006a) devido à impossibilidade de modificar uma interface existente (LARSSON, 2000).

Outra característica diferenciada, introduzida pelo DBC, é o alto grau de encapsulamento que o conceito de componente apresenta. Como os componentes não têm relacionamentos diretos com outros componentes, é possível isolar o impacto da modificação de um componente desde que as suas interfaces permaneçam intactas (LARSSON, 2000). Em razão disso, o versionamento de componentes está fortemente dependente do versionamento das interfaces e essa dependência contribui para a diminuição da complexidade associada à construção e manutenção dos diversos componentes de uma arquitetura.

No restante desta seção, são apresentadas, com maior detalhamento, algumas abordagens referentes a sistemas de controle de versões no DBC (CHRISTENSEN,

1999a; ATKINSON *et al.*, 2001; DASHOFY *et al.*, 2001; CHEN *et al.*, 2003). Diferentemente do que pode ser constatado para o desenvolvimento convencional, o apoio de controle de versões para o DBC é deficiente e necessita maior maturidade, que só será obtida como conseqüência do aumento das pesquisas na área.

3.4.1 RCM

A grande maioria dos sistemas de controle de versões trata arquivos do sistema operacional como unidade de abstração para o versionamento. Essa postura se torna especialmente inadequada no contexto de DBC, onde diversas entidades ficam usualmente armazenadas em um mesmo arquivo. Para possibilitar o controle de versões de arquiteturas de componentes, foi proposto o RCM (CHRISTENSEN, 1999a), criando uma camada de abstração em relação aos arquivos que armazenam os modelos e código fonte desses componentes.

O RCM, que é implementado no contexto do ambiente de desenvolvimento de software Ragnarok (CHRISTENSEN, 1999b), atua no nível lógico, controlando a versão de componentes. Esse nível lógico mapeia, por meio de atributos, o nível físico, englobando os artefatos que implementam o componente, como, por exemplo, modelos e código fonte.

Um componente é representado por uma tupla (CID,VID,S_{sub},S_{rel}), onde CID representa o identificador do componente, VID representa a versão do componente, S_{sub} contém as dependências para artefatos no nível físico que implementam o componente e S_{rel} contém as dependências para outros componentes no nível lógico.

Utilizando essa estrutura, as funcionalidades de GCS foram remodeladas para atender às necessidades de controle de versões de componentes. A funcionalidade de *check-in* foi concebida de forma a atuar recursivamente, criando uma nova versão para cada componente relacionado que tenha sido modificado. De forma semelhante, a funcionalidade de *check-out* atua recursivamente, buscando todos os componentes que dependem em algum nível do componente solicitado.

Além disso, são fornecidos recursos de ramos e diferença arquitetural entre componentes, que é o fundamento necessário para a funcionalidade de junção. A junção ocorre via união dos conjuntos S_{sub} e S_{rel} entre os respectivos componentes. Apesar dessa abordagem ser suscetível a efeitos colaterais, o autor argumenta que, para casos onde as junções ocorrem com razoável freqüência, tudo acontece dentro do esperado. Todavia, para possibilitar a junção de artefatos no nível físico, é necessário fazer uso

dos algoritmos de junção referentes a cada tipo de artefato. Além disso, embora seja fornecido um mecanismo automático para a junção de componentes, é salientada a necessidade de intervenção manual nas situações onde os componentes se diferem substancialmente.

Apesar de trazer características desejáveis de GCS para o domínio de DBC, a abordagem não permite que, a partir de uma determinada configuração, existam dependências para mais de uma versão de um mesmo componente. Essa restrição é questionável em cenários reais, onde, por exemplo, um componente C_1 desenvolvido em Java depende dos componentes C_2 na versão 1.0 e Log4j na versão 1.2.8. Contudo, o componente C_2, por ser mais antigo, depende do Log4j na versão 1.1.3. Desta forma, o componente C_1 tem dependências em diferentes níveis para diferentes versões de um mesmo componente, o Log4j, como exibido na Figura 3.3. Esta situação, que não é permitida pelo RCM, poderia ser tratada ao invés de proibida (LÜER e VAN DER HOEK, 2004).

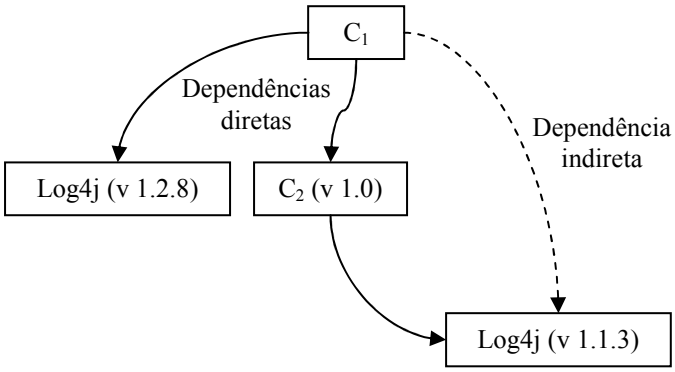


Figura 3.3: Dependências para diferentes versões de um mesmo componente.

Mais ainda, o tratamento dos artefatos pertencentes ao nível físico necessita do uso dos atuais sistemas de GCS, impossibilitando o RCM de atuar em níveis mais granulares, que envolvem, no caso particular da orientação a objetos, os conceitos de classes, atributos e métodos.

3.4.2 KobrA

A abordagem sugerida pelo KobrA (ATKINSON *et al.*, 2001) para o sistema de controle de versões não faz uso de configuração como um elemento explícito. Para o KobrA, os artefatos são descritos por meio de um conjunto de dependências para outros artefatos, além das propriedades do próprio artefato. Desta forma, uma configuração pode ser obtida quando, a partir de um artefato qualquer, são percorridas todas as dependências e encontrados todos os artefatos compatíveis.

Essas dependências são descritas em diferentes perspectivas. Um componente, por exemplo, está relacionado com versões compatíveis de especificações, realizações e implementações por meio de dependências do tipo <<*contains*>>. Dentro do componente existem informações sobre quais combinações desses elementos geram configurações consistentes. Além disso, um componente também pode se relacionar com outros componentes, determinando estruturas de decomposição. Esse relacionamento ocorre no contexto de um IC de maior granularidade, que mantém as informações referentes à suas partes sem a necessidade do uso do conceito explícito de configuração. Desta forma, cada IC armazena a configuração dos outros ICs que o compõem, recursivamente. Uma grande vantagem dessa recursão é possibilitar que, com uma metáfora homogênea, recursos de versionamento possam ser aplicados uniformemente, abrangendo desde arquiteturas completas de componentes até elementos individuais de modelagem UML.

Um outro mecanismo de dependências utilizado, do tipo <<*derived*>>, consiste no relacionamento existente entre as ocorrências de um mesmo componente no ambiente de desenvolvimento de componentes e no ambiente de desenvolvimento com componentes. Esse tipo de relacionamento é fundamental para possibilitar a notificação de modificação e posterior intercâmbio dos conjuntos de modificações entre essas ocorrências do componente.

Para controlar a versão dos ICs, é sugerido o uso de um vetor com quatro informações de versionamento: (1) revisão, tratada pelo KobrA como modificação semântica, (2) variante, (3) edição, tratada pelo KobrA como modificação cosmética, e (4) estado de qualidade. Além disso, a própria dependência entre os ICs também armazena a informação do estado de qualidade. Assim, é possível definir mecanismos automáticos com linguagens como OCL para evoluir os artefatos e atribuir estados de qualidade que indiquem a necessidade de uma posterior verificação manual. Por exemplo, no caso da modificação da especificação de um componente, as novas versões de realização, implementação, componentes relacionados e do próprio componente poderiam ser associadas automaticamente à nova versão da especificação. Contudo, o estado de qualidade dessa associação deveria relatar a necessidade de inspeção.

3.4.3 xADL

DASHOFY et al. (2001) apresentam uma linguagem extensível para a descrição de arquiteturas, denominada xADL, que possibilita, entre outros recursos, o

versionamento dos elementos que compõem a arquitetura. A xADL é considerada extensível pois todos os seus recursos são mapeados em módulos individuais, implementados em XML Schema (W3C, 2001). Um desses módulos é utilizado para prover recursos de grafos de versões para componentes, interfaces e conectores.

Esse recurso de versionamento provido pela xADL permite que diferentes versões de um componente possam ser utilizadas em diferentes locais de uma mesma arquitetura. Além disso, o grafo de versões pode relatar tanto a evolução de um único componente quanto a evolução de uma arquitetura como um todo.

Outros módulos providos pela xADL, que se relacionam com a GCS, permitem a descrição dos elementos variantes e opcionais das arquiteturas. Com o módulo de condições booleanas ativo, é possível definir fórmulas lógicas para estabelecer as dependências entre os elementos variantes, opcionais e os demais elementos.

Além disso, existe um módulo responsável pela representação das diferenças entre arquiteturas. Com o uso desse módulo juntamente com as ferramentas apropriadas, também pertencentes à xADL, é possível comparar arquiteturas com o intuito de obter a diferença ($D = diff[A_1, A_2]$) ou a arquitetura resultado da aplicação de uma diferença sobre uma arquitetura base ($A_2 = merge[A_1, D]$).

Uma das aplicações da xADL consiste no apoio à representação de linhas de produtos. Essas linhas de produtos podem ser refinadas de diferentes maneiras, encadeando na geração de diferentes produtos. CHEN et al. (2003) apresentam uma abordagem para a comparação entre duas versões de produtos oriundas de uma determinada linha de produtos e a posterior junção do resultado dessa comparação na linha de produtos.

Os algoritmos utilizados para efetuar comparação e junção, que são baseados em algoritmos definidos anteriormente por WESTHUIZEN et al. (2002), são úteis tanto para as arquiteturas de componentes voltadas para um produto específico ou para as arquiteturas de componentes de linhas de produtos. Esses algoritmos diferem dos anteriores em relação a características necessárias, introduzidas para suportar linhas de produtos, como, por exemplo, o uso de similaridade ao invés de identificadores e o uso de granularidade fina para comparar os elementos das linhas de produtos.

Apesar do algoritmo de comparação e junção não fazer uso de identificadores, o seu principal recurso para a detecção de similaridade entre componentes é o nome do componente. Esse tipo de técnica pode gerar resultados indesejáveis em domínios novos ou instáveis, onde os engenheiros de software não têm certeza sobre a nomenclatura

mais adequada. Além disso, não é fornecido nenhum apoio para a comparação e junção de diferentes linhas de produtos. Esse apoio seria interessante para possibilitar a detecção de outras linhas de produtos mais abrangentes.

3.4.4 Considerações finais sobre sistemas de controle de versões no DBC

Das abordagens apresentadas, a RCM e a KobrA atuam principalmente no versionamento de componentes. A RCM define o mapeamento entre as versões de componentes e as versões dos demais artefatos, incluindo código fonte. Contudo, obriga que somente uma versão do componente seja utilizada por projeto. O Kobra, por sua vez, faz uso do conceito de configuração por meio dos próprios relacionamentos entre artefatos e fornece apoio explícito para reutilização. Foram também apresentadas abordagens para o controle de versões sobre arquiteturas de linha de produtos.

Independentemente da abordagem, o foco principal é no controle de versões dos componentes em alto nível de abstração, propagando as ações de GCS para as demais ferramentas de controle de versões nos níveis de abstração inferiores. Todavia, o apoio ao controle de versões nos níveis de abstração intermediários, que lidam com modelos de análise e projeto, continua deficiente.

3.5 Sistemas de gerenciamento de construção no DBC

De forma semelhante ao desenvolvimento de software convencional, o DBC necessita de mecanismos que possibilitem a estruturação dos ICs para construir um ambiente de trabalho condizente com o paradigma utilizado. Além dos problemas já conhecidos, novos problemas surgem quando o objetivo deixa de ser a construção de uma única aplicação e passa a ser a construção de componentes para atender a famílias de aplicações. Dentre esses problemas estão o tratamento da recursividade existente na dependência entre componentes e a necessidade de definição de partes opcionais e variantes dentro de um componente.

Apesar de autocontidos, componentes se caracterizam fortemente pela reutilização de serviços providos por outros componentes. Desta forma, a seleção de uma versão de um componente dentro do repositório de GCS implica a seleção de versões de outros componentes. Este encadeamento de dependências entre componentes é, normalmente, descrito via um modelo de sistema (*system model*) (KWON *et al.*, 1999; ESTUBLIER *et al.*, 2005). Contudo, o procedimento de seleção ocorre de forma recursiva, restringindo o espaço de versões a cada iteração. Esse tipo de

comportamento, que pode levar à necessidade de substituição de alguns componentes da seleção inicial por motivos de inconsistência, difere do que é habitual no desenvolvimento convencional de sistemas, e, conseqüentemente, requer um tratamento diferenciado.

Além da dimensão das versões, os componentes podem ser selecionados segundo dimensões de obrigatoriedade e variabilidade, ou a combinação de ambas. O tratamento de obrigatoriedade e variabilidade, que consiste na troca de comportamento em determinados pontos do ciclo de vida do software (SVAHNBERG *et al.*, 2005), também é diferenciado em relação ao desenvolvimento convencional de software. Apesar de existirem várias propostas conhecidas no assunto (e.g.: diretivas de compilação, arquivos de configuração, etc.) (SVAHNBERG *et al.*, 2005), a preocupação dessas propostas é referente ao problema da construção de um único sistema, não levando em conta as características existentes em coleções de sistemas semelhantes.

A liberação de sistemas construídos no DBC também diferencia da liberação de sistemas construídos no desenvolvimento tradicional. A principal característica é a separação do papel de engenheiro de software em dois novos papéis: o de engenheiro de componentes e o de engenheiro de aplicações reutilizando componentes. Mais ainda, na maioria dos cenários, o próprio engenheiro de componentes exerce o papel de reutilizador de componentes quando parte do componente em desenvolvimento depende de serviços providos por componentes já existentes.

Nesse novo contexto, a liberação de linhas bases passa a ocorrer tanto quando os componentes recém construídos são providos às equipes de desenvolvimento de aplicações, quanto quando as aplicações são fornecidas aos clientes propriamente ditos. Esse processo de liberação deve ser acompanhado de forma minuciosa, catalogando informações sobre os provedores e os usuários das linhas bases em questão, para possibilitar a notificação futura no caso de surgimento de erro ou novas versões, respectivamente.

As equipes de desenvolvimento de componentes devem executar testes sobre os componentes a serem liberados, e fornecer, sempre que possível, os dados, planos e resultados dos testes junto com os componentes. Apesar dos componentes já serem testados no ambiente produtor, é fundamental que também sejam testados no ambiente consumidor, para assegurar que a qualidade declarada pelo produtor realmente existe fora do ambiente controlado de desenvolvimento do produtor (LARSSON, 2000).

No restante desta seção, são apresentadas, com maior detalhamento, algumas abordagens referentes ao gerenciamento de construção no DBC (LARSSON e CRNKOVIC, 2000; LEBSACK *et al.*, 2001; ZHANG *et al.*, 2001; VAN DER HOEK, 2004).

3.5.1 Navegador de dependências

Em arquiteturas de componentes, é crucial a obtenção de configurações consistentes, compostas por versões de componentes compatíveis. Para apoiar essa tarefa de manutenção da consistência das configurações, LARSSON et al. (2000) propõem um modelo para a verificação de dependências entre componentes. Para permitir o controle das dependências, é introduzida uma representação baseada em grafo. Essa representação é armazenada em um repositório de versões para possibilitar a evolução controlada da mesma.

O principal objetivo desse trabalho é responder a cinco perguntas, que são (LARSSON, 2000): (1) Quais componentes foram adicionados ou removidos depois de uma modificação na configuração? (2) Quais dependências foram adicionadas, removidas ou afetadas por uma determinada modificação na configuração? (3) Se um componente é modificado, quais outros componentes no sistema são afetados? (4) Qual é o efeito no sistema quando um novo componente é implantado? (5) Qual é a diferença entre duas configurações determinadas?

Para atender a essas necessidades, é proposto um esquema de identificação de componentes que utiliza o nome, o momento de criação, o tamanho e um número gerado pelo compilador como identificador composto. Além disso, são formalizadas operações de busca por dependências em grafos e detectadas duas estruturas possíveis para a representação dos grafos: matrizes e listas.

A partir da representação matricial de dependências, composta por binários que indicam se existe dependência entre dois elementos, são definidos algoritmos que possibilitam a detecção das diferenças. Para isso, o tamanho das matrizes é inicialmente igualado e, posteriormente, uma matriz é subtraída da outra, resultando na matriz diferença.

Por meio de um navegador de dependências (*dependency browser*), é feita uma análise das duas matrizes que foram comparadas e da matriz resultado da comparação, visando responder, mesmo que superficialmente, às perguntas propostas como objetivo do trabalho. A superficialidade das respostas é devido ao pouco conhecimento inserido

no modelo de dependências. Como esse modelo lida somente com componentes em tempo de execução, não é possível tratar questões referentes ao projeto interno dos mesmos.

3.5.2 CMPro

Em um trabalho semelhante ao navegador de dependências, LEBSACK et al. (2001) apresentam a ferramenta CMPro, que possibilita o controle de dependências de componentes COTS (*Commercial Off-The-Shelf*). A principal motivação do CMPro é permitir a identificação e a propagação das informações referentes a componentes COTS pelos diversos projetos de desenvolvimento com componentes numa mesma organização.

Mesmo existindo a possibilidade de construir componentes dentro da organização, LEBSACK et al. (2001) argumentam que componentes COTS podem reduzir drasticamente o tempo e o custo necessários para povoar um repositório de componentes. Apesar desse benefício, existem algumas desvantagens, como, por exemplo, a inexistência de informações referentes aos componentes COTS nos sistemas de controle de versões da organização, pois o desenvolvimento e a manutenção dos componentes COTS ocorrem fora da organização. Além disso, os componentes COTS não são usualmente distribuídos com documentação completa que inclua modelos e código fonte, forçando a reutilização do tipo caixa preta.

Devido à norma IEEE Std 828 (IEEE, 2005) tratar superficialmente questões referentes à gerência de componentes COTS por meio de subcontratos, alguns problemas ainda necessitam de solução, como, por exemplo, o acompanhamento do conhecimento referente aos diversos pontos em que componentes COTS são reutilizados.

O CMPro se propõe a solucionar esses problemas por meio do uso de um banco de dados de GCS que contenha a representação lógica de componentes COTS armazenada como IC. O componente propriamente dito não é controlado, mas é criado um IC que representa esse componente logicamente e possibilita a definição de dependências entre os componentes e as aplicações que fazem uso desse componente.

As informações necessárias para povoar esse banco de dados de GCS são, entre outras: nome do projeto, número da versão, data de modificação, localização, componentes dependentes e versões anteriores do mesmo componente. Essas informações podem ser utilizadas tanto para análises de impacto quanto para auditorias.

Com o uso do conceito de projeto, é possível tratar um componente de duas formas distintas: (1) componente caixa preta, que é considerado um IC granular, faz parte de um projeto e interage com outros componentes, ou (2) componente caixa branca, que é considerado um IC composto por diversos outros ICs granulares e é representado no CMPro pelo conceito de projeto. Essa segunda opção, apesar de citada, não é utilizada no CMPro, que lida principalmente com o desenvolvimento com reutilização calcado no uso de componentes COTS caixa preta, deixando em segundo plano preocupações referentes a componentes caixa branca.

Para que o CMPro forneça os serviços de geração de relatórios descritos como essenciais para a análise de impacto e auditoria, é necessária a aquisição e o armazenamento das informações referentes às dependências entre componentes COTS. O processo manual para a aquisição e manutenção dessas informações pode tornar o banco de dados de GCS desatualizado devido à complexidade envolvida. Seria desejável que mecanismos automatizados acessassem informações já existentes nos sistemas de controle de versões e no sistema de controle de modificações para povoar essa base de dados de GCS.

3.5.3 JBCM

ZHANG et al. (2001) propõem um sistema de controle de versões voltado para o DBC baseado em um modelo proposto por MEI et al. (2001). Esse sistema, nomeado JBCM, se baseia na existência de dois tipos de componentes: (1) constituintes primitivos e (2) constituintes compostos. Os constituintes primitivos são implementados via de linguagens de programação e os constituintes compostos são obtidos pela conexão de constituintes primitivos via linguagens de descrição de componentes ou linguagens de descrição de arquiteturas. Desta forma, os constituintes compostos são tratados como linhas bases de constituintes primitivos.

A filosofia adotada por ZHANG et al. (2001) em relação à metáfora de controle de versões de componentes condiz com os requisitos de uniformidade levantados por ESTUBLIER (2000). Essa filosofia consiste em fornecer uma única metáfora para tratar elementos primitivos ou compostos. Logo, constituintes primitivos evoluem por meio de versões e constituintes compostos evoluem por meio de linhas bases, sempre que seus constituintes primitivos evoluem. Portanto, o conceito de versão está para o constituinte primitivo assim como o conceito de linhas bases está para o constituinte

composto, fazendo com que um constituinte composto seja uma configuração formada por outros constituintes, sejam eles primitivos ou compostos.

Apesar da abordagem tratar do controle de versões de componentes, a implementação utiliza algoritmos baseados no RCS (TICHY, 1985). Esses algoritmos fazem uso de arquivos do sistema operacional para simular as versões de constituintes primitivos. Os arquivos que fazem parte funcional dos constituintes primitivos também são tratados por meio desses algoritmos.

3.5.4 Variabilidade ao longo do ciclo de vida

O processo de seleção de variabilidade em sistemas depende usualmente de técnicas específicas, que se aplicam em fases determinadas do ciclo de vida do software. Por exemplo: durante o início da especificação de aplicações, no contexto da engenharia de domínio (NEIGHBORS, 1980; ARANGO, 1988; PRIETO-DIAZ, 1990), é possível selecionar as características desejadas utilizando técnicas de recorte, como a proposta por MILER (2000); durante a codificação de aplicações, é possível usar diretivas de compilação; durante a instalação de aplicações, é possível fazer uso de sistemas de seleção de módulos; durante a execução de aplicações, é possível utilizar mecanismos de carga dinâmica de *plug-ins*.

Contudo, devido à especificidade dessas técnicas, um mesmo projeto necessita do uso de cada uma delas em fases específicas, aumentando a sua complexidade. Para contornar esse problema, VAN DER HOEK (2004) propôs o uso de uma infra-estrutura genérica para modelar as obrigatoriedades e variabilidades por meio de linhas de produtos e estruturas específicas que selecionam os artefatos modelados nas diversas fases do ciclo de vida.

A estrutura genérica compreende um formalismo de representação de variabilidade, implementado na xADL 2.0 (DASHOFY *et al.*, 2001), e uma ferramenta para especificação e seleção de linhas de produtos, implementada no Ménage (GARG *et al.*, 2003). A estrutura específica compreende um conjunto de diversas ferramentas, uma para cada fase do ciclo de vida, que interpretam a linha de produtos resultante de sucessivas seleções e aplicam os cortes estipulados nos artefatos de suas competências.

Para mostrar a viabilidade da solução, diferentes aplicações tiveram suas arquiteturas modeladas segundo essa técnica e foram alvo de seleções de variabilidade em momentos distintos: projeto, invocação e execução. Dois tipos de seleção foram tratados nesse estudo, que são: variações e versões. Como resultado, foi detectada a

necessidade de uma maior automação no apoio ao controle de versões de componentes de software, pois o mapeamento manual entre as tecnologias de DBC e a infra-estrutura existente de GCS é altamente suscetível a erro.

3.5.5 Considerações finais sobre sistemas gerenciamento de construção no DBC

Tanto a abordagem do navegador de dependências quanto a abordagem CMPro atuam na manutenção de dependências entre componentes, com enfoque em componentes em tempo de execução e componentes COTS, respectivamente. O navegador de dependências, por atuar em componentes em tempo de execução, faz uso de um modelo simples, o que dificulta a inferência de relações complexas entre os componentes.

O JBCM, em contrapartida ao CMPro, possibilita o controle de construção sobre componentes caixa-branca, atribuindo o conceito de linha base a componentes compostos e associando mecanismos existentes de controle de versões aos componentes primitivos. Em ambas as abordagens, existem mapeamentos entre componentes físicos e elementos lógicos, usando arquivos do sistema operacional.

A dificuldade de controlar a construção e liberação de componentes é relatada tanto por LEBSACK et al. (2001) quanto por VAN DER HOEK (2004), indicando que mecanismos automatizados são necessários para apoiar a seleção de componentes e povoar o espaço de trabalho com a configuração correta dos artefatos que descrevem esses componentes.

3.6 Integração dos espaços de trabalho de GCS no DBC

De forma análoga à necessidade de integração dos espaços de trabalho de GCS e desenvolvimento convencional, o DBC necessita que os conceitos de GCS sejam encapsulados nos conceitos já existentes de componente, interface e conector. Contudo, apesar dessa necessidade existir, o DBC não dispõe da mesma infra-estrutura de GCS existente para o desenvolvimento convencional.

Para que possa existir uma integração real entre GCS e DBC, é necessário que os sistemas de controle de versões, controle de modificações e gerenciamento de construção passem a atuar no nível de abstração requerido pelos conceitos existentes no DBC. Para atingir esse objetivo, existem duas possíveis abordagens: (1) mascaramento dos sistemas de base ou (2) redefinição dos sistemas de base. A abordagem de

mascaramento do sistema de base consiste em fornecer uma metáfora de DBC para o engenheiro de software, acoplada ao mapeamento dessa metáfora para os sistemas de GCS convencionais. Já a abordagem de redefinição dos sistemas de base consiste na reconstrução dos sistemas de GCS, para que eles passem a fornecer de forma nativa a metáfora de DBC.

No restante desta seção, são apresentadas, com maior detalhamento, algumas abordagens referentes à integração dos espaços de trabalho de GCS no DBC (GARG *et al.*, 2003; WALLS e RICHARDS, 2003; BORLAND, 2006b).

3.6.1 JBuilder

O ambiente de desenvolvimento JBuilder (BORLAND, 2006b), na sua edição para empresas, permite a construção de componentes EJB (SUN, 2006a) por meio de diagramas com notação semelhante à UML. O código desses componentes é gerado automaticamente e diversos descritores são produzidos, permitindo que a implantação do componente possa ser otimizada para a grande maioria dos servidores de aplicação existentes no mercado.

Independentemente do sistema de controle de versões utilizado, é possível obter recursos de GCS sobre os componentes modelados. A plataforma do JBuilder permite que diversos sistemas de controle de versões possam ser integrados, como, por exemplo, o CVS e o ClearCase. Esses sistemas têm seus comandos mapeados para um conjunto de comandos genéricos definidos pelo JBuilder. Desta forma, a troca do sistema de controle de versões não afeta a maneira com que o engenheiro de software interage com os recursos de GCS sobre os componentes. Contudo, devido às características complexas associadas à modelagem concorrente e distribuída, o JBuilder faz uso da política pessimista (bloqueio) sempre que algum engenheiro de software necessita editar o modelo de componentes.

A construção dos componentes ocorre a partir do modelo, passando pela geração do código fonte e posterior criação do pacote contendo os componentes compilados e os descritores de implantação. Esse processo é totalmente automatizado, possibilitando inclusive a automação do próprio processo de implantação por meio do uso de *plug-ins* específicos dos servidores de aplicação.

Apesar de existir uma boa integração do sistema de controle de versões e do sistema de gerenciamento de construção com o JBuilder, não existe nenhum apoio em relação ao sistema de controle de modificações. Além disso, a modelagem de

componentes dentro do JBuilder é fortemente dependente da tecnologia EJB, não possibilitando a separação entre a especificação e a realização do componente.

Finalmente, o modelo de trabalho sugerido pelo JBuilder utiliza componentes principalmente como estruturas de controle de complexidade. As preocupações referentes à reutilização ocorrem somente dentro da própria aplicação, pois não existe nenhum apoio para o controle e versionamento da biblioteca de componentes em produção. Outra deficiência acontece no apoio à substituição dos componentes, pois devido à forma automática em que as interfaces são criadas, com relacionamento um-para-um com os componentes, o uso de interfaces como contratos de serviços prestados por diversos componentes fica prejudicado.

3.6.2 XDoclet

Uma outra forma de construir componentes EJB é via o uso de programação orientada a atributos. O XDoclet (WALLS e RICHARDS, 2003) é uma implementação em Java, com licença livre, para a programação orientada a atributos.

A filosofia por trás do XDoclet consiste em integrar no código de uma única classe todas as informações necessárias para gerar um componente. Essas informações são declaradas usando comentários do tipo Javadoc (SUN, 2006b) e pré-processadas pela ferramenta XDoclet, por meio de rotinas de construção descritas utilizando Ant. Esse pré-processamento analisa tanto o código da classe quanto os atributos colocados propositalmente nas seções de comentários da declaração da classe, dos métodos e dos atributos. Como resultado, são geradas as interfaces do componente, os descritores de implantação, o código completo do componente e as classes de apoio.

Desta forma, o código da classe com os comentários XDoclets pode ser visto como um IC fonte para o componente EJB, que é um IC derivado. Esse IC fonte pode ser armazenado em qualquer repositório de controle de versões e desenvolvido de forma paralela e concorrente por equipes distribuídas, característica essa que não é possível no caso do JBuilder devido a complexidade existente em efetuar junções sobre diagramas editados concorrentemente.

Apesar das vantagens aparentes, o XDoclet vai no sentido oposto ao das outras abordagens quando acumula em um único elemento as informações referentes a interfaces, componentes, descritores e classes de apoio. Esse tipo de abordagem tende a gerar problemas relacionados com escalabilidade. Além disso, de forma análoga ao JBuilder, as interfaces são relacionadas em um-para-um com os componentes,

dificultando a substituição do componente e inibindo o uso de interfaces como elementos contratuais.

3.6.3 Ménage

O ambiente Ménage (GARG *et al.*, 2003) foi construído para permitir a evolução de arquiteturas de linha de produtos descritas via xADL. Um dos objetivos almejados pelo autor é prover transparência nas atividades de GCS relacionadas ao versionamento dos componentes. O principal argumento defendido é que a atividade de construção de arquiteturas já é, por si só, demasiadamente complexa, e que a inclusão das atividades de GCS poderia dificultar ainda mais o desenvolvimento.

Para atender a esse objetivo, o conceito de versionamento foi incorporado aos conceitos de componente, interface e conectores, possibilitando a criação de novas versões desses elementos via comandos *check-in* e *check-out* integrados ao ambiente. Além disso, o conceito de linha base pode ser obtido pela aplicação do conceito de versão para os componentes que definem subarquiteturas.

Um vasto conjunto de ferramentas integradas ao Ménage apóia tanto a construção quanto a evolução de arquiteturas xADL. Dentre essas ferramentas, a ferramenta de crítica traz contribuições especiais para a GCS que são críticas específicas para verificar a corretude sintática dos grafos de versões de componentes, interfaces e conectores e as condições de guarda de variantes e opções.

Outra ferramenta importante permite a seleção de componentes de forma intuitiva, por meio do preenchimento de propriedades de corte. Quando são atribuídos valores a um conjunto de propriedades de cortes, as condições booleanas descritas na Seção 3.4.3 são calculadas e é determinado se os elementos devem ser removidos da arquitetura. Ao final desse processamento, a arquitetura original da linha de produtos pode ser transformada em uma arquitetura específica de um produto ou em uma arquitetura de linha de produtos simplificada. Esse segundo caso ocorre quando, mesmo após a seleção, ainda restam condições booleanas não determinadas.

Apesar de prover uma solução satisfatória de integração de espaço de trabalho, o Ménage lida somente com as questões referentes ao sistema de controle de versões, deixando a desejar nos aspectos relacionados ao controle de modificações. Por exemplo, não são definidos processos de controle para as atividades de criação da linha de produtos e de obtenção dos produtos. Sem esses processos, torna-se difícil a

manutenção dos rastros das solicitações de modificação que resultaram na evolução da própria linha de produtos, ou de um determinado produto.

3.6.4 Considerações finais sobre integração dos espaços de trabalho de GCS no DBC

As abordagens JBuilder e XDoclet, apresentadas nesta seção, apóiam o desenvolvimento de componentes EJB, possibilitando a construção automatizada desses componentes de forma integrada ao ambiente de desenvolvimento. Entretanto, o apoio à reutilização desses componentes é limitado. Nenhuma dessas abordagens considera a separação entre as interfaces e os componentes propriamente ditos. Ou seja, para cada componente criado, é criada automaticamente uma interface, que, por sua vez, é associada ao componente. Esse tipo de postura privilegia a substituição de componentes em detrimento à reutilização, pois não são identificadas as necessidades (i.e., interfaces) para só então determinar a solução (i.e., componente). Além disso, o JBuilder define a política pessimista como obrigatória para o versionamento do modelo de componentes, e o XDoclet agrupa todas as informações relacionadas a um componente dentro de um único artefato.

De forma análoga ao JBuilder e ao XDoclet, o Ménage não possibilita o controle de modificações integrado ao ambiente de desenvolvimento. Porém, o apoio à integração do controle de versões em linhas de produtos é provido a contento.

3.7 Considerações finais

Apesar da existência de uma ampla infra-estrutura de GCS para o desenvolvimento convencional, a integração da GCS com paradigmas específicos do desenvolvimento de software ainda é muito carente (ESTUBLIER *et al.*, 2005). O DBC é um exemplo de paradigma específico do desenvolvimento de software que necessita um maior apoio na evolução controlada de seus artefatos.

O DBC tem um alto grau de complexidade inerente às suas atividades, e já foi detectado que uma das maiores causas de falha em soluções inovadoras e consideradas promissoras da GCS está relacionada com a complexidade adicional que essas soluções introduzem nas atividades já existentes do desenvolvimento de software (ESTUBLIER *et al.*, 2005). Por esse motivo, a aplicação de GCS no contexto de DBC se torna um grande desafio, que só será superado com um maior investimento em pesquisa nessa área.

Neste capítulo, foram apresentadas abordagens que visam diminuir essa carência de GCS no DBC. Todavia, essas abordagens levantam novos problemas em cada um dos cinco tópicos discutidos. No Capítulo 5, são apresentados alguns caminhos para minimizar o efeito desses problemas, possibilitando que outros trabalhos mais aprofundados possam ser desenvolvidos.

Capítulo 4 – O Projeto Odyssey

4.1 Introdução

Este trabalho de pesquisa foi concebido no contexto do Projeto Odyssey, que tem como objetivo fornecer um ambiente para reutilização de software que aplica técnicas de Engenharia de Domínio, Linha de Produtos e DBC, denominado ambiente Odyssey (WERNER *et al.*, 2003).

A reutilização de software no ambiente Odyssey ocorre via duas perspectivas, que são: desenvolvimento para reutilização e desenvolvimento com reutilização. O desenvolvimento para reutilização tem como objetivo construir componentes reutilizáveis que atendam a famílias de aplicações ou a domínios de conhecimento específicos. O desenvolvimento com reutilização tem como objetivo construir aplicações que seguem uma determinada família de aplicações ou que estão situadas em um determinado domínio, reutilizando os componentes já existentes. O ambiente Odyssey faz uso de extensões de diagramas da UML para representar o conhecimento de uma família de aplicações ou de um domínio, e permite que esses diagramas sejam reutilizados para facilitar a construção de novas aplicações. Além disso, diversas outras ferramentas apóiam as atividades específicas de reutilização no ambiente Odyssey.

O restante deste capítulo apresenta os acontecimentos que levaram ao surgimento do Projeto Odyssey (Seção 4.2), como ele evoluiu desde a sua concepção (seções 4.3, 4.4 e 4.5), como ele tratou questões relacionadas à GCS até então (Seção 4.6), o seu estado atual e os seus próximos passos (Seção 4.7).

4.2 As suas origens

Em 1992, com o término do doutorado da Prof.ª Cláudia Werner (WERNER, 1992), surgiu o grupo de reutilização de software da COPPE/UFRJ, atuando desde então em atividades de ensino, pesquisa e extensão.

De 1995 a 1997, o grupo de reutilização atuou no Projeto Memphis, que tinha como objetivo a construção de um ambiente de desenvolvimento de software baseado em reutilização, utilizando a linguagem de programação Eiffel (MEYER, 1991) e o banco de dados O2 (BANCILHON *et al.*, 1992), em estações UNIX. O ambiente

Memphis (WERNER *et al.*, 1997) era instanciado a partir da estação TABA (ROCHA *et al.*, 1990) e fazia uso do método de Booch (BOOCH, 1993) no seu processo definido.

Em paralelo, de 1996 a 1999, o grupo de reutilização também atuou no Projeto Âmbar, que visava fornecer um ambiente de desenvolvimento de software baseado em reutilização para o domínio das engenharias. Este projeto fazia parte do RECOPE (Redes Cooperativas de Pesquisa) (FINEP, 2003), que visava integrar universidades, centros de pesquisa e indústria para a realização de pesquisas.

4.3 O seu histórico até 2002

Em 1998 foi iniciado o Projeto Odyssey como parte de um projeto integrado do CNPq, denominado Ambientes de Desenvolvimento de Software Orientados a Domínios, adotando Java como linguagem de programação, recém lançada em 1995, e a UML como notação dos seus diagramadores, também recém lançada em 1997. A linguagem Java foi escolhida pois trazia consigo a promessa de independência de plataforma, tanto de desenvolvimento quanto de produção. Além disso, apesar de se tratar de uma linguagem de programação nova, já existia uma vasta biblioteca de componentes disponível, abordando questões como interface gráfica, comunicação via rede e desenho 2D. A escolha da notação UML foi ainda mais natural, pois se tratava da tão desejada unificação de diversas notações de modelagem de software, liderada por James Rumbaugh, Grady Booch e Ivar Jacobson em resposta a uma solicitação de propostas (RFP - *Request for Proposal*) da OMG (*Object Management Group*) (OMG, 2006b).

Ainda em 1998, foi feita por Márcio Barros, doutorando do grupo de reutilização na época, a primeira liberação interna do ambiente Odyssey, até então uma ferramenta CASE contando somente com os diagramadores de classes e casos de uso da UML, persistidos via mecanismo nativo da linguagem Java para serialização de objetos. Posteriormente, no mesmo ano, foi desenvolvida a primeira ferramenta do ambiente Odyssey, para apoiar a adaptação do processo de aquisição de conhecimento no contexto de análise de domínio (ROSETI, 1998).

Em 1999, o ambiente Odyssey se tornou mais maduro, contando com diversos outros diagramadores da UML, além de um diagramador específico para características, todos persistidos via um gerente de objetos denominado GOA++ (MATTOSO *et al.*, 2000). Além dessas funcionalidades básicas, foi adicionada uma infra-estrutura para a documentação de componentes, denominada FrameDoc (MURTA, 1999), e

mecanismos para o estabelecimento e consulta de rastreabilidade entre artefatos em diferentes níveis de abstração. No final desse ano, o ambiente Odyssey teve a sua primeira liberação pública, apresentada na Sessão de Ferramentas do Simpósio Brasileiro de Engenharia de Software (SBES) (WERNER *et al.*, 1999).

Em 2000, o Projeto Odyssey já contava com a definição de processos tanto de desenvolvimento para reutilização, denominado Odyssey-DE (BRAGA, 2000; BLOIS, 2006), quanto de desenvolvimento com reutilização, denominado Odyssey-AE (MILER, 2000). Nesse momento, o Odyssey também apresentava diversas ferramentas adicionais: navegação inteligente entre modelos (BRAGA, 2000), obrigatoriedade e restrição no diagrama de características (MILER, 2000) e geração de código em Java, Delphi e C++. Contudo, até esse momento, os processos Odyssey-DE e Odyssey-AE ainda não podiam ser orquestrados pelo próprio ambiente Odyssey devido à inexistência de uma máquina de processos. Nesse ano, houve uma segunda liberação pública do ambiente Odyssey, também apresentada na Sessão de Ferramentas do SBES (WERNER *et al.*, 2000).

Em 2001, o ambiente Odyssey passou a contar com um novo mecanismo de persistência, denominado MOR (MURTA *et al.*, 2001b), que possibilitava o armazenamento de objetos em bancos de dados relacionais de forma transparente, fazendo uso dos mecanismos de introspecção da linguagem Java. Além disso, também foram adicionadas as ferramentas de engenharia reversa de código fonte Java para modelos de classes UML, denominada Ares (VERONESE e NETTO, 2001), de críticas de consistência em modelos UML, denominada Oráculo (DANTAS, 2001), e de seleção de estilos arquiteturais a partir de requisitos não funcionais (XAVIER, 2001).

Em 2002, novas ferramentas para a publicação, busca e recuperação de componentes na Internet foram adicionadas ao ambiente Odyssey (COSTA, 2002; PINHEIRO, 2002). Além disso, o ambiente Odyssey passou a contar com ferramentas para a colaboração síncrona (MANGAN *et al.*, 2002) e para a instanciação de padrões e detecção de anti-padrões em modelos de projeto UML (DANTAS *et al.*, 2002). Finalmente, uma máquina de processos, denominada Charon (MURTA, 2002), passou a fazer parte do ambiente Odyssey, fornecendo apoio automatizado para a execução dos processos Odyssey-DE e Odyssey-AE. Nesse ano, houve uma terceira liberação pública do ambiente Odyssey, também apresentada na Sessão de Ferramentas do SBES (WERNER *et al.*, 2002).

A partir desse momento, o ambiente Odyssey recebe uma nova denominação: ambiente Odyssey Share. Isso ocorreu devido ao surgimento de uma nova preocupação no Projeto Odyssey: viabilizar o trabalho colaborativo sobre os modelos de domínio. Para atender a essa demanda, foram feitas diversas pesquisas na utilização de diferentes técnicas de CSCW (*Computer Supported Collaborative Work*) e de GCS nos anos seguintes.

4.4 O seu histórico de 2002 a 2006

Este trabalho de pesquisa foi iniciado em 2002. Contudo, outros trabalhos também foram desenvolvidos no contexto do Projeto Odyssey no período de 2002 a 2006.

Em meados de 2002, o ambiente Odyssey já tinha um porte razoável, com diversas ferramentas fortemente acopladas ao seu núcleo. Esse acoplamento se tornava, cada vez mais, um empecilho para a sua evolução. Com o intuito de contornar esse problema, foi iniciada uma grande reestruturação do ambiente Odyssey, separando o núcleo, denominado *kernel*, de suas ferramentas, denominadas *plug-ins*.

Em 2003, um novo mecanismo de colaboração, denominado Ariane (VIEIRA, 2003), foi adicionado ao ambiente Odyssey, permitindo a análise de bases de dados compartilhadas com o intuito de apoiar a percepção. Além disso, o ambiente Odyssey também passou a contar com um mecanismo para a geração de componentes de negócio a partir de modelos de análise (TEIXEIRA, 2003).

Em 2004, a reestruturação do ambiente Odyssey foi finalmente terminada. Contudo, devido a essa reestruturação, tornou-se necessário um mecanismo que possibilitasse a carga por demanda das ferramentas no núcleo. Esse mecanismo faz parte deste trabalho de pesquisa e é apresentado em detalhes nos capítulos 5 e 6.

Em 2005, o apoio à colaboração no ambiente Odyssey é ainda aumentado. Além do Ariane, que extrai informações de percepção de bases de dados compartilhadas, o ambiente Odyssey passou a contar com o MAIS (LOPES, 2005), que extrai informações de percepção diretamente de modelos de software compartilhados, e com o GAW (SILVA, 2005), que extrai informações de percepção de grupo, mostrando as sobreposições de trabalho dos engenheiros de software. Além disso, o ambiente Odyssey também recebeu uma ferramenta parametrizável para a extração de medidas de modelos de projeto UML (MELO JR., 2005).

Finalmente, em 2006, se encerra a época focada em aspectos de colaboração no ambiente Odyssey, na qual motivou a troca da sua denominação para ambiente Odyssey Share (MANGAN, 2006). Nesse mesmo período, o ambiente Odyssey também recebeu importantes contribuições no que se refere ao projeto arquitetural de domínio (BLOIS, 2006). Dentre elas, foi adicionada a verificação de consistência e aumentado o poder de expressão de variabilidades nos modelos de características (OLIVEIRA, 2006), juntamente com o apoio para a transformação bidirecional de modelos desde análise até código (MAIA, 2006), de acordo com a abordagem MDA (*Model Driven Architecture*) (MILLER e MUKERJI, 2003). Além disso, o ambiente Odyssey passou a contar com uma infra-estrutura para modelagem precisa (CORREA, 2006), que possibilita a elaboração, verificação e validação de restrições OCL anexadas a modelos.

4.5 O seu estado atual

Atualmente, estão sendo desenvolvidos no ambiente Odyssey trabalhos referentes à recuperação de arquiteturas (VASCONCELOS, 2004). Esses trabalhos fazem uso não só das técnicas de engenharia reversa estática, já existentes no ambiente Odyssey, mas também propõem novas técnicas para a engenharia reversa dinâmica. O núcleo do ambiente Odyssey conta hoje com as seguintes funcionalidades:

- Diagramação UML (componentes, pacotes, classes, casos de uso, seqüência, estado, atividades e implantação);
- Diagramação voltada à reutilização (contexto, características e negócio);
- Rastreabilidade entre artefatos;
- Documentação de artefatos;
- Navegação inteligente;
- Busca por componentes do domínio;
- Geração automática de componentes de negócio;
- Geração de código;
- Execução de processos; e
- Controle de acesso e persistência.

Além das funcionalidades já existentes no núcleo, outras funcionalidades podem ser adicionadas por meio da carga por demanda de ferramentas. As ferramentas disponíveis hoje no ambiente Odyssey, sem contar com as que estão descritas no restante deste trabalho, provêem as seguintes funcionalidades:

- Engenharia reversa estática (classes);
- Engenharia reversa dinâmica (seqüência);
- Transformações bidirecionais entre modelos;
- Geração de código via esquemas configuráveis;
- Mensagens instantâneas via Jabber (JSF, 2006);
- Percepção multi-síncrona;
- Percepção de grupo;
- Detecção de padrões;
- Agrupamento de artefatos em componentes;
- Manutenção da consistência de arquiteturas;
- Críticas de consistência; e
- Ajuda on-line.

O ambiente Odyssey está atualmente na sua liberação 1.5, que pode ser obtida em http://reuse.cos.ufrj.br/odyssey.

4.6 Gerência de configuração e o Projeto Odyssey

De 1998 até 2000, a GCS do próprio Projeto Odyssey era feita de forma manual. Num sistema manual de GCS, uma pessoa, denominada bibliotecário, fica encarregada de controlar todos os ICs (HASS, 2003). Quando algum engenheiro de software necessita trabalhar sobre um IC, este solicita diretamente ao bibliotecário, que, por sua vez, verifica se o IC pode ser alterado e o fornece para o engenheiro de software.

Ao final das modificações, o engenheiro de software retorna o IC modificado para o bibliotecário, que o reintegra no sistema como um todo. Vale ressaltar que o bibliotecário pode adotar tanto uma política pessimista, não permitindo que dois ou mais engenheiros de software trabalhem sobre um mesmo IC ao mesmo tempo, ou uma política otimista, tendo que juntar posteriormente os trabalhos concorrentes sobre um mesmo IC.

No caso do Projeto Odyssey, a política adotada era pessimista, mas com uma característica especial: a granularidade dos ICs era replanejada sempre que necessário, visando minimizar a quantidade de bloqueios. Por exemplo, caso um dado IC estivesse sendo solicitado por três engenheiros de software ao mesmo tempo, o bibliotecário analisaria a necessidade de cada engenheiro de software e decomporia o IC em dois ou três outros ICs, minimizando ou eliminando os bloqueios desnecessários.

Contudo, com o crescimento do Projeto Odyssey, esse sistema manual foi se tornando cada vez mais complexo e difícil de ser executado. Para contornar esse problema, foi criada a ferramenta Token (MURTA *et al.*, 2000), que permitia o acesso via Internet e fazia uso do banco de dados relacional MySQL (MYSQL AB, 2006) para persistir as características dos ICs. A ferramenta Token consistia basicamente na automação do sistema manual que já era utilizado até então.

A Figura 4.1 apresenta a tela principal da Token. Na parte superior, são listados alguns ICs como resultado de uma consulta. No centro, são fornecidas opções para a listagem das versões existentes e *check-out* de um dado IC. Na parte inferior, é fornecida a opção para *check-in* de ICs já modificados.

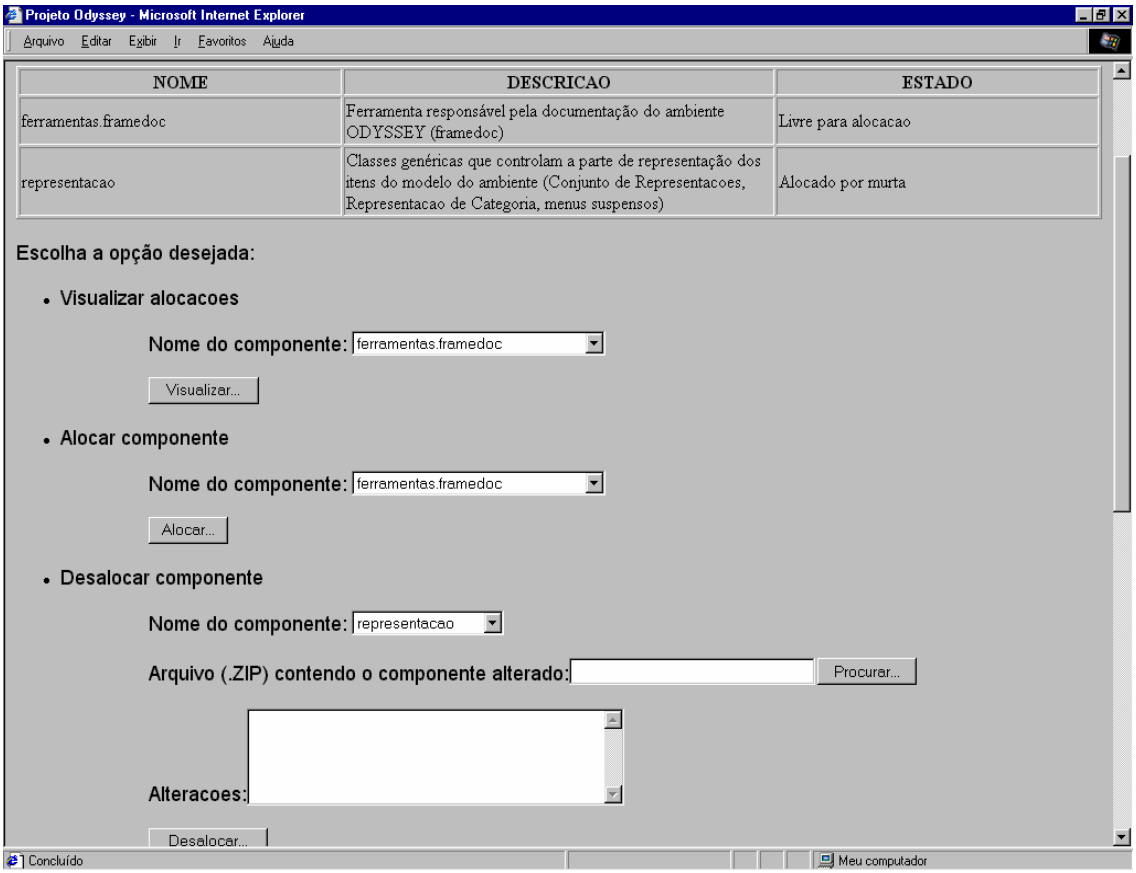


Figura 4.1: Ferramenta Token apoiando a GCS do Projeto Odyssey.

A utilização da Token no Projeto Odyssey motivou a criação de um mecanismo equivalente como ferramenta do próprio ambiente Odyssey, que possibilitasse o controle de concorrência sobre os modelos. Essa ferramenta, denominada LockED (TEIXEIRA *et al.*, 2001b; TEIXEIRA *et al.*, 2001a), é exibida na Figura 4.2.

Apesar da ferramenta LockED ter trazido uma contribuição importante para o ambiente Odyssey, permitindo que engenheiros de software trabalhassem em paralelo sobre partes diferentes de um mesmo modelo, ela apresentava algumas deficiências

graves do ponto de vista de GCS. Em primeiro lugar, ela não tratava o controle de versões em si, mas somente o controle de concorrência. Ou seja, o repositório não guardava as versões anteriores. Além disso, ela trabalhava sobre objetos serializados diretamente do meta-modelo do Odyssey para produzir os ICs durante o *check-in* ou o *check-out*, tornando inviável o seu uso em outros ambientes ou ferramentas CASE. Finalmente, a única política utilizada pela LockED para o controle de concorrência era a pessimista. Contudo, esse tipo de política pode ser inapropriada em determinadas situações por limitar o trabalho em paralelo (ESTUBLIER, 2001). Essas deficiências do LockED motivaram diversos aspectos deste trabalho de pesquisa, como detalhados nos capítulos 5 e 6.

Figura 4.2: Ferramenta LockEd apoiando o acesso concorrente a elementos de modelo no ambiente Odyssey.

Finalmente, em julho de 2003, o Token foi substituído pelo CVS no controle de versões do Projeto Odyssey, com o intuito de promover um maior dinamismo devido à utilização de políticas otimistas para o controle de concorrência. Juntamente com o CVS, é utilizado o Bugzilla para o controle de modificações, o Ant para o gerenciamento de construção e o Eclipse para a programação.

Atualmente, o Projeto Odyssey conta, também, com um mecanismo de construção noturna (*nightly build*), que recupera a versão mais atual do código fonte, compila e roda a bateria de testes programada com o JUnit. Os resultados desse ciclo de

construção são apresentados de forma automática, diariamente, em um site interno do projeto.

4.7 Considerações finais

Este capítulo apresentou o Projeto Odyssey, descrevendo as suas origens e o seu histórico. Nos seus 8 anos de existência, ele foi celeiro para 6 teses de doutorado, sendo 4 já concluídas. Além disso, também foram desenvolvidos no contexto do projeto 14 dissertações de mestrado e 5 projetos finais de curso. Nesse período, o ambiente Odyssey e suas ferramentas foram apresentados 24 vezes em 8 edições da Sessão de Ferramentas do SBES, tendo sido premiados 4 vezes em primeiro lugar e 3 vezes em segundo lugar.

Atualmente, o grupo de reutilização tem trabalhado na integração do ambiente Odyssey com a biblioteca Brechó, em desenvolvimento pelo mesmo grupo. A biblioteca Brechó tem como objetivo fornecer mecanismos de armazenamento, busca e recuperação de componentes, eventualmente produzidos pelo ambiente Odyssey. A integração entre o ambiente Odyssey e a biblioteca Brechó visa facilitar a publicação de liberações dos componentes desenvolvidos no ambiente Odyssey e possibilitar que a biblioteca Brechó faça a mediação entre o ambiente Odyssey e as suas ferramentas, carregadas dinamicamente.

Os próximos passos do Projeto Odyssey estão fortemente direcionados para questões relacionadas com a evolução de software. Dentre elas, mecanismos mais precisos para a detecção de rastreabilidade, abordagens para a extração de informações de alto nível a partir de sistemas legados e estratégias para visualização de grandes quantidades de informação.

Capítulo 5 – A Abordagem Odyssey-SCM

5.1 Introdução

A partir da revisão da literatura apresentada nos capítulos 2 e 3, foi possível identificar as principais fraquezas no apoio dado pela GCS no DBC. Essas fraquezas, analisadas em conjunto com trabalhos executados anteriormente no contexto do Projeto Odyssey, como discutido no Capítulo 4, motivaram os requisitos principais para a abordagem proposta, que são:

1. Processo de controle de modificações compatível com as normas atuais de GCS, que leve em consideração as particularidades do DBC apresentadas no Capítulo 3;
2. Modelagem gráfica do processo via uma notação padrão, que possibilite a substituição do sistema de controle de modificações, caso necessário;
3. Modelagem gráfica das informações que devem ser coletadas nas atividades do processo de controle de modificações;
4. Execução do processo de controle de modificações, coletando as informações necessárias em cada atividade e provendo informações adicionais sobre casos de reutilização de determinados componentes;
5. Controle de versões em artefatos de alto nível de abstração, de acordo com um meta-modelo padrão, permitindo identificar, em granularidade fina, quais elementos foram impactados por uma dada solicitação de modificação e com escalabilidade nas dimensões de duração e tamanho dos projetos;
6. Manutenção dos rastros entre os elementos arquiteturais (componentes, interfaces e conectores) e o código fonte que efetivamente implementa esses elementos;
7. Acesso a comandos de GCS em elementos arquiteturais e propagação desses comandos para o código fonte;
8. Construção, empacotamento, liberação e posterior implantação dinâmica dos componentes nas aplicações alvo; e
9. Detecção de ligações de rastreabilidade entre elementos de alto nível de abstração e apresentação da razão da existência dessas ligações de rastreabilidade de forma automática.

Desta forma, a abordagem proposta, denominada Odyssey-SCM (SCM: *Software Configuration Management*) (MURTA, 2004; MURTA *et al.*, 2004a; MURTA *et al.*, 2005; MURTA *et al.*, 2006a), visa fornecer processos e ferramental de apoio para a aplicação de técnicas de GCS no contexto do DBC, atendendo aos requisitos supracitados. A abordagem Odyssey-SCM é composta por cinco subabordagens com enfoque nos elementos da taxonomia definida no Capítulo 2, que são:

- Odyssey-SCMP (SCMP: *Software Configuration Management Process*): Adaptações do processo de GCS para o contexto de DBC (requisito 1);
- Odyssey-CCS (CCS: *Change Control System*): Sistema de controle de modificações com ênfase nos requisitos específicos do DBC (requisitos 2, 3 e 4);
- Odyssey-VCS (VCS: *Version Control System*): Sistema de controle de versões focado em artefatos de alto nível de abstração (requisito 5);
- Odyssey-BRD (BRD: *Build, Release and Deploy*): Gerenciamento de construção, liberação, empacotamento e implantação de componentes (requisitos 6, 7 e 8); e
- Odyssey-WI (WI: *Workspace Integration*): Integração dos espaços de trabalho de GCS e DBC (requisito 9).

Cada uma dessas subabordagens agrega funcionalidades importantes ao Odyssey-SCM, com o intuito de prover um maior controle ao DBC, em especial, na evolução de artefatos em altos níveis de abstração. Todavia, um requisito não funcional importante é a necessidade da abordagem não sobrecarregar demasiadamente as atividades de DBC existentes com as novas atividades relacionadas a GCS. Esse requisito não funcional é levado em consideração por todas as subabordagens, em especial pelo Odyssey-WI.

A Figura 5.1 apresenta um panorama estático do relacionamento entre a abordagem Odyssey-SCM, suas subabordagens e as equipes produtoras e consumidoras de componentes. Segundo a abordagem proposta, toda e qualquer atividade de reutilização deverá ser passível de controle pelo Odyssey-SCM, possibilitando que o conhecimento referente à GCS seja extraído, processado e armazenado para posterior consulta. Na camada inferior da Figura 5.1 se situam os repositórios de solicitações de modificação e de versões. Esses repositórios são integrados por meio de um repositório

de rastros, que armazena relacionamentos entre as solicitações de modificação e as versões de artefatos. Na camada intermediária da Figura 5.1, se situam os sistemas de GCS, que atuam no controle de modificações, controle de versões, gerenciamento de construção e integração dos espaços de trabalho. Finalmente, na camada superior da Figura 5.1, se situa o processo de GCS propriamente dito.

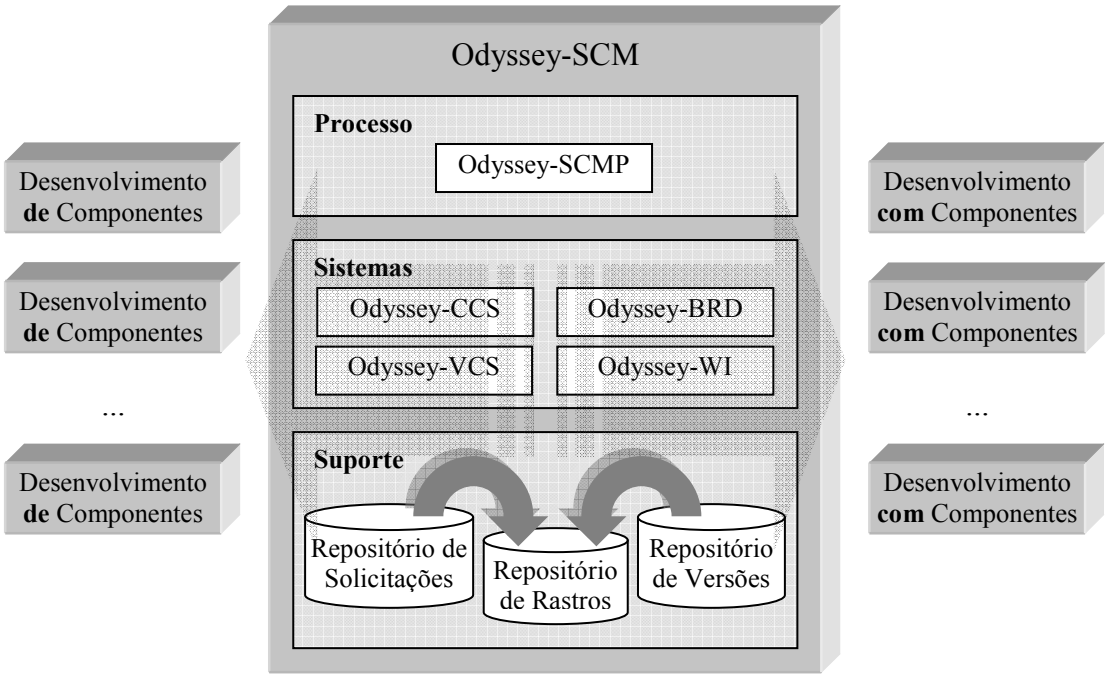


Figura 5.1: Panorama estático do Odyssey-SCM.

A abordagem Odyssey-SCM também pode ser analisada sob a perspectiva dos serviços providos pelas suas subabordagens. Esse panorama dinâmico, exibido na Figura 5.2, segue o conjunto de passos detalhados a seguir:

1. O usuário solicita uma modificação sobre um dado aplicativo utilizando o Odyssey-CCS, de acordo com o processo Odyssey-SCMP;
    1.1. O Odyssey-CCS apóia na análise da responsabilidade de manutenção dos componentes. Caso a solicitação não seja de responsabilidade da equipe em questão, a solicitação ou parte dela é encaminhada para a equipe produtora do componente afetado;
2. Caso a solicitação ou parte dela seja de responsabilidade da equipe em questão, um engenheiro de software é designado para implementá-la;
3. Esse engenheiro de software faz uso de um ADS (e.g. ambiente Odyssey) para implementar as modificações;
    3.1. O Odyssey-VCS pode ser utilizado para acessar a versão pertinente dos modelos referentes aos componentes a serem modificados;

3.2. Um módulo do Odyssey-BRD, denominado ArchTrace, pode ser usado para identificar e acessar o código fonte que implementa os componentes que devem ser modificados;

3.3. Em paralelo, o Odyssey-WI pode ser utilizado para detectar outros pontos do software que devem ser modificados (análise de impacto);

4. Periodicamente (ou via comando do engenheiro de software), o Odyssey-BRD acessa o repositório de código fonte, obtendo todos os artefatos que implementam os componentes existentes. Cada componente é compilado, testado e empacotado;
5. Caso o componente esteja em um estado estável, é feita a liberação do mesmo e disponibilização em uma biblioteca de componentes;
6. Finalmente, o mecanismo de carga dinâmica do aplicativo permite que a nova versão do componente, que contempla a modificação requerida, seja instalada, caso seja o desejo do usuário.

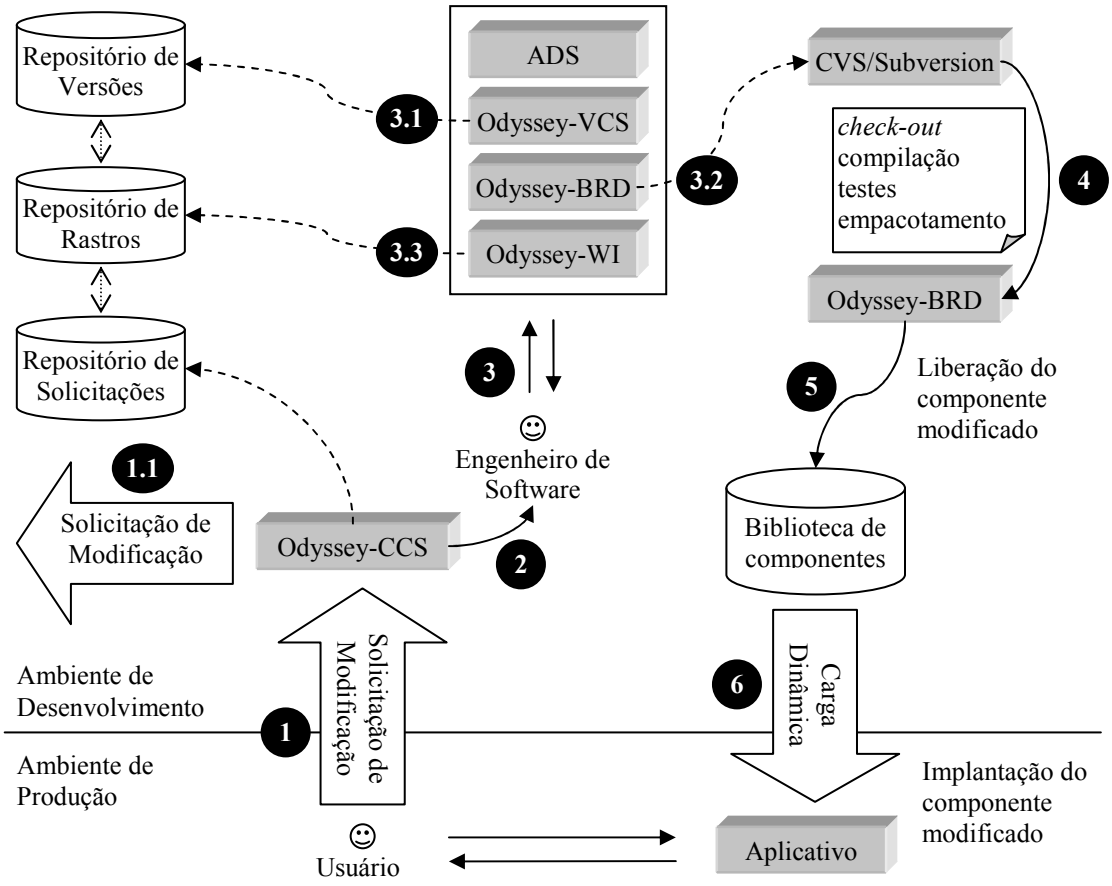


Figura 5.2: Panorama dinâmico do Odyssey-SCM.

Vale ressaltar que as subabordagens Odyssey-CCS (LOPES, 2006), Odyssey-VCS (OLIVEIRA, 2005) e Odyssey-WI (DANTAS, 2005) foram desenvolvidas em

dissertações de mestrado no contexto deste trabalho de pesquisa. O detalhamento do problema, definição de caminhos para a solução e integração das soluções individuais foram feitos em fases distintas desta tese de doutorado (MURTA, 2004). O detalhamento das soluções específicas das subabordagens, como também as suas implementações, foram desenvolvidos em cada uma das dissertações de mestrado supracitadas, sempre em sintonia com esta tese de doutorado. Finalmente, a avaliação das soluções propostas foi executada nesta tese de doutorado, conforme apresentado no Capítulo 7.

Cada uma dessas subabordagens é detalhada nas demais seções deste capítulo. Nesse detalhamento, inicialmente, cada abordagem é contextualizada e os problemas específicos são identificados. Posteriormente, é apresentada a proposta de solução referente à subabordagem.

5.2 Odyssey-SCMP

Conforme visto no Capítulo 2, existem diversas normas para a aplicação de GCS no desenvolvimento convencional de software. Dentre essas normas, as que mais se destacam são a ISO 10007 (ISO, 1995a), a IEEE Std 828 (IEEE, 2005) e a IEEE Std 1042 (IEEE, 1987). Além disso, os modelos CMM (JALOTE, 1999), CMMI (CHRISSIS *et al.*, 2003) e MPS.BR (SOFTEX, 2006b) consideram a GCS como área de processo e determinam objetivos e resultados para atender às suas necessidades.

Apesar dessa grande variedade de normas e modelos para a GCS, nenhum deles considera questões específicas do DBC, até porque o foco dessas normas não é em paradigmas específicos de desenvolvimento de software, mas sim nos requisitos genéricos para a aplicação de GCS. Essas normas tratam das cinco funções de GCS, que são (1) identificação da configuração, (2) controle da configuração, (3) contabilização da situação da configuração, (4) avaliação e revisão da configuração e (5) gerenciamento de liberação e entrega, como discutido no Capítulo 2. Contudo, somente uma equipe de desenvolvimento de software é considerada nessas funções. No cenário de DBC, o número de equipes de desenvolvimento de software aumenta, assim como o relacionamento entre elas.

5.2.1 Detalhamento do problema

Um processo genérico de DBC engloba, ao menos, duas etapas: desenvolvimento **de** componentes e desenvolvimento **com** componentes. Essas etapas

são executadas, respectivamente, pela equipe produtora de componentes e pela equipe consumidora de componentes, como apresentado na Figura 5.3.

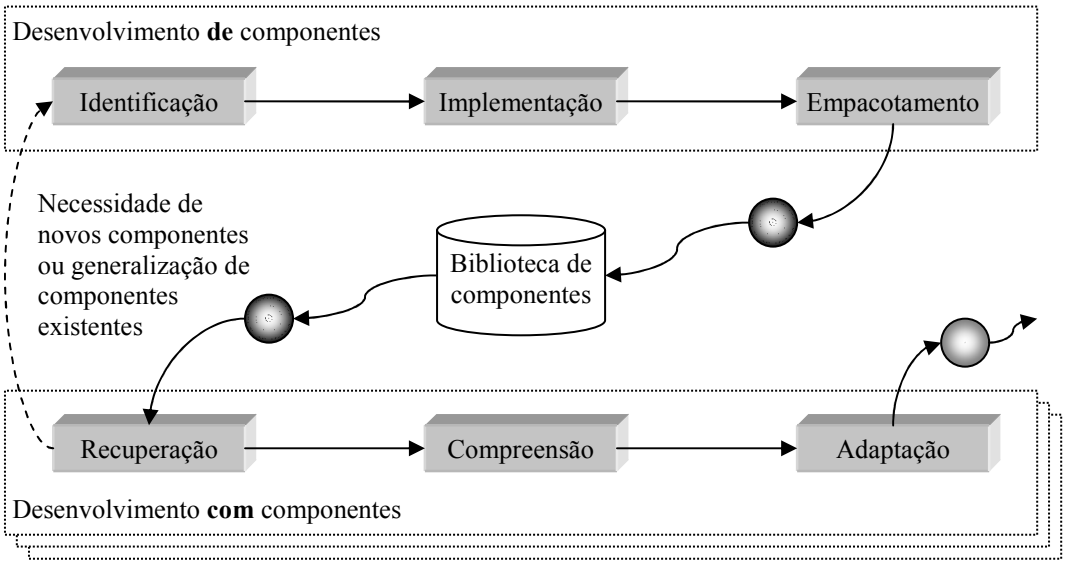


Figura 5.3: Processo de DBC convencional.

Neste processo, a equipe consumidora busca por componentes em uma biblioteca ou solicita a criação de componentes, seja do zero ou via generalização de componentes existentes, caso não seja possível encontrar os componentes desejados na biblioteca. Em um segundo momento, o componente obtido deve ser compreendido e adaptado para o problema em questão. Por outro lado, a equipe produtora deve identificar as necessidades de componentes das equipes consumidoras, implementar esses componentes, empacotar juntamente com a documentação pertinente e disponibilizar na biblioteca de componentes. Este processo básico contempla somente o desenvolvimento, não levando em consideração as necessidades de manutenção, como apresentado na Figura 5.4.

A principal diferença entre os processos apresentados na Figura 5.3 e na Figura 5.4 é a adição de infra-estrutura que possibilite tratar futuras solicitações de manutenção sobre os componentes. Para isso, é essencial prover um mecanismo que possibilite a solicitação e o acompanhamento das modificações sobre os componentes. Mais ainda, é necessário ter controle de versões sobre os componentes e distinguir a biblioteca de componentes, que contém versões de produção (ou seja, liberações) dos componentes, do repositório de versões de desenvolvimento. Finalmente, é necessário rastrear os diferentes estados de um determinado componente, possibilitando recuperar corretamente os artefatos no ambiente de desenvolvimento que geraram um determinado componente no ambiente de produção.

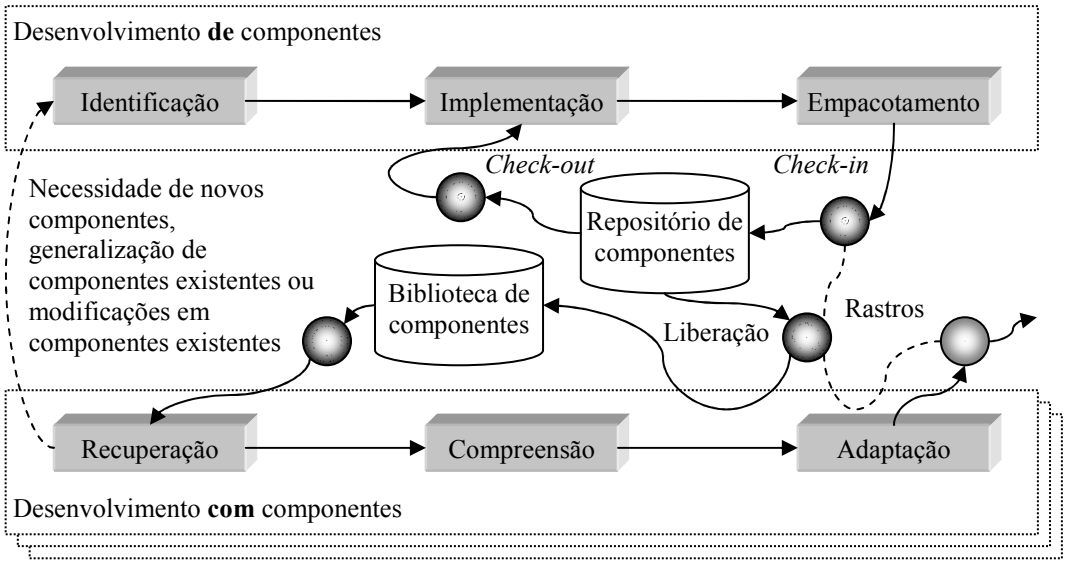


Figura 5.4: Processo de DBC contemplando as atividades de manutenção.

Contudo, este processo descrito na Figura 5.4 é recursivo, pois também podem existir equipes híbridas, que fazem uso de componentes para desenvolver outros componentes. As equipes hibridas podem ser vistas como equipes que atuam concomitantemente nos papéis de produtoras e consumidoras. Desta forma, podem ser identificados, ao menos, quatro participantes no processo de DBC, no que tange a GCS: equipes produtoras, equipes híbridas, equipes consumidoras e usuários finais. Esses participantes devem interagir sempre que uma modificação for necessária. Diferentemente do processo convencional de GCS, uma solicitação de modificações pode ser propagada para outras equipes caso o componente afetado tenha sido reutilizado, como exibido na Figura 5.5. Essa propagação deve levar em consideração questões contratuais e análises de custo-benefício em relação ao desenvolvimento local da funcionalidade.

Neste cenário, como exibido na Figura 5.5.a, uma solicitação de modificação é efetuada pelo usuário final. A equipe consumidora responsável pelo software alvo dessa solicitação de modificação detecta, utilizando ferramental de apoio apropriado, que parte da solicitação diz respeito a um componente reutilizado. Devido a condições contratuais favoráveis, essa parte da solicitação é propagada para a equipe que produziu o componente, recursivamente. Desta forma, o processo de GCS passa a se comportar como um processo multi-nível, onde processos de determinadas equipes interagem com processos de outras equipes, delegando parte das solicitações sempre que necessário, como exibido na Figura 5.5.b.

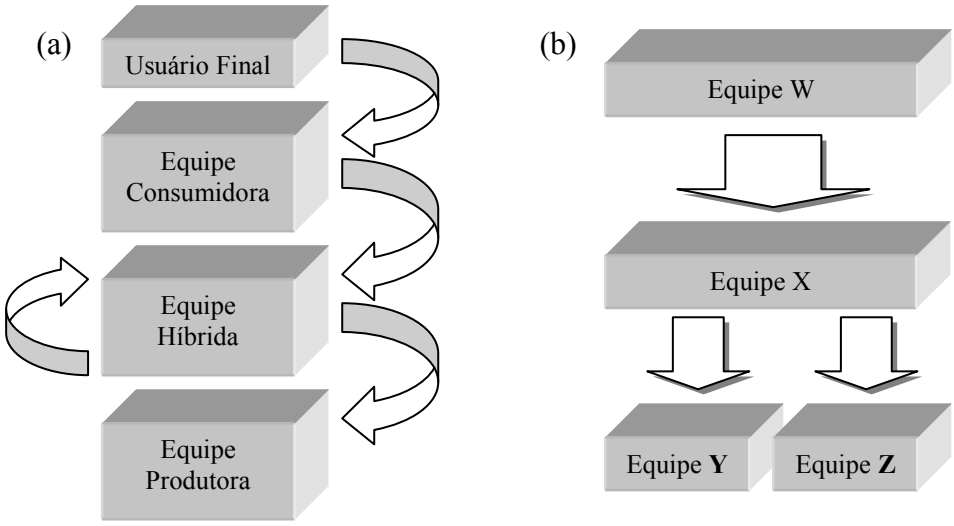


Figura 5.5: Propagação de solicitações de modificação do processo de GCS no DBC.

A reutilização de software no contexto de DBC pode ocorrer em diferentes etapas do ciclo de vida do componente. A reutilização de serviços de um componente em execução, ou a reutilização de componentes executáveis, porém não implantados, são as formas ideais de reutilização devido ao baixo custo de desenvolvimento adicional envolvido. Entretanto, pode não ser possível obter o componente desejado nesse estágio de desenvolvimento ou até mesmo não ser viável adaptar o componente para as necessidades devido a incompatibilidades de plataforma.

Nesses casos, a reutilização de artefatos de mais alto nível de abstração pode ser uma boa alternativa. Quanto mais alto for o nível de abstração do artefato, menor será a dependência desse artefato com tecnologias específicas. Por esta razão, a probabilidade de reutilizar o artefato sem um excesso de adaptações aumentará. Contudo, artefatos muito abstratos tendem a ser demasiadamente genéricos, perdendo utilidade para situações específicas.

Desta forma, como exibido na Figura 5.6, um componente pode ser reutilizado em qualquer etapa do seu desenvolvimento, contudo, quanto mais precoce for essa reutilização, maior será o esforço de continuação de desenvolvimento despendido pela equipe consumidora. Por outro lado, componentes muito específicos podem ser difíceis de reutilizar ou podem requerer um alto grau de adaptação.

De forma análoga ao desenvolvimento, a responsabilidade sobre a manutenção de um componente está intimamente relacionada com o ponto em que o componente foi reutilizado. Quando um componente é reutilizado, todas as informações relacionadas ao fornecedor desse componente e às condições contratuais devem ser mantidas pelo sistema de GCS. Essas informações, juntamente com o conhecimento sobre quais partes

do componente foram reutilizadas e quais partes do componente foram desenvolvidas ou adaptadas pela equipe consumidora, serão úteis para apoiar a decisão de propagar ou não uma solicitação de modificação, como exibido na Figura 5.7.

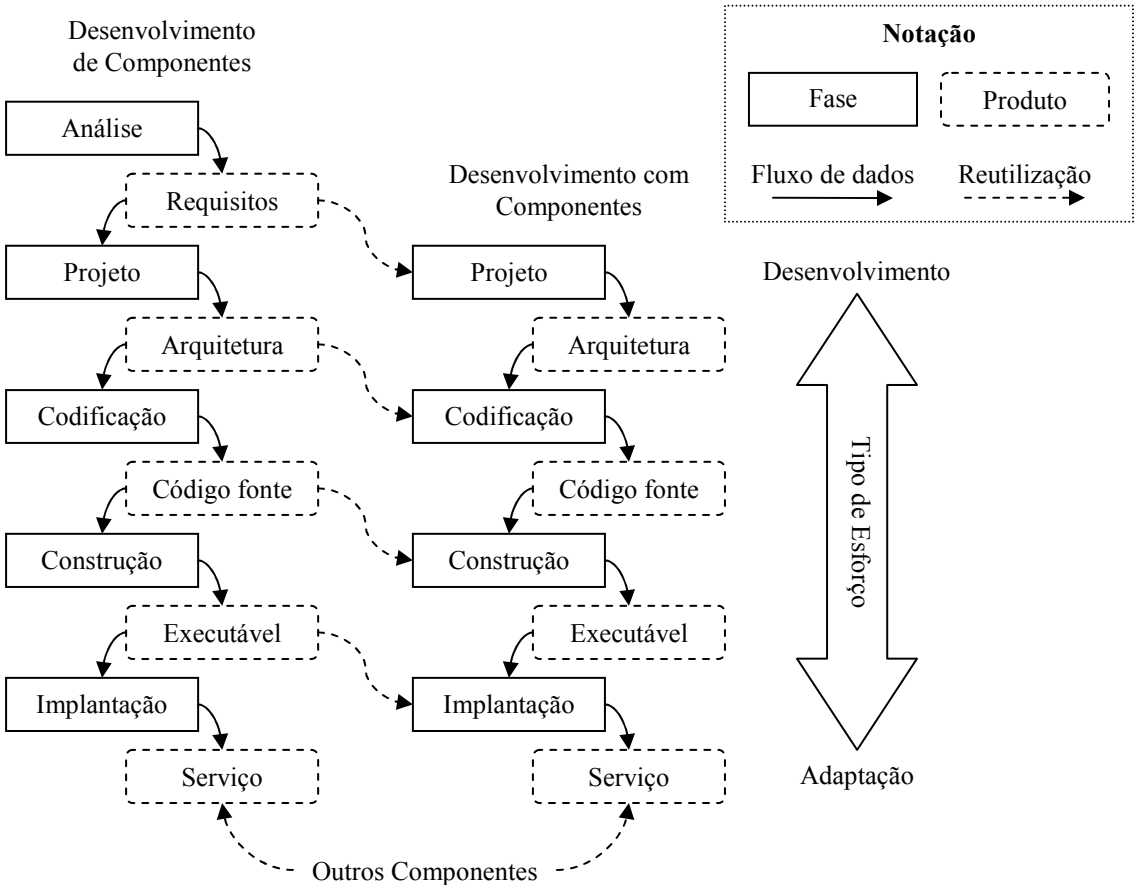


Figura 5.6: Esforço envolvido na reutilização de componentes em diferentes níveis de abstração.

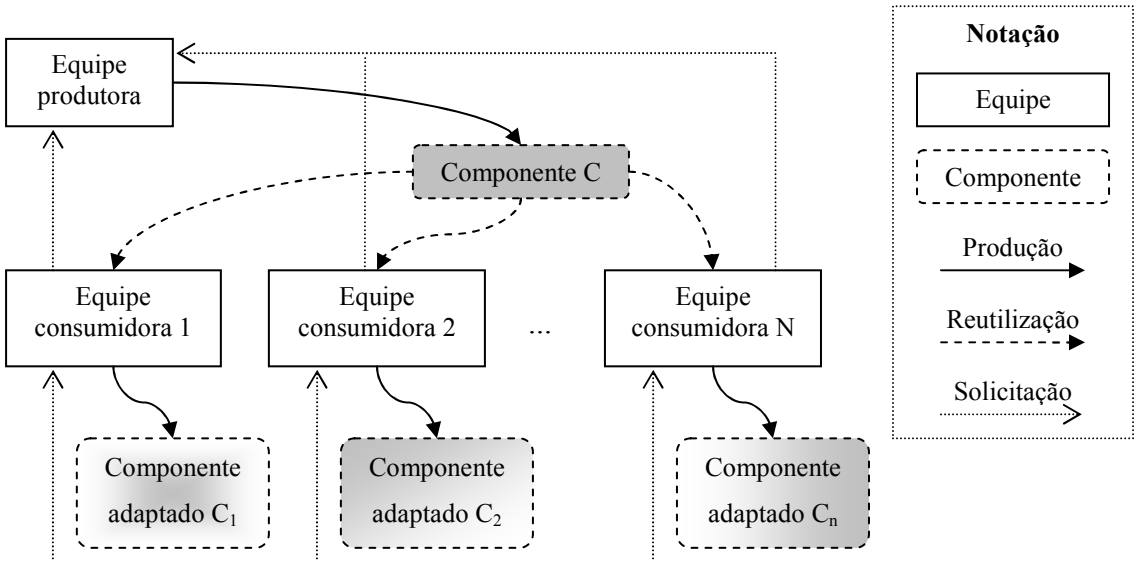


Figura 5.7: Propagação de solicitações de modificação.

A detecção correta de responsabilidades de manutenção é fundamental para evitar retrabalho e possibilitar que os componentes atinjam um elevado grau de

qualidade. Quando um componente é reutilizado por diferentes equipes consumidoras, cada equipe consumidora efetua desenvolvimento adicional e adaptações sobre o componente, como já discutido anteriormente. Apesar da ocorrência dessas modificações, parte do componente mantém concordância com o componente reutilizável original. Por esta razão, modificações no componente devem ser precedidas de uma análise apropriada, para detectar quem é o verdadeiro responsável pela modificação.

Caso uma modificação de responsabilidade da equipe produtora seja implementada pela equipe consumidora, as demais equipes consumidoras estarão utilizando um componente de menor qualidade ou funcionalidade, ou então serão obrigadas a repetir o trabalho efetuado pela primeira equipe consumidora. Contudo, caso a modificação seja especificamente relacionada com adaptações ou necessidades de uma determinada equipe consumidora, o esforço para implementar essa modificação no contexto da equipe produtora pode não compensar, devido a um pequeno interesse das demais equipes consumidoras.

5.2.2 Abordagem proposta

O Odyssey-SCMP (DANTAS *et al.*, 2003) visa adaptar as cinco funções clássicas de GCS, levando em consideração os requisitos peculiares do DBC descritos anteriormente.

5.2.2.1 Função de identificação da configuração

Na função de identificação são executadas tarefas referentes à seleção e documentação dos ICs, que podem estar em diferentes níveis de abstração (análise, projeto, codificação, teste, etc).

Inicialmente, em função da demanda das equipes consumidoras, pode ser priorizada uma ordem de construção de novos componentes. Neste contexto, os ICs podem ser classificados como internos, quando são desenvolvidos pela própria equipe consumidora, ou externos, quando são vindos da equipe produtora, seja ela pertencente à própria organização ou a terceiros.

A documentação dos ICs pode variar em relação a esta classificação. Para cada participante da equipe, pode ser definido um nível de acesso que determine a visibilidade da informação. O que, no entanto, deve ser ressaltado, é que toda a documentação que estiver disponível para o IC, inclusive a descrição de algoritmos e

código fonte, deve ser adicionada ao próprio IC, para que o mesmo possa ser projetado e testado de forma independente.

A granularidade do IC também deve ser levada em consideração. Heurísticas podem ser utilizadas para estipular o que considerar como ICs, visto que este pode englobar qualquer artefato produzido durante o desenvolvimento de software ou um conjunto de artefatos. Uma possível heurística seria considerar elementos fortemente coesos e fracamente acoplados.

Deve-se ainda considerar a possibilidade de definição de interface como IC em DBC. Apesar de se assumir a estabilidade dos contratos de interface como premissa para a compatibilidade entre componentes, em certas circunstâncias as próprias interfaces evoluem, e todos os clientes de componentes que requerem ou provêem essas interfaces devem ser notificados para que possam se adequar, de forma não traumática, a essas modificações contratuais.

5.2.2.2 Função de controle da configuração

A função de controle é responsável pela autorização, implementação e verificação das modificações sobre os ICs. As tarefas que compõem a função de controle são: (1) solicitação da modificação, (2) classificação da modificação, (3) análise de impacto da modificação, (4) avaliação da modificação, (5) implementação da modificação, (6) verificação da modificação e (7) atualização da linha base com a modificação. Posteriormente ao estabelecimento de uma linha base, os ICs só podem ser alterados utilizando esse processo formal de controle da mudança.

No **contexto de desenvolvimento de componentes**, a solicitação da modificação parte das equipes consumidoras de componentes. Durante a classificação, é necessário levar em consideração o impacto da modificação dentre as diversas equipes consumidoras para definir prioridades. Além disso, quanto à análise de impacto, se a modificação resultar na criação de novos componentes, é necessário avaliar, além de custo e tempo de desenvolvimento, o interesse das demais equipes consumidoras.

No **contexto de desenvolvimento com componentes**, a análise de impacto da equipe consumidora deve interagir com o processo de GCS da equipe produtora para estimar o impacto da modificação, caso essa modificação atinja alguma parte reutilizada de componentes. Essa interação entre os processos de análise de impacto visa avaliar se é realmente viável modificar o componente no contexto da equipe produtora. Caso a decisão de modificar o componente junto à equipe produtora não seja viável, ainda

existem as opções de reutilizar um componente equivalente de outro fornecedor, desde que esse novo componente obedeça ao contrato de interfaces do componente atual, ou ainda, modificar o componente internamente à equipe consumidora, desde que esse componente seja distribuído como caixa-branca.

Se o componente for um COTS, usualmente distribuído como caixa-preta, a modificação deve ocorrer junto ao fornecedor. Todavia, mesmo nesse cenário ainda existe a possibilidade da definição de adaptadores sobre o componente (GAMMA *et al.*, 1995).

Na etapa de avaliação da modificação, a opção de aceitar a modificação passa a ser dividida entre aceitar optando pelo desenvolvimento ou optando pela reutilização de componentes existentes. Desta forma, uma solicitação de modificação no processo de GCS da equipe consumidora pode motivar o início de um processo de reutilização de componentes.

Finalmente, a equipe consumidora deve verificar o componente realizando, preferencialmente, testes de integração e regressão. Esses testes não podem ser abandonados mesmo depois da equipe produtora assegurar que o componente, em conformidade com as suas interfaces, atende à especificação anterior ou especificação acrescida de uma nova funcionalidade. Os testes de integração e regressão também são importantes como forma de averiguar se os requisitos não funcionais continuam sendo atendidos no contexto de uso do componente.

5.2.2.3 Função de contabilização da situação da configuração

A função de contabilização da situação da configuração é encarregada de armazenar as informações coletadas pelas demais funções de GCS e relatar essas informações às pessoas autorizadas. Na aplicação de GCS no DBC, não há um consenso sobre quais informações, além das informações tradicionais, devem ser coletadas. Contudo, fica patente a necessidade de manter informações sobre os produtores e consumidores de um dado componente e as condições contratuais em que os consumidores reutilizaram o componente.

No **contexto de desenvolvimento de componentes**, após a execução de ciclos da função de controle, a função de contabilização da situação da configuração deve relatar as modificações aos interessados. Contudo, para possibilitar que a função de contabilização da situação da configuração seja capaz de notificar às equipes consumidoras sobre a modificação, a equipe produtora deve manter o rastro de

reutilização em relação a todas as equipes consumidoras que adquiriram o componente, assim como um controle das versões utilizadas por cada uma dessas equipes e o contrato de reutilização estabelecido.

No **contexto de desenvolvimento com componentes**, sempre que uma solicitação feita pelo usuário final afeta um componente reutilizado, a função de contabilização da situação da configuração deve fornecer informações sobre a equipe produtora do componente afetado e as condições contratuais de reutilização. Essas informações são fundamentais no processo de avaliação da modificação, que irá decidir como a modificação será atendida.

Em ambos os contextos, as informações contratuais estabelecidas durante o processo de reutilização são importantes em especial para decisões referentes ao custo da modificação. O contrato é o instrumento que estabelece os direitos e deveres entre equipes produtoras e consumidoras, e pode apoiar na solução de disputas entre essas duas equipes.

5.2.2.4 Função de avaliação e revisão da configuração

A função de avaliação e revisão da configuração consiste na execução de auditorias físicas e funcionais sobre a linha base antes da sua liberação, com o intuito de assegurar a sua completude e corretude.

Esta função é pouco afetada pelo DBC. Contudo, vale ressaltar a mudança de perspectiva entre equipes produtoras e consumidoras sob um mesmo componente. Do ponto de vista da equipe produtora, um componente é uma configuração composta pelos diversos ICs que descrevem esse componente. Por exemplo, especificações, modelos, casos de teste, código fonte, entre outros. A auditoria no contexto da equipe produtora é mais abrangente, focada totalmente no componente, visando verificar a existência de cada um desses ICs (auditoria física) e se os resultados de testes são realmente satisfatórios (auditoria funcional).

Por outro lado, do ponto de vista da equipe consumidora, esse componente será somente mais um IC dentre os demais ICs que compõem a aplicação final. Desta forma, sob a perspectiva de auditoria, a granularidade tratada é maior: o foco está na aplicação final, e não no componente. Assim sendo, durante a auditoria no contexto da equipe consumidora, a existência do componente (auditoria física) e os resultados dos testes da aplicação (auditoria funcional) serão verificados. Esses testes podem exercitar

parcialmente o componente, visto que nem toda a funcionalidade dos componentes são utilizadas em todas as aplicações (VINCENZI *et al.*, 2005).

5.2.2.5 Função de gerenciamento de liberação e entrega

A função de gerenciamento de liberação e entrega é responsável por controlar formalmente a construção, liberação e entrega de software.

Como discutido anteriormente, a equipe consumidora deve, ao receber uma versão de um componente, empacotá-la em um IC dentro de seu processo de GCS e documentá-la para que fique identificado o produtor do componente, as restrições e os direitos contratuais envolvidos na aquisição. Essas informações são fundamentais para possibilitar a solicitação de modificações sobre o componente no futuro.

Por outro lado, a equipe produtora também deve armazenar os dados dos consumidores de cada componente produzido. Esta informação é útil durante o processo de liberação, permitindo identificar quais consumidores devem receber as novas versões de um dado componente.

5.3 Odyssey-CCS

Apesar da existência de diversos sistemas de controle de modificações, a maioria desses sistemas é limitada em relação ao processo e às informações que são coletadas durante a execução do processo de controle de modificações. Além disso, os sistemas existentes não levam em consideração aspectos particulares do DBC, como os discutidos na Seção 5.2.1.

5.3.1 Detalhamento do problema

Como apresentado no Capítulo 3, referente à revisão da literatura de GCS aplicada no DBC, os processos de GCS relacionados com o DBC ainda estão imaturos e tendem a sofrer modificações até que uma maior estabilidade possa ser alcançada. Ainda não existe um consenso em relação a quais informações devem ser coletadas em cada atividade do processo. Também não existem normas que estabeleçam quais atividades são obrigatórias e quais atividades são opcionais na adoção de GCS no DBC.

Por estas razões, é essencial possibilitar a adaptação do sistema de controle de modificações a novos processos e permitir a configuração da coleta de informações em função das necessidades dos outros sistemas. Mais ainda, essa adaptação e essa

configuração devem acontecer não somente para os novos processos, mas também nas instâncias dos processos em execução, sem que ocorra perda de informações anteriores.

Finalmente, como previamente discutido na Seção 5.2.1, um dos principais desafios da aplicação de GCS no contexto de DBC é o problema da "cadeia de responsabilidade de manutenção". Este problema consiste em identificar qual equipe é responsável pela manutenção de um dado componente, visto que este pode fazer uso ou fazer parte de outros componentes.

5.3.2 Abordagem proposta

O Odyssey-CCS (LOPES *et al.*, 2005; LOPES *et al.*, 2006a; LOPES *et al.*, 2006b) consiste em uma abordagem configurável para o controle de modificações que permite a modelagem do processo de controle de modificações e que faz uso de padrões de documentação para a coleta de informações durante as atividades do processo. Essa infra-estrutura pode ser utilizada pelos participantes de ambientes de DBC de duas formas: o usuário final pode solicitar modificações à equipe de desenvolvimento com componentes, e a equipe de desenvolvimento com componentes pode repassar parte das solicitações para a equipe de desenvolvimento de componentes, dependendo de como os artefatos afetados foram adaptados durante o processo de reutilização.

Para possibilitar a configuração do processo de GCS para DBC, a abordagem prevê a existência de processos primitivos e compostos. Os processos primitivos, que representam as atividades de GCS, permitem, entre outras características, a definição dos papéis autorizados a executar a atividade em questão, a definição das ferramentas de apoio e a definição dos produtos que são consumidos ou produzidos pela atividade. Por sua vez, os produtos são representados por formulários que podem ser preenchidos durante a execução das atividades que os produz. Já os processos compostos fazem uso de um grafo para descrever os seus subprocessos. A Figura 5.8 exibe um modelo UML simplificado com os conceitos envolvidos no sistema de controle de modificações proposto.

Na parte superior do modelo, são exibidos os elementos que representam o nível configurável da abordagem. Para descrever um processo completo de controle de modificações pode ser necessário utilizar grafos em diferentes níveis de abstração, possibilitando a reutilização de subprocessos. Esse recurso é fornecido pelos processos compostos.

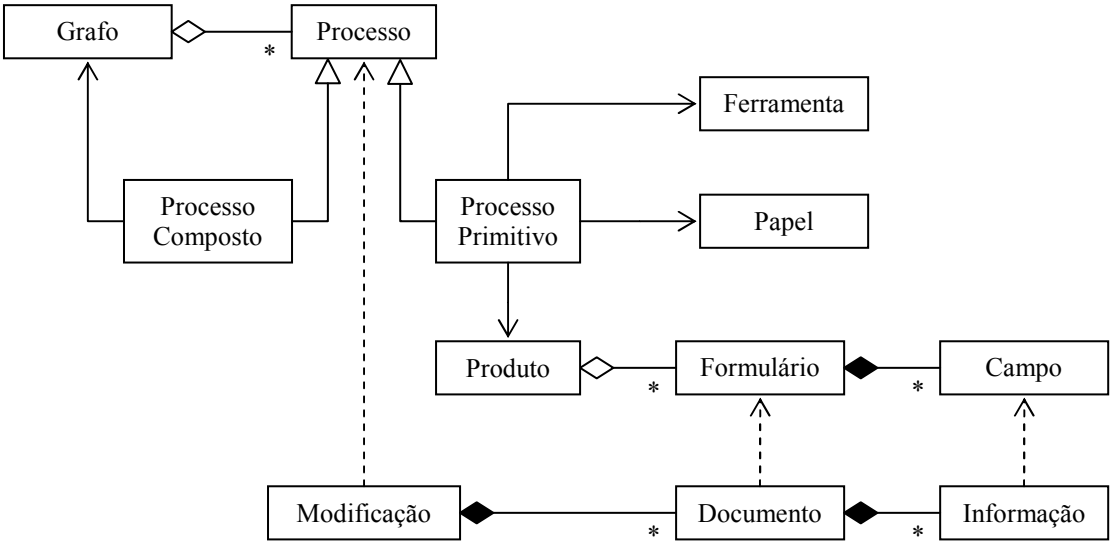


Figura 5.8: Modelo UML simplificado de controle de modificações.

De forma mais concreta, os processos primitivos possibilitam a interação entre os usuários do sistema e o sistema propriamente dito. Para cada processo primitivo, podem ser atribuídos os formulários necessários para a geração dos seus produtos a serem produzidos. Por exemplo, a atividade de "análise de impacto" pode ser modelada como um processo primitivo, executada pelo "analista de impacto" e produzindo o produto "laudo técnico". Para que esse produto possa ser produzido, um template pode ser definido, contendo os campos "impacto em custo", "impacto em cronograma", "impacto em esforço", entre outros. Esses formulários são configuráveis em função das necessidades do processo e facilmente adaptáveis caso o processo precise evoluir.

Na parte inferior do modelo, são exibidos os elementos necessários para a execução do processo configurado. Para cada execução do processo completo de controle de modificações, é criado um objeto que representa a modificação propriamente dita. Esse objeto é composto por todos os documentos produzidos pelo preenchimento dos formulários durante a execução das atividades de GCS. Desta forma, é possível obter qualquer informação produzida por atividades anteriores no processo.

A utilização do Odyssey-CCS consiste, principalmente, nas etapas referentes à preparação do ambiente e execução dos processos. A etapa de preparação do ambiente é executada pelo gerente de configuração responsável por implantar o processo de GCS adotado pela organização. Usualmente, é estabelecido um processo padrão de GCS para a organização como um todo, ou para unidades organizacionais específicas, e esse processo é adaptado para os projetos, constituindo os processos de GCS definidos dos projetos.

O gerente de configuração deve modelar os formulários necessários e, para cada formulário criado, modelar os campos que compõem esse formulário. Posteriormente, o processo de controle de modificações, que faz parte da função de controle da configuração do processo de GCS, também deverá ser modelado no Odyssey-CCS. Os produtos modelados no processo de controle de modificações devem ser associados aos respectivos formulários. Além disso, devem ser determinados os participantes que exercerão os papéis associados às atividades modeladas no processo de controle de modificações. Após essa etapa, a preparação do sistema está finalizada e o sistema pode entrar em execução.

Neste momento, vale ressaltar a semântica da instanciação do processo de controle de modificações. Diferentemente do processo de desenvolvimento de software, que tem usualmente uma única instância em cada projeto, o processo de controle de modificações normalmente tem diversas instâncias no mesmo projeto. Cada solicitação de modificação consiste em uma nova instância do processo de controle de modificação. Desta forma, algumas instâncias podem ter um ciclo de vida curto, durando pouco mais de uma semana. Por outro lado, solicitações complexas podem ter um ciclo de vida quase tão longo quanto o próprio ciclo de vida do software. Assim sendo, em um dado instante podem existir dezenas, centenas, ou até mesmo milhares de instâncias do processo de controle de modificações em execução paralela no mesmo projeto.

Para cada vez em que um processo for instanciado, um objeto que representa a modificação que motivou a instanciação desse processo será criado. Esse objeto atua como repositório para todos os documentos gerados pelos participantes durante as atividades do processo. A geração desses documentos ocorre durante o preenchimento dos formulários associados aos produtos produzidos pelas atividades do processo. Ao término do ciclo de vida de uma modificação, o objeto que representa a modificação contém todo o conhecimento referente ao processo, organizado hierarquicamente nas estruturas de atividades, produtos, formulários e campos.

Esse conhecimento, acumulado pelas diversas instanciações de processos de controle de modificações, pode ser útil para possibilitar a execução das funções de contabilização da situação da configuração e de avaliação e revisão da configuração. Além disso, a análise sobre essas informações pode permitir a detecção de deficiências no próprio processo de controle de modificações.

Finalmente, o problema da cadeia de responsabilidade de manutenção, discutido na Seção 5.2.1, é tratado por meio da manutenção de informações referentes à

reutilização dos componentes, via um mecanismo denominado mapa de reutilização, como apresentado na Figura 5.9.

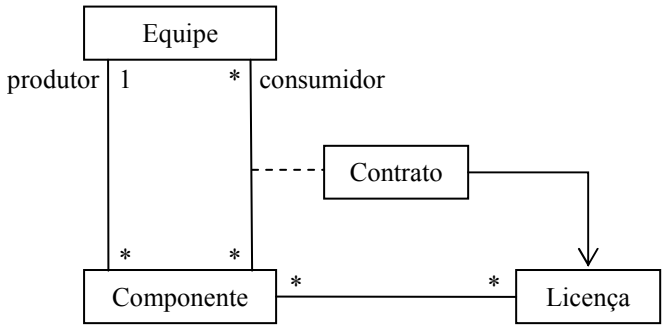


Figura 5.9: Modelo UML simplificado do mapa de reutilização.

O mapa de reutilização armazena, para cada componente, o seu produtor e os seus consumidores, juntamente com as cláusulas contratuais estabelecidas no momento da reutilização. Para simplificar o processo de definição das cláusulas contratuais, é possível definir um conjunto de licenças e selecionar quais licenças estão disponíveis para cada componente. Esta seleção é feita pelo próprio produtor do componente. Desta forma, no momento da reutilização, são apresentadas as licenças disponíveis ao consumidor. A licença selecionada pelo consumidor será utilizada na composição do contrato, que pode conter informações complementares, como, por exemplo, validade e custos envolvidos.

5.4 Odyssey-VCS

Usualmente, um componente é descrito e construído usando artefatos pertencentes a diversos níveis de abstração, estruturados em diferentes modelos de dados. Atualmente, existe apoio satisfatório para o controle de versões sobre artefatos de baixo nível de abstração, como, por exemplo, código fonte. Contudo, existe uma grande carência no apoio aos artefatos de alto nível de abstração, como, por exemplo, modelos de análise e projeto.

Entretanto, esses artefatos de alto nível de abstração são usualmente produzidos por ferramentas CASE que seguem modelos de dados específicos. A implementação de sistemas de controle de versões dependentes desses modelos de dados seria muito custosa, especialmente no que se refere ao acompanhamento da evolução e à integração desses modelos. Uma abordagem possível para lidar com esses artefatos é fazer uso de modelos de dados padrões.

Além disso, questões referentes à utilização de metáfora homogênea para os diversos tipos de ICs, a possibilidade de desenvolvimento concorrente desses ICs e a distribuição pela Internet são requisitos desejáveis para qualquer sistema de controle de versões.

5.4.1 Detalhamento do problema

Mesmo restringindo o contexto para artefatos de alto nível de abstração, ainda existem diferentes tipos de ICs que são sujeitos ao controle de versões. No caso específico de artefatos UML, esses ICs podem ser modelos, pacotes, classes, casos de uso, atributos, métodos, entre outros. Como pode ser percebido, apesar de todos esses artefatos pertencerem à UML, eles estão em níveis diferentes de uma mesma hierarquia de composição. Por exemplo, métodos e atributos fazem parte de classes, e classes fazem parte de modelos.

Para possibilitar a discussão sobre o relacionamento entre esses diversos elementos, foram definidos os conceitos de unidade de comparação e unidade de versionamento. O conceito de unidade de comparação estabelece os artefatos na hierarquia de composição que devem ser utilizados na comparação de modificações. De forma análoga, o conceito de unidade de versionamento estabelece os artefatos na hierarquia de composição que necessitam conter informações sobre versão.

Quando abordagens convencionais, como o CVS ou ClearCase, são adotadas para o controle de versões, a **unidade de comparação** utilizada é linha de arquivo do sistema operacional. No caso de ICs contendo documentos texto, essa unidade de comparação é aceitável, pois uma linha é delimitada pelos caracteres especiais CR e LF, o que estabelece equivalência para parágrafos nesses documentos. A Figura 5.10.a exibe a unidade de comparação em documentos texto utilizado pelas abordagens convencionais de controle de versões.

Porém, nem sempre é compatível o casamento entre o conceito de linha de arquivo do sistema operacional, utilizado pelas abordagens convencionais como unidade de comparação, e os conceitos utilizados logicamente pelas estruturas que serão versionadas. No caso de código fonte orientado a objetos, por exemplo, a unidade de comparação será mapeada para algo maior ou igual a um comando e menor ou igual a um bloco de comandos, como exibido na Figura 5.10.b. Isso ocorre porque em linguagens orientadas a objetos, usualmente uma mesma linha pode conter mais de um

comando, e um bloco de comandos, circunscrito por delimitadores (chaves, por exemplo), pode ocupar mais de uma linha.

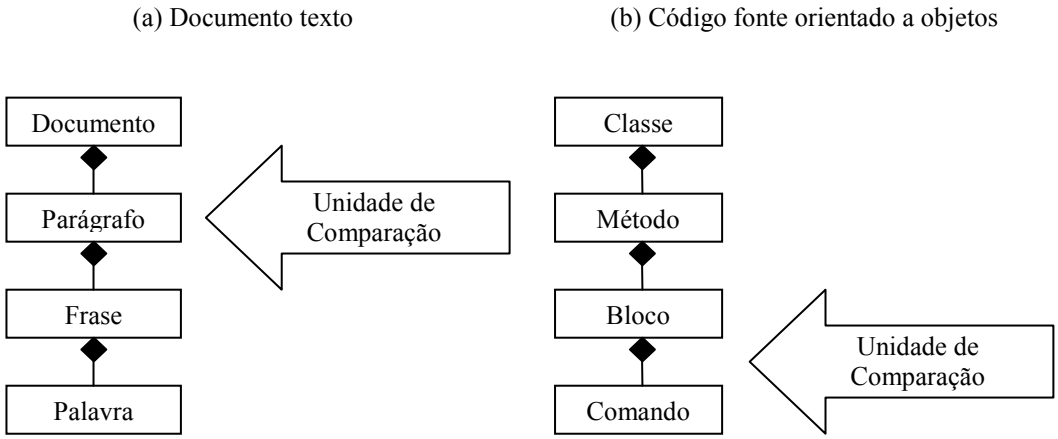


Figura 5.10: Unidade de comparação em documento texto e código fonte orientado a objetos.

Fazendo análise semelhante ao caso de parágrafos em documentos texto, pode ser constatada a falta de coesão na unidade de comparação utilizada pelas abordagens convencionais para código fonte orientado a objetos. Essa falta de coesão irá gerar dificuldades na detecção de conflitos, pois a unidade de comparação é fina demais. Para essa situação, uma unidade de comparação mais adequada seria o método como um todo.

Desta forma, o uso de unidade de comparação demasiadamente fina pode impossibilitar a detecção de conflitos existentes ou possíveis colaborações entre os participantes do desenvolvimento de software. Por outro lado, o uso de unidade de comparação demasiadamente grossa pode inviabilizar a diferenciação entre as partes isoladas do sistema, culminando na detecção incorreta de um maior número de conflitos. O conceito de coesão pode ser útil para a definição de uma unidade de comparação correta. Contudo, vale lembrar que a coesão é dependente do modelo de dados específico do IC. Por exemplo, em documento texto, o parágrafo tem um grau adequado de coesão. No caso de código fonte orientado a objetos, o método pode ser visto como a unidade mínima de coesão.

A adoção das abordagens convencionais também introduz restrições referentes à **unidade de versionamento**. A unidade de versionamento utilizada por essas abordagens é o próprio arquivo do sistema operacional. Desta forma, essa unidade de versionamento é perfeitamente adequada para o caso de documentos texto, como exibido na Figura 5.11.a, onde existe um mapeamento para o documento como um todo. Contudo, no caso de código fonte orientado a objetos, o mapeamento já não é

satisfatório, como exibido na Figura 5.11.b, pois um arquivo pode conter uma ou mais classes. Seria mais adequada a utilização de classes como unidade de versionamento nessa situação.

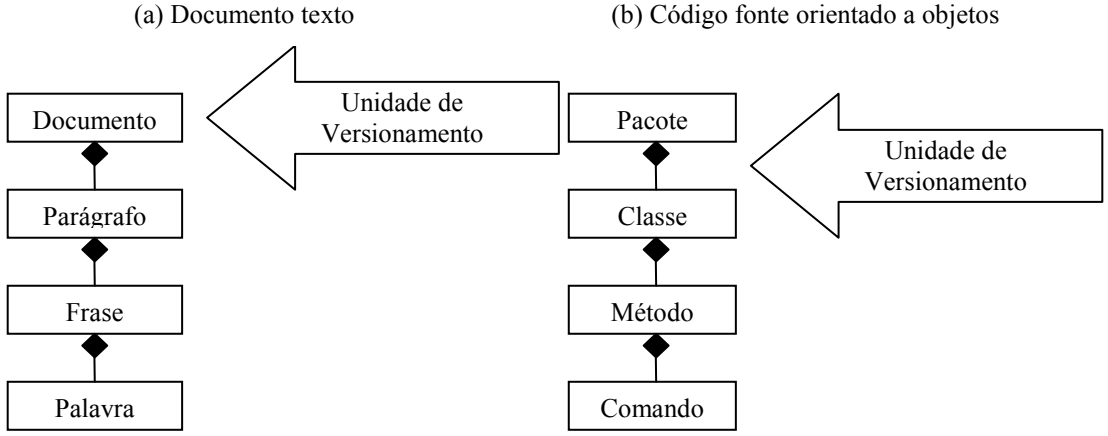


Figura 5.11: Unidade de versionamento em documento texto e código fonte orientado a objetos.

A escolha de uma unidade de versionamento demasiadamente fina sobrecarrega desnecessariamente o sistema de controle de versões, tornando custosa (lenta) a identificação das versões devido ao excesso delas. Por outro lado, a escolha de uma unidade de versionamento demasiadamente grossa impossibilita o controle da evolução dos elementos que compõem o sistema e aumenta o número de bloqueios, caso essa política esteja sendo utilizada.

Todavia, quando os ICs são modelos, a situação é ainda mais catastrófica caso as abordagens convencionais venham a ser adotadas. A UML, por ser um meta-modelo MOF, pode ser externada via XMI, que consiste em um esquema XML. Esses modelos XMI, que são usualmente armazenados em arquivos e colocados sob versionamento, têm uma unidade de comparação extremamente fina, conforme exibido na Figura 5.12. Por exemplo, um atributo de uma classe modelada em UML pode ocupar diversas linhas do arquivo XMI gerado. Desta forma, cada uma dessas linhas tem uma baixa coesão e um grande acoplamento com as demais linhas se analisada isoladamente. Por essa razão, grande parte dos conflitos reais será ignorada, possivelmente gerando versões inconsistentes do modelo.

Nesta mesma situação, a unidade de versionamento será extremamente grossa, pois o modelo completo do sistema é armazenado em um único arquivo XMI. Essa postura inviabiliza o controle individual das versões de elementos do modelo, como, por exemplo, casos de uso e classes. A Figura 5.12 exibe esse cenário, onde um sistema completo, que pode ser composto por milhares de classes, tem unidade de

versionamento equivalente ao sistema como um todo. Nesse cenário, onde os ICs são modelos, seria mais indicado configurar a unidade de comparação para elementos do tipo característica (*Feature*) do meta-modelo da UML, como, por exemplo, métodos e atributos, e configurar a unidade de versionamento para elementos do tipo classificador (*Classifier*) do meta-modelo da UML, como, por exemplo, classes e casos de uso.

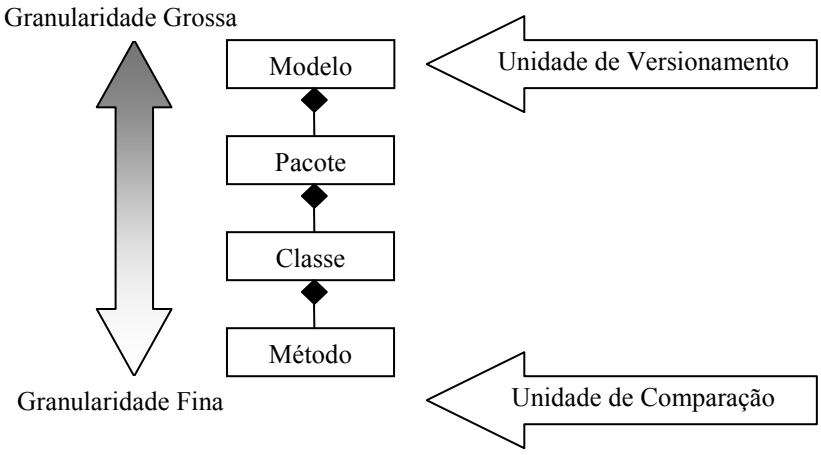


Figura 5.12: Unidades de comparação e de versionamento em modelos XMI.

5.4.2 Abordagem proposta

O Odyssey-VCS (OLIVEIRA *et al.*, 2004; OLIVEIRA *et al.*, 2005) consiste em um mecanismo de controle de versões que atua sobre modelos descritos de acordo com o meta-modelo da UML (OMG, 2001), utilizando de políticas configuráveis para as unidades de comparação e de versionamento. Diferentemente das abordagens convencionais, que têm unidades de comparação e versionamento fixas (em arquivos e linhas, respectivamente), o Odyssey-VCS permite que esses elementos sejam configuráveis em função do próprio meta-modelo da UML. Para que isto seja possível, é necessário fazer uso da hierarquia de composição estabelecida no meta-modelo da UML. Com essa informação, juntamente com a definição das unidades de comparação e versionamento, o Odyssey-VCS é capaz de identificar quais elementos devem ser analisados no momento da detecção de conflitos e quais elementos devem ter informações de versionamento persistidas.

Usualmente, as ferramentas CASE dividem elementos de modelo em duas categorias: elementos semânticos e elementos sintáticos. Os elementos semânticos representam conceitos, enquanto elementos sintáticos são representações dos elementos semânticos em diagramas. Desta forma, características como cor, posição e tamanho são armazenadas nos elementos sintáticos, enquanto os dados reais da entidade que está

sendo modelada são armazenados nos elementos semânticos. No contexto da abordagem proposta, ICs são elementos semânticos de ferramentas CASE baseadas em UML. Para ser mais preciso, qualquer subtipo de *Elemento de Modelo* (*ModelElement*) do meta-modelo da UML pode ser considerado um IC pelo Odyssey-VCS. Devido a esta característica, o Odyssey-VCS é capaz de versionar relacionamentos entre elementos UML, pois relacionamentos também são elementos de modelo UML. Exemplos de elementos de modelo UML são: casos de uso, atores, classes, associações entre classes, operações, atributos, componentes, entre outros.

Devido à complexidade do modelo de dados em questão (UML), não é desejável ter um único comportamento de versionamento para todo tipo de IC. Além disso, a maioria das normas de GCS recomenda a definição de planos individuais de GCS para cada projeto, e uma seção importante do plano de GCS é a identificação dos ICs. A seção de identificação dos ICs descreve em que nível de granularidade a GCS deverá atuar e onde podem ser obtidos os ICs identificados. Contudo, as abordagens convencionais só atuam no nível de arquivos, limitando as possibilidades de definição de ICs. No Odyssey-VCS é possível definir ICs em granularidade fina. Por exemplo, classe pode ser definida como um IC atômico para um dado projeto. Por outro lado, operação e atributo podem ser definidos como ICs atômicos em um outro projeto. A flexibilidade do Odyssey-VCS permite uma definição mais precisa de ICs, atuando de forma alinhada com as recomendações das normas de GCS.

Além do uso do meta-modelo da UML como modelo de dados, o Odyssey-VCS tem o seu próprio modelo de versionamento, que também é um meta-modelo MOF. No modelo de versionamento do Odyssey-VCS, cada instância do tipo *Projeto* está associada a diversas instâncias do tipo *Item de Configuração*, e cada instância do tipo *Item de Configuração* está associada a diversas instâncias do tipo *Versão*. Por sua vez, cada instância do tipo *Versão* está associada a uma instância do tipo *Elemento de Modelo*, que é um super-tipo genérico do meta-modelo da UML. Desta forma, o Odyssey-VCS é capaz de versionar elementos do meta-modelo da UML, mantendo os dados de versionamento no próprio meta-modelo do Odyssey-VCS. A Figura 5.13 apresenta uma visão simplificada do relacionamento entre o modelo de versionamento e o modelo de dados (UML).

Vale notar que a abordagem proposta tem algumas características de Gerenciamento de Dados de Produto (*Product Data Management*) (ESTUBLIER *et al.*, 1998): o Odyssey-VCS utiliza um modelo de dados orientado a objetos, o modelo de

versionamento não está embutido no modelo de dados (ele referencia o modelo de dados) e os objetos são armazenados em um banco de dados, evitando dependências ao sistema de arquivos, como será detalhado no Capítulo 6.

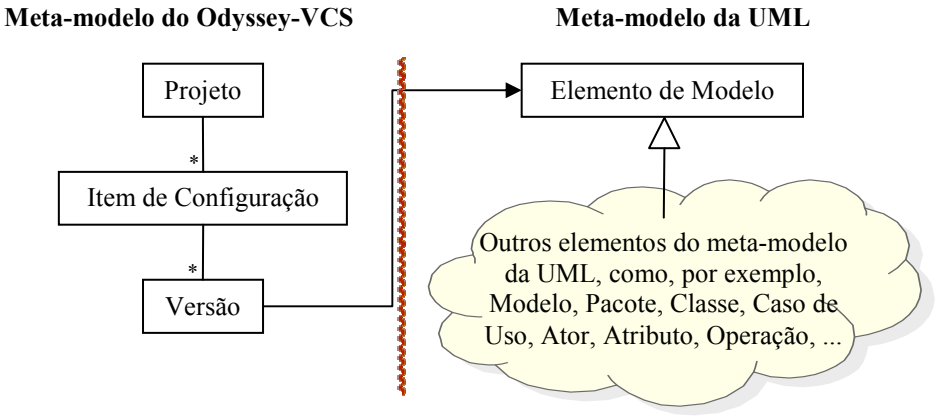


Figura 5.13: Relacionamento entre o modelo de versionamento e o modelo de dados.

O Odyssey-VCS faz uso de uma estratégia otimista para o controle de concorrência. Essa estratégia otimista, que já é utilizada em sistemas de controle de versão baseados em arquivo desde os anos 80 (CONRADI e WESTFECHTEL, 1998), permite que engenheiros de software modifiquem o mesmo modelo em paralelo e juntem as suas contribuições durante o processo de *check-in*, como mostrado na Figura 5.14. Esta estratégia alavanca o trabalho em equipe, mas necessita da definição de algoritmos de junção, como detalhado no Capítulo 6.

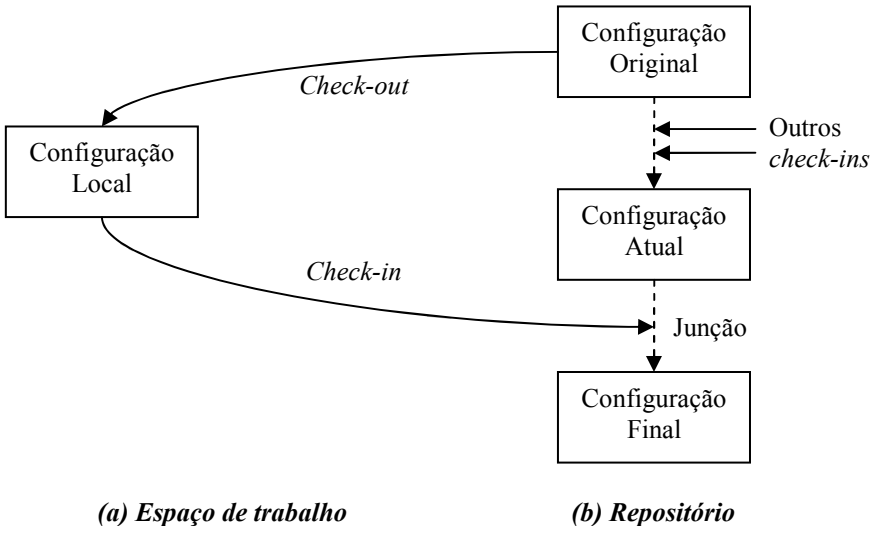


Figura 5.14: Estratégia otimista para o controle de concorrência.

A Configuração Original, mostrada na Figura 5.14.b, é o ponto de começo de um ciclo de desenvolvimento. A Configuração Local é criada quando um dado engenheiro de software executa o processo de *check-out*, seguido de modificações. Durante esse período de tempo, outros engenheiros de software também trabalham sobre a

Configuração Original, gerando a Configuração Atual. Finalmente, a Configuração Local é enviada ao repositório via um processo de *check-in* e juntada à Configuração Atual, dando origem à Configuração Final.

Quando um conflito é detectado durante o processo de junção, o *check-in* como um todo é cancelado e o usuário é informado sobre os detalhes do conflito que ocorreu. Além disso, a Configuração Original, Configuração Local e Configuração Atual são providas ao usuário, para que uma junção manual possa ser efetuada. Após a junção manual, que pode ser apoiada por ferramentas externas, o engenheiro de software reenvia o modelo UML resultante para o repositório.

5.5 Odyssey-BRD

Atualmente, existem diversas soluções para a construção de executáveis a partir de código fonte. Dentre elas estão o make (FELDMAN, 1979) e o Ant (HATCHER e LOUGHRAN, 2004). Contudo, o conceito de construção não está relacionado somente com geração de executáveis, mas sim no processo mais amplo de obtenção de ICs derivados a partir de um conjunto de ICs fonte. No caso de DBC, é necessário fornecer a metáfora de GCS embutida nos próprios conceitos de componente, conectores e interfaces.

5.5.1 Detalhamento do problema

No contexto de DBC, componentes, interfaces e conectores são utilizados para especificar arquiteturas de componentes. Contudo, a utilização de arquiteturas de componentes leva a um problema clássico da engenharia de software: é necessário manter a rastreabilidade entre a arquitetura e o código fonte, visto que ambos evoluem com o passar do tempo. Arquiteturas são utilizadas para evolução em tempo de execução (OREIZY *et al.*, 1998), seleção de produtos a partir de linhas de produtos (BOSCH, 2000), abordagens avançadas de teste (RICHARDSON e WOLF, 1996), análise de impacto (ZHAO *et al.*, 2002) e diversas outras atividades que não funcionariam corretamente sem um mapeamento detalhado entre a arquitetura e a sua implementação.

O fato de ambos, arquitetura e código fonte, poderem evoluir independentemente consiste em um fator a mais de complexidade para o problema. O estabelecimento de um mapeamento inicial entre arquitetura e código fonte pode ser possível. Contudo, não é razoável esperar que os engenheiros de software mantenham

esse mapeamento manualmente sempre que a arquitetura ou o código fonte evoluírem, em especial para sistemas de grande escala, que estão sofrendo modificações constantemente.

Diversas abordagens atuam sobre este problema utilizando diferentes estratégias. Essas abordagens podem ser genericamente divididas em dois grupos. No primeiro grupo, existe um rastro perfeito entre a arquitetura e o código fonte, pois um está embutido no outro. Por exemplo, tanto ArchJava (ALDRICH *et al.*, 2002) quanto XDoclet (WALLS e RICHARDS, 2003) embutem a definição da arquitetura no código fonte. Apesar dessa solução ser 100% efetiva em manter o rastro, ela não é válida em todas as situações. Em alguns casos, a arquitetura do sistema é especificada por meio de uma linguagem de descrição de arquiteturas (ADL), separada do código fonte. Nesses casos, é comum ter pessoas diferentes utilizando ferramentas diferentes para manter essas duas representações do software.

No segundo grupo, as ligações de rastreabilidade são detectadas novamente sempre que uma quantidade razoável de modificações é feita sobre as diferentes representações do software. As técnicas de mineração de dados, (SHIRABAD *et al.*, 2001; YING *et al.*, 2004; ZIMMERMANN *et al.*, 2004), recuperação de informação (ANTONIOL *et al.*, 2002; HUFFMAN HAYES *et al.*, 2003; MARCUS e MALETIC, 2003; SETTIMI *et al.*, 2004) e análise sintática (BRIAND *et al.*, 2003) fazem parte desse grupo. Essas técnicas, apesar de úteis para uma detecção inicial das ligações de rastreabilidade, não são sensíveis às pequenas divergências entre as representações, perdendo um valioso conhecimento para a manutenção contínua do sincronismo entre a arquitetura e sua implementação. Mais ainda, essas técnicas atuam somente em dimensões específicas do relacionamento entre arquiteturas e código fonte. Por exemplo, técnicas de mineração de dados analisam somente a dimensão histórica, enquanto técnicas de recuperação de informação são focadas em similaridades entre termos, e técnicas de análise sintática levam em conta somente a estrutura dos artefatos.

A manutenção das ligações de rastreabilidade entre elementos arquiteturais (componentes, interfaces e conectores) e código fonte também é fundamental para que a GCS possa atuar em níveis mais altos de abstração, propagando as ações para os níveis inferiores. Por exemplo, ao definir uma linha base para um determinado componente, não é desejado que as versões do código fonte que implementam esse componente mudem sem um processo formal de controle de modificações. Mais ainda, para a execução de comandos de *check-out* visando obter o código fonte de um determinado

componente, é necessário identificar exatamente qual código fonte implementa esse componente. A manutenção do sincronismo entre a arquitetura e o código fonte pode facilitar substancialmente essa tarefa.

Finalmente, após a construção e empacotamento do componente, é necessário disponibilizá-lo no ambiente de produção. Contudo, a necessidade de determinados componentes na aplicação alvo pode variar em função de diferentes demandas. Por exemplo, em uma ferramenta CASE, determinados usuários necessitam somente dos mecanismos de modelagem e de geração de código, enquanto outros usuários precisam de recursos referentes à engenharia reversa. Desta forma, técnicas de variabilidade (SVAHNBERG *et al.*, 2005) podem ser adotadas para possibilitar a adaptação da aplicação alvo em tempo de execução.

5.5.2 Abordagem proposta

O Odyssey-BRD é decomposto em duas principais vertentes, que são o ArchTrace (MURTA *et al.*, 2006b; MURTA *et al.*, 2006c) e a Carga Dinâmica (MELO JR. *et al.*, 2004; MURTA *et al.*, 2004b). O ArchTrace consiste em um mecanismo para a evolução das ligações de rastreabilidade existentes entre a arquitetura e a sua implementação. Por outro lado, a Carga Dinâmica consiste em um mecanismo para o tratamento da variabilidade de sistemas em produção, possibilitando a seleção de quais componentes devem ser implantados na aplicação alvo em tempo de execução. O restante desta seção detalha cada uma dessas vertentes.

5.5.2.1 ArchTrace

O ArchTrace faz uso de uma arquitetura flexível, baseada em políticas configuráveis, com o objetivo de apoiar a evolução de ligações de rastreabilidade. A operação do ArchTrace consiste em monitorar tanto o ambiente utilizado para a definição de arquiteturas quanto o ambiente de controle de versões do código fonte. A partir de eventos gerados nesses ambientes, um mecanismo de notificação avisa que houve modificações nos artefatos monitorados. Internamente, o ArchTrace executa todas as políticas disponíveis para a atualização das ligações de rastreabilidade existentes, como apresentado na Figura 5.15. Por exemplo, quando um engenheiro de software executa *check-in* referente a uma modificação no código fonte, o ArchTrace pode executar uma política que adiciona uma ligação de rastreabilidade entre o elemento arquitetural apropriado e a nova versão do código fonte. Além disso, uma

segunda política também pode ser executada para remover a ligação de rastreabilidade antiga, entre o elemento arquitetural e a versão anterior do código fonte.

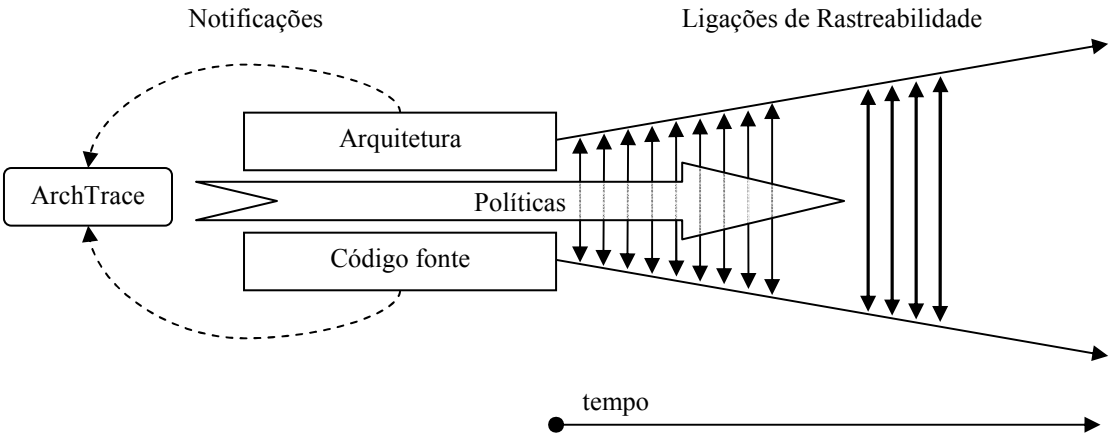


Figura 5.15: Visão geral da abordagem ArchTrace.

Um aspecto importante do ArchTrace é ser extensível em relação ao seu conjunto de políticas utilizadas na evolução de ligações de rastreabilidade. Apesar de existir um conjunto inicial de políticas, detalhado no Caplítulo 6, novas políticas podem ser acrescentadas a esse conjunto inicial. Por exemplo, a preocupação principal do ArchTrace não é na detecção inicial das ligações de rastreabilidade, mas sim na evolução das ligações de rastreabilidade já detectadas, por ser um problema complexo e ainda não resolvido completamente (MEDVIDOVIC e ROSENBLUM, 1997). Contudo, técnicas como mineração de dados, recuperação de informação e análise sintática podem ser vistas como políticas adicionais ao ArchTrace, aumentando as suas capacidades atuais com um maior apoio para a fase de detecção inicial.

As políticas do ArchTrace são intencionalmente simples, cada uma capturando comportamentos específicos, relacionados à evolução das ligações de rastreabilidade. Por exemplo, existem políticas específicas para o *check-in* de novas versões de elementos arquiteturais, para o *check-in* de novas versões do código fonte, para o tratamento de elementos imutáveis, para a remoção de ligações de rastreabilidade obsoletas, entre outras. A combinação dessas políticas caracteriza o estilo de trabalho do engenheiro de software. Desta forma, o ArchTrace permite que políticas sejam habilitadas ou desabilitadas, adaptando a sua execução às necessidades específicas de determinados perfis de usuários e momentos do ciclo de vida do software.

É importante notar que a execução de determinadas políticas pode levar à execução de outras políticas. Desta forma, o conjunto inicial de políticas colabora ativamente na atividade de atualização das ligações de rastreabilidade. Usualmente, um

único *check-in* efetuado pelo arquiteto ou engenheiro de software pode desencadear na execução de diversas políticas.

O ArchTrace diferencia quatro tipos de políticas: políticas de evolução de elemento arquitetural, políticas de evolução da implementação, políticas de pré-rastreabilidade e políticas de pós-rastreabilidade. As políticas de evolução de elemento arquitetural são acionadas sempre que o arquiteto faz modificações na arquitetura. Da mesma forma, as políticas de evolução da implementação são acionadas sempre que o engenheiro de software faz modificações no código fonte. Ambas são portas de entrada para o tratamento dos eventos notificados ao ArchTrace e atuam principalmente na sugestão inicial de novas ligações de rastreabilidade ou remoção de ligações de rastreabilidade existentes.

As políticas de pré-rastreabilidade podem ser vistas como mecanismos de verificação sobre as ligações de rastreabilidade que estão prestes a serem adicionadas ou removidas. Essas políticas são acionadas sempre que outras políticas tentam alterar o conjunto ligações de rastreabilidade existentes. A sua principal função é evitar que o conjunto de ligações de rastreabilidade existentes fique inconsistente. Caso alguma inconsistência seja detectada, a política de pré-rastreabilidade pode cancelar a ação que gera a inconsistência. Por exemplo, caso uma outra política qualquer tente modificar o conjunto de ligações de rastreabilidade de uma versão imutável de um componente, uma política do tipo pré-rastreabilidade deve cancelar essa ação, pois uma versão imutável consiste em uma linha base, e não deve ser alterada sem que uma nova versão seja gerada.

Finalmente, as políticas de pós-rastreabilidade são acionadas sempre que a adição ou remoção de ligações de rastreabilidade se concretiza (ou seja, é aprovada pelas políticas de pré-rastreabilidade). A principal função desse tipo de política é atualizar outras ligações de rastreabilidade em resposta a modificações no conjunto atual de ligações de rastreabilidade. Por exemplo, após adicionar uma ligação de rastreabilidade entre um elemento arquitetural existente e nova versão de código fonte é necessário remover a ligação de rastreabilidade para a versão anterior do código fonte. Essa ação é usualmente efetuada por políticas do tipo pós-rastreabilidade. Desta forma, a execução de políticas de pós-rastreabilidade pode levar a um novo ciclo de execuções de políticas de pré-rastreabilidade e pós-rastreabilidade, recursivamente.

5.5.2.2 Carga Dinâmica

O processo de construção e empacotamento dos componentes pode fazer uso de ferramentas tradicionais, como, por exemplo, o make (FELDMAN, 1979) e o Ant (HATCHER e LOUGHRAN, 2004), apoiadas pelo conhecimento contido no ArchTrace. Esse processo produz distribuições binárias do componente, de acordo com a tecnologia utilizada. Contudo, essas distribuições binárias dos componentes precisam ser implantadas, de forma sistemática, no ambiente de produção.

A abordagem de Carga Dinâmica foi definida para satisfazer a restrições sobre quais componentes devem ser implantados em conjunto no ambiente de produção. Alguns componentes são indesejáveis ou incompatíveis com outros componentes já implantados. Por outro lado, outros componentes podem ser redundantes, como, por exemplo, os diferentes mecanismos de persistência de uma ferramenta CASE. Para atenuar esse problema de forma flexível, o mecanismo de Carga Dinâmica possibilita a seleção, obtenção e implantação dos componentes em tempo de execução.

Para viabilizar a identificação de quais componentes estão disponíveis e as dependências entre os componentes é utilizado um descritor que segue um esquema XML predefinido. A partir do processamento desse descritor, a Carga Dinâmica consegue calcular as dependências dos componentes selecionados pelo usuário e recuperar os componentes de uma biblioteca previamente estabelecida. Vale ressaltar que esse descritor é, na verdade, uma visão arquitetural do sistema alvo, que estabelece uma configuração com todas as versões dos componentes que podem ser utilizados e descreve em detalhe as dependências desses componentes.

Após a recuperação dos componentes necessários, se dá início ao processo de implantação em tempo de execução. Esse processo está intimamente ligado à linguagem de programação utilizada, pois diferentes linguagens fazem uso de diferentes mecanismos para a ligação das partes dos sistemas. Por exemplo, no caso de Java, uma classe especial é responsável pela carga de novas classes no sistema. Essa classe, denominada *ClassLoader*, deve ser adaptada para passar a considerar os componentes obtidos dinamicamente. Além disso, um processo de introspecção é necessário para identificar os pontos de entrada de execução dos componentes. Finalmente, vale ressaltar que os componentes devem seguir algum tipo de interface padrão para viabilizar a sua conexão. Sem esse tipo de recurso, o processo de conexão se torna ainda mais complexo, fugindo do escopo desse trabalho.

5.6 Odyssey-WI

O sucesso da adoção de novos métodos e ferramentas no desenvolvimento de software está intimamente relacionado com os benefícios providos e o esforço adicional necessário para atingir esses benefícios. O caso da GCS aplicada ao DBC não é diferente. O DBC por si só é composto por atividades de alta complexidade, e o aumento da burocracia na execução dessas atividades pode inviabilizar a adoção da GCS.

Por esta razão, uma atenção especial deve ser despendida na integração do ferramental e processos de GCS com o ferramental e processos existentes no DBC. Para que essa interação seja a mais suave possível, as funcionalidades de GCS devem ficar encapsuladas na metáfora atual de DBC, que engloba componentes, interfaces e conectores, conforme proposto pelo Odyssey-BRD.

5.6.1 Detalhamento do problema

A integração entre GCS e DBC pode ser dividida em dois níveis de complexidade: (1) integração simples e (2) integração avançada. A integração simples abrange os aspectos referentes à integração entre as próprias ferramentas de GCS, entre as ferramentas de GCS e de DBC, o acesso aos recursos de GCS na interface de usuário dos ambientes de DBC e o mecanismo de carga de componentes no ambiente de produção. Esses aspectos de integração são usualmente tratados dentro do próprio escopo dos demais sistemas propostos no Odyssey-SCM.

A integração avançada busca a propagação de conhecimento entre os ambientes de GCS e de DBC. A função de contabilização da situação da configuração, da GCS, é responsável por armazenar e relatar informações durante o desenvolvimento e manutenção de software. Essas informações, existentes no contexto dos sistemas de controle de modificações e de controle de versões, são valiosas para a execução das atividades do DBC.

Sempre que uma solicitação de modificação é aprovada, e a implementação é iniciada, modelos de análise e projeto podem ser alterados para atender às novas necessidades. Quando elementos de modelo estão sendo alterados, é importante detectar quais outros elementos também necessitam de alterações. Para viabilizar esta atividade, denominada análise de impacto, é necessário fazer uso de ligações de rastreabilidade que permitam identificar os elementos que precisam ser modificados em conjunto.

Genericamente, dois tipos de ligações de rastreabilidade podem ser considerados (GOTEL e FINKELSTEIN, 1994): ligações de rastreabilidade de pré-especificação e ligações de rastreabilidade de pós-especificação. As ligações de rastreabilidade de pré-especificação estão relacionadas com as origens dos requisitos e estão mais ligadas à área de engenharia de requisitos. Por outro lado, as ligações de rastreabilidade de pós-especificação estão relacionadas com as diversas abstrações necessárias para concretizar os requisitos e estão intimamente ligadas à área de GCS.

Dentre as ligações de rastreabilidade de pós-especificação, podem ser detectados ao menos dois tipos diferentes de ligações de rastreabilidade (KOWALCZYKIEWICZ e WEISS, 2002): (1) ligações de rastreabilidade intermodelos, que são ligações de rastreabilidade entre os diversos níveis de abstração de um mesmo elemento conceitual e (2) ligações de rastreabilidade intramodelo, que são ligações de rastreabilidade entre diferentes elementos conceituais em um mesmo nível de abstração. Independentemente do tipo das ligações de rastreabilidade de pós-especificação, a sua detecção é considerada um problema em aberto na engenharia de software, devido à complexidade envolvida no descobrimento dos reais motivos do seu surgimento.

5.6.2 Abordagem proposta

O uso de técnicas de mineração de dados pode apoiar na detecção dessas dependências de manutenção, encontrando regras do tipo: "engenheiros de software que modificam esses elementos também modificam esses outros elementos". Esse tipo de conhecimento pode ser útil para apoiar diversas atividades do DBC e da GCS, como, por exemplo: sugerir modificações futuras, prevenir erros oriundos de modificações incompletas, detectar efeitos colaterais de modificações, apoiar na análise de impacto de modificações e detectar falhas de projeto devido ao alto acoplamento entre artefatos não correlatos.

O Odyssey-WI (DANTAS *et al.*, 2004; DANTAS *et al.*, 2005; DANTAS *et al.*, 2006) faz uso dessas técnicas sobre repositórios de versões de ICs e sobre repositórios de solicitações de modificação para detectar ligações de rastreabilidade de pós-especificação, intramodelo e intermodelos, entre elementos UML. Essas ligações de rastreabilidade detectadas a partir das análises das modificações anteriores, denominadas **rastros de modificação**, podem ser utilizadas para apoiar o engenheiro de software durante a manutenção, pois quando uma modificação é implementada, a

própria implementação da modificação pode motivar o surgimento de novas modificações, como exibido na Figura 5.16.

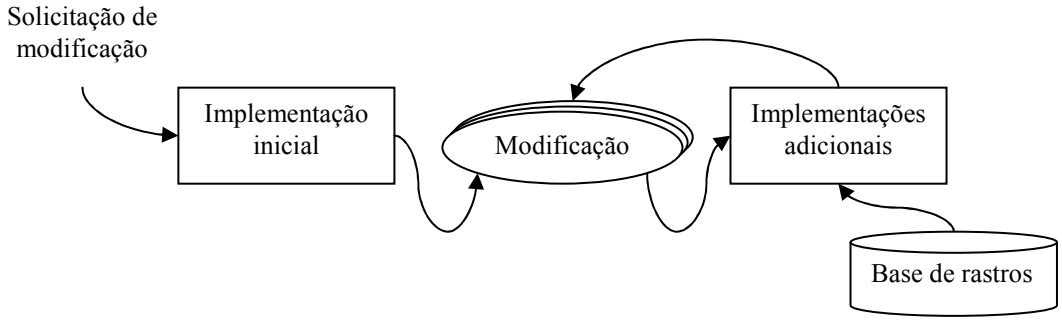


Figura 5.16: Detecção de conjuntos de modificações.

Desta forma, para cada modificação implementada, é possível correlacionar as informações existentes no sistema de controle de modificações com os artefatos modificados no sistema de controle de versões. Essa correlação apóia no surgimento gradativo tanto dos rastros de modificação intermodelos quanto dos rastros de modificação intramodelo, como exibido na Figura 5.17.

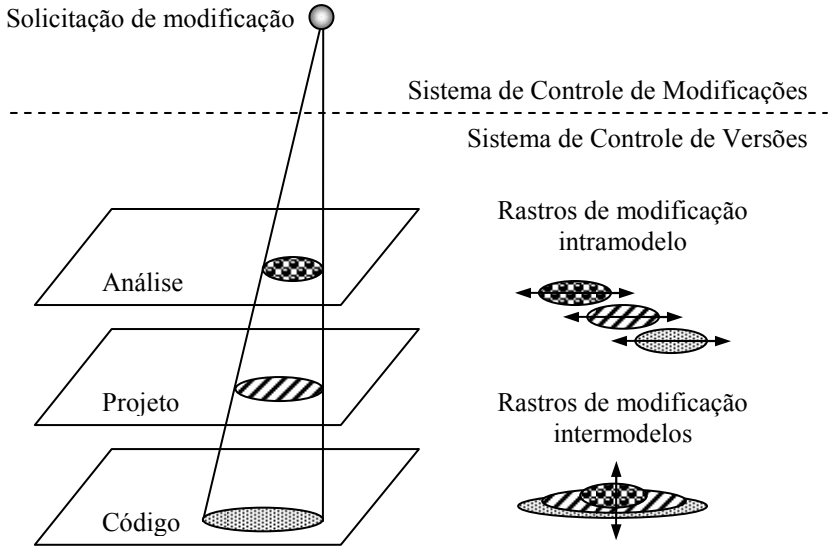


Figura 5.17: Rastros de modificação.

Contudo, a existência isolada dos rastros de modificação, apesar de útil, suscita dúvidas sobre o porquê do seu surgimento. Ainda que não seja possível identificar automaticamente a real razão da existência de um rastro de modificação, é possível reunir informações das diversas modificações que motivaram a sua existência. Essas informações, quando organizadas de acordo com a estrutura 5W+1H (GUTWIN e GREENBERG, 2002), identificam **quem** implementou as modificações, **quando** as modificações foram implementadas, **onde** deveria ter sido alterado para implementar as

modificações, **como** as modificações deveriam ter sido implementadas, **o quê** era necessário e **por quê** existiam essas necessidades. Essas informações são fundamentais para apoiar na fundamentação dos rastros de modificação.

As atividades do processo de controle de modificações, juntamente com informações existentes no sistema de controle de versões, apóiam na coleta dessas informações da seguinte forma:

- **Quem?** Obtido nos registros de *check-in* do sistema de controle de versões e na atividade de implementação do sistema de controle de modificações, que indica o responsável pela implementação;
- **Quando?** Obtido nos registros de *check-in* do sistema de controle de versões e na atividade de implementação do sistema de controle de modificações, que indica o momento da implementação;
- **Onde?** Obtido nos registros de *check-in* do sistema de controle de versões, e na atividade de análise de impacto do sistema de controle de modificações, que indica os ICs possivelmente afetados pela implementação da modificação;
- **Como?** Obtido na atividade de análise de impacto do sistema de controle de modificações, que indica possíveis estratégias para implementar a modificação;
- **O quê?** Obtido na atividade de solicitação da modificação do sistema de controle de modificações, que relata a necessidade de modificação; e
- **Por quê?** Obtido na atividade de solicitação da modificação do sistema de controle de modificações, que justifica a necessidade da modificação.

5.7 Considerações finais

Este capítulo apresentou a abordagem Odyssey-SCM, que é decomposta em cinco subabordagens: Odyssey-SCMP, Odyssey-CCS, Odyssey-VCS, Odyssey-BRD e Odyssey-WI. As principais características, referentes à subabordagem Odyssey-SCMP, são:

- Customização do processo de GCS para o DBC;
- Colaboração entre participantes do processo multi-nível de produção, consumo e utilização de componentes;

– Propagação de partes de solicitações em função da responsabilidade contratual de manutenção;

– Embasamento nas normas ISO 10007, IEEE Std 282 e IEEE Std 1042, e nos modelos MPS.BR e CMMI; e

– Implantação do processo no sistema Odyssey-CCS.

As principais características, referentes à subabordagem Odyssey-CCS, são:

– Modelagem gráfica, simulação e execução do processo de controle de modificações;

– Configuração da coleta de informações necessárias para as atividades do processo;

– Metáfora centrada na modificação, permitindo o acesso a todos os documentos produzidos durante o ciclo de vida da modificação; e

– Estabelecimento e utilização de informações referentes à reutilização de componentes.

As principais características, referentes à subabordagem Odyssey-VCS, são:

– Modelo de dados focado em modelos UML; e

– Políticas configuráveis para unidades de comparação e versionamento.

As principais características, referentes à subabordagem Odyssey-BRD, são:

– Mapeamento entre os conceitos de DBC e GCS, permitindo a manipulação dos diversos artefatos que compõem um componente via o próprio componente;

– Evolução desse mapeamento de acordo com políticas extensíveis; e

– Obtenção e implantação dinâmica dos componentes.

As principais características, referentes à subabordagem Odyssey-WI, são:

– Utilização de mineração de dados sobre repositórios de GCS para detecção de ligações de rastreabilidade de pós-especificação intramodelo e intermodelos, apoiando a detecção de modificações incompletas, análises de impacto e reengenharia dos componentes; e

– Contextualização dos rastros detectados via análise dos próprios dados minerados.

Apesar de oferecer contribuições pontuais a subsistemas específicos de GCS, a coleção dessas contribuições pontuais confere ao Odyssey-SCM uma abordagem ampla de GCS no contexto de DBC, com contribuições tanto individualmente, para cada uma das cinco subabordagens, quanto como solução integrada para o problema de GCS no DBC. Além disso, com esse apoio mínimo, necessário aos cinco tópicos discutidos, abrem-se novas portas para pesquisas futuras. O Capítulo 6 detalha a abordagem Odyssey-SCM, apresentando tanto as decisões de implementação que foram tomadas, quanto um exemplo de utilização do protótipo implementado.

Capítulo 6 – O Protótipo Odyssey-SCM

6.1 Introdução

O Capítulo 5 apresentou uma visão geral da abordagem Odyssey-SCM, discutindo seus requisitos, arquitetura e principais subabordagens. Neste capítulo, são apresentados aspectos tecnológicos e implementacionais do Odyssey-SCM. Para cada subabordagem do Odyssey-SCM é apresentado um detalhamento das decisões adotadas e uma comparação com outras abordagens existentes na literatura.

Desta forma, este capítulo está organizado em outras sete seções, além desta introdução. A Seção 6.2 apresenta um exemplo intencionalmente simples que será utilizado nas demais seções, visando facilitar a compreensão. A Seção 6.3 descreve o processo Odyssey-SCMP em detalhes, fazendo uso de uma notação utilizada pelo MPS.BR para descrição das atividades de um processo (SOFTEX, 2006a). A Seção 6.4 apresenta a máquina de processos, o mecanismo de coleta de informações e o mapa de reutilização utilizados pelo Odyssey-CCS. A Seção 6.5 discute a arquitetura interna do Odyssey-VCS, assim como o algoritmo para junção de elementos de modelo UML. A Seção 6.6 descreve o mecanismo de evolução de ligações de rastreabilidade entre arquitetura e código fonte e o mecanismo para carga de componentes em ambiente de produção por demanda, ambos pertencentes ao Odyssey-BRD. A Seção 6.7 apresenta o algoritmo de mineração de dados e o mecanismo para contextualização dos rastros de modificação utilizados pelo Odyssey-WI. Finalmente, a Seção 6.8 conclui este capítulo discutindo como o Odyssey-SCM poderia contribuir em situações mais complexas que o exemplo tratado ao longo do capítulo.

Vale ressaltar que as demais seções estão organizadas de forma similar aos capítulos anteriores, visando permitir, por meio de uma leitura transversal, a visualização dos diferentes níveis de abstração de cada subabordagem.

6.2 Exemplo

Com o intuito de uniformizar os exemplos apresentados no decorrer deste capítulo, é utilizado um cenário fictício e intencionalmente simples de desenvolvimento e manutenção de um sistema de controle de hotelaria, adaptado da literatura

(TEIXEIRA, 2003). Os principais casos de uso, componentes e classes do sistema são ilustrados na Figura 6.1.

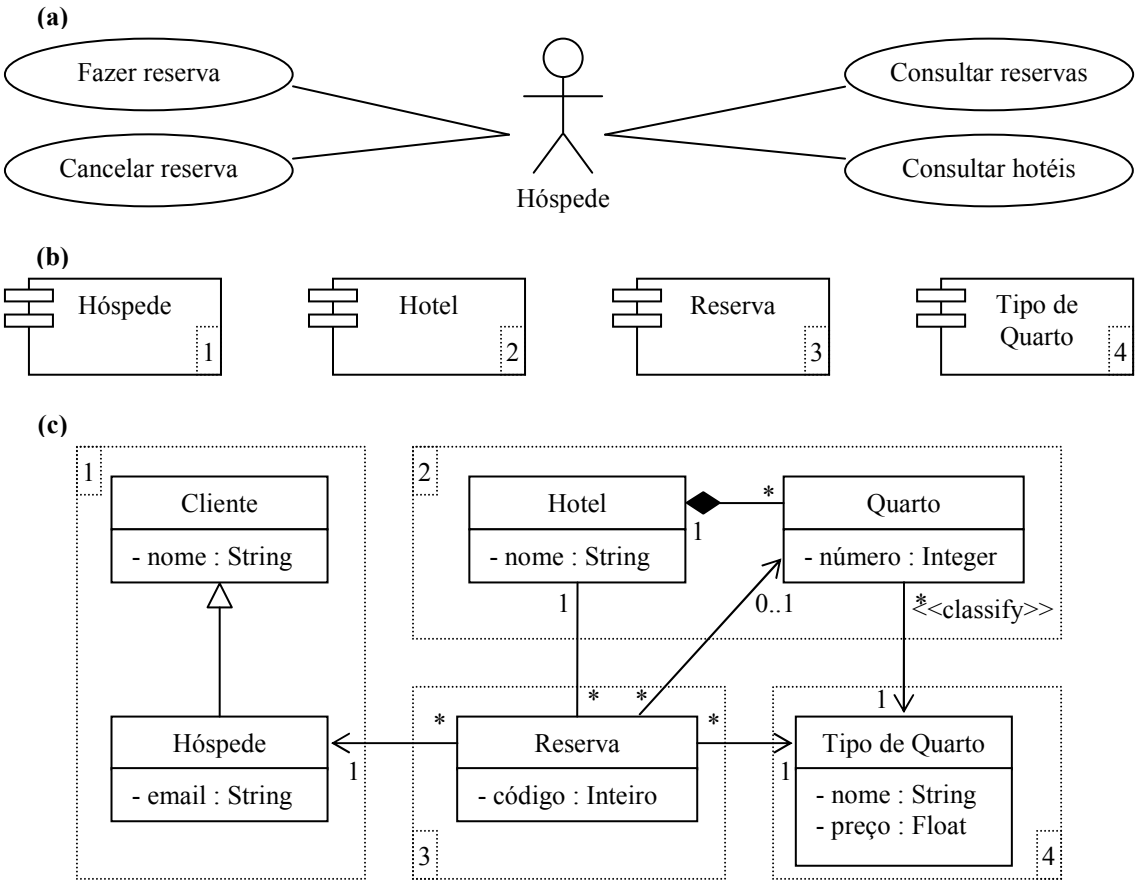


Figura 6.1: Casos de uso (a), componentes (b) e classes (c) do exemplo de hotelaria.

Analisando a Figura 6.1, é possível notar que o componente "Hóspede" é implementado pelas classes "Cliente" e "Hóspede"; o componente "Hotel" é implementado pelas classes "Hotel" e "Quarto"; o componente "Reserva" é implementado pela classe "Reserva"; e o componente "Tipo de Quarto" é implementado pela classe "Tipo de Quarto". Para demonstrar o funcionamento do Odyssey-SCM, seis modificações referentes a novos requisitos ou correções sobre requisitos existentes são efetuadas sobre o sistema exemplo. Essas modificações estão listadas na Tabela 6.1.

Tabela 6.1: Modificações realizadas no exemplo de hotelaria.

N°	Descrição
1	Verificar, durante a reserva, a disponibilidade de quartos para o período desejado.
2	Mostrar o tipo de quarto, características e preço no momento de consultas sobre a reserva.
3	Procurar por quartos em outros hotéis da mesma área quando nenhum quarto estiver disponível.
4	Não permitir duplicação de reserva de um quarto no mesmo período.
5	Tornar o quarto disponível no caso de cancelamento de reserva.
6	Mostrar hotéis reservados anteriormente como opções preferidas.

Durante a fase de implementação das modificações, alguns artefatos precisam ser modificados. A Tabela 6.2 apresenta os artefatos modificados para viabilizar a implementação de cada modificação apresentada na Tabela 6.1. Esses artefatos são apresentados juntamente com a versão produzida durante a implementação da modificação (número entre parênteses). Vale ressaltar que todos os artefatos apresentados na Figura 6.1 estavam na versão 1 antes das modificações serem implementadas, e que foi adotada uma estratégia de versionamento global, onde os números de versão subseqüentes são gerados por *check-in* e não por artefato. Além disso, é importante notar que uma modificação pode ser implementada por meio de mais de um *check-in*.

Tabela 6.2. Artefatos modificados no exemplo de hotelaria.

N°	Elementos Modificados		
	Casos de Uso	**Classes**	**Componentes**
1	Fazer Reserva (2).	Hóspede (3),Tipo de Quarto (3), Cliente (2), Quarto (2), Hotel (2), Reserva (2).	Reserva (2), Hotel (2), Hóspede (3), Tipo de Quarto (3).
2	Consultar Reserva (4).	Tipo de Quarto (4), Reserva (4).	Tipo de Quarto (4), Reserva (4).
3	Fazer Reserva (5).	Reserva (5), Hotel (5).	Reserva (5), Hotel (5).
4	Fazer Reserva (6).	Reserva (6), Hotel (6).	Reserva (6), Hotel (6).
5	Cancelar Reserva (7).	Reserva (7), Hotel (8).	Reserva (7), Hotel (8).
6	Consultar Hotéis (9).	Hotel (9), Hóspede (9).	Hotel (9), Hóspede (9).

6.3 Odyssey-SCMP

Esta seção detalha o processo Odyssey-SCMP de acordo com a notação utilizada pelo MPS.BR para descrição das atividades de um processo (SOFTEX, 2006a). Essa notação apresenta, para cada atividade do processo, as seguintes informações: o seu nome, uma breve descrição, critérios de entrada e saída, responsáveis e participantes na execução, ferramentas necessárias para a execução, produtos consumidos e produzidos, e outras atividades que devem ser executadas antes e depois da atividade em questão. Vale ressaltar que somente as ferramentas definidas neste trabalho são descritas no processo. Contudo, diversas outras ferramentas complementares podem ser utilizadas concomitantemente.

6.3.1 Detalhamento da solução

O processo Odyssey-SCMP detalhado nesta seção pode ser visto como um processo padrão, idealizado para equipes híbridas. Desta forma, ele deve ser adaptado quando for utilizado por equipes puramente produtoras ou consumidoras. Esse processo é composto pelas seguintes atividades:

Nome da Atividade:	**Preparação do ambiente de GCS.**
Descrição:	Preparar o plano de GCS, identificando quais artefatos devem ser considerados como ICs, e estabelecer a arquitetura que descreve as dependências entre esses artefatos. Neste momento as ferramentas devem ser configuradas para tratar os ICs identificados.
Pré-atividade:	-
Critérios de Entrada:	Demanda de componentes por consumidores.
Critérios de Saída:	Ambiente de GCS preparado.
Responsáveis:	Gerente de configuração.
Participantes:	Equipes consumidoras.
Produtos Requeridos:	Lista de requisitos dos componentes solicitados.
Produtos Gerados:	Arquitetura de componentes e descritor de comportamento (unidades de versionamento e comparação).
Ferramentas:	Odyssey-CCS, Odyssey-VCS, Odyssey-WI e Odyssey-BRD.
Pós-atividade:	Estabelecimento de linha base do componente.

Nome da Atividade:	**Estabelecimento de linha base do componente.**
Descrição:	Estabelecer, em marcos específicos do projeto (usualmente ao término das fases de análise, projeto e codificação), linhas bases do componente que serão utilizadas como referência para as fases posteriores.
Pré-atividade:	Preparação do ambiente de GCS.
Critérios de Entrada:	Término de fase do projeto.
Critérios de Saída:	Linha base do componente na fase estabelecida.
Responsáveis:	Gerente de configuração.
Participantes:	-
Produtos Requeridos:	ICs (produzidos na fase).
Produtos Gerados:	Linha base do componente (funcional, alocada ou de produto).
Ferramentas:	Odyssey-VCS e Odyssey-BRD.
Pós-atividade:	Solicitação de modificação ou Liberação do componente.

Nome da Atividade:	**Solicitação de modificação.**
Descrição:	Solicitar uma modificação, descrevendo o nome e a organização do solicitante, a data da solicitação, a versão dos ICs afetados, um indicativo de urgência, a necessidade e a justificativa da modificação.
Pré-atividade:	Estabelecimento de linha base do componente ou Liberação do componente.
Critérios de Entrada:	Necessidade de modificação sobre um componente.
Critérios de Saída:	Modificação solicitada.
Responsáveis:	Equipes consumidoras.
Participantes:	Gerente de configuração.
Produtos Requeridos:	Mapa de reutilização.
Produtos Gerados:	Solicitação de modificação.
Ferramentas:	Odyssey-CCS.
Pós-atividade:	Classificação da modificação.

Nome da Atividade:	**Classificação da modificação.**

Descrição:	Classificar a prioridade da modificação em relação às demais modificações solicitadas. Neste momento, é importante considerar a importância dessa modificação para as demais equipes consumidoras que reutilizam os componentes afetados.
Pré-atividade:	Solicitação de modificação.
Critérios de Entrada:	Ter uma modificação solicitada, mas ainda não priorizada.
Critérios de Saída:	Ter classificado a modificação em relação às demais modificações solicitadas.
Responsáveis:	Gerente de configuração.
Participantes:	Equipes consumidoras.
Produtos Requeridos:	Solicitação de modificação e mapa de reutilização.
Produtos Gerados:	Solicitação de modificação (classificada).
Ferramentas:	Odyssey-CCS.
Pós-atividade:	Análise de impacto da modificação.

Nome da Atividade:	**Análise de impacto da modificação.**
Descrição:	Analisar o impacto da modificação em questão em relação aos ICs afetados, alternativas, custo e cronograma de implementação, e analisar o interesse das demais equipes consumidoras pela modificação. Caso a modificação atinja componentes reutilizados, é necessário interagir com o processo de controle de modificações da equipe produtora para obter estimativas de custo e cronograma.
Pré-atividade:	Classificação da modificação.
Critérios de Entrada:	Modificação classificada como próxima a ser analisada.
Critérios de Saída:	Impacto da modificação analisado.
Responsáveis:	Analista de impacto.
Participantes:	Gerente de configuração, equipes consumidoras e equipes produtoras.
Produtos Requeridos:	Solicitação de modificação e mapa de reutilização.
Produtos Gerados:	Relatório de impacto.
Ferramentas:	Odyssey-CCS, Odyssey-WI e Odyssey-BRD.
Pós-atividade:	Avaliação da modificação.

Nome da Atividade:	**Avaliação da modificação.**
Descrição:	Avaliar a viabilidade de implementação da modificação em questão. O resultado dessa avaliação pode ser: (1) rejeitar a modificação, (2) postergar a modificação, devido à necessidade de uma análise de impacto mais profunda, (3) delegar a modificação para as equipes produtoras dos componentes afetados, (4) implementar a modificação, substituindo os componentes afetados por outros componentes equivalentes, ou (5) implementar a modificação, alterando os componentes afetados.
Pré-atividade:	Análise de impacto da modificação.
Critérios de Entrada:	Modificação analisada.
Critérios de Saída:	Modificação avaliada.
Responsáveis:	Comitê de controle da configuração.

Participantes:	-
Produtos Requeridos:	Solicitação de modificação e relatório de impacto.
Produtos Gerados:	Laudo de avaliação.
Ferramentas:	Odyssey-CCS.
Pós-atividade:	Solicitação de modificação (na equipe produtora), análise de impacto da modificação ou implementação da modificação.

Nome da Atividade:	**Implementação da modificação.**
Descrição:	Implementar a modificação em todos os níveis de abstração pertinentes (análise, projeto, código, teste, etc.). Caso a implementação seja via substituição dos componentes afetados, seguir o processo de aquisição. Caso a implementação seja via alteração dos componentes afetados, seguir o processo de solução técnica.
Pré-atividade:	Avaliação da modificação.
Critérios de Entrada:	Modificação aprovada para implementação.
Critérios de Saída:	Modificação implementada.
Responsáveis:	Engenheiros de software (analistas, projetistas, programadores, etc.).
Participantes:	Gerente de configuração.
Produtos Requeridos:	Solicitação de modificação, relatório de impacto, laudo de avaliação e ICs (afetados).
Produtos Gerados:	Relatório de implementação ou relatório de aquisição e ICs (criados/modificados).
Ferramentas:	Odyssey-CCS, Odyssey-VCS, Odyssey-WI e Odyssey-BRD.
Pós-atividade:	Verificação da modificação.

Nome da Atividade:	**Verificação da modificação.**
Descrição:	Verificar se a implementação está correta via técnicas de revisão ou testes.
Pré-atividade:	Implementação da modificação.
Critérios de Entrada:	Modificação implementada.
Critérios de Saída:	Modificação verificada.
Responsáveis:	Revisores ou equipe de testes.
Participantes:	Gerente de configuração.
Produtos Requeridos:	ICs (criados/modificados), casos e dados de teste (se pertinente).
Produtos Gerados:	Relatório de verificação.
Ferramentas:	Odyssey-CCS, Odyssey-VCS e Odyssey-BRD.
Pós-atividade:	Implementação da modificação ou atualização da linha base do componente com a modificação.

Nome da Atividade:	**Atualização da linha base do componente com a modificação.**
Descrição:	Incorporar a modificação na linha base do componente.
Pré-atividade:	Verificação da modificação.
Critérios de Entrada:	Modificação verificada.
Critérios de Saída:	Linha base do componente atualizada.
Responsáveis:	Integrador.

Participantes:	Gerente de configuração.
Produtos Requeridos:	Linha base do componente e ICs (criados/modificados).
Produtos Gerados:	Linha base do componente (atualizada).
Ferramentas:	Odyssey-CCS, Odyssey-VCS e Odyssey-BRD.
Pós-atividade:	Auditoria sobre a linha base do componente.

Nome da Atividade:	**Auditoria sobre a linha base do componente.**
Descrição:	Realizar auditoria física e funcional sobre a linha base do componente, verificando tanto a estrutura dos ICs quando a sua corretude em relação aos requisitos. Além disso, o próprio sistema de GCS também pode ser alvo de auditoria. A auditoria deve ser realizada sempre que o comitê de controle da configuração decidir gerar uma liberação a partir da linha base de um dado componente. Caso a auditoria seja executada com sucesso, a liberação pode ser gerada. Caso contrário, as não conformidades devem ser reportadas na forma de solicitações de modificação.
Pré-atividade:	Atualização da linha base do componente com a modificação.
Critérios de Entrada:	Necessidade de liberação de uma nova versão do componente.
Critérios de Saída:	Linha base do componente (auditada).
Responsáveis:	Auditor.
Participantes:	Gerente de configuração.
Produtos Requeridos:	Linha base do componente.
Produtos Gerados:	Laudo de auditoria.
Ferramentas:	Odyssey-CCS, Odyssey-VCS e Odyssey-BRD.
Pós-atividade:	Solicitação de modificação ou liberação do componente.

Nome da Atividade:	**Liberação do componente.**
Descrição:	Construir, liberar e implantar o componente.
Pré-atividade:	Auditoria sobre a linha base do componente.
Critérios de Entrada:	Linha base do componente aprovada pela auditoria.
Critérios de Saída:	Nova versão do componente implantada.
Responsáveis:	Equipe de liberação.
Participantes:	Gerente de configuração e cliente.
Produtos Requeridos:	Linha base do componente.
Produtos Gerados:	Liberação do componente.
Ferramentas:	Odyssey-BRD.
Pós-atividade:	Solicitação de modificação.

6.3.2 Considerações finais sobre o Odyssey-SCMP

O processo Odyssey-SCMP, detalhado nesta seção, foi idealizado levando em consideração as normas IEEE Std 828 (IEEE, 2005), IEEE Std 1042 (IEEE, 1987) e ISO 10007 (ISO, 1995a). Apesar do foco principal ser a função de controle da configuração, algumas atividades tratam superficialmente aspectos das demais funções.

Para o exemplo de hotelaria, este processo padrão deve ser utilizado na criação dos processos definidos, tanto da equipe produtora, que desenvolve os componentes,

quanto da equipe consumidora, que reutiliza os componentes para criar a aplicação como um todo.

6.4 Odyssey-CCS

A abordagem Odyssey-CCS, apresentada no Capítulo 5, foi implementada em Java (GOSLING *et al.*, 2005), de acordo com o estilo arquitetural MVC (*model-view-controller*) (REENSKAUG, 2003). A camada de visão faz uso das tecnologias JSP e Servlets (BODOFF *et al.*, 2004), enquanto a camada de controle faz uso da tecnologia *Enterprise JavaBeans* (BODOFF *et al.*, 2004), do tipo *Session Bean*. Já a camada de modelo, implementada sobre o repositório MOF MDR (MATULA, 2005), também é acessada via *Enterprise JavaBeans*, do tipo *Bean Managed Persistence Entity Bean*. O restante desta seção apresenta as decisões de projeto tomadas durante a implementação do Odyssey-CCS.

6.4.1 Detalhamento da solução

A abordagem Odyssey-CCS, como descrita no Capítulo 5, levanta quatro principais necessidades: (1) modelagem de processos, (2) modelagem de formulários (3) execução dos processos e (4) identificação da responsabilidade de manutenção.

Visando viabilizar a **modelagem de processos** segundo uma notação padronizada, foi utilizado o meta-modelo SPEM (*Software Process Engineering Metamodel*) (OMG, 2005b). O meta-modelo SPEM, definido pela OMG, faz uso de uma notação estendida de diversos modelos da UML com o intuito de permitir a modelagem de processos. Por exemplo, o modelo de atividades é utilizado para representar os fluxos de controle e de dados entre as atividades do processo. Por outro lado, o modelo de casos de uso é utilizado para representar os papéis responsáveis por executar as atividades do processo. Além disso, o modelo de classes é utilizado para representar estruturas hierárquicas e de composição entre os produtos de trabalho consumidos ou produzidos pelas atividades do processo.

O uso de um meta-modelo padrão, como o SPEM, aumenta a possibilidade de substituição das ferramentas que interagem para viabilizar a modelagem e execução do processo. O meta-modelo SPEM, assim como a UML, é um meta-modelo MOF, o que o torna compatível com a especificação XMI, que permite o intercâmbio de informação via um esquema XML específico. Além disso, também por ser um meta-modelo MOF, o SPEM é compatível com a especificação JMI (DIRCKZE, 2002), que permite a

manipulação dos elementos do meta-modelo diretamente em Java, via interfaces específicas.

Dentre os elementos de modelagem SPEM, além das atividades propriamente ditas e seus produtos de trabalho, existem elementos de controle, que são: paralelismo, sincronismo e decisão. O elemento de paralelismo possibilita que diversas atividades sejam executadas em paralelo. Por outro lado, o elemento de sincronismo possibilita que sejam definidos pontos de controle onde todas as atividades em paralelo tenham que estar finalizadas para que o fluxo de execução siga em frente. Finalmente, o elemento de decisão permite que, em função de uma escolha tomada por algum participante do processo, o fluxo de execução tome uma determinada direção em detrimento de outras.

Apesar da existência de diversas ferramentas para modelagem SPEM via *profile* UML, não foram encontradas ferramentas que atuassem diretamente no meta-modelo SPEM. Para contornar este problema, foi implementado um modelador SPEM no Odyssey-CCS, como componente dinâmico (*plug-in*) do ambiente Odyssey, apresentado na Figura 6.2. O modelador SPEM faz uso dos ícones sugeridos pela própria especificação SPEM.

O processo Odyssey-SCMP, parcialmente modelado na Figura 6.2, será posteriormente exportado via notação XMI para o ambiente de execução de processos. Após a implantação desse processo no ambiente de execução de processos, cada solicitação de modificação do exemplo de hotelaria, apresentada na Tabela 6.1, será tratada como uma instância de execução desse processo.

Cada atividade do processo agrega novos conhecimentos ao ciclo de vida das modificações. Na atividade "Solicitar Modificação", por exemplo, informações referentes ao tipo, razão e gravidade da modificação, entre outras, são necessárias para possibilitar a execução das demais atividades. Entretanto, na atividade "Analisar" ou "Avaliar", o conjunto de informações necessárias é completamente distinto. Com isso, é necessário definir formulários específicos para a coleta de informações de cada atividade. Para possibilitar a **modelagem de formulários** de documentação e posterior utilização desses formulários, a abordagem FrameDoc (MURTA, 1999; MURTA *et al.*, 2001a) foi adaptada para o problema em questão.

Na sua concepção original, o FrameDoc se destinava à documentação de componentes reutilizáveis. Todavia, suas características são propensas ao cenário em questão: criar formulários customizados de documentação e associar esses formulários aos produtos das atividades de GCS previamente modeladas. Contudo, devido a

evoluções tecnológicas e à plataforma escolhida para o Odyssey-CCS, foi necessário reimplementar a abordagem FrameDoc. Essa reimplementação, utilizando *Enterprise JavaBeans* e o repositório MOF MDR, permite a definição dos seguintes tipos de campo nos formulários: caixa de texto, área de texto, caixa de seleção (*check-box*), caixa de seleção única (*combo-box*), submissão de arquivo e referência. A Figura 6.3 apresenta o Odyssey-CCS sendo utilizado para modelar o formulário de solicitação de modificação do exemplo de hotelaria. Durante a implantação do processo apresentado na Figura 6.2, esse formulário é associado ao produto "Solicitação de Modificação". Desta forma, sempre que uma instância do processo atingir este produto, o formulário será apresentado ao responsável pela atividade que produz o produto em questão, e os documentos gerados a partir do preenchimento do formulário serão associados à modificação para consulta futura.

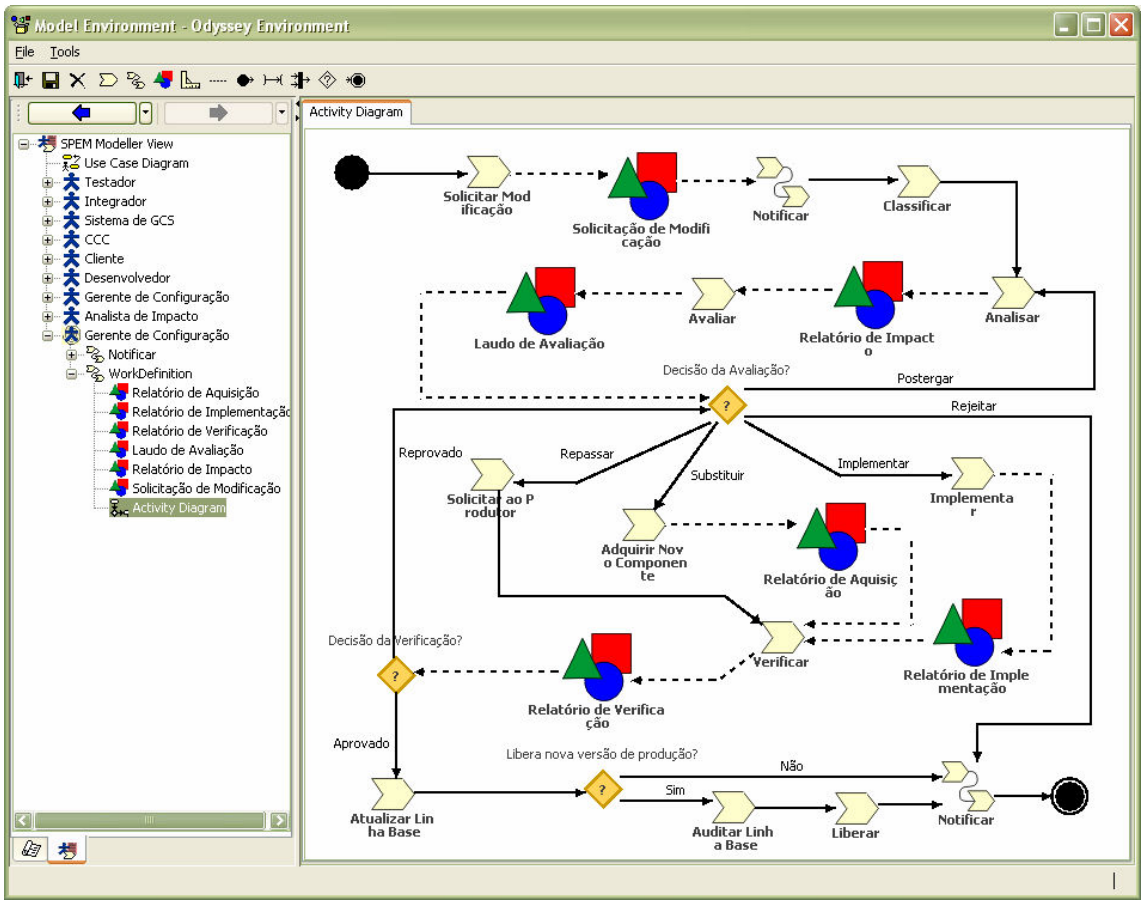


Figura 6.2: Processo Odyssey-SCMP modelado no Odyssey-CCS.

Esses formulários podem ser utilizados nas demais atividades do processo de GCS, com intuitos diferentes. Do ponto de vista de análise de impacto, eles podem, por exemplo, apoiar a análise de causa por meio de campos específicos para manutenções corretivas. Já do ponto de vista de aquisição de componentes, eles podem, por exemplo,

coletar informações referentes aos fornecedores, viabilizando o apoio ao desenvolvimento de software por meio de diversas camadas contratuais de produção e consumo de componentes.

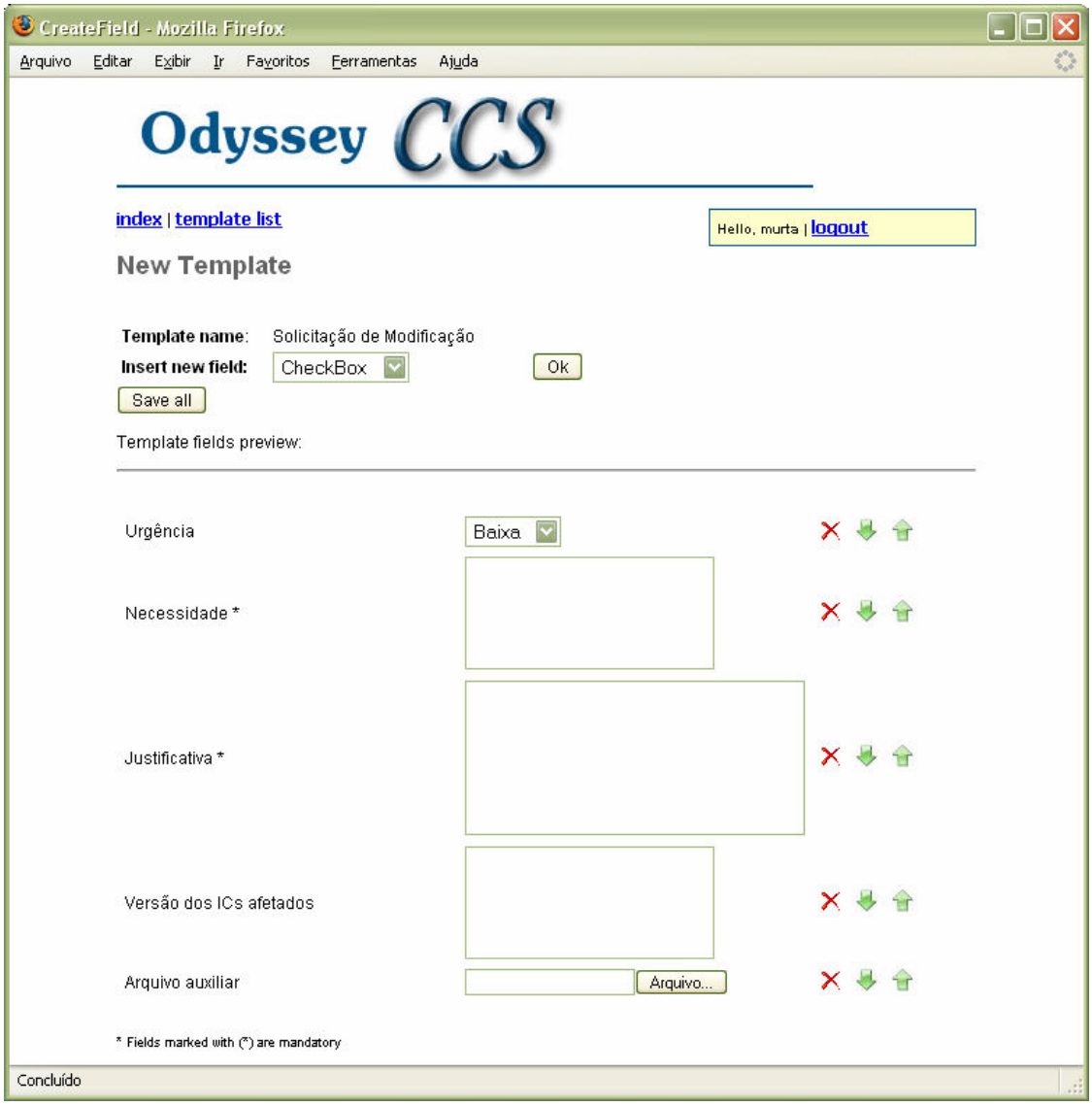


Figura 6.3: Formulário de solicitação de modificação modelado no Odyssey-CCS.

Com o intuito de viabilizar a **execução dos processos** de controle de modificações, foi utilizada a máquina de processos Charon (MURTA, 2002; MURTA *et al.*, 2002a; MURTA *et al.*, 2002b), adaptada para atender aos requisitos específicos do problema em questão. A versão original da Charon possibilitava a execução de processos modelados via uma notação estendida do diagrama de atividades da UML. Esta versão original foi adaptada para atuar sobre processos modelados de acordo com a especificação SPEM.

O processo modelado é transformado em fatos Prolog pela máquina de processos Charon, constituindo uma base de conhecimento. Sobre essa base de conhecimento,

agentes específicos para a simulação, execução e acompanhamento do processo são acionados. O agente de simulação verifica se o processo modelado é válido, ou seja, se o fluxo de execução de cada instância é capaz de atingir o final do processo. Caso o processo seja válido, novas instâncias podem ser criadas. Como discutido anteriormente, cada instância representa uma nova solicitação de modificação. Para uma dada instância do processo, o agente de execução verifica se alguma atividade ou decisão já foi finalizada e infere as próximas atividades ou decisões. Finalmente, o agente de acompanhamento informa as atividades e decisões pendentes e possibilita que o engenheiro de software defina quais atividades devem ser finalizadas ou selecione o fluxo desejado para uma dada decisão. A Figura 6.4 exibe o fluxo dessas atividades no contexto da máquina de processos Charon.

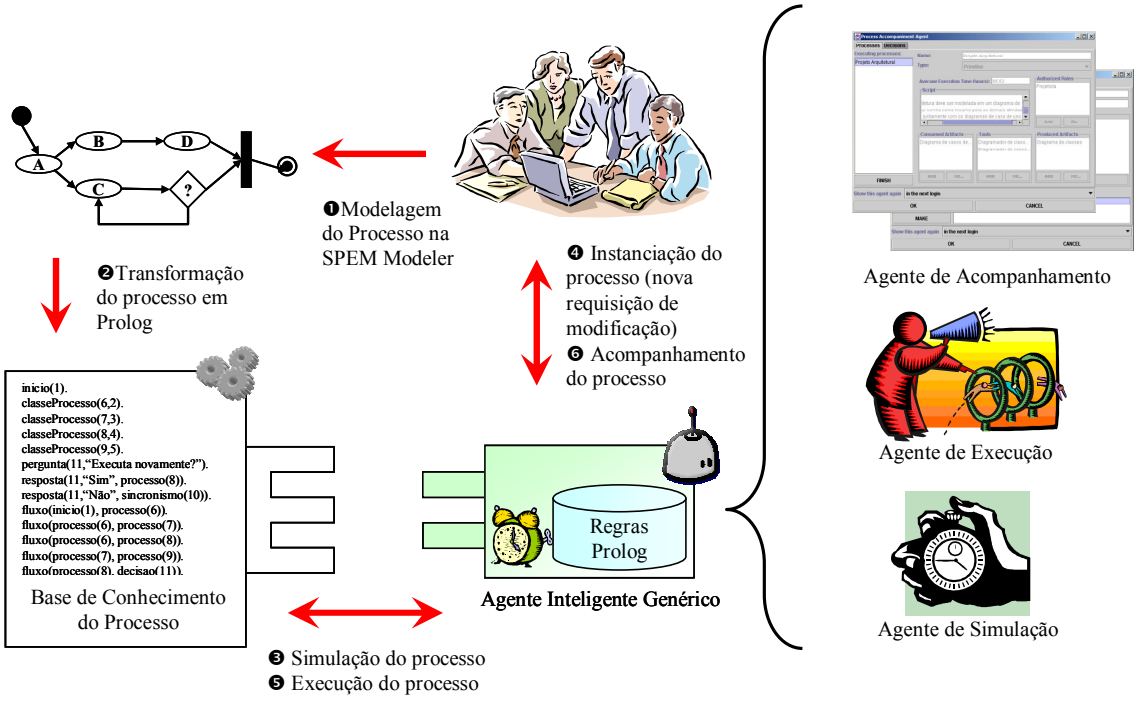


Figura 6.4: Visão geral da máquina de processos Charon.

Sempre que o agente de acompanhamento da Charon detecta atividades ou decisões pendentes para o engenheiro de software corrente, o Odyssey-CCS é informado. Desta forma, o Odyssey-CCS interage com o engenheiro de software fornecendo as informações necessárias para que as atividades e decisões pendentes sejam processadas, como exibido na Figura 6.5. É possível observar na Figura 6.5 que a modificação de número 1, descrita na Tabela 6.1, está na atividade de avaliação. De acordo com o processo Odyssey-SCMP, modelado na Figura 6.2, a avaliação deve

decidir dentre cinco opções, detalhadas na Seção 6.3.1. Em paralelo, a modificação de número 2 ainda está na atividade de classificação.

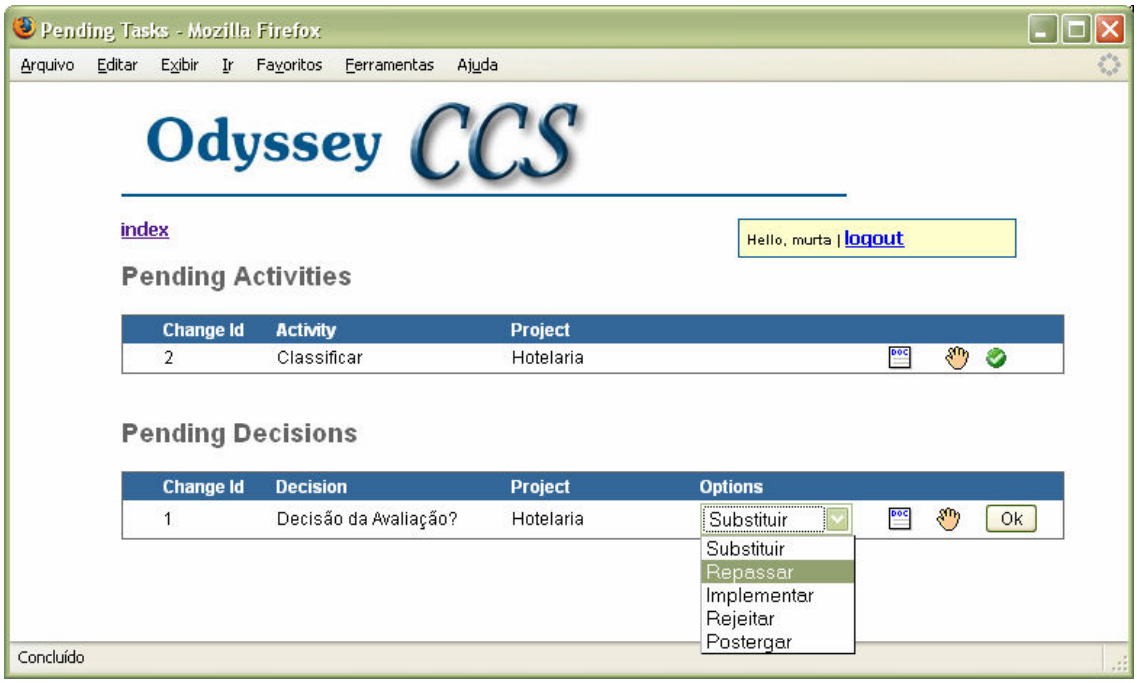


Figura 6.5: Atividades e decisões pendentes, sendo exibidas pelo Odyssey-CCS.

Além de prover os agentes de simulação, execução e acompanhamento, a máquina de processos Charon define um *framework* para a criação de novos agentes. Esses novos agentes são descritos por regras de inferência Prolog e conectados automaticamente às bases de conhecimento, sempre que necessário. Essas regras de inferência Prolog fazem uso de um meta-modelo genericamente definido para todas as bases de conhecimento da Charon, com o intuito de manipular o conhecimento existente em função de um objetivo específico.

Desta forma, a máquina de processos Charon permite que o Odyssey-CCS possa ser estendido para prover relatórios ou funcionalidades mais elaboradas, que direta ou indiretamente necessitam acesso às informações coletadas durante a execução dos processos. Por exemplo, novos agentes podem ser criados para a coleta de medidas via a análise das informações geradas pelo processo.

Finalmente, visando identificar a **responsabilidade de manutenção** dos componentes, o mapa de reutilização, descrito no Capítulo 5, foi implementado em uma biblioteca de componentes em construção na COPPE, denominada Brechó. Como discutido anteriormente, o principal propósito do mapa de reutilização é armazenar os produtores, consumidores e os contratos de reutilização firmados entre eles. Para atingir este objetivo, os produtores informam, durante o cadastramento do componente na

Brechó, sob quais licenças o componente pode ser reutilizado. Por outro lado, quando o consumidor decide reutilizar um dado componente, é necessário selecionar uma licença dentre as disponíveis, estabelecendo assim um contrato de reutilização.

Esse contrato de reutilização pode ser consultado bidirecionalmente. Ou seja, é possível identificar o produtor de um dado componente, condição necessária para a propagação de solicitações de modificação. Além disso, é possível identificar os vários consumidores de um componente, condição necessária para comunicação de novas funcionalidades ou correções de defeito.

Vale salientar que, na realidade, o contrato não é firmado para um componente, mas sim para uma versão específica de um componente. Desta forma, é possível saber exatamente quais consumidores utilizam quais versões de um dado componente e, indiretamente, em relação a qual versão uma dada solicitação de modificação se refere. A Figura 6.6 exibe uma consulta ao mapa de reutilização referente aos consumidores do componente "Hotel", descrito na Figura 6.1.

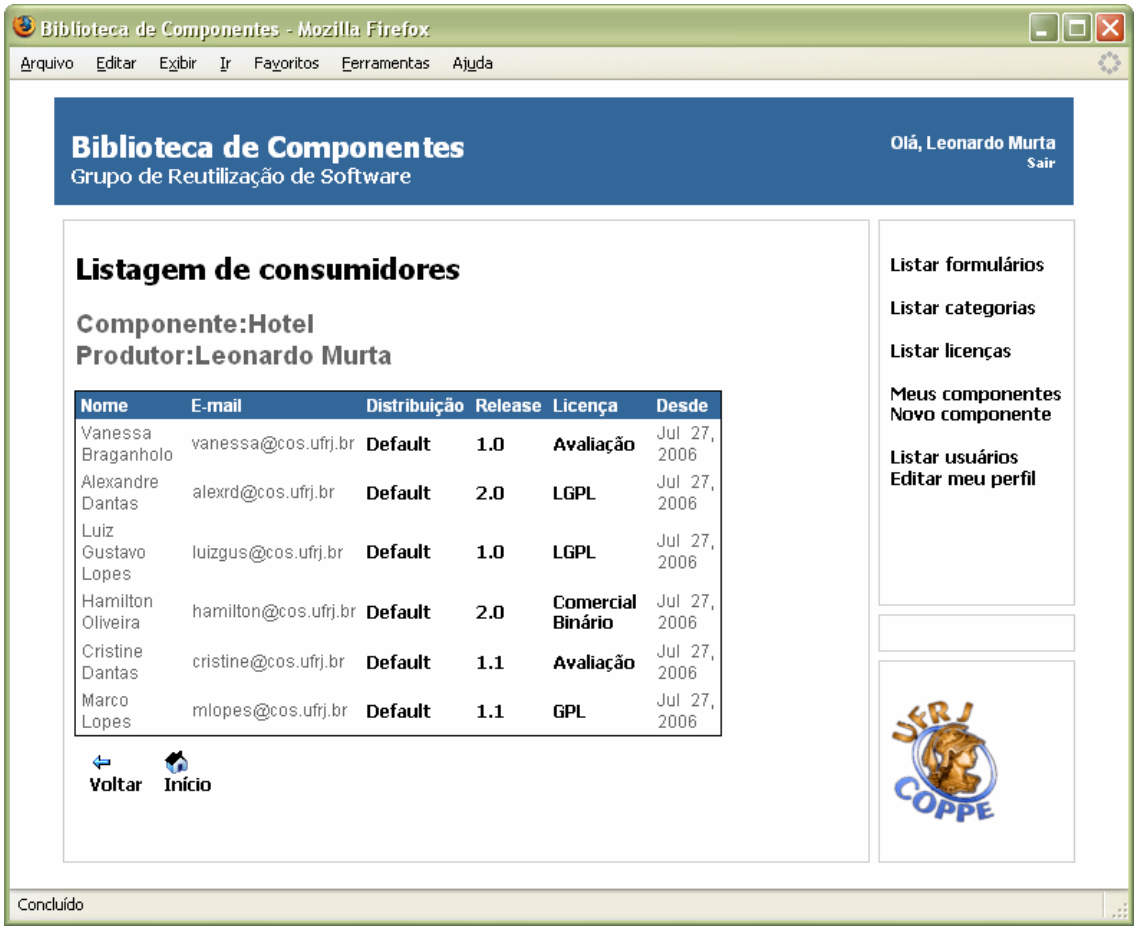


Figura 6.6: Mapa de reutilização sendo consultado pelo Odyssey-CCS.

Atualmente, novas características estão sendo adicionadas à Brechó, que complementam as necessidades discutidas anteriormente. Dentre elas, está o controle

sobre dependências entre versões de componentes e o empacotamento dinâmico de componentes, possibilitando tanto a visualização de impacto de modificações entre componentes quanto a customização das partes de um componente a serem entregues a um dado consumidor em função da licença escolhida.

6.4.2 Considerações finais sobre o Odyssey-CCS

Apesar da carência de pesquisas em controle de modificações aplicada ao DBC, existem alguns poucos trabalhos na fronteira dessas áreas. O processo MwR (KWON *et al.*, 1999), implementado na ferramenta TERRA, apesar de possibilitar o registro e consulta sobre solicitações de modificação e componentes, não leva em consideração as diferentes versões de um dado componente, tampouco leva em consideração as equipes consumidoras que reutilizaram esse componente. Como discutido anteriormente, não é viável a detecção da responsabilidade de manutenção sem esse tipo de informação. Esses problemas foram contornados pelo Odyssey-CCS com o uso do mapa de reutilização juntamente com a configuração flexível do processo e dos formulários de coleta de informações.

O KobrA (ATKINSON *et al.*, 2001) é um outro trabalho importante para a manutenção de sistemas baseados em componentes. Apesar das características positivas do KobrA, relacionadas com a composição de componentes e dependências entre modificações, como visto no Capítulo 3, ele não trata a detecção da responsabilidade de manutenção. Além disso, não foi possível encontrar implementações computacionais da abordagem.

Finalmente, as bibliotecas comerciais *Flashline Registry* (FLASHLINE, 2006), *Logidex* (LOGICLIBRARY, 2006) e *Select Component Manager* (SELECT BUSINESS SOLUTIONS, 2006) mantêm informações sobre consumidores de componentes e interdependências entre os componentes. Apesar dessas informações serem úteis na resolução do problema da cadeia de responsabilidade de manutenção, essas bibliotecas aparentam considerar somente um único tipo de licença para todos os componentes. Além disso, a informação de quem consome quais componentes, que é de grande valia para a equipe produtora, fica indisponível à mesma durante o processo de controle de modificações.

6.5 Odyssey-VCS

A abordagem Odyssey-VCS, apresentada no Capítulo 5, foi implementada em Java devido a grande oferta de bibliotecas para manipulação de modelos nesta linguagem. Essa implementação evitou dependências aos sistemas de controle de versões existentes, possibilitando a quebra da metáfora atual que mapeia ICs em arquivos do sistema operacional. O Odyssey-VCS foi construído sobre o repositório MOF MDR, reutilizando as funcionalidades básicas de persistência de objetos. O restante desta seção apresenta as decisões de projeto tomadas durante a implementação do Odyssey-VCS.

6.5.1 Detalhamento da Solução

A arquitetura do Odyssey-VCS é composta por três camadas principais: cliente, transporte e servidor. O principal elemento da **camada cliente**, apresentada na Figura 6.7.a, são as ferramentas CASE. Nesta camada, qualquer ferramenta CASE pode ser utilizada, desde que seja compatível com a notação UML e utilize XMI para importar e exportar os modelos gerados. A integração entre a ferramenta CASE adotada e o Odyssey-VCS pode ser feita via dois mecanismos distintos: *plug-in* e ferramenta cliente. O mecanismo de *plug-in*, utilizado para integrar o Odyssey com o Odyssey-VCS, permite que os comandos do Odyssey-VCS sejam executados diretamente da ferramenta CASE. Para que esse mecanismo fosse possível, foi definida uma API (*Application Programming Interface*) de importação e exportação de XMI no ambiente Odyssey, denominada Odyssey-XMI. Por outro lado, a ferramenta cliente, utilizada para integrar o Poseidon (GENTLEWARE, 2006) com o Odyssey-VCS, necessita que a ferramenta CASE exporte o modelo via XMI, e opera sobre esse XMI.

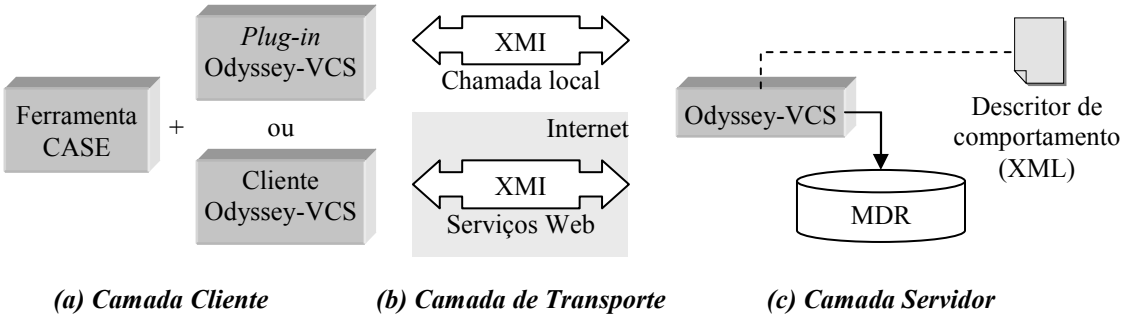


Figura 6.7: Visão geral do Odyssey-VCS.

A **camada de transporte**, apresentada na Figura 6.7.b, é responsável por permitir o desenvolvimento distribuído de modelos UML pela Internet. Esta camada

utiliza tanto chamadas locais quanto Serviços Web (BOOTH *et al.*, 2005) como protocolo de transporte. Antes de ser enviado pela Internet, o XMI passa por um módulo de compressão, que faz uso do algoritmo zlib (GAILLY e ADLER, 2006) para melhorar o desempenho da camada de transporte como um todo. Esse tipo de abordagem vem sendo adotada no lugar de transporte de diferenças (*deltas*) devido ao aumento do poder de processamento das máquinas atuais (ESTUBLIER *et al.*, 2005).

Finalmente, a **camada servidor**, apresentada na Figura 6.7.c, se comporta de forma diferenciada em função das necessidades específicas de cada projeto. O descritor de comportamento permite a definição de quais elementos serão considerados unidades de comparação e de versionamento para um dado projeto. Desta forma, o XMI recebido via comando de *check-in* é transformado em objetos e esses objetos são combinados com os objetos existentes no repositório via um procedimento de junção (*merge*), sempre levando em consideração as unidades de comparação e versionamento. O resultado desse procedimento de junção é armazenado no MDR para consultas futuras.

Vale ressaltar que o Odyssey-VCS não versiona o arquivo XMI diretamente. Esse arquivo, que serve somente como meio de transporte, é convertido em uma representação orientada a objetos pelo MDR e os objetos são comparados e versionados de forma individual. Essa representação orientada a objetos segue a especificação UML e é acessada pelo Odyssey-VCS via JMI, que possibilita a geração de APIs a partir de meta-modelos MOF e a utilização dessas APIs para acessar os elementos persistidos em repositórios MOF. A implementação atual do Odyssey-VCS persiste integralmente cada versão do modelo UML. Além disso, o uso do MDR garante ao Odyssey-VCS consistência estrutural dos modelos UML. Entretanto, não são aplicadas as regras de boa formação definidas pela especificação da UML, ficando a cargo de ferramentas externas.

Para identificar os elementos que devem ser considerados unidade de comparação e de versionamento, o descritor de comportamento faz uso dos próprios nomes das interfaces JMI que representam elementos UML. A Figura 6.8 exibe o descritor de comportamento utilizado no exemplo de hotelaria. Nesse descritor, elementos do tipo classe (*Class*) são considerados unidades de comparação (<UC>true</UC>). Isso significa que um conflito será detectado caso dois ou mais engenheiros de software trabalhem de forma concomitante sobre uma mesma classe. Esse conflito ocorrerá mesmo se os engenheiros de software estiverem atuando sobre partes diferentes da classe.

```
<?xml version="1.0" ?>
<behaviors>
  <type name="org.omg.uml.modelmanagement.Model">
    <UV>true</UV>
    <UC>false</UC>
  </type>
  <type name="org.omg.uml.modelmanagement.UmlPackage">
    <UV>true</UV>
    <UC>false</UC>
  </type>
  <type name="org.omg.uml.foundation.core.UmlClass">
    <UV>true</UV>
    <UC>true</UC>
  </type>
  <type name="org.omg.uml.foundation.core.Operation">
    <UV>true</UV>
    <UC>true</UC>
  </type>
  <type name="org.omg.uml.foundation.core.Attribute">
    <UV>true</UV>
    <UC>true</UC>
  </type>
  <type name="org.omg.uml.behavioralelements.usecases.UseCase">
    <UV>true</UV>
    <UC>true</UC>
  </type>
  <type name="org.omg.uml.behavioralelements.usecases.Actor">
    <UV>false</UV>
    <UC>false</UC>
  </type>
</behaviors>
```

Figura 6.8: Exemplo de configuração do comportamento do Odyssey-VCS em termos de unidades de comparação e versionamento.

Além disso, uma classe é composta de atributos (*Attribute*) e operações (*Operation*), e ambos são considerados unidades de versionamento (<UV>true</UV>). Ou seja, sempre que algum engenheiro de software modificar um atributo ou uma operação, o Odyssey-VCS criará uma nova versão do elemento, armazenando informações complementares como, por exemplo, o nome do engenheiro de software, o momento que a modificação ocorreu e a razão informada durante o *check-in*. Essas informações podem ser consultadas no futuro, respondendo a perguntas do tipo: "Quais assinaturas o método 'pagaConta' teve no passado?", "Quem modificou o atributo 'preço' nos últimos 6 meses?" ou "Por que o atributo 'preço' teve o seu tipo alterado de *Float* para *Currency*?". Por outro lado, o elemento ator (*Actor*) não está configurado como unidade de versionamento (<UV>false</UV>). Isso significa que nenhuma informação de versionamento será armazenada em relação a esse elemento. Desta forma, um ator só poderá ser acessado como conseqüência do acesso a algum outro

elemento que seja unidade de versionamento e referencie esse ator (um pacote, por exemplo).

É importante notar que, como cada projeto define seu descritor próprio, é possível ter comportamentos diferentes em projetos distintos. Por exemplo, se um projeto não definir classe como unidade de comparação, nenhum conflito será detectado caso duas ou mais pessoas trabalhem concomitantemente sobre partes distintas de uma classe nesse projeto.

Para possibilitar a combinação de modificações efetuadas em paralelo, foi definido e implementado um algoritmo de junção no Odyssey-VCS. Esse algoritmo, que é inspirado em algoritmos clássicos de junção (CONRADI e WESTFECHTEL, 1998), leva em consideração as configurações descritas na Seção 5.4.2 do Capítulo 5, que são:

- Configuração Original (O): Contém, no repositório, as versões dos elementos de modelo no momento do *check-out*;
- Configuração Local (L): Contém, no espaço de trabalho, as versões dos elementos de modelo no momento do *check-in*;
- Configuração Atual (A): Contém, no repositório, as versões dos elementos de modelo no momento do *check-in*; e
- Configuração Final (F): Contém, no repositório, as versões dos elementos de modelo após o *check-in* (junção da Configuração Local com a Configuração Atual, utilizando a Configuração Original como pivô).

A ação do algoritmo de junção, descrito na Tabela 6.3, é obtida após analisar a presença/ausência de elementos de modelo nessas configurações e o valor interno desses elementos de modelo durante os *check-ins*. No início do processamento do algoritmo de junção, a Configuração Final está vazia. Durante o processamento, os elementos de modelo podem ser adicionados à Configuração Final, caso pertinente. As relações entre configurações, utilizadas na Tabela 6.3, são definidas da seguinte forma:

- e_X: Elemento "e" da configuração "X"; e
- $e_X \equiv e_Y$: Verdadeiro caso o elemento "e" seja similar nas configurações "X" e "Y". Vale ressaltar que a computação de similaridade leva em consideração somente o valor interno dos elementos de modelo. A estrutura de composição é analisada dinamicamente de acordo com a configuração da unidade de comparação no descritor de comportamento.

Tabela 6.3: Algoritmo de junção do Odyssey-VCS.

Caso	$e \in O$	$e \in A$	$e \in L$	$e_O \equiv e_A$	$e_O \equiv e_L$	Ação
1	V	V	V	V	V	Adiciona e_A (ou e_L) em F
2	V	V	V	V	F	Adiciona e_L em F
3	V	V	V	F	V	Adiciona e_A em F
4	V	V	V	F	F	Notifica um conflito: "modificações concorrentes sobre o mesmo elemento"
5	V	V	F	V	N/A	Nenhuma (não adiciona "e" em F)
6	V	V	F	F	N/A	Notifica um conflito: "modificação e remoção concorrentes sobre o mesmo elemento"
7	V	F	V	N/A	V	Nenhuma (não adiciona "e" em F)
8	V	F	V	N/A	F	Notifica um conflito: "modificação e remoção concorrentes sobre o mesmo elemento"
9	V	F	F	N/A	N/A	Nenhuma (não adiciona "e" em F)
10	F	V	V	N/A	N/A	N/A
11	F	V	F	N/A	N/A	Adiciona e_A em F
12	F	F	V	N/A	N/A	Adiciona e_L em F
13	F	F	F	N/A	N/A	N/A

O Caso 3 da Tabela 6.3 mostra um cenário onde um dado elemento (e.g.: caso de uso) existe em todas as configurações ($e \in O$, $e \in A$, $e \in L$), foi modificado na Configuração Atual ($e_O \not\equiv e_A$), mas não foi tocado na Configuração Local ($e_O \equiv e_L$). Neste caso, o Odyssey-VCS promove o elemento da Configuração Atual para a Configuração Final. Por outro lado, o Caso 6 mostra um cenário onde um dado elemento (e.g.: operação) foi removido da Configuração Local ($e \notin L$), mas existe em todas as outras configurações ($e \in O$, $e \in A$). Contudo, o elemento foi modificado por outros engenheiros de software em paralelo ($e_O \not\equiv e_C$). Como resultado desse cenário, o Odyssey-VCS notifica um conflito, justificando que o mesmo elemento foi modificado e removido concomitantemente por diferentes engenheiros de software. Finalmente, o Caso 10 mostra uma situação impossível de acontecer. Nessa situação, um elemento que não existia na Configuração Original ($e \notin O$) é criado concomitantemente na Configuração Local ($e \in L$) e na Configuração Atual ($e \in A$). Entretanto, elementos não são identificados pelo seu conteúdo, mas sim por um identificador MOF fictício. Desta forma, mesmo que elementos criados concomitantemente tenham o mesmo conteúdo, não serão considerados similares pelo Odyssey-VCS.

Por exemplo, durante a implementação da modificação número 1, descrita na Tabela 6.1, dois engenheiros de software denominados João e Maria, utilizando respectivamente as ferramentas CASE Poseidon e Odyssey, modificaram concorrentemente a versão inicial do modelo de hotelaria. João desejava trocar o nome do atributo "email", pertencente à classe "Hóspede", para "telefone", adicionar o atributo "sexo" na classe "Cliente" e adicionar um novo caso de uso denominado

"Modificar Reserva". Por outro lado, Maria desejava trocar o tipo do atributo "preço" da classe "Tipo de Quarto" de "Float" para "Currency" e incluir duas novas operações na classe "Hóspede": "getEmail() : String" e "setEmail(email : String) : void".

As modificações feitas pelo João, mostradas na Figura 6.9, foram as primeiras a serem enviadas ao repositório. Nenhum conflito foi detectado e o *check-in* foi incorporado com sucesso na Configuração Atual do repositório (Caso 2 da Tabela 6.3). Após esse momento, a versão mais atual da classe "Hóspede", residente no repositório, não tem mais o atributo "email". Contudo, a versão da classe "Hóspede" residente no espaço de trabalho de Maria está desatualizada, e ainda contém o atributo "email", como mostrado na Figura 6.10.

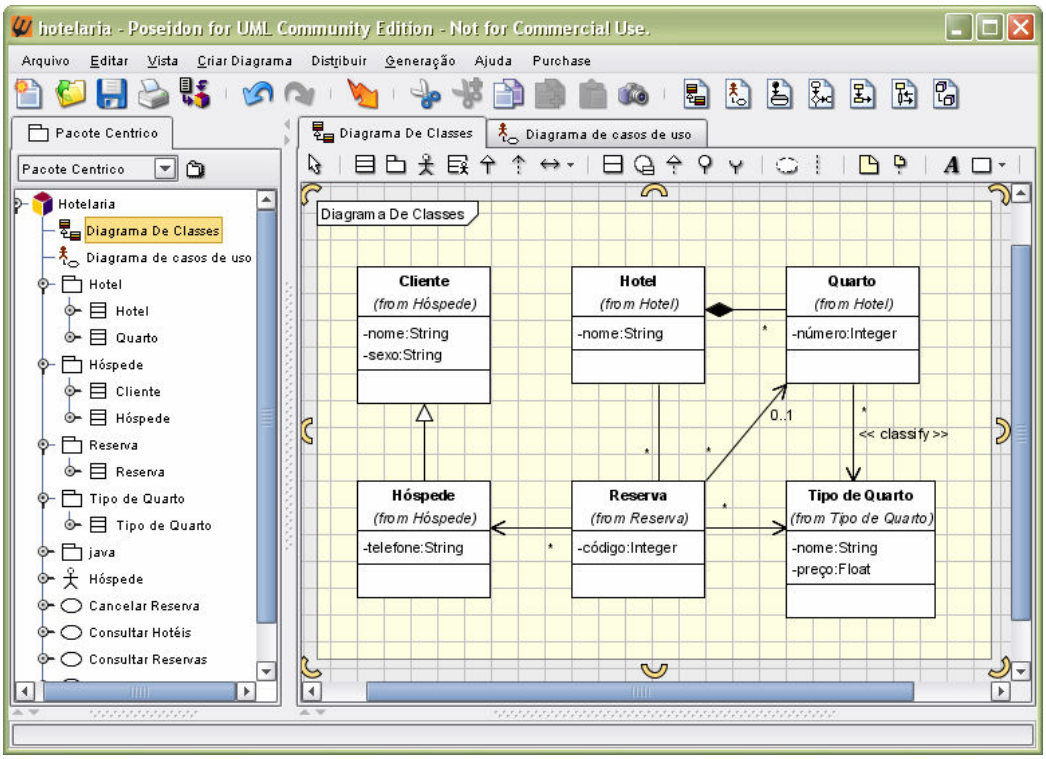


Figura 6.9: Modificações feitas pelo João no Poseidon.

Quando Maria tenta executar *check-in*, todos os elementos de modelo são corretamente combinados com as versões atuais no repositório (casos 3 e 12 da Tabela 6.3), com exceção da classe "Hóspede". Apesar de os subelementos da classe "Hóspede" terem sido processados corretamente, a classe "Hóspede" ficaria semanticamente incorreta caso a junção ocorresse com sucesso, pois as operações "getEmail() : String" e "setEmail(email : String) : void" foram criadas com o intuito de manipular o atributo "email", que foi renomeado para "telefone" por João. Felizmente, o tipo classe (*Class*) foi definido como unidade de comparação (Figura 6.8). Por esse motivo, o *check-in* de Maria lança um conflito, como mostrado na Figura 6.11.

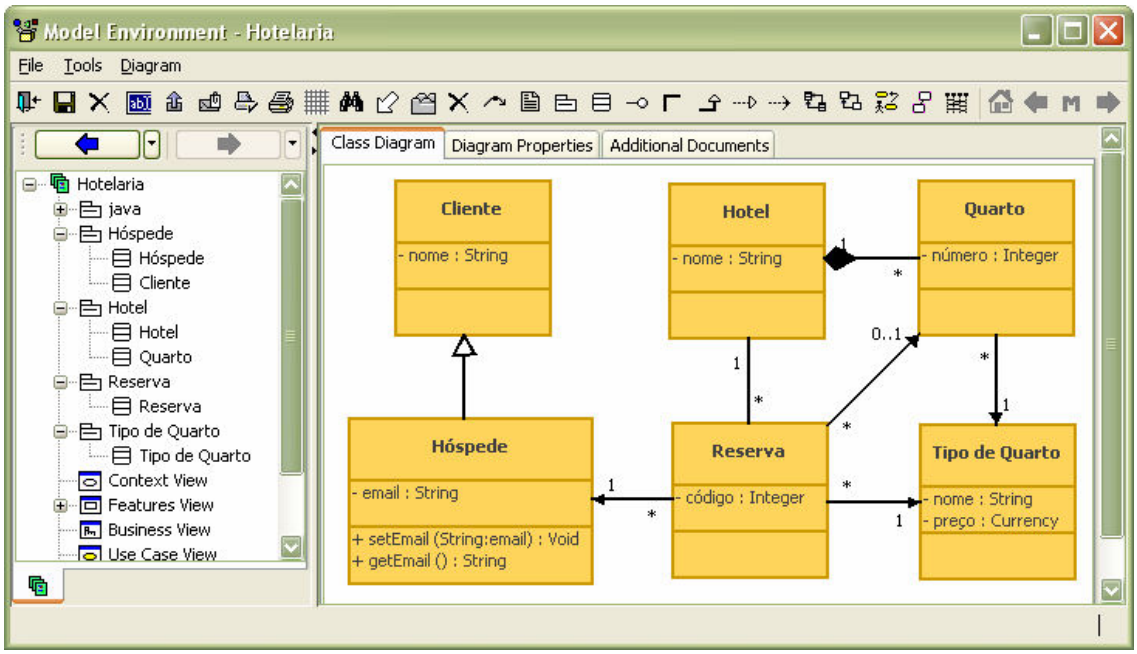


Figura 6.10: Modificações feitas pela Maria no Odyssey.

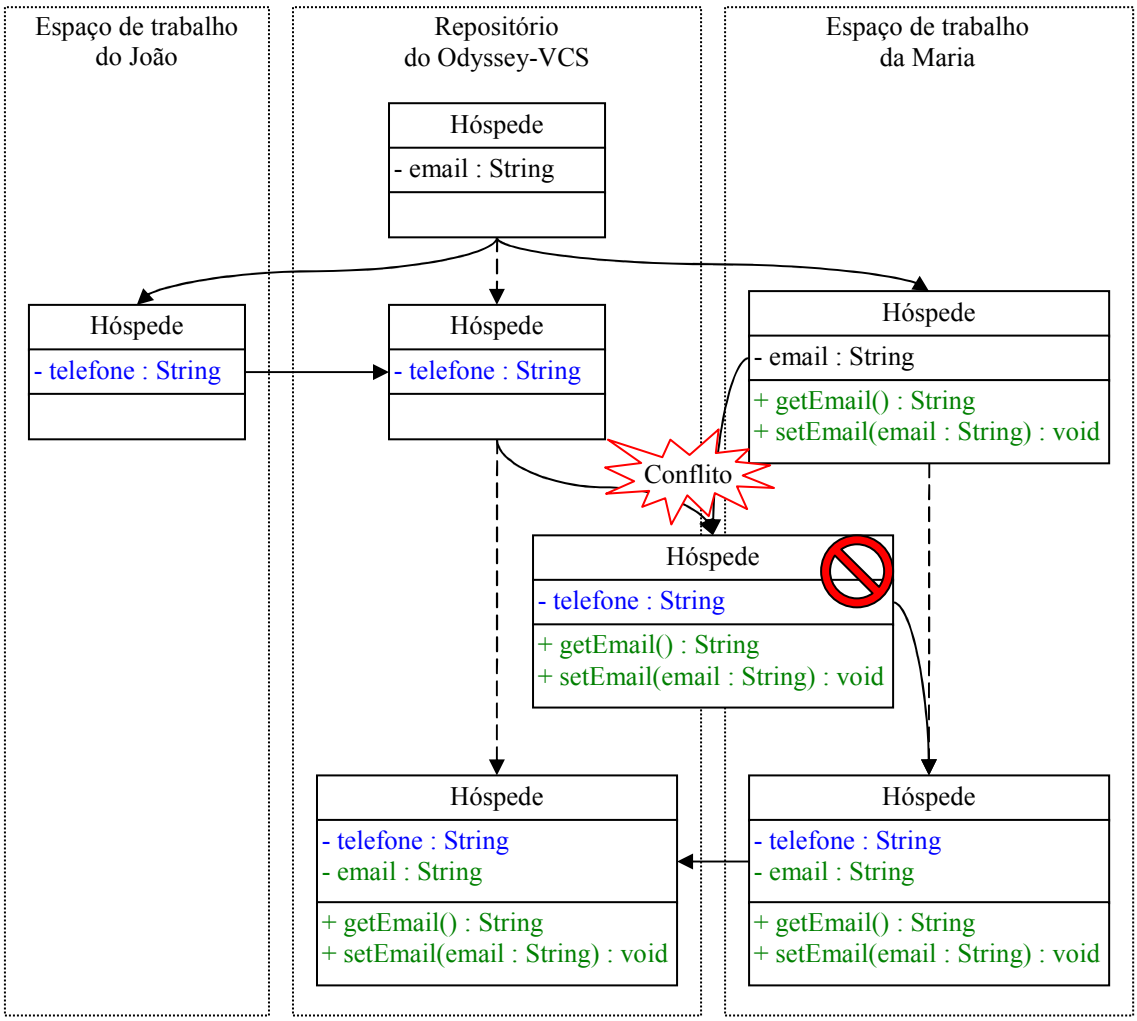


Figura 6.11: Cenário de detecção de conflito durante junção.

O Odyssey-VCS fornece toda a informação necessária para que Maria resolva o conflito, fazendo uso de ferramentas externas. Após resolver o conflito, Maria finalmente consegue enviar a sua contribuição para o repositório. A Figura 6.12 mostra o estado final do repositório após os *check-ins* de João e de Maria.

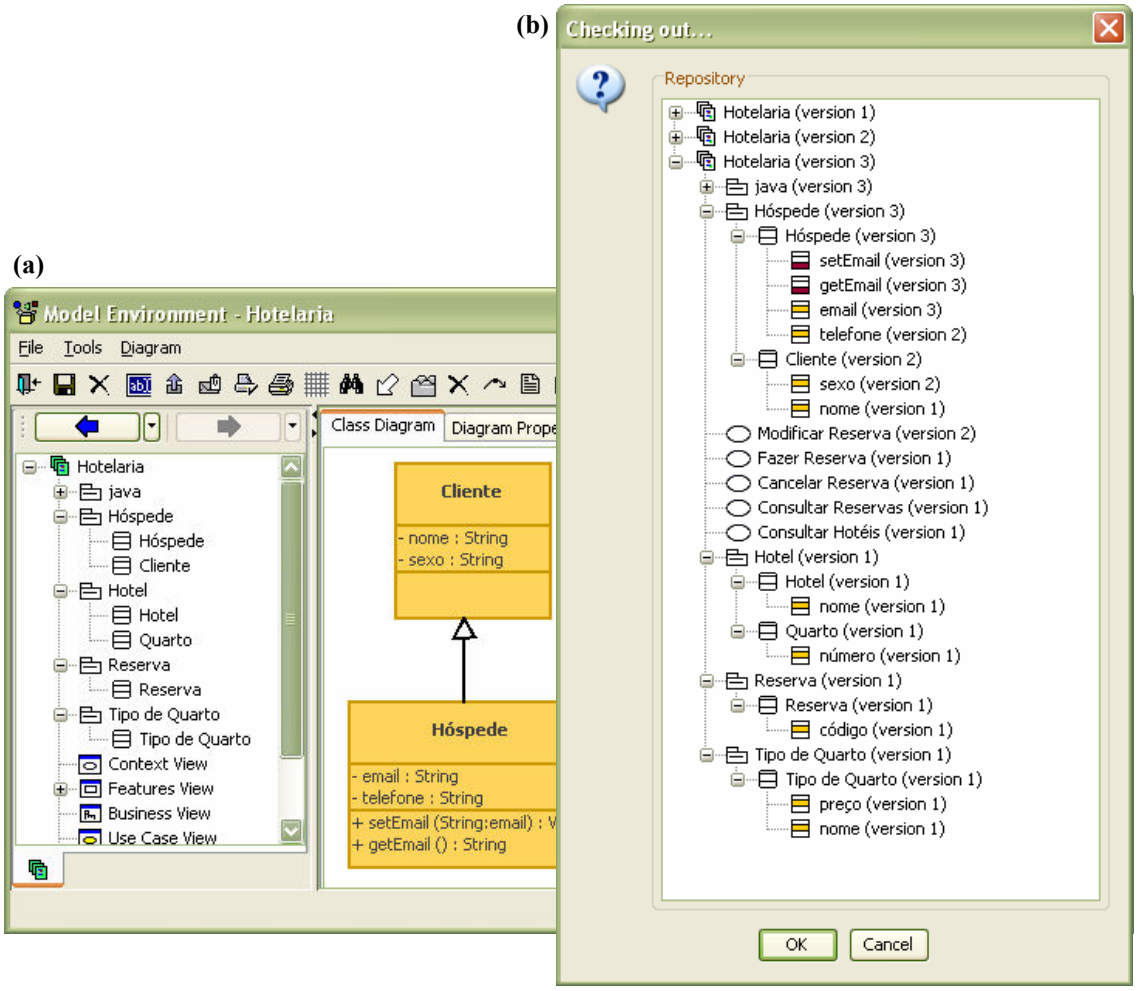


Figura 6.12: Odyssey (a) acessando o repositório via *plug-in* do Odyssey-VCS (b).

6.5.2 Considerações finais sobre o Odyssey-VCS

A maioria dos sistemas de controle de versão comerciais e livres atua sobre arquivos (WHITE, 2000; FOGEL e BAR, 2001; ROCHE e WHIPPLE, 2001; COLLINS-SUSSMAN *et al.*, 2004). Como discutido anteriormente, esses sistemas têm diversas limitações para manipular artefatos descritos via modelos de dados complexos. Contudo, os sistemas de controle de versão convencionais são estáveis e amplamente utilizados para controlar a evolução de código fonte.

Algumas ferramentas CASE, como, por exemplo, Enterprise Architect (SPARX SYSTEMS, 2006) e Poseidon (GENTLEWARE, 2006), fazem uso desses sistemas para controlar a evolução de modelos UML, estando sujeitas aos problemas descritos no

Capítulo 5. O Odyssey-VCS pode ser visto como um sistema complementar, que possibilita o controle de versão sobre modelos UML em granularidade fina, enquanto os sistemas tradicionais tratam os demais artefatos manipulados no projeto.

Existem algumas abordagens que fazem uso de outros modelos de dados, como, por exemplo, entidade-relacionamento ou orientado a objetos. Contudo, essas abordagens atuam sobre código fonte de linguagens de programação específicas. Por exemplo, GOLDSTEIN et al. (1984), HABERMANN et al. (1986) e RENDER et al. (1991) apóiam respectivamente o versionamento de código fonte Smalltalk, C e Pascal. O Odyssey-VCS também pode ser visto como complementar a essas abordagens, pois atua sobre elementos de modelo UML.

Algumas poucas abordagens fazem uso de modelos de dados mais complexos para controlar versões de modelos de análise e projeto. Por exemplo, OHST et al. (2002) fazem uso de árvores sintáticas armazenadas em arquivos XML para detectar modificações estruturais em granularidade fina de modelos UML. Contudo, o mero uso de XML não garante aderência aos padrões da UML. O esquema utilizado por eles não segue a especificação XMI, levando a incompatibilidades com as ferramentas CASE existentes. Por outro lado, NGUYEN et al. (2004) utilizam um sistema de versionamento hipermídia para controlar a evolução de artefatos UML de análise e projeto. Esse trabalho tem um grande foco no versionamento dos relacionamentos entre os elementos de modelo. Contudo, também utiliza um meta-modelo UML proprietário, reduzindo a compatibilidade com as ferramentas CASE existentes.

Finalmente, a OMG está trabalhando numa especificação para versionamento MOF (OMG, 2005a). Mesmo ainda não tendo sido lançada uma versão final dessa nova especificação, é possível notar alguma semelhança entre o Odyssey-VCS e a filosofia que guia a confecção dessa especificação. Ambos fazem uso de um meta-modelo à parte para armazenar informações sobre o versionamento e guardam as versões em raízes de persistência separadas, com informações históricas associadas.

6.6 Odyssey-BRD

A abordagem Odyssey-BRD, apresentada no Capítulo 5, foi implementada em Java. Essa implementação fez uso de sistemas existentes para viabilizar o estabelecimento de ligações de rastreabilidade entre diferentes representações de um mesmo software. No nível arquitetural, foi utilizada a linguagem de descrição de arquiteturas xADL (DASHOFY *et al.*, 2001), manipulada pela infra-estrutura

ArchStudio (DASHOFY *et al.*, 2002). No nível de código fonte, foi utilizado o sistema de controle de versões Subversion (COLLINS-SUSSMAN *et al.*, 2004). O restante desta seção apresenta as decisões de projeto tomadas durante a implementação do Odyssey-BRD.

6.6.1 Detalhamento da solução

De forma semelhante à Seção 5.5.2 do Capítulo 5, esta seção é decomposta em duas subseções, que discutem os aspectos de projeto detalhado, implementação e exemplificação do ArchTrace e da Carga Dinâmica, que são as duas principais vertentes do Odyssey-BRD.

6.6.1.1 ArchTrace

A arquitetura do ArchTrace, apresentada na Figura 6.13, é composta de 6 componentes principais. Desses, 4 são fixos e 2 são customizáveis. Os componentes customizáveis ("Conector de Arquiteturas" e "Conector de Repositórios") representam a linguagem de descrição de arquiteturas e o repositório de gerência de configuração utilizados. Atualmente, a linguagem de descrição de arquiteturas utilizada é a xADL e o repositório de gerência de configuração utilizado é o Subversion. Contudo, tanto a linguagem de descrição de arquitetura quanto o repositório de gerência de configuração podem ser substituídos devido a essa camada de abstração adotada pelo ArchTrace. Além de abstrair implementações específicas, esses componentes enviam eventos para o componente "Escutador de Eventos" sempre que a arquitetura ou sua implementação evoluem.

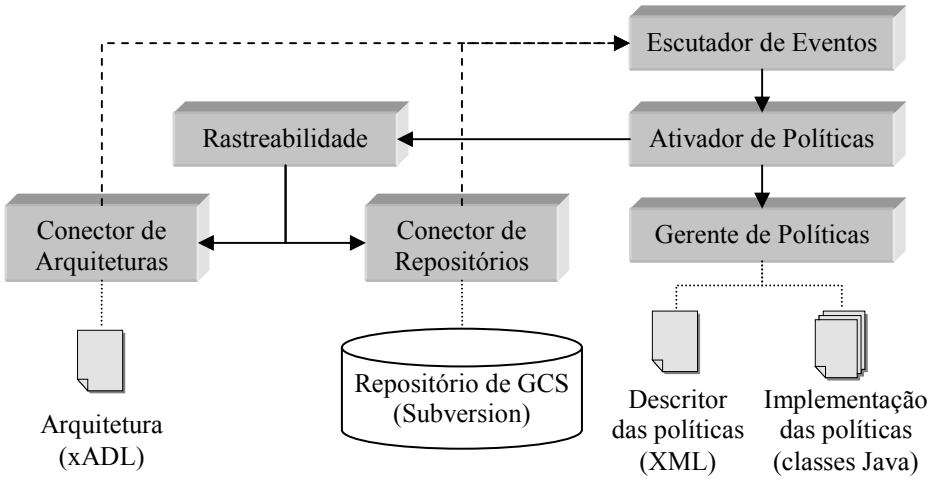


Figura 6.13: Visão geral do ArchTrace.

Para cada evento enviado, o componente “Escutador de Eventos” processa o evento e repassa para o componente “Ativador de Políticas”, que é responsável por calcular as políticas que devem ser ativadas para tratar o evento, assim como a ordem em que essas políticas devem ser ativadas. Sempre que uma política necessita modificar as ligações de rastreabilidade existentes, o componente de “Rastreabilidade” é acionado. Finalmente, o componente “Gerente de Políticas” é responsável por identificar as políticas existentes, descritas via arquivo XML, e carregar essas políticas no ArchTrace, colocando-as disponíveis para o componente “Ativador de Políticas”.

A principal razão de utilizar xADL em detrimento de outras linguagens de descrição de arquiteturas é a sua estrutura extensível via adição de esquemas XML externos. O ArchTrace estendeu a xADL com um esquema específico para armazenar ligações de rastreabilidade entre elementos arquiteturais e elementos de implementação, persistidos em arquivos. Esse esquema xADL definido pelo ArchTrace se baseia em um esquema existente, denominado “Implementação”, que permite anexar aos elementos arquiteturais informações relacionadas com implementação. Desta forma, o esquema definido pelo ArchTrace possibilita associar qualquer elemento arquitetural da xADL com código fonte ou demais artefatos de implementação persistidos em repositórios de gerência de configuração, como apresentado na Figura 6.14.

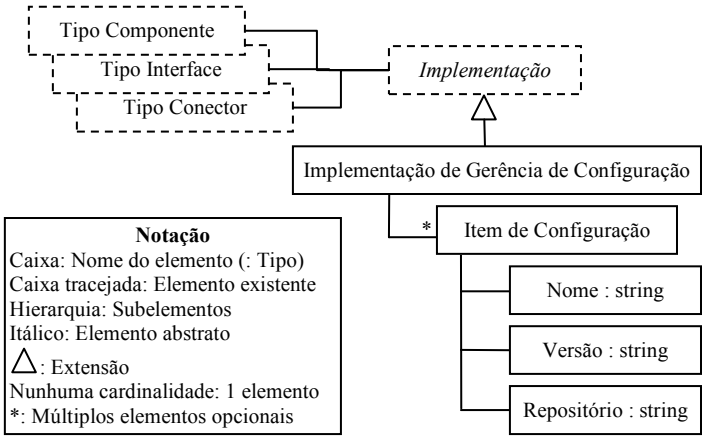


Figura 6.14: Esquema xADL definido pelo ArchTrace.

O ArchTrace faz uso de uma API específica para viabilizar a definição de políticas utilizando a linguagem Java. Essa API, apresentada na Figura 6.15, contempla os quatro tipos de política descritos na Seção 5.5.2.1 do Capítulo 5, que são: políticas de evolução de elemento arquitetural, políticas de evolução da implementação, políticas de pré-rastreabilidade e políticas de pós-rastreabilidade.

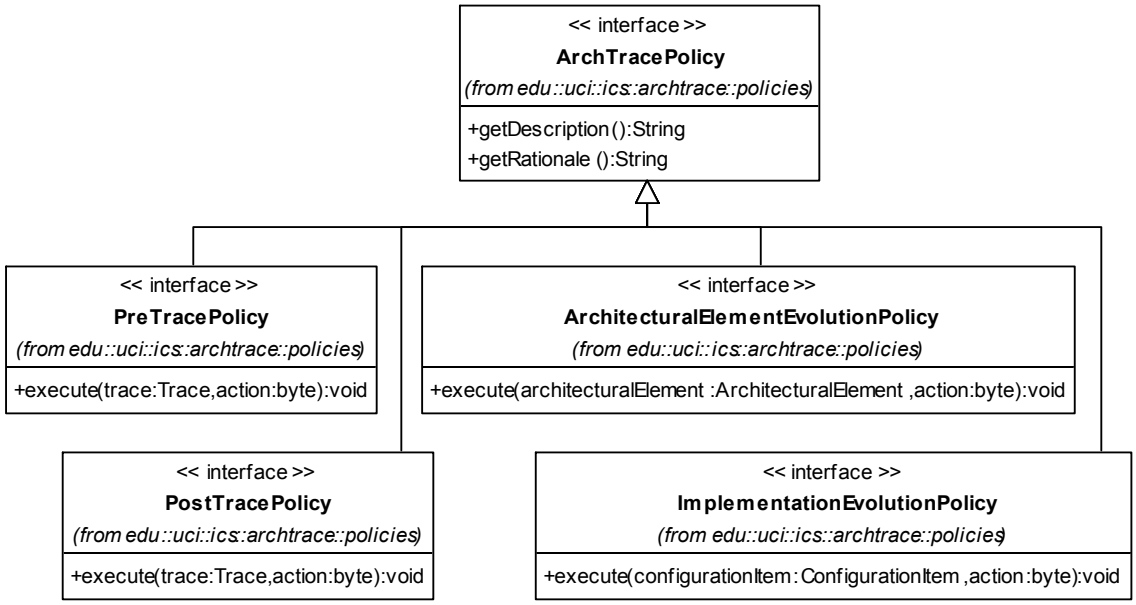


Figura 6.15: API para definição de políticas no ArchTrace.

A versão atual do ArchTrace conta com 9 políticas já implementadas, descritas na Tabela 6.4. As quatro primeiras políticas visam monitorar as ações dos engenheiros de software e das demais políticas para evitar que o sistema como um todo entre em estado inconsistente. As três políticas seguintes visam, a partir da ação do engenheiro de software ou de outras políticas, executar ações complementares, que apóiam no estabelecimento e evolução das ligações de rastreabilidade. Finalmente, as duas últimas políticas respondem diretamente aos eventos de evolução da arquitetura ou de sua implementação. Esse conjunto inicial de políticas pode ser estendido para contemplar outras técnicas de rastreabilidade.

Antes de colocar o ArchTrace em execução, algumas configurações iniciais devem ser feitas. Em primeiro lugar, a arquitetura e o repositório de gerência de configuração devem ser definidos. Além disso, devem ser configuradas as políticas que serão utilizadas para o projeto em questão. A Figura 6.16 mostra a configuração das políticas para o exemplo de hotelaria. Nesta configuração, é possível notar que somente a política 3 está desabilitada. Isso ocorre, pois as políticas 3 e 5 são formas distintas de tratar o mesmo problema, que devem ser selecionadas em função das necessidades específicas de cada projeto. Enquanto a política 3 evita o problema, impedindo a criação da ligação de rastreabilidade, a política 5 contorna o problema, modificando as ligações de rastreabilidade existentes.

Tabela 6.4: Políticas implementadas no ArchTrace.

#	Tipo	Descrição	Raciocínio
1	Restrição interativa (*pre-rastr.*)	Sugere ligações de rastreabilidade para a versão mais recente de um IC caso ligações de rastreabilidade tenham sido estabelecidas para versões anteriores.	Quando diferentes versões de um mesmo IC têm diferentes nomes ou estão em diferentes diretórios, uma ligação de rastreabilidade equivocada pode ser estabelecida.
2	Restrição automática (*pre-rastr.*).	Não permite a criação ou remoção de ligações de rastreabilidade em elementos arquiteturais imutáveis.	Não é desejável evoluir elementos imutáveis (liberados para produção) sem que uma nova versão (de desenvolvimento) seja criada.
3	Restrição automática (*pre-rastr.*)	Não permite a criação de ligações de rastreabilidade para mais de uma versão de um mesmo IC.	Algumas linguagens de programação não permitem mais de uma versão do mesmo IC no mesmo ambiente de execução.
4	Restrição automática (*pre-rastr.*)	Não permite a criação de ligações de rastreabilidade para subitem se já existe para o IC composto.	É redundante estabelecer ligações de rastreabilidade para partes se já está estabelecido para o todo.
5	Regra automática (*pos-rastr.*)	Remove ligações de rastreabilidade de versões anteriores de um IC se outras ligações de rastreabilidade forem estabelecidas para versões mais recentes.	Algumas linguagens de programação não permitem mais de uma versão do mesmo IC no mesmo ambiente de execução.
6	Regra automática (*pos-rastr.*)	Remove ligações de rastreabilidade de subitens se ligações de rastreabilidade forem estabelecidas para o IC composto.	É redundante estabelecer ligações de rastreabilidade para partes se já está estabelecido para o todo.
7	Regra interativa (*pos-rastr.*)	Sugere ligações de rastreabilidade relacionadas quando uma ligação de rastreabilidade é criada.	Algumas ligações de rastreabilidade futuras podem ser inferidas a partir das já estabelecidas via mineração de dados.
8	Regra automática (*evol. arq.*)	Copia todas as ligações de rastreabilidade existentes para a nova versão do elemento arquitetural.	A nova versão de um elemento arquitetural pode herdar as ligações de rastreabilidade da versão anterior.
9	Regra automática (*evol. impl.*)	Atualiza automaticamente as ligações de rastreabilidade quando novas versões de um IC estão disponíveis.	As ligações de rastreabilidade devem apontar para as versões mais recentes do ICs, sempre que possível.

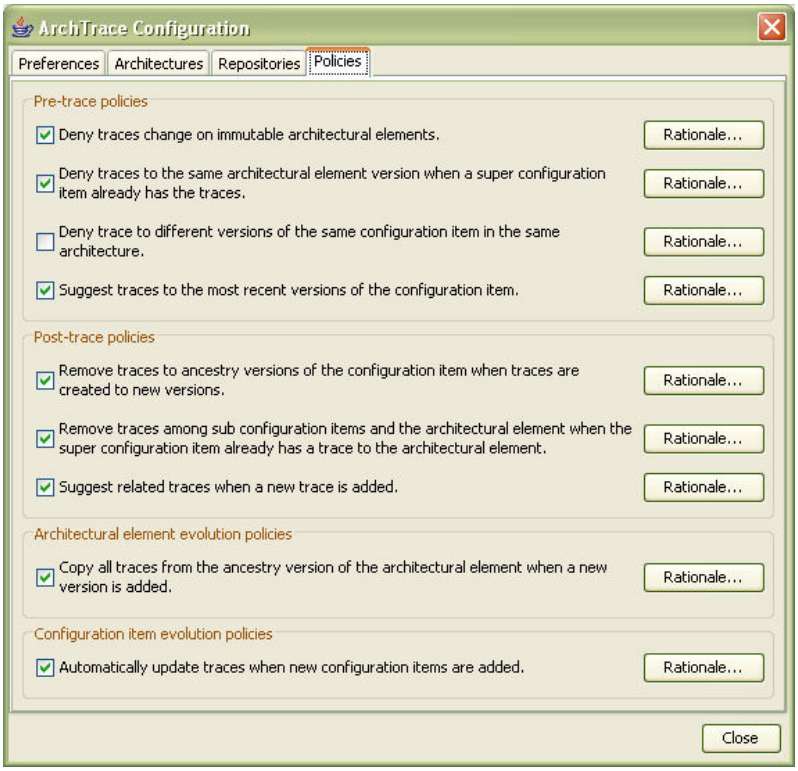


Figura 6.16: Configuração do ArchTrace.

A partir desse momento, a interface gráfica do ArchTrace deve ser utilizada para estabelecer as ligações de rastreabilidade iniciais. Esse passo pode ser feito de forma manual ou utilizando outras abordagens descritas na literatura, discutidas na Seção 5.5.1 do Capítulo 5. Finalmente, sempre que alguma modificação for feita na arquitetura ou no código fonte, o ArchTrace aplicará as políticas para atualizar as ligações de rastreabilidade.

No exemplo de hotelaria, o mapeamento inicial entre componentes e classes Java foi feito de acordo com a Figura 6.1.c. Com o passar do tempo, foi necessário reestruturar o projeto, inserindo alguns prefixos nos nomes das classes e movendo as classes para diretórios específicos. Esse tipo de reestruturação invalidaria as ligações de rastreabilidade existentes. Contudo, o ArchTrace é capaz de evoluir as ligações de rastreabilidade existentes, aplicando as políticas 9, 5 e 2. Para cada nova versão de código fonte, a política 9 analisa a versão anterior e estabelece novas ligações de rastreabilidade (mesmo que a nova versão tenha outro nome ou esteja em outro local). Já a política 5 remove as ligações de rastreabilidade antigas quando as novas são estabelecidas pela política 9. Contudo, caso alguma ligação de rastreabilidade envolva elementos imutáveis, ou seja, que já foram liberados para produção, a política 2 não permite que as políticas 9 ou 5 atuem sobre eles. A Figura 6.17 mostra a tela principal do ArchTrace após essa reestruturação. Na parte inferior da Figura 6.17, são exibidas as mensagens das políticas após a execução. É possível notar que as ligações de rastreabilidade do componente “Hóspede” (selecionado na esquerda) foram corretamente atualizadas (caixas de marcação ligadas na direita). Contudo, a primeira versão da classe “TipoQuarto.java” (selecionada da direita) continua implementando o componente “Tipo de Quarto” (caixa de marcação ligada na esquerda), por este ser imutável.

A interface gráfica do ArchTrace permite, além da consulta sobre as ligações de rastreabilidade, a utilização das mesmas em processos de *check-in* e *check-out* no nível de abstração de componentes. Por meio de menus *popup* sobre os elementos arquiteturais, o engenheiro de software pode fazer *check-out* dos ICs que implementam esses elementos. Para isso, um algoritmo de recorte é utilizado, copiando para o espaço de trabalho somente os ICs que tem ligações de rastreabilidade para o elemento arquitetural escolhido.

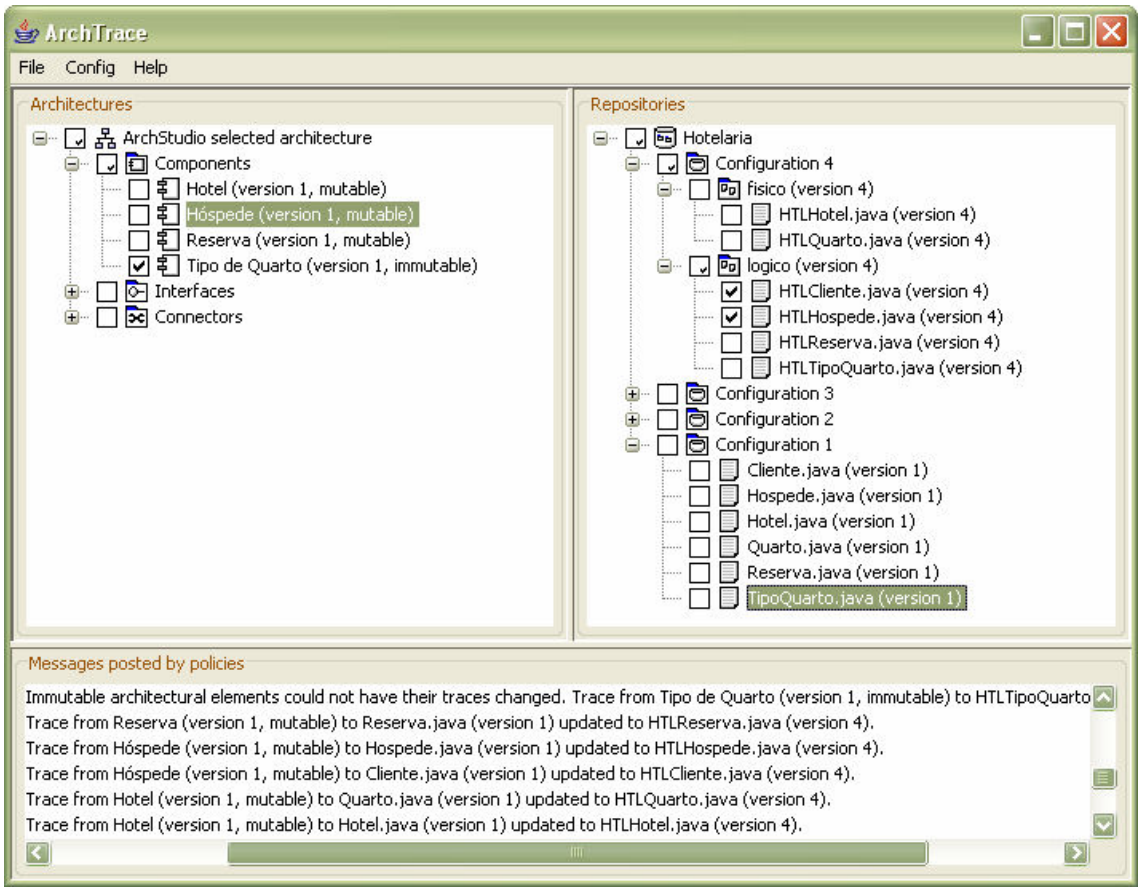


Figura 6.17: ArchTrace evoluindo ligações de rastreabilidade do exemplo de hotelaria.

6.6.1.2 Carga Dinâmica

Após a implementação e o empacotamento dos componentes, que pode ser feito com o auxílio de ferramentas convencionais, como make (FELDMAN, 1979) ou Ant (HATCHER e LOUGHRAN, 2004), se inicia a implantação. Para facilitar esse processo de implantação, a Carga Dinâmica possibilita que componentes possam ser selecionados em tempo de execução e ativados no sistema alvo. A implementação atual da Carga Dinâmica foi feita no contexto do ambiente Odyssey, na linguagem Java. Contudo, essa implementação pode ser adaptada para outros sistemas, caso necessário. Para que um componente possa ser carregado dinamicamente, três condições devem ser atendidas.

A primeira condição diz respeito à implementação do componente. O componente deve implementar uma interface predefinida pelo mecanismo de Carga Dinâmica. Essa interface especifica métodos para inserção de menus na aplicação alvo. A implementação atual insere tanto menus em janelas específicas quanto *popup* menus sobre itens específicos.

A segunda condição diz respeito ao empacotamento do componente. O componente deve declarar, no manifesto de empacotamento, um atributo predefinido

pelo mecanismo de Carga Dinâmica. Esse atributo informa qual classe do componente implementa a interface discutida na primeira condição.

A terceira condição diz respeito à implantação do componente propriamente dita. Para que a Carga Dinâmica seja capaz de selecionar, dentre os componentes disponíveis, quais devem ser implantados, é utilizado um descritor de componentes. Esse descritor de componentes, com DTD (*Document Type Definition*) (W3C, 2004) apresentado na Figura 6.18, divide os componentes em três tipos: *kernel*, *plugin* e *library*. Os componentes do tipo *kernel* não são opcionais. Por isso devem sempre ser implantados na instalação inicial da aplicação alvo. Os componentes do tipo *plugin* podem ser carregados posteriormente, via comando do engenheiro de software. Finalmente, os componentes do tipo *library* são bibliotecas necessárias para o funcionamento dos demais componentes. A instalação de um componente do tipo *library* não depende de comandos diretos do engenheiro de software, mas sim da necessidade desse componente para que outros componentes funcionem corretamente.

```
<!ELEMENT components (component)* >
<!ELEMENT component (dependency)* >
<!ELEMENT component EMPTY >
<!ATTLIST component
        type (kernel | plugin | library) #REQUIRED
        name ID #REQUIRED
        description CDATA #IMPLIED
        location CDATA #REQUIRED
>
<!ATTLIST dependency
        name IDREF #REQUIRED
>
```

Figura 6.18: DTD para descrição de componentes da Carga Dinâmica.

A Figura 6.19 apresenta parte do descritor de componentes do exemplo de hotelaria. Esse descritor pode ser visto como um detalhamento da linha base de produto, pois identifica univocamente cada um de seus componentes. Por exemplo, é possível notar que a versão 1.0 do sistema de hotelaria necessita obrigatoriamente da versão 1.2 do componente “Hotel”. Contudo, o componente “Reserva” é opcional. Sendo assim, o sistema de hotelaria pode atuar em modo consulta, sem o componente “Reserva”, ou em modo completo, quando o componente “Reserva” é instalado.

Contudo, para que o componente “Reserva” funcione corretamente, é necessário instalar também o componente “Hóspede” e o componente “Axis” (responsável por acesso via Serviços Web). Além disso, como o componente “Axis” tem dependências adicionais, essas dependências também têm que ser instaladas. A Figura 6.20 mostra o

mecanismo de Carga Dinâmica em ação[23], após a solicitação de instalação do componente "Reserva". Como o componente "Hóspede" já estava instalado, o mecanismo de Carga Dinâmica não o baixa novamente, minimizando o tempo total de implantação.

```
...
<component type="kernel" name="hotelaria-1.0.jar" ... >
      <dependency name="hotel-1.2.jar" />
</component>

<component type="plugin" name="reserva-1.3.jar" ... >
      <dependency name="hospede-1.0.jar " />
      <dependency name="axis-1.2.1.jar" />
</component>

<component type="library" name="hotel-1.2.jar" ... />
<component type="library" name="hospede-1.0.jar" ... />
<component type="library" name="axis-1.2.1.jar" ... >
      <dependency name="saaj-1.2.jar"/>
      <dependency name="jaxrpc-1.1.jar"/>
      <dependency name="commons-discovery-0.2.jar"/>
      <dependency name="commons-logging-1.0.4.jar"/>
</component>
<component type="library" name="commons-discovery-0.2.jar" ... />
<component type="library" name="commons-logging-1.0.4.jar" ... />
<component type="library" name="jaxrpc-1.1.jar" ... />
<component type="library" name="saaj-1.2.jar" ... />
```

Figura 6.19: Parte do descritor de componentes do exemplo de hotelaria.

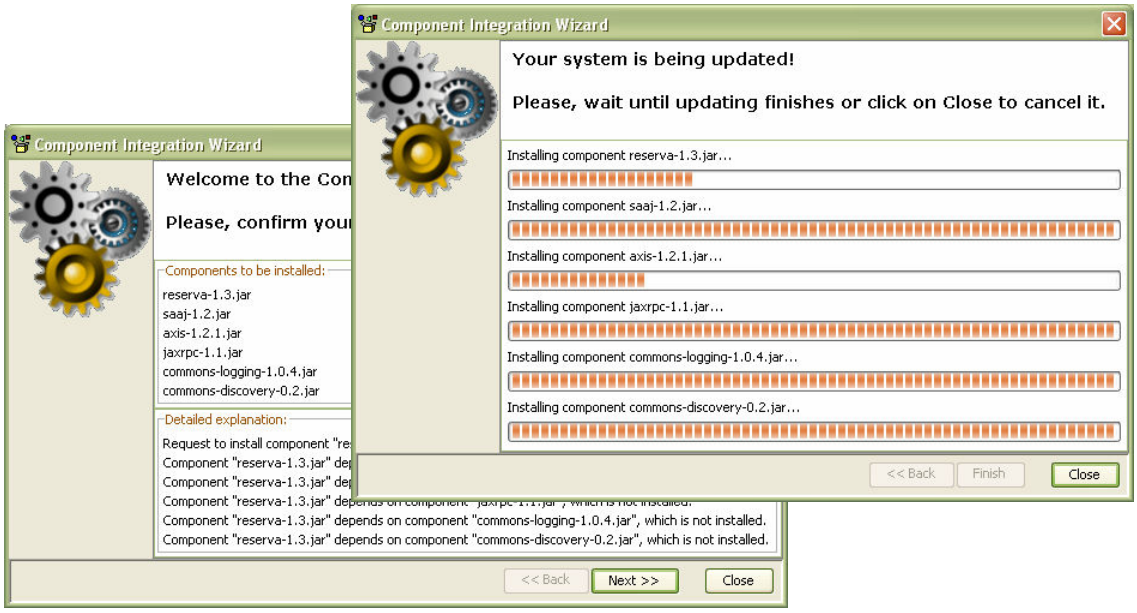


Figura 6.20: Carga Dinâmica de componentes.

[23] Vale ressaltar que a implementação atual do mecanismo de Carga Dinâmica não é genérica, e foi feita no contexto do ambiente Odyssey. A utilização da Carga Dinâmica no exemplo de hotelaria é meramente ilustrativa, com o intuito de manter um único exemplo durante todo o capítulo. Contudo, como explicitado anteriormente, é possível adaptá-lo para outros sistemas.

Após a instalação dos componentes, o mecanismo de Carga Dinâmica faz uso da infra-estrutura de introspecção da linguagem Java para instanciá-los e carregá-los em memória, solicitando os menus e instalando-os na aplicação alvo, sem que seja necessário reiniciá-la.

6.6.2 Considerações finais sobre o Odyssey-BRD

Algumas abordagens (ALDRICH *et al.*, 2002; WALLS e RICHARDS, 2003) combinam a definição de arquitetura no próprio código fonte, evitando a necessidade de rastreabilidade. Contudo, muitas situações demandam representações separadas para arquitetura e código fonte (OMMERING *et al.*, 2000). Por exemplo, quando diferentes equipes utilizam diferentes ferramentas para especificar o software e para a sua posterior construção. Nessas situações, é necessário manter a rastreabilidade entre as representações.

Dentre as abordagens que lidam com rastreabilidade entre diferentes representações do software (SHIRABAD *et al.*, 2001; ANTONIOL *et al.*, 2002; BRIAND *et al.*, 2003; HUFFMAN HAYES *et al.*, 2003; MARCUS e MALETIC, 2003; DE LUCIA *et al.*, 2004; SETTIMI *et al.*, 2004; YING *et al.*, 2004; ZIMMERMANN *et al.*, 2004), a maioria está focada somente na detecção das ligações de rastreabilidade, em detrimento da evolução das mesmas. Desta forma, essas abordagens reconstroem as ligações de rastreabilidade de tempos em tempos, ignorando algumas informações que poderiam apoiar numa reconstrução progressiva e contínua.

Existem abordagens adicionais que, apesar de não apoiarem na detecção das ligações de rastreabilidade em si, permitem verificar a consistência entre diferentes representações do software. REISS (2002), NENTWICH et al. (2003) e ABI-ANTOUN et al. (2005) transformam representações específicas em uma representação genérica, permitindo a construção de restrições entre as representações, como, por exemplo, regras de boa formação e transformações diretas. Diferentemente dessas abordagens, ArchTrace não atua somente detectando inconsistências, mas tentando evitar que elas venham a ocorrer. Além disso, ArchTrace faz uso da dimensão histórica para manter continuamente a consistência entre as representações

Finalmente, essas abordagens não deveriam ser vistas como competidoras do ArchTrace, mas sim como complementares. Desta forma, políticas adicionais poderiam ser implementadas no ArchTrace contemplando totalmente ou parcialmente essas abordagens. Assim, essas políticas atuariam colaborando com as políticas atuais, em

especial nas situações onde as políticas atuais não atendem a contento, como apresentado no Capítulo 7.

Em relação à Carga Dinâmica de componentes, diversas outras abordagens também fornecem característica semelhante, como, por exemplo, Eclipse (ECLIPSE FOUNDATION, 2006a), JBuilder (BORLAND, 2006b) e NetBeans (NETBEANS COMMUNITY, 2006). Contudo, a maioria das abordagens obriga a adoção da sua plataforma, o que pode ser indesejável para sistemas já existentes, como é o caso do ambiente Odyssey. Além disso, elas usualmente forçam que o engenheiro de software reinicie a aplicação para que os componentes recém instalados sejam ativados. Este ponto é contornado pela Carga Dinâmica com o uso de introspecção e de um carregador de classes (*class loader*) especial.

6.7 Odyssey-WI

A abordagem Odyssey-WI, apresentada no Capítulo 5, foi implementada em Java. O Odyssey-WI faz uso dos dados de controle de modificações do Odyssey-CCS e de controle de versões do Odyssey-VCS para detectar e contextualizar rastros de modificação entre elementos de modelo UML. Para isso, o Odyssey-WI interage diretamente com o repositório MOF MDR, onde tanto o Odyssey-CCS quanto o Odyssey-VCS persistem seus dados. O restante desta seção apresenta as decisões de projeto tomadas durante a implementação do Odyssey-WI.

6.7.1 Detalhamento da solução

Para que a abordagem Odyssey-WI atinja o seu objetivo, que consiste na detecção automática de rastros de modificação entre elementos de modelo UML, e a contextualização desses rastros com informações adicionais, é necessário tratar duas questões principais: a técnica a ser utilizada na mineração de dados e o mecanismo de coleta de informações para contextualização dos rastros.

Dentre as diversas técnicas de mineração de dados existentes, a técnica que melhor se encaixa com as características do problema em questão é a de regras de associação (AGRAWAL *et al.*, 1993), mais especificamente, o algoritmo Apriori (AGRAWAL e SRIKANT, 1994). A técnica de regras de associação extrai conjuntos freqüentes de itens que ocorrem nas transações de banco de dados. A Tabela 6.5 apresenta, para cada termo utilizado na área de mineração de dados, o seu mapeamento

para um domínio clássico (supermercado) e para o domínio de GCS, utilizado pelo Odyssey-WI.

Tabela 6.5: Terminologia de mineração de dados.

Domínio	Item	Transação	Banco de dados	Resultado da mineração
Supermercado	Produto	Venda	SGBD relacional	Preferências do cliente
GCS	IC	Modificação	Repositório de GCS	Rastros de modificação

É importante notar que o Odyssey-WI atua somente como um apoio para o engenheiro de software. Os rastros de modificação detectados não constituem verdade absoluta, mas sim sugestões coletadas de experiências passadas. Visando categorizar a relevância dessas sugestões, cada rastro de modificação tem uma interpretação probabilística baseada na quantidade de evidência coletada dos dados que eles derivam.

Essa evidência é representada por duas medidas de mineração de dados (AGRAWAL *et al.*, 1993): suporte e confiança. A medida de suporte quantifica a co-ocorrência de ICs nas modificações implementadas (probabilidade conjunta). Por outro lado, a medida de confiança quantifica a co-ocorrência de ICs nas modificações implementadas, dado que um determinado IC consultado ocorre (probabilidade condicional).

O algoritmo Apriori minera rastros de modificação que satisfazem valores mínimos de suporte e confiança. Por esse motivo, é necessário configurar no Odyssey-WI esses valores mínimos, como apresentado na Figura 6.21.a. Para o exemplo de hotelaria, esses valores foram configurados em 50% e 60%, respectivamente para suporte e confiança. O significado dessas medidas é que o IC consultado tem que ter sido modificado juntamente com os ICs rastreados em ao menos 50% de todas as modificações. Por outro lado, os ICs rastreados têm que ter sido modificados em ao menos 60% de todas as modificações que afetaram o IC consultado.

Quanto maiores os valores mínimos de suporte e confiança, menos rastros de modificação são apresentados, deixando, possivelmente, alguns rastros de modificação relevantes de fora. Por outro lado, quanto menores os valores mínimos de suporte e confiança, mais rastros de modificação são apresentados, mas com um maior número de rastros de modificação falso-positivos. Foi possível observar que os valores mínimos de suporte e confiança devem iniciar maiores para projetos novos, e serem diminuídos até uma posição estacionária. Um projeto novo, usualmente, tem em seu repositório de GCS poucas modificações implementadas. O número de modificações implementadas é um fator importante no cálculo das medidas de suporte e confiança. Em repositórios de

GCS pequenos, cada modificação tem uma grande influência nos valores de suporte e confiança, aumentando o número de rastros de modificação falso-positivos. Por outro lado, em repositórios de GCS grandes, uma modificação individual tem muito pouca influência na medidas de suporte e confiança, permitindo a redução dos valores mínimos. Algumas outras estratégias para a seleção de valores mínimos de suporte e confiança podem ser obtidas na literatura (ZIMMERMANN *et al.*, 2004).

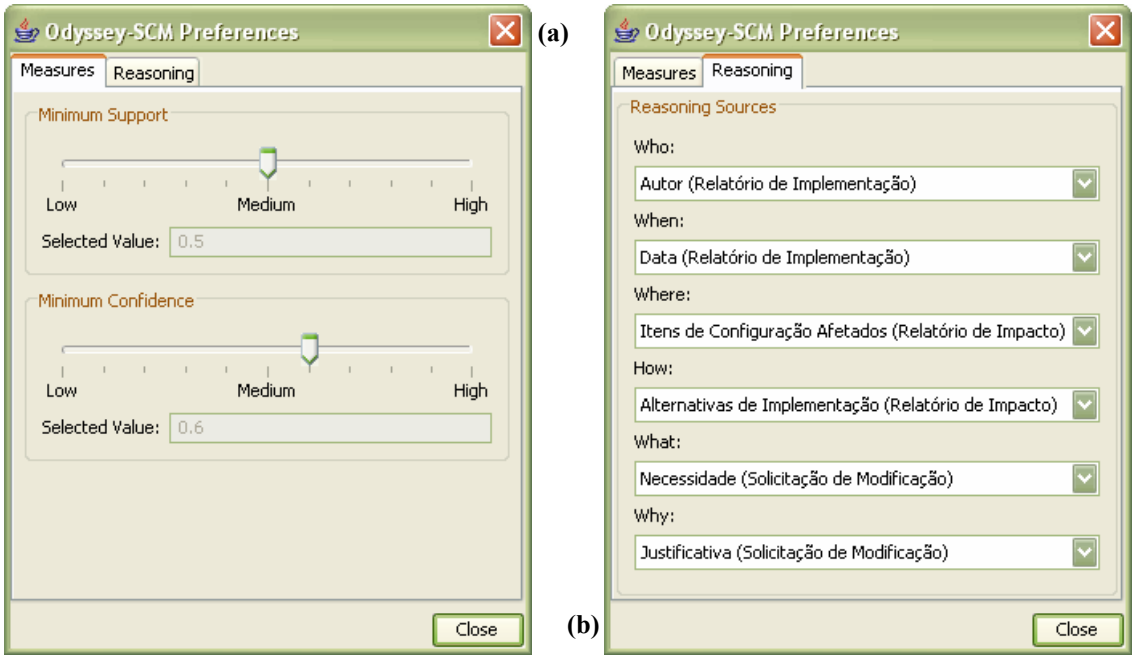


Figura 6.21: Configuração das medidas de mineração de dados (a) e da coleta de informações de contextualização dos rastros de modificação (b) no Odyssey-WI.

Visando permitir a coleta de informações para a contextualização dos rastros de modificação, o Odyssey-WI faz uso da integração entre o Odyssey-VCS e o Odyssey-CCS. Essa integração é construída durante a atividade de implementação da modificação. Nesse momento, o engenheiro de software informa ao Odyssey-SCM qual modificação está sendo implementada. A partir daí, todo *check-in* efetuado no Odyssey-VCS é relacionado com a modificação em questão no Odyssey-CCS. Com isso, é possível consultar, para uma dada modificação no Odyssey-CCS, quais ICs foram manipulados no Odyssey-VCS, e para uma dada versão de ICs no Odyssey-VCS, todos os documentos gerados a partir dos formulários preenchidos no Odyssey-CCS.

O Odyssey-WI permite que sejam configurados os formulários e campos que devem ser inspecionados no Odyssey-CCS para a contextualização dos rastros de modificação. Além disso, são lidas automaticamente do Odyssey-VCS informações como autor, data e mensagem postada durante o *check-in*, além dos próprios ICs modificados. No exemplo de hotelaria, os formulários e campos lidos para cada tipo de

informação a ser coletada são exibidos na Tabela 6.6. A Figura 6.21.b mostra essa configuração sendo feita no Odyssey-WI.

Tabela 6.6: Formulários e campos inspecionados no Odyssey-CCS para a contextualização dos rastros de modificação do exemplo de hotelaria.

Tipo de informação	Formulário de coleta	Campo de coleta
Quem?	Relatório de Implementação	Autor
Quando?	Relatório de Implementação	Data
Onde?	Relatório de Impacto	ICs Afetados
Como?	Relatório de Impacto	Alternativas de Implementação
O quê?	Solicitação de Modificação	Necessidade
Por quê?	Solicitação de Modificação	Justificativa

Finalmente, no exemplo de hotelaria, após terem sido implementadas as 6 modificações apresentadas na Tabela 6.1, uma nova solicitação de modificação é submetida ao Odyssey-CCS. Após aprovada, durante a atividade de implementação, o engenheiro de software constata que a classe "Reserva" precisa ser modificada e faz uso do Odyssey-WI para identificar possíveis rastros de modificação. Neste cenário, a classe "Reserva" é o IC consultado, e os rastros de modificação indicam possíveis ICs impactados por uma modificação na classe "Reserva". A Figura 6.22 apresenta o resultado da consulta ao Odyssey-WI.

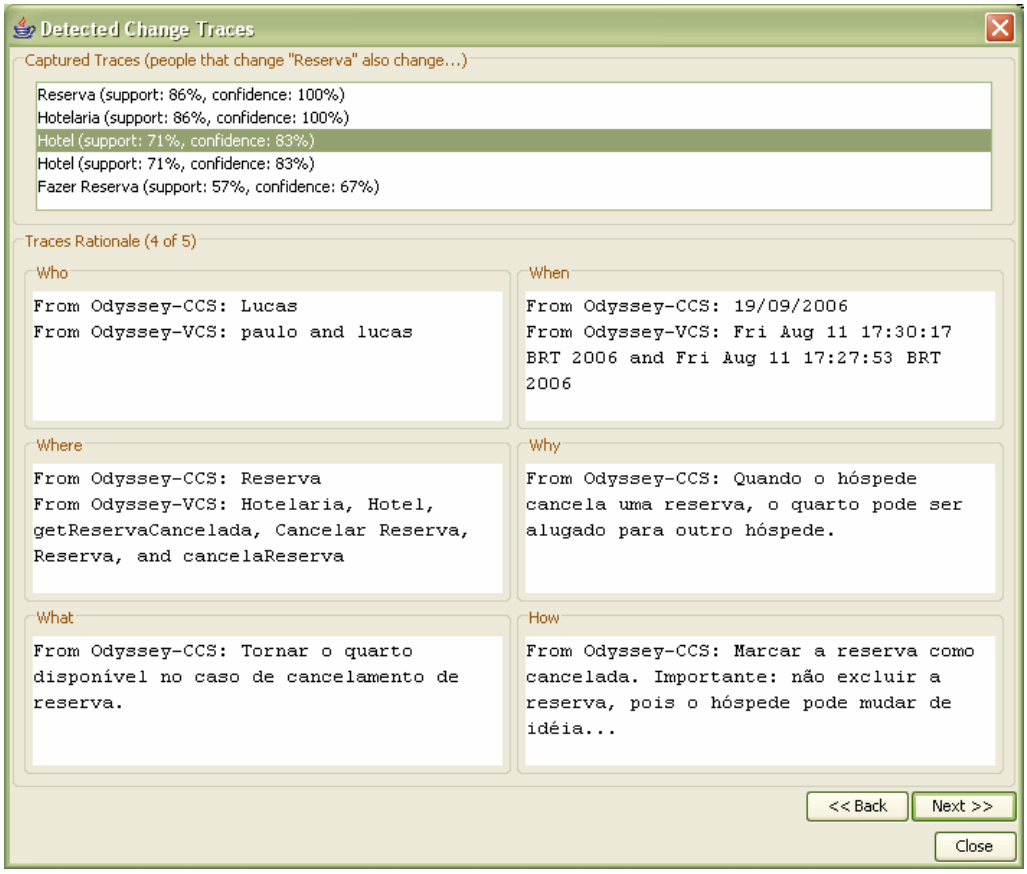


Figura 6.22: Odyssey-WI apresentando os rastros de modificação da classe "Reserva".

É possível observar na parte de cima da Figura 6.22 que em 83%[24] das vezes que a classe "Reserva" foi modificada, a classe "Hotel" também o foi. Além disso, em 67% das vezes que a classe "Reserva" foi modificada, o caso de uso "Fazer Reserva" também o foi. Outros elementos de modelo com confiança abaixo de 60% foram filtrados devido à configuração feita na Figura 6.21.a.

A parte de baixo da Figura 6.22 apresenta as informações de contextualização dos rastros de modificação coletadas do Odyssey-VCS e do Odyssey-CCS, de acordo com a configuração apresentada na Figura 6.21.b. É possível notar algumas divergências entre as informações providas pelo Odyssey-VCS e do Odyssey-CCS. De forma genérica, as informações coletadas de sistemas de controle de versões representam fatos mais precisos, pois elas são computadas automaticamente no momento de *check-in*, e não via entrada do engenheiro de software. Por exemplo, a data fornecida pelo Odyssey-VCS é o momento exato do *check-in*, enquanto a data fornecida pelo Odyssey-CCS é o momento em que a atividade foi finalizada, que pode ser posterior devido ao esquecimento do engenheiro de software.

6.7.2 Considerações finais sobre o Odyssey-WI

Nas últimas décadas, pesquisadores têm experimentado a utilização de repositórios de GCS para entender o software, assim como a sua evolução. GALL et al. (1997) utilizaram dados de liberação para detectar acoplamento lógico entre módulos. BALL et al. (1997) fizeram análises sobre classes C++ armazenadas em repositórios de GCS, buscando detectar a proximidade entre as mesmas em função das modificações implementadas. De forma similar, BIEMAN et al. (2003) identificaram classes em um sistema comercial que poderiam ser alvo de reengenharia.

Além disso, outros trabalhos também fizeram uso da dimensão histórica de repositórios de GCS. SHIRABAD et al. (2001) utilizaram aprendizado indutivo para detectar níveis de relevância entre arquivos. EICK et al. (2001) analisaram o histórico de modificações e aplicaram índices de decaimento do código para identificar fatores de risco em sistemas. DRAHEIM et al. (2003) analisaram atividades do processo de

[24] Apesar da Tabela 6.2 apresentar 4 modificações que afetam as classes "Hotel" e "Reserva" concomitantemente, e um total de 5 modificações que afetam a classe "Reserva", o que daria confiança de 80% (4/5), o *check-in* da primeira versão do modelo antes das modificações também deve ser considerado, o que produz confiança de 83% (5/6).

desenvolvimento e aplicaram algumas medidas sobre o repositório de GCS visando identificar a relação de qualidade entre processo e produto. Finalmente, ZIMMERMANN et al. (2004) evidenciaram que a mineração sobre repositórios de GCS pode ser útil para apoiar modificações futuras, detectando dependências ocultas e prevenindo modificações incompletas.

Contudo, essas abordagens atuam sobre repositórios de GCS baseados em arquivos. Por essa razão, nenhuma delas apóia a detecção de rastros de modificação em elementos UML de granularidade fina. Além disso, nenhuma delas fornece indicadores da razão de surgimento dos rastros de modificação, extraídos automaticamente de uma infra-estrutura integrada de GCS. Esses dois pontos são tratados pelo Odyssey-WI.

6.8 Considerações finais

Este capítulo apresentou as decisões de projeto e implementação do Odyssey-SCM, assim como um exemplo de sua utilização. O exemplo utilizado, que consiste em manutenção sobre um sistema de hotelaria, é propositalmente pequeno e simples, para facilitar a compreensão sobre o funcionamento do Odyssey-SCM. Contudo, as funcionalidades providas pelo Odyssey-SCM se tornam ainda mais importantes quando aplicadas a sistemas maiores e mais complexos, onde a capacidade de compreensão do engenheiro de software sobre o sistema é limitada.

Nesse contexto de sistemas grandes e complexos, se torna ainda mais importante a existência de processos definidos, derivados de um processo padrão, como o Odyssey-SCMP, guiando os engenheiros de software de forma previsível pelas atividades de GCS. Além disso, a modelagem e automação desses processos no Odyssey-CCS aumentam a precisão e o controle sobre a execução.

A importância do Odyssey-CCS também é aumentada em situações onde o número de modificações em andamento não é 6, mas sim centenas ou milhares. Nesse cenário, a ausência de informações precisas sobre o estado de cada modificação pode levar o projeto ao caos. Por outro lado, a construção e manutenção de sistemas grandes e complexos demandam equipes numerosas. Sendo assim, o apoio dado pelo Odyssey-VCS, que possibilita o trabalho concomitante de múltiplos engenheiros de software sobre o mesmo modelo, se torna ainda mais necessário.

Nesses cenários, o número de componentes e a quantidade de dependências entre eles também tende a aumentar consideravelmente, e nem sempre a arquitetura conceitual desses componentes está implementada de forma análoga no código fonte,

fenômeno esse denominado derivação arquitetural (*architectural drift*). O Odyssey-BRD apóia no tratamento desse problema, evoluindo as ligações de rastreabilidade entre a arquitetura conceitual e a sua implementação. Além disso, ele também apóia a implantação desses componentes no ambiente de produção, levando em consideração as dependências existentes.

Finalmente, a contribuição do Odyssey-WI pode ser ainda aumentada em projetos com repositórios de GCS grandes e com alta rotatividade de pessoal, onde poucos engenheiros de software conhecem a fundo o relacionamento entre os ICs. Nesse cenário, a análise de impacto, e, conseqüentemente, a implementação de modificações sem o apoio provido pelo Odyssey-WI, pode demandar um tempo excessivamente longo e ser mais suscetível a erros.

O Capítulo 7 apresenta uma avaliação feita sobre o Odyssey-SCM, visando caracterizar o seu comportamento quando aplicado a sistemas maiores e mais complexos do que o sistema de exemplo apresentado neste capítulo.

Capítulo 7 – Avaliação da Abordagem Odyssey-SCM

7.1 Introdução

Este capítulo apresenta uma avaliação da abordagem Odyssey-SCM. O foco principal desta avaliação é em aspectos relacionados a desempenho e acurácia. Contudo, avaliações adicionais devem ser feitas sob outras perspectivas (ISO, 2001), como, por exemplo, confiabilidade, usabilidade, manutenibilidade e portabilidade.

Dentre os cinco módulos do Odyssey-SCM, a avaliação ocorreu sobre o Odyssey-VCS, o Odyssey-BRD e o Odyssey-WI. A avaliação utilizou, sempre que possível, o próprio ambiente Odyssey como objeto de estudo, servindo tanto como avaliação da abordagem propriamente dita quanto como fonte de informações sobre a evolução do projeto Odyssey no decorrer dos últimos anos.

As seções 7.2, 7.3 e 7.4 apresentam o planejamento, a execução e a análise das avaliações feitas, respectivamente, sobre o Odyssey-VCS, Odyssey-BRD e Odyssey-WI. Finalmente, a Seção 7.5 sumariza as conclusões obtidas a partir dessas avaliações.

7.2 Avaliação do Odyssey-VCS

A necessidade de controle de versões para artefatos de alto nível de abstração, como, por exemplo, modelos UML, é amplamente enfatizada na literatura (CONRADI e WESTFECHTEL, 1998; OHST e KELTER, 2002; CHRISSIS *et al.*, 2003; ESTUBLIER *et al.*, 2005; IEEE, 2005). Contudo, uma característica importante desse tipo de sistema é a escalabilidade (ESTUBLIER, 2000). Por esta razão, foi executado um estudo que visa caracterizar o Odyssey-VCS em comparação com sistemas de controle de versões convencionais, baseados em arquivos, mais precisamente, CVS (FOGEL e BAR, 2001) e Subversion (COLLINS-SUSSMAN *et al.*, 2004). O objetivo desse estudo é compreender os efeitos relacionados à escalabilidade quando elementos de modelo UML estão sendo versionados.

O estudo foi executado sobre dados históricos de dois sistemas existentes. O primeiro sistema, o Odyssey-XMI, consiste de uma pequena ferramenta para importação e exportação de modelos UML no formato XMI. Esta ferramenta tem em torno de 2.000 linhas físicas de código Java e é um *plug-in* do ambiente Odyssey. O segundo sistema é o *kernel* do ambiente Odyssey (WERNER *et al.*, 2003) propriamente dito. O *kernel* do

ambiente Odyssey tem em torno de 60.000 linhas físicas de código Java. Apesar de serem sistemas pequenos do ponto de vista de GCS, são grandes o suficiente para demonstrar as diferenças entre versiomento baseado em arquivos (sistemas existentes) e versionamento baseado em modelos (Odyssey-VCS). Deste ponto em diante, chamaremos esses sistemas de S1 e S2, respectivamente.

Para que esse estudo fosse executado, foram coletados dados de versionamento do sistema S1 entre 24 de outubro de 2003 e 14 de setembro de 2005 e do sistema S2 entre 26 de novembro de 2003 e 11 de dezembro de 2003. Os períodos escolhidos, 23 meses e 1 mês respectivamente para S1 e S2, visaram possibilitar tanto uma análise na dimensão de duração do projeto (S1) quanto na dimensão de tamanho do projeto (S2). Esses dados foram reorganizados para possibilitar a reprodução dos *check-outs* e *check-ins* originais. Após essa etapa de reorganização, os *check-outs* e *check-ins* originais foram reproduzidos novamente em repositórios CVS, Subversion e Odyssey-VCS. No caso do Odyssey-VCS, um passo adicional de engenharia reversa foi introduzido antes de cada *check-in*, para obter o modelo UML correspondente. O resultado obtido foi a reprodução dos cenários originais de desenvolvimento e manutenção. Esta estratégia possibilitou uma análise retrospectiva de desempenho dos três sistemas em ação.

Nas próximas subseções, é detalhado o planejamento do estudo retrospectivo[25], a preparação do ambiente para o estudo, as estatísticas coletadas de cada sistema, a execução do estudo propriamente dita e a análise sobre os resultados obtidos dessa execução. O resultado final desse estudo é dual: uma demonstração do Odyssey-VCS em ação, ilustrando o seu uso para versionar modelos UML, e, genericamente, um entendimento aprofundado das qualidades e deficiências de abordagens de granularidade fina para versionamento de modelos.

7.2.1 Planejamento do estudo

Esta avaliação consiste em um estudo observacional via um *benchmark* "in vitro" (TRAVASSOS *et al.*, 2002) para caracterizar (BASILI, 1996) o Odyssey-VCS em termos de sistemas existentes baseados em arquivos. O objetivo desse estudo é observar o Odyssey-VCS em um ambiente controlado, mas utilizando dados reais, e

[25] Na medicina, um estudo retrospectivo (*retrospective study*) consiste em um estudo que analisa o passado, usualmente via registros médicos e entrevistas com pacientes que sabidamente têm uma determinada doença, visando explicar o presente (MEDICINENET, 2003).

detectar as suas fraquezas e possíveis desafios genéricos a versionamento baseado em modelos. Neste cenário, duas variáveis independentes (fatores) são detectadas: sistema de controle de versão e projeto de software. Os valores possíveis (tratamentos) da variável "sistema de controle de versão" são CVS, Subversion e Odyssey-VCS. Por outro lado, os valores possíveis (tratamentos) da variável "projeto de software" são S1 e S2. Como discutido anteriormente, S1 e S2 são sistemas diferentes, em termos de complexidade e tamanho.

Apesar do CVS e do Subversion serem sistemas de controle de versão baseados em arquivos, eles são amplamente utilizados em projetos comerciais e de código livre. Por essa razão, eles foram escolhidos como objetos de controle (*baselines*) do estudo. Em nenhum momento, é assumido que controle de versão sobre arquivos tem a mesma complexidade de controle de versão sobre modelos. Contudo, é importante entender o quão diferentes são esses dois cenários para determinados projetos de software.

As principais variáveis dependentes desse estudo são a duração de *check-in* e *check-out*. Outras variáveis dependentes, como o número de ICs e o tamanho do espaço de trabalho e do repositório também foram observadas para cada combinação possível dos fatores (<CVS, Subversion, Odyssey-VCS> x <S1, S2>). Finalmente, também foi medido o tempo gasto na transformação de arquivos XMI em estruturas orientadas a objetos que são manipuladas pelo Odyssey-VCS e vice-versa. Essa transformação ocorre durante as operações de *check-in* e *check-out*, pois o Odyssey-VCS trabalha sobre um repositório MOF, o MDR.

7.2.2 Preparação do ambiente

A execução do estudo demanda dois ingredientes principais: (1) dados históricos disponíveis e organizados e (2) uma ferramenta para reproduzir esses dados. Os dados históricos foram obtidos de um sistema de GCS existente. Esse sistema, que é o CVS, não organiza os dados por transação de *check-in*. Com o uso do Subversion é possível acessar diretamente configurações específicas do software devido à identificação global de versões. Isto ocorre em decorrência da atribuição de um número único de versão para todos os ICs modificados em um dado *check-in*. Por esta razão, a ferramenta *cvs2svn* foi empregada para migrar os dados existentes, armazenados no repositório CVS, para um repositório Subversion.

Tendo dados históricos de ambos S1 e S2 ordenados pelos *check-ins*, uma ferramenta foi desenvolvida para exportar cada configuração existente e efetuar novos

check-ins nos repositórios monitorados. Uma visão geral do processo implementado por esta ferramenta é apresentado na Figura 7.1. Este processo, que é composto de cinco passos, é repetido para cada configuração (126 configurações para S1 e 52 configurações para S2), possibilitando que as estatísticas pudessem ser coletadas para análise futura.

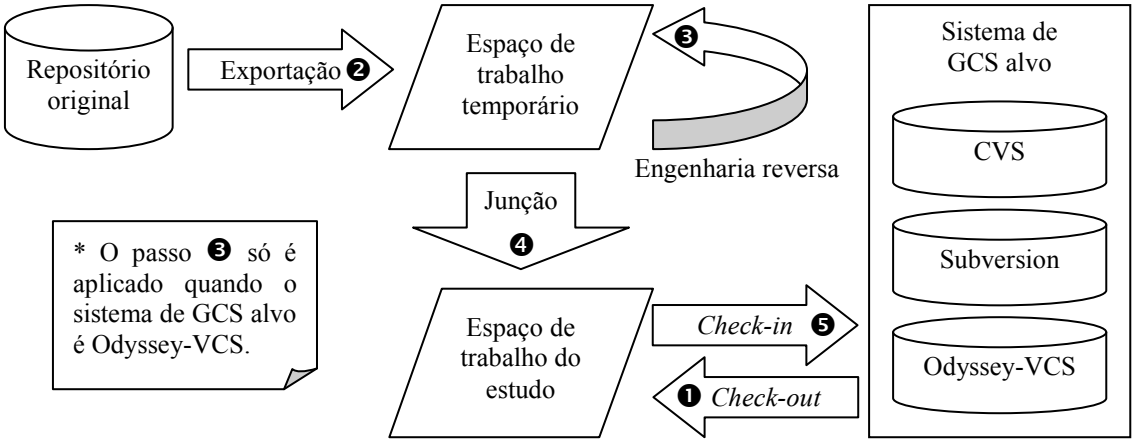


Figura 7.1: Visão geral do estudo.

Antes do início do estudo, o seguinte estado pode ser observado: o repositório original contém todas as versões de S1 ou S2, ambos os espaços de trabalho estão vazios e os repositórios dos sistemas de GCS alvo também estão vazios.

O primeiro passo (❶) preenche o espaço de trabalho do estudo. Este passo é executado para cada iteração, com exceção da primeira. Ele ocorre via *check-out* da última configuração armazenada no sistema de GCS alvo (CVS, Subversion ou Odyssey-VCS). Ambos CVS e Subversion constroem uma estrutura de arquivos e diretórios no espaço de trabalho do estudo. Contudo, o resultado provido pelo Odyssey-VCS é um único arquivo XMI contendo o modelo. O espaço de trabalho do estudo continuará vazio se nenhuma configuração ainda tiver sido adicionada ao sistema de GCS alvo, o que ocorre somente na primeira iteração.

No segundo passo (❷), a configuração é exportada do repositório original, que contém S1 ou S2, para o espaço de trabalho temporário. Se o sistema sob estudo for o Odyssey-VCS, um terceiro passo (❸) deve ser executado. Este passo aplica engenharia reversa sobre o espaço de trabalho temporário, obtendo o modelo de classes UML a partir do código fonte. Este passo é apoiado por duas ferramentas desenvolvidas no contexto do projeto Odyssey: Ares (VERONESE *et al.*, 2002), que é responsável pela engenharia reversa propriamente dita, e Odyssey-XMI, que possibilita exportar o resultado obtido pela Ares em XMI.

Após a execução dos passos anteriores, o espaço de trabalho temporário e o espaço de trabalho do estudo contêm artefatos no mesmo nível de abstração: arquivos e diretórios para o caso do CVS e Subversion, e arquivo XMI contendo modelos UML para o caso do Odyssey-VCS. Contudo, o espaço de trabalho temporário contém os artefatos originais na versão N, e o espaço de trabalho do estudo contém artefatos versionados na versão N – 1. Esses artefatos versionados são compostos dos artefatos originais na versão N – 1, juntamente com as informações de versionamento (diretório CVS, diretório .svn ou *tagged values*, para CVS, Subversion e Odyssey-VCS, respectivamente). O quarto passo (❹) consiste na junção dos artefatos originais, na versão N, que se localizam no espaço de trabalho temporário, com os artefatos já versionados, na versão N – 1, que se localizam no espaço de trabalho do estudo. Este algoritmo de junção é mais complexo do que uma simples cópia, pois ambos CVS e Subversion devem ser notificados quando novos elementos são adicionados ou removidos do espaço de trabalho (comandos "cvs add"/"cvs remove" ou "svn add"/"svn delete").

Finalmente, o quinto passo (❺) consiste na execução de *check-in* do espaço de trabalho do estudo no sistema de GCS alvo. Após este passo, ambos os espaços de trabalho são esvaziados e todo o processo reinicia para a próxima configuração do repositório original. Este processo como um todo é repetido para cada um dos projetos de software (S1 e S2) e para cada um dos sistemas de GCS alvo (CVS, Subversion e Odyssey-VCS), totalizando seis execuções completas.

7.2.3 Coleta de estatísticas

O estudo visa coletar e analisar estatísticas obtidas dos sistemas CVS, Subversion e Odyssey-VCS. Para possibilitar uma coleta automática, foi implementada a ferramenta de reprodução dos *check-ins* e *check-outs*, detalhada na Figura 7.1. Além disso, a ferramenta Odyssey-VCS foi instrumentada para coletar medidas referentes à leitura e escrita de XMI. Um total de 20 medidas foi coletado para cada configuração de cada projeto de software. Essas medidas são: o número, autor e data da configuração; o tamanho do espaço de trabalho, o número de ICs no espaço de trabalho, o tamanho do repositório e as durações de *check-in* e *check-out*, para cada sistema de GCS alvo; e a duração da leitura e da escrita do arquivo XMI pelo Odyssey-VCS.

7.2.4 Execução do estudo

O estudo foi executado em um computador Pentium IV, 2.4 GHz, Hyper-threading, com 1 GB de memória e RAID 1 de 80 GB, composto de dois discos serial ATA (SATA) de 80 GB, executando Windows XP e Cygwin. Uma atenção especial foi dada para minimizar interferências externas, como, por exemplo, antivírus e outros processos residentes que poderiam afetar os resultados finais. O estudo foi executado em ambiente local para focar no desempenho dos sistemas de GCS alvo. Por esta razão, as estatísticas coletadas não levam em conta os atrasos relacionados à comunicação em rede.

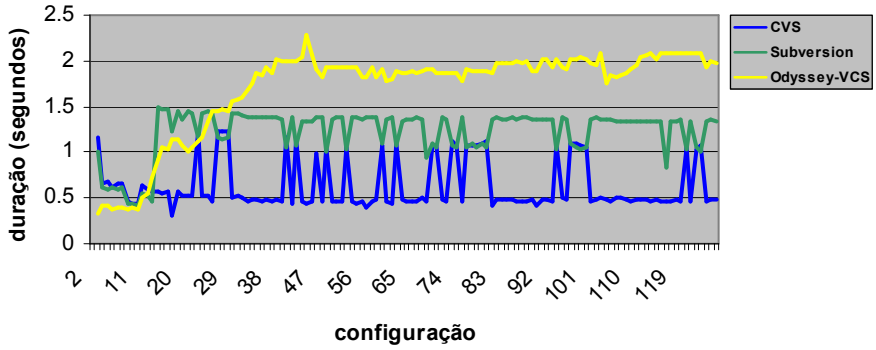
O descritor de comportamento do Odyssey-VCS foi configurado para considerar pacotes, classes, métodos e atributos como unidades de versionamento, e classes, métodos e atributos como unidades de comparação. Os resultados coletados da execução foram salvos pela ferramenta detalhada na Figura 7.1 em um arquivo separado por vírgula (CSV), que foi posteriormente processado utilizando a ferramenta Microsoft Excel.

7.2.5 Análise dos resultados

Visando caracterizar o Odyssey-VCS, foi analisado, inicialmente, o desempenho de *check-out* e *check-in* em relação ao sistema S1, como mostrado na Figura 7.2. Em ambos os casos, pode ser notado que o Odyssey-VCS não teve um desempenho tão bom quanto CVS e Subversion. Independentemente disto, é importante notar que a duração dos *check-outs* e *check-ins* não é afetada pelo número de configurações previamente processadas. Esta situação demonstra escalabilidade na dimensão da duração do projeto. Exemplificando, a duração dos *check-outs* e *check-ins* é em torno de 2 segundos para qualquer configuração entre 38 e 127, o que representa quase 2 anos de desenvolvimento. Contudo, este cenário levanta uma questão importante: “O Odyssey-VCS se comporta da mesma forma em relação ao tamanho do projeto?”.

Para responder essa questão, a mesma análise foi executada em relação ao sistema S2, que é significativamente maior que o sistema S1 (aproximadamente 30 vezes maior). Os resultados, apresentados na Figura 7.3, confirmam a escalabilidade na dimensão da duração do projeto, visto que a duração de *check-outs* e *check-ins* não é afetada pelo número de configurações já existentes no repositório. Contudo, a Figura 7.3 mostra, também, que o Odyssey-VCS não escala na dimensão do tamanho do projeto.

(a) *Check-out*



(b) *Check-in*

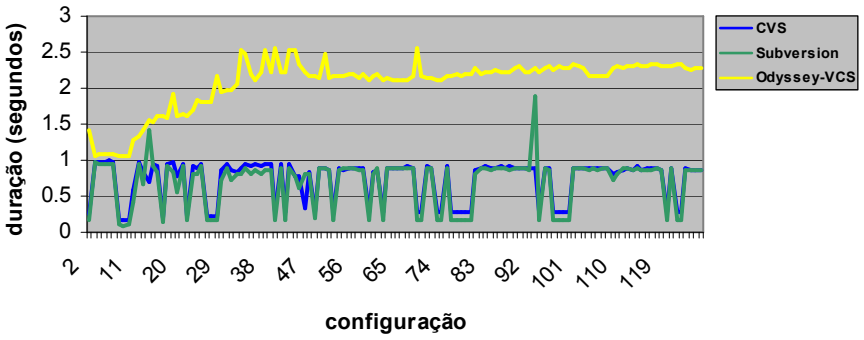
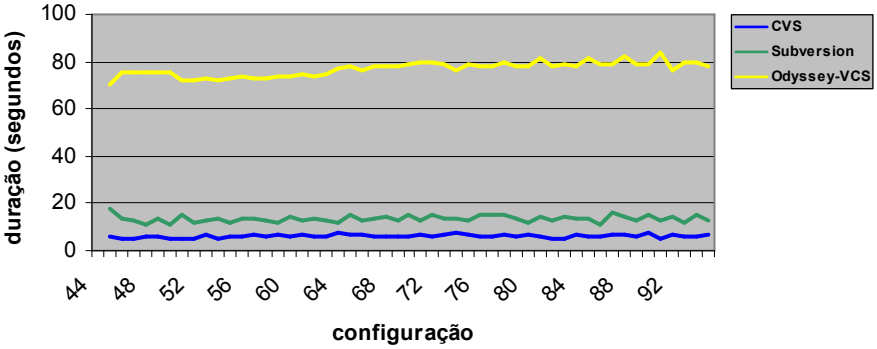


Figura 7.2: Resultados de *check-out* e *check-in* em relação ao sistema S1.

(a) *Check-out*



(b) *Check-in*

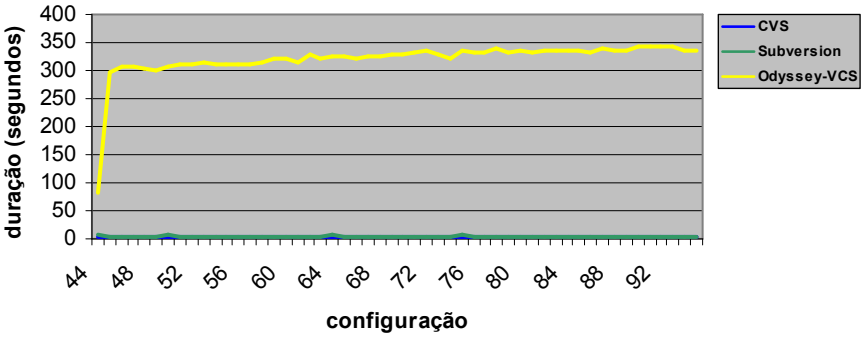


Figura 7.3: Resultados de *check-out* e *check-in* em relação ao sistema S2.

Este fato levanta outras questões importantes: “Por que o Odyssey-VCS não escala na dimensão do tamanho do projeto?” e “Isto é uma limitação específica do Odyssey-VCS ou está relacionado a tecnologias genéricas utilizadas por ele (MOF, MDR, XMI, etc.)?”. Visando responder a essas questões, as seguintes análises mais aprofundadas de entendimento foram executadas: (1) comparação do número de ICs nos espaços de trabalho dos três sistemas de GCS alvo; (2) análise do tamanho do espaço de trabalho dos três sistemas de GCS alvo; e (3) uma análise interna ao Odyssey-VCS, verificando a responsabilidade do MDR no tempo total de *check-out* e *check-in*.

A primeira análise de entendimento comprova a ocorrência do fenômeno de explosão de versões para elementos de granularidade fina. Em relação a S1, é possível notar que o número médio de ICs processados pelo Odyssey-VCS é consideravelmente maior que pelo CVS ou Subversion. CVS considera somente arquivos (mapeados em classes Java) como ICs. Por outro lado, o Subversion considera tanto diretórios (mapeados em pacotes Java) quanto arquivos como ICs. Contudo, o Odyssey-VCS foi configurado para considerar pacotes, classes, métodos e atributos como ICs. Devido a isso, o Odyssey-VCS processou 19 vezes mais ICs que o CVS e 11 vezes mais ICs que o Subversion. Esses resultados são ainda mais notáveis em relação ao sistema S2. Neste caso, o Odyssey-VCS processou 34 vezes mais ICs que o CVS e 29 vezes mais ICs que o Subversion, como apresentado na Figura 7.4. Vale a pena ressaltar que o número de versões de ICs é ainda maior, visto que um IC usualmente tem mais de uma versão associada a ele.

A segunda análise de entendimento foca no tamanho médio do espaço de trabalho. Como discutido anteriormente, o espaço de trabalho do Odyssey-VCS é construído via engenharia reversa do código fonte. Devido a isto, toda informação contida no modelo UML é derivada do código fonte. Contudo, ao analisar o tamanho do espaço de trabalho, é possível notar que o modelo UML é até 5 vezes maior que o respectivo código fonte, que inclui informações de versionamento armazenadas nos diretórios CVS ou .svn. Esta comparação é apresentada na Figura 7.5.

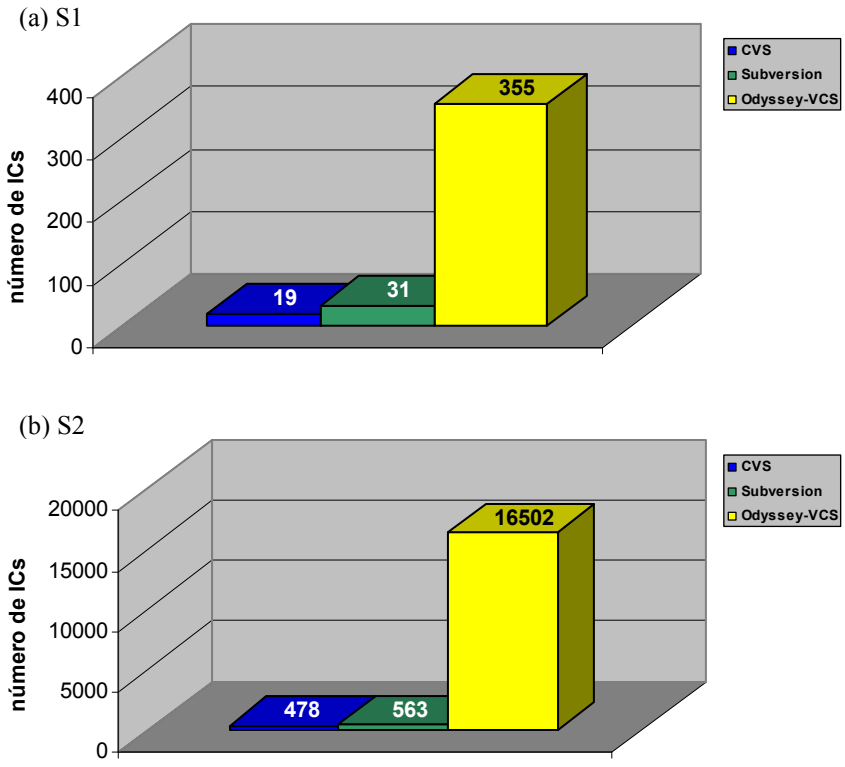


Figura 7.4: Número médio de ICs nos espaços de trabalho de S1 e S2.

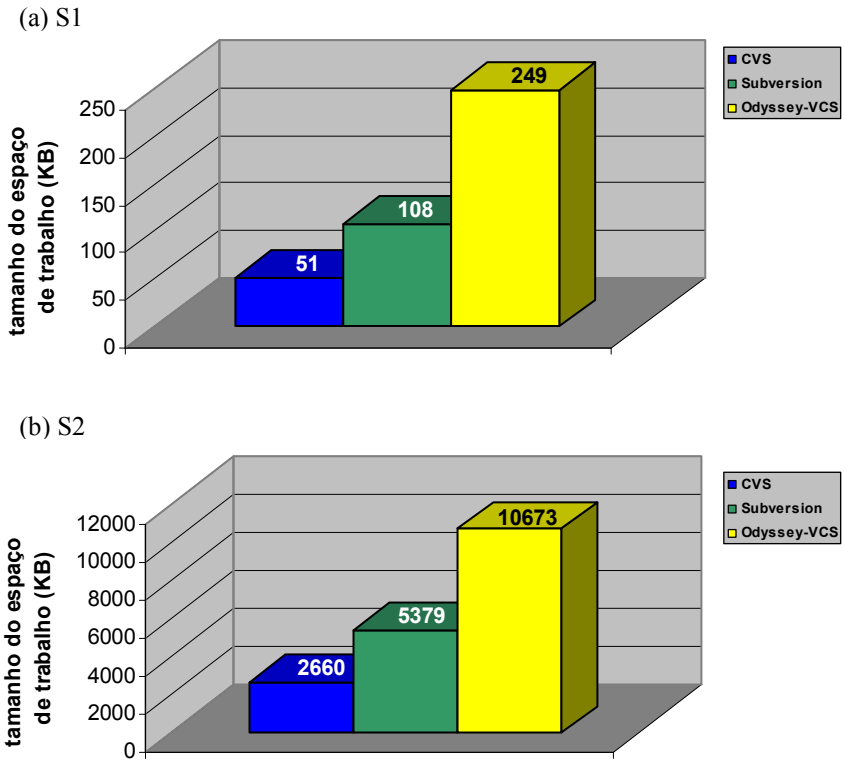


Figura 7.5: Tamanho médio dos espaços de trabalho de S1 e S2.

Finalmente, a terceira análise de entendimento é em relação ao esforço de transformação desses arquivos XMI grandes em estruturas orientadas a objetos, que são persistidas no repositório do Odyssey-VCS. A leitura e a escrita de XMI são executadas pelo MDR. Durante o *check-out*, a versão selecionada do modelo é acessada diretamente por índices de Árvore B e transformada em XMI. Devido a isto, 100% do tempo gasto com *check-out* é referente à escrita de XMI. Durante o *check-in*, o XMI é transformado numa estrutura orientada a objetos, que é, por sua vez, processada pelo Odyssey-VCS. Em média, de 75% a 81% do tempo gasto com *check-in* é referente à leitura do XMI, como exibido na Figura 7.6.

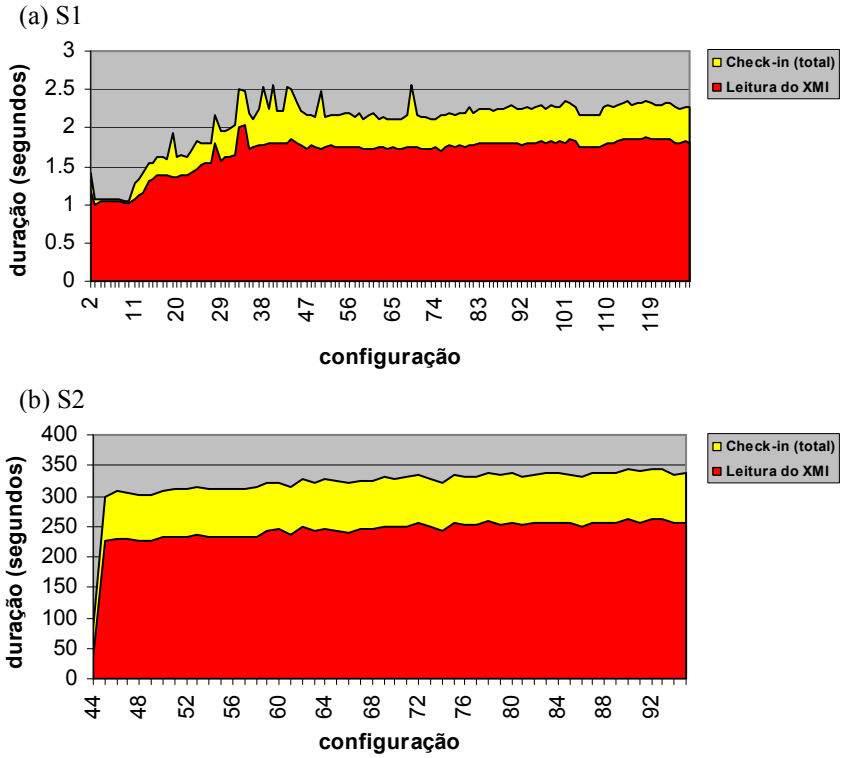


Figura 7.6: Leitura do XMI durante o *check-in*.

7.2.6 Considerações finais sobre a avaliação do Odyssey-VCS

Este estudo de caracterização analisou o Odyssey-VCS em termos de escalabilidade. Duas dimensões foram consideradas: duração e tamanho do projeto. A dimensão de duração do projeto foca em situações onde o repositório contém uma grande quantidade de artefatos. Por outro lado, a dimensão de tamanho do projeto foca em situações onde o espaço de trabalho contém uma grande quantidade de artefatos.

O resultado do estudo mostrou que o Odyssey-VCS teve um bom desempenho em relação à duração do projeto. O mecanismo atual de persistência do Odyssey-VCS é

o MDR, que implementa uma Árvore B para persistir fisicamente os ICs versionados. Árvore B permite buscas, inserções e remoções em tempo logarítmico.

Contudo, o Odyssey-VCS não teve um bom desempenho em relação ao tamanho do projeto. Após uma análise adicional, algumas explicações puderam ser encontradas. Em primeiro lugar, o número de ICs é muito grande quando elementos UML de granularidade fina são versionados (até 34 vezes maior em alguns casos). Além disso, o padrão XMI propriamente dito não escala para representar sistemas grandes. No caso apresentado, S2 tem em torno de 60.000 linhas físicas de código java e o seu modelo obtido via engenharia reversa tem em torno de 10,5 megabytes. Finalmente, o processo de leitura de arquivos XMI grandes é muito lento. O MDR levou em torno de 4 minutos para ler esse arquivo XMI de 10,5 megabytes durante o *check-in* e transformá-lo em uma estrutura orientada a objetos armazenada na Árvore B, e em torno de 1 minuto para reconstruir o arquivo XMI durante o *check-in*.

Desta forma, pode ser concluído que a especificação atual para exportação e troca de modelos UML (i.e.: XMI) e algumas implementações de repositório MOF (i.e.: MDR) não foram projetadas para lidar com sistemas grandes. Este problema motiva novas frentes de trabalho em formas alternativas para intercâmbio de modelos UML, como, por exemplo, camadas padrões de compressão (Odyssey-VCS já faz uso de uma camada com zlib) e diferenças entre XMIs (a especificação já define rótulos específicos para diferenças, mas a maioria das ferramentas não é compatível com esses rótulos). Outra pesquisa importante é em relação a algoritmos rápidos para leitura e escrita de grandes arquivos XMI em estruturas orientadas a objetos e posterior armazenamento em estruturas de dados, como, por exemplo, Árvore B.

7.3 Avaliação do Odyssey-BRD

A avaliação do Odyssey-BRD foca no seu componente principal, o ArchTrace. Visando avaliar a efetividade do ArchTrace e seu conjunto atual de políticas, foi executado um estudo retrospectivo novamente sobre o ambiente Odyssey (WERNER *et al.*, 2003).

Para executar este estudo, foram coletados dados de versionamento do Odyssey entre 9 de julho de 2003 e 1 de março de 2005. Esses dados foram reorganizados e utilizados para reproduzir os *check-ins* originais do Odyssey em um repositório instrumentado com o ArchTrace. Com isso, durante a reprodução dos *check-ins*, foi possível receber no ArchTrace todos os eventos que aconteceram no passado,

reproduzindo o cenário original de desenvolvimento e manutenção. Esta estratégia possibilitou observar se o ArchTrace se comportaria corretamente na evolução de ligações de rastreabilidade caso estivesse em operação no período supracitado.

7.3.1 Planejamento do estudo

Assumindo que software necessariamente evolui, e que ligações de rastreabilidade também devem evoluir em resposta à evolução do software para evitar inconsistências, o objetivo do estudo é analisar a execução do ArchTrace e avaliar a efetividade das políticas em relação à evolução de ligações de rastreabilidade no contexto do projeto Odyssey.

O estudo é composto de quatro passos principais: (1) a definição inicial das ligações de rastreabilidade do Odyssey em 9 de julho de 2003; (2) a reprodução dos *check-ins* do Odyssey de 9 de julho de 2003 e até 1 de março de 2005, que motivam o ArchTrace a evoluir as ligações de rastreabilidade existentes; (3) a definição das ligações de rastreabilidade ideais em 1 de março de 2005; e (4) a comparação entre as ligações de rastreabilidade evoluídas pelo ArchTrace com as ligações de rastreabilidade ideais, definida por engenheiros de software do projeto Odyssey.

O primeiro passo consiste na detecção inicial de ligações de rastreabilidade entre a arquitetura do Odyssey e seu código fonte, ambos datando de 9 de julho de 2003. Este conjunto inicial de ligações de rastreabilidade foi detectado manualmente por engenheiros de software do Odyssey, com apoio parcial de políticas do ArchTrace. Contudo, técnicas existentes de detecção de ligações de rastreabilidade poderiam ser utilizadas para esse fim.

O segundo passo consiste na evolução das ligações de rastreabilidade durante 20 meses de desenvolvimento e manutenção do Odyssey. Este passo foi executado pelas políticas do ArchTrace em resposta à evolução de elementos arquiteturais ou código fonte. Após esse período de desenvolvimento e manutenção, o conjunto inicial de ligações de rastreabilidade se transformou em um novo conjunto de ligações de rastreabilidade evoluídas pelo ArchTrace. Este conjunto é denominado T_e.

O terceiro passo consiste na detecção das ligações de rastreabilidade ideais entre a arquitetura e o código fonte do Odyssey, ambos datando de 1 de março de 2005. Este conjunto de ligações de rastreabilidade ideais, denominado T_i, também foi criado manualmente por engenheiros de software do Odyssey.

Finalmente, o quarto passo consiste na comparação entre o conjunto de ligações de rastreabilidade ideais (T_i) e o conjunto de ligações de rastreabilidade evoluídas pelo ArchTrace (T_e). Esta comparação ($T_i \cap T_e$) pode mostrar o quão precisas as políticas do ArchTrace foram na evolução de ligações de rastreabilidade.

O resultado desde estudo é apresentado descrevendo o efeito de algumas decisões de desenvolvimento e manutenção na evolução das ligações de rastreabilidade. Além disso, a efetividade das políticas é avaliada, mostrando que $|T_i \cap T_e| \div |T_i|$ por cento das ligações de rastreabilidade evoluídas estão corretas, onde |X| é o número de elementos contidos no conjunto X. Na Tabela 7.1, as políticas utilizadas no ArchTrace são sumarizadas para facilitar a leitura do restante da seção.

Tabela 7.1: Sumário das políticas do ArchTrace.

Política	Descrição
1	Sugere ligações de rastreabilidade para a versão mais recente de um IC caso ligações de rastreabilidade tenham sido estabelecidas para versões anteriores.
2	Não permite a criação ou remoção de ligações de rastreabilidade em elementos arquiteturais imutáveis.
3	Não permite a criação de ligações de rastreabilidade para mais de uma versão de um mesmo IC.
4	Não permite a criação de ligações de rastreabilidade para subitens se já existe ligação de rastreabilidade para o IC composto.
5	Remove ligações de rastreabilidade de versões anteriores de um IC se outras ligações de rastreabilidade forem estabelecidas para versões mais recentes.
6	Remove ligações de rastreabilidade de subitens se ligações de rastreabilidade forem estabelecidas para o IC composto.
7	Sugere ligações de rastreabilidade relacionadas quando uma ligação de rastreabilidade é criada.
8	Copia todas as ligações de rastreabilidade existentes para a nova versão do elemento arquitetural.
9	Atualiza automaticamente as ligações de rastreabilidade quando novas versões de um IC estão disponíveis.

7.3.2 Preparação do ambiente

Como discutido anteriormente, o projeto Odyssey faz uso do CVS (FOGEL e BAR, 2001) desde 9 de julho de 2003 para controlar a evolução dos seus artefatos. Contudo, a implementação atual do ArchTrace está apta a monitorar repositórios Subversion. Devido a isto, o primeiro passo na preparação do ambiente do estudo foi converter o repositório CVS em um repositório Subversion. Mais uma vez, este passo foi executado usando a ferramenta cvs2svn.

O repositório do Odyssey é composto de um ramo principal, utilizado para desenvolvimento e manutenções evolutivas, e alguns ramos auxiliares, utilizados para manutenções corretivas e provas de conceito. Para efeito deste estudo, somente o ramo principal foi considerado, visando focar no desenvolvimento e manutenções principais.

A Tabela 7.2 apresenta algumas estatísticas coletadas durante a execução da ferramenta cvs2svn.

Tabela 7.2: Estatísticas do ambiente Odyssey.

Total de arquivos no CVS	2703	Tamanho do repositório CVS	40158 KB
Total de versões no CVS	8463	Total de configurações no Subversion	307
Total de etiquetas no CVS	13	Data da primeira configuração	9 de julho de 2003
Total de ramos no CVS	7	Data da última configuração	1 de março de 2005

A definição da arquitetura do Odyssey e o conjunto inicial de ligações de rastreabilidade foram estabelecidos. A arquitetura do Odyssey foi definida via documentação existente. Além disso, foram restauradas as duas primeiras configurações. A configuração 1 representa o repositório vazio, iniciado pelo cvs2svn. A configuração 2 tem a primeira configuração real do Odyssey, migrada do CVS.

A primeira configuração do Odyssey foi utilizada juntamente com a arquitetura do Odyssey para estabelecer as ligações de rastreabilidade iniciais no ArchTrace. Este conjunto inicial de ligações de rastreabilidade foi detectado de acordo com o conhecimento dos engenheiros de software do Odyssey. Contudo, durante esta configuração inicial das ligações de rastreabilidade, a política 7 foi acionada, sugerindo ligações de rastreabilidade relacionadas após a execução de mineração de dados (algoritmo Apriori) sobre as ligações de rastreabilidade existentes, como mostrado na Figura 7.7.

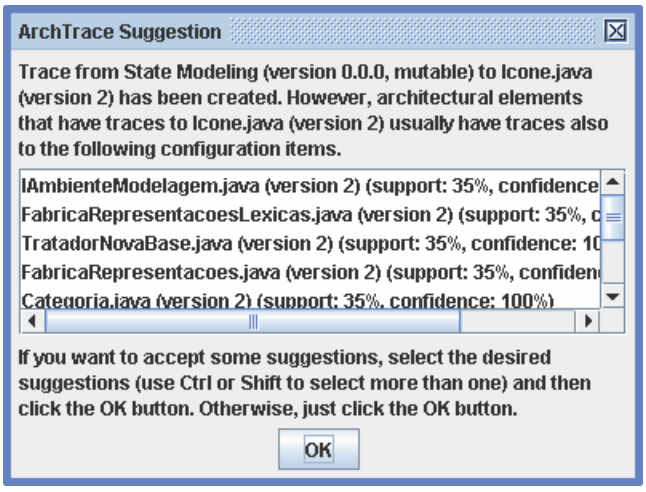


Figura 7.7: Detecção de ligações de rastreabilidade no ArchTrace.

Finalmente, todas as políticas foram habilitadas, com exceção das políticas 1, 3 e 7. A política 3 não foi projetada para trabalhar em conjunto com as políticas 5 e 9. Além disso, as políticas 1 e 7 foram projetadas para trabalhar de forma interativa e necessitam intervenção manual do usuário. Estas situações são evitadas no estudo para focar na

atualização automática de ligações de rastreabilidade. Neste momento, o ArchTrace está pronto para monitorar os 305 *check-ins* remanescentes.

7.3.3 Coleta de estatísticas

Este estudo retrospectivo visa analisar diferentes medidas coletadas na execução do ArchTrace. A coleta manual de medidas é suscetível a erros e demorada. Por outro lado, a coleta automática de medidas é usualmente feita via instrumentação de código fonte. Contudo, essa coleta automática de medidas pode ser vista como uma preocupação ortogonal ao projeto do ArchTrace, afetando diferentes partes do código fonte. Para resolver esse problema, foi utilizada a programação orientada a aspectos (KICZALES *et al.*, 1997), evitando uma instrumentação intrusiva.

O aspecto de coleta de medidas, que foi combinado com o ArchTrace, é composto de 19 pontos de interseção (*pointcuts*) que coletam as seguintes medidas (27 no total) para cada *check-in*: o número, autor e data da configuração; o número de ICs adicionados, removidos e modificados; o número de execuções de cada uma das 9 políticas; o número de ligações de rastreabilidade adicionadas e removidas manualmente; o número de ligações de rastreabilidade adicionadas e removidas automaticamente; o número de adições e remoções de ligações de rastreabilidade perdidas; o número de ligações de rastreabilidade indiretamente adicionadas e removidas manualmente; o número de ligações de rastreabilidade indiretamente adicionadas e removidas automaticamente; e o número de adições e remoções de ligações de rastreabilidade perdidas indiretamente;

Nesse contexto, ligações de rastreabilidade indiretas são ligações de rastreabilidade implicitamente detectadas quando uma dada ligação de rastreabilidade é estabelecida para um artefato composto. Por exemplo, caso uma ligação de rastreabilidade seja estabelecida para um diretório, todos os arquivos e subdiretórios dentro desse diretório são, também, recursivamente ligados. Usualmente, o efeito de perder uma ligação de rastreabilidade de um artefato composto é pior quando esse artefato agrega muitos outros artefatos. Esta situação pode diminuir consideravelmente a cobertura total da arquitetura sobre o código fonte.

7.3.4 Execução do estudo

A execução do estudo é composta por dois passos principais: (1) a reprodução dos *check-ins* e (2) a análise das ligações de rastreabilidade perdidas. O primeiro passo é

executado depois da decomposição do repositório original nos *check-ins* individuais. Cada *check-in* individual é reproduzido no repositório monitorado pelo ArchTrace, permitindo que o aspecto de coleta de medidas possa atuar, coletando todas as medidas necessárias. Este passo é apoiado por uma ferramenta projetada especialmente para a automação do estudo, apresentada na Figura 7.8.

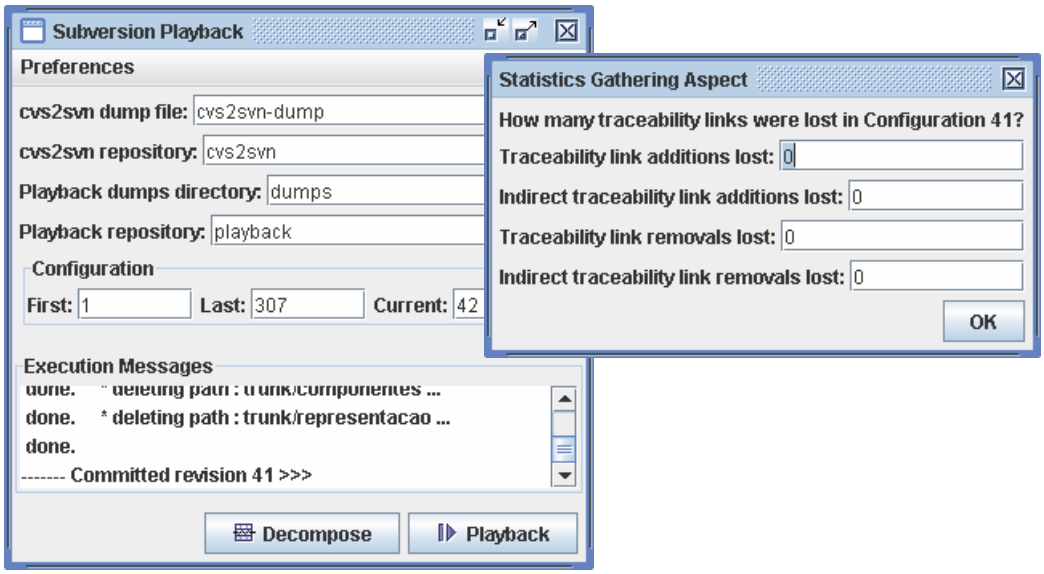


Figura 7.8: Reprodução incremental dos *check-ins*.

O segundo passo é executado após cada *check-in* individual ser reproduzido. Quando um *check-in* individual é reproduzido, o ArchTrace evolui as ligações de rastreabilidade existentes, e o especialista (engenheiro de software do Odyssey) verifica se existem ligações de rastreabilidade perdidas. Dois tipos de ligações de rastreabilidade devem ser informadas ao aspecto de coleta de medidas, mostrado na Figura 7.8: (1) adições perdidas, ou seja, ligações de rastreabilidade que idealmente existem, mas não foram detectadas pelo ArchTrace, e (2) remoções perdidas, ou seja, ligações de rastreabilidade que não existem idealmente, mas que foram incorretamente detectadas pelo ArchTrace. As ligações de rastreabilidade indiretas, mostradas na Figura 7.8, representam as ligações de rastreabilidade que deveriam ter sido adicionadas ou removidas recursivamente, quando outra ligação de rastreabilidade relacionada a um artefato composto foi adicionada ou removida.

Durante a execução do estudo, foram feitas três liberações principais do Odyssey: 1.0.0, 1.1.0, e 1.2.0. Quando uma liberação é feita, as versões atuais dos componentes do Odyssey são transformadas em imutáveis e novas versões desses componentes são criadas. Este procedimento permite que análises futuras da arquitetura

do Odyssey possam ser executadas, identificando claramente qual versão de cada componente fazia parte de cada liberação do ambiente Odyssey como um todo.

Além disso, quatro componentes foram adicionados à arquitetura do Odyssey após 9 de julho de 2003. Junto com esses componentes, um conjunto inicial de ligações de rastreabilidade foi, também, manualmente estabelecido pelos engenheiros de software do Odyssey. Após esse momento, somente o ArchTrace foi utilizado para evoluir as ligações de rastreabilidade estabelecidas.

7.3.5 Análise dos resultados

Durante os 20 meses de desenvolvimento e manutenção do Odyssey, 3031 arquivos foram adicionados, renomeados ou movidos, 154 arquivos foram removidos e 1563 modificações foram feitas sobre os arquivos existentes. Como mostrado na Figura 7.9, a maioria dos ICs foi adicionada em julho de 2003. Contudo, alguns ICs também foram adicionados e removidos em novembro de 2003. Este cenário ocorreu devido a uma reorganização do Odyssey. Após novembro de 2003, a maioria das atividades estava relacionada com modificações sobre ICs existentes, com somente uma pequena quantidade de adição ou remoção de ICs.

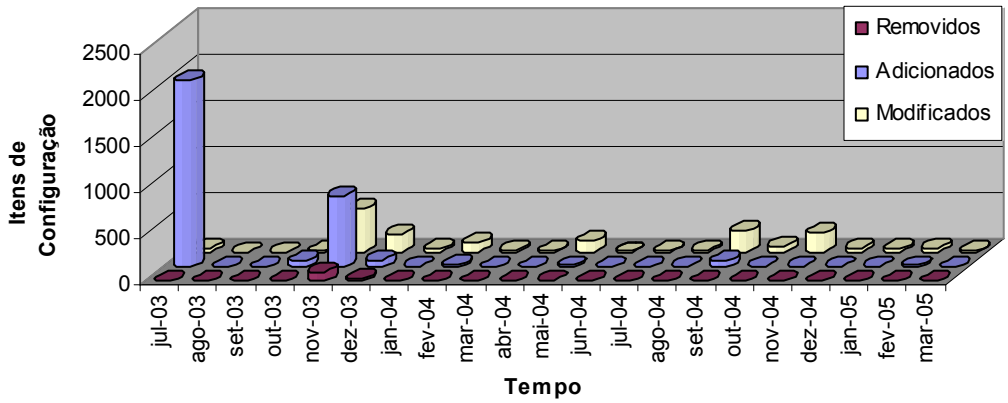


Figura 7.9: Evolução dos ICs.

As políticas do ArchTrace foram ativadas em resposta a evolução da arquitetura ou da implementação. A Figura 7.10 mostra que a política 7, que é responsável pela mineração de dados sobre as ligações de rastreabilidade existentes, foi acionada em julho de 2003 como uma resposta para a adição manual das ligações de rastreabilidade iniciais, como mostrado na Figura 7.11.

Além disso, a política 8, que é responsável por copiar as ligações de rastreabilidade existentes para novas versões de elementos arquiteturais, foi acionada em maio, agosto e setembro de 2004, devido às liberações do Odyssey, como mostrado

na Figura 7.10. Após as liberações do Odyssey, a política 2 foi acionada para evitar a evolução de ligações de rastreabilidade relacionadas a elementos arquiteturais imutáveis, como mostrado na Figura 7.10. Esses elementos arquiteturais imutáveis não devem ter as suas ligações de rastreabilidade modificadas porque eles pertencem a liberações já efetuadas do Odyssey.

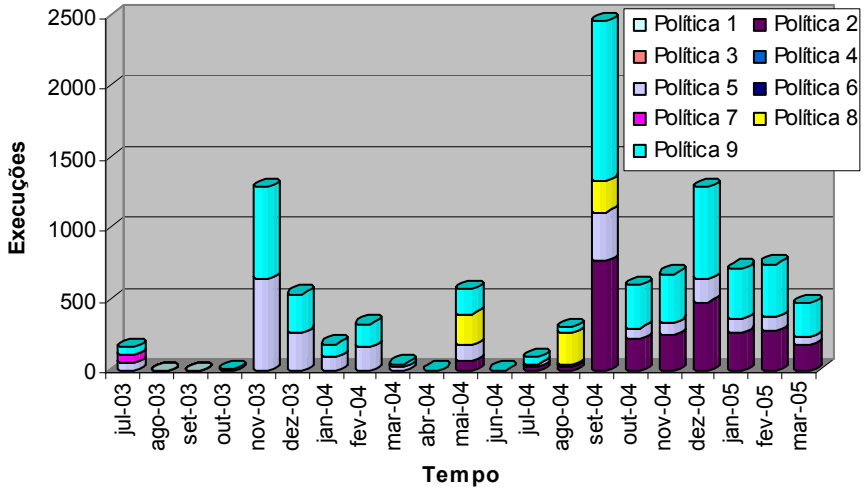


Figura 7.10: Execuções de políticas.

Finalmente, um cenário importante aconteceu em novembro de 2003, quando uma reorganização do Odyssey foi executada. Essa reorganização afetou o nome de pacote e a localização de classes existentes. Contudo, as políticas 5 e 9 foram capazes de lidar com a situação e atualizar as ligações de rastreabilidade para refletir a nova organização do código fonte. Este cenário é composto por três fases: (1) a reorganização sendo feita no nível do código fonte (Figura 7.9), (2) as políticas sendo acionadas para lidar com a situação (Figura 7.10), e (3) as ligações de rastreabilidade sendo reorganizadas para aderir à nova organização do código fonte (Figura 7.11).

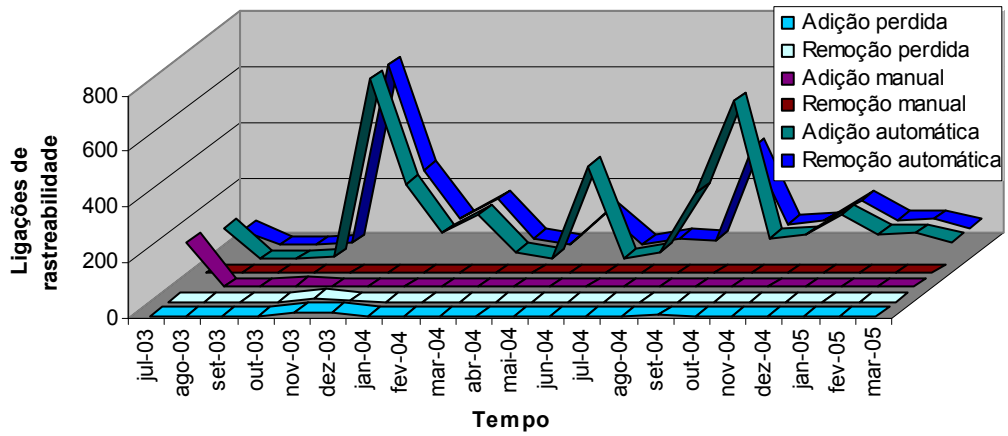
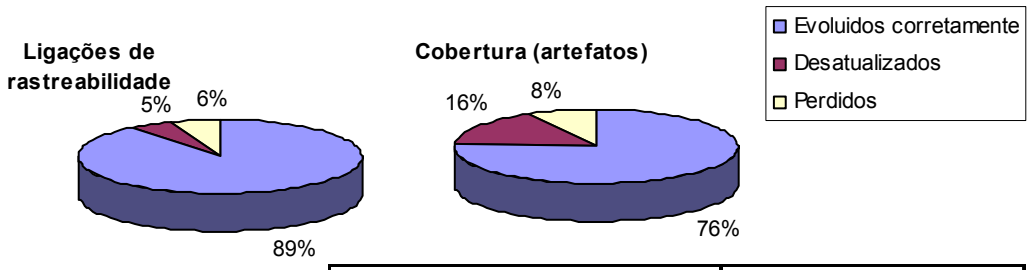


Figura 7.11: Evolução das ligações de rastreabilidade.

Finalmente, para concluir o estudo, foi comparado o conjunto de ligações de rastreabilidade evoluídas pelo ArchTrace (T_e) com o conjunto de ligações de rastreabilidade ideais (T_i), detectado por engenheiros de software do Odyssey. T_e contém um total de 222 ligações de rastreabilidade e uma cobertura de 638 artefatos. Por outro lado, T_i contém 235 ligações de rastreabilidade e uma cobertura de 691 artefatos.

A Figura 7.12 sumariza os resultados da análise, ilustrando que, após os 20 meses de evolução, o conjunto de ligações de rastreabilidade evoluídas pelo ArchTrace (T_e) tem 12 ligações de rastreabilidade desatualizadas, afetando 113 artefatos. Além disso, 13 ligações de rastreabilidade foram perdidas (|Ti-Te|), afetando 53 artefatos devido a algumas dessas ligações de rastreabilidade perdidas serem referentes a artefatos compostos (i.e.: diretórios). No final, as políticas do ArchTrace corretamente identificaram 89% do conjunto de ligações de rastreabilidade ideais e rastrearam 76% do código fonte para os elementos arquiteturais correspondentes no contexto do projeto Odyssey.



	Ligações de rastreabilidade		Cobertura (artefatos)	
	#	%	#	%
Evoluídos corretamente	210	89%	525	76%
Desatualizados	12	5%	113	16%
Perdidos	13	6%	53	8%
Total	235	100%	691	100%

Figura 7.12: Sumário da análise.

Para colocar este resultado numa perspectiva padrão, duas medidas de recuperação de informação foram utilizadas (BAEZA-YATES e RIBEIRO-NETO, 1999): precisão (*precision*), que representa a fração dos documentos recuperados que são relevantes, e revocação (*recall*), que representa a fração dos documentos relevantes que foram recuperados. No contexto desde estudo, a precisão mostra o percentual das ligações de rastreabilidade detectadas que é correto ($|T_i \cap T_e| \div |T_e| = 95\%$; indicando que 5% das ligações de rastreabilidade encontradas são incorretas) e a revocação mostra o percentual das ligações de rastreabilidade ideais que foi corretamente evoluído

($|T_i \cap T_e| \div |T_i| = 89\%$; indicando que somente 11% das ligações de rastreabilidade foram perdidas).

7.3.6 Considerações finais sobre a avaliação do Odyssey-BRD

A análise mostrou que o ArchTrace operou quase sempre de forma correta, mesmo durante uma reorganização do Odyssey. Todavia, parte das ligações de rastreabilidade ficou desatualizada. Este problema ocorreu devido à remoção equivocada de um diretório, que foi restabelecido em configurações posteriores. Por esta razão, o ArchTrace não foi capaz de detectar a evolução dos rastros de modificação relacionados, mas foi possível manter as ligações de rastreabilidade para os artefatos antes da reorganização, devido a dimensão de versionamento. Sendo assim, seria possível restabelecer as ligações de rastreabilidade manualmente para a versão mais atual. Caso contrário, essas ligações de rastreabilidade seriam perdidas completamente.

Outras ligações de rastreabilidade foram perdidas quando novos artefatos foram introduzidos fora do contexto de artefatos existentes. Nessas situações, políticas de mineração de dados não são úteis devido à falta de informação prévia em relação a esses novos artefatos. Uma possível solução para atenuar este problema seria a construção de políticas que fazem uso de técnicas de recuperação de informação (DE LUCIA *et al.*, 2004) ou análise sintática (BRIAND *et al.*, 2003) para detectar ligações de rastreabilidade. Essas políticas não dependem do histórico do artefato. Elas utilizam, respectivamente, os termos e a estrutura sintática dos artefatos para sugerir ligações de rastreabilidade candidatas. Contudo, o uso isolado dessas políticas, sem a colaboração com as políticas existentes do ArchTrace, pode levar a uma alta quantidade de ligações de rastreabilidade perdidas quando reorganizações do código, uso incorreto de termos ou erros de projeto acontecerem.

Vale também notar que as políticas atualmente implementadas no ArchTrace atuam de maneira conservativa. Somente as ligações de rastreabilidade corretas são estabelecidas automaticamente. Quando uma ligação de rastreabilidade detectada pelas políticas do ArchTrace não é 100% correta, as políticas interagem com o usuário para verificar se a ligação de rastreabilidade deve ser realmente estabelecida. Contudo, a configuração do ArchTrace para este estudo somente habilitou políticas não interativas após o estabelecimento das ligações de rastreabilidade iniciais. Desta forma, teoricamente, os resultados poderiam ser ainda melhores, caso essas políticas interativas

fossem habilitadas. Além disso, cada ligação de rastreabilidade detectada é registrada para permitir auditorias futuras.

Finalmente, o Odyssey é um sistema relativamente estável, que já estava em desenvolvimento por alguns anos quando o estudo se iniciou. Não está clara a contribuição do conjunto atual de políticas no caso de um projeto novo, sendo monitorado desde o seu nascimento.

Além do ArchTrace, o módulo Odyssey-BRD é, também, composto pelo componente de Carga Dinâmica, que está em operação no ambiente Odyssey desde 20 de outubro de 2003. Neste período, 7 liberações candidatas foram feitas até que em 9 de maio de 2005 foi feita a sua primeira liberação estável. Desde então, outras 5 liberações estáveis foram feitas. A liberação mais recente totaliza mais de 15 componentes, carregados dinamicamente no Odyssey, sendo que esses componentes dependem de mais de 40 bibliotecas diferentes, também carregadas por demanda.

7.4 Avaliação do Odyssey-WI

Esta seção apresenta uma avaliação do desempenho de detecção de rastros de modificação da abordagem Odyssey-WI. O desempenho de detecção de rastros de modificação é avaliado objetivamente por meio de algumas medidas tradicionais de recuperação de informação, que indicam o quão precisa a abordagem é em termos de quantos rastros de modificação corretos foram detectados (quanto maior, melhor) e quantos rastros de modificação incorretos foram detectados (quanto menor, melhor). Espera-se que uma boa abordagem recupere um grande número de rastros de modificação sem adicionar ruído (rastros de modificação incorretos) no resultado.

ZIMMERMANN et al. (2004) avaliaram o desempenho de detecção de rastros de modificação entre arquivos. O objetivo desta avaliação é complementar o trabalho de ZIMMERMANN et al. (2004), focando em rastros de modificação entre elementos de modelo. Novamente, esta avaliação consiste em um estudo retrospectivo sobre o ambiente Odyssey (WERNER *et al.*, 2003).

O foco do Odyssey-WI é na detecção de rastros de modificação em modelos UML. Contudo, o Odyssey não tem versões completas do seu modelo UML disponíveis. Por esta razão, mais uma vez, uma estratégia de engenharia reversa foi utilizada para reconstruir os modelos UML referentes a cada configuração existente do código fonte. Esta estratégia permite que a avaliação possa ser feita em curto espaço de tempo e sobre um projeto real. Caso contrário, seria necessário um longo espaço de

tempo para obter uma quantidade suficiente de versões de modelos UML em um projeto real. Uma outra vantagem importante dessa estratégia é em relação ao viés. O estudo não tinha sido nem projetado durante o desenvolvimento do Odyssey. Por esta razão, o estudo não influenciou ações tomadas naquele momento.

7.4.1 Preparação do ambiente

Para executar o estudo, foram coletados dados de versionamento do Odyssey durante o período de 26 de novembro de 2003 até 1 de março de 2005, totalizando mais de 15 meses de desenvolvimento e contendo 264 configurações (*check-ins*). Esses dados foram reorganizados, submetidos a um processo de engenharia reversa e divididos em dois conjuntos: *conjunto de treinamento* e *conjunto de avaliação*. O *conjunto de treinamento* é composto de versões datando entre 26 de novembro de 2003 e 5 de setembro de 2004, totalizando mais de 9 meses de desenvolvimento. O *conjunto de avaliação* é composto de versões datando entre 8 de setembro de 2004 e 1 de março de 2005, totalizando em torno de 6 meses de desenvolvimento.

Algumas restrições foram aplicadas sobre os dados históricos do Odyssey antes de executar a avaliação para reduzir possíveis fatores de confusão[26]. Em primeiro lugar, somente classes UML foram processadas. Classes são compostas por atributos e operações. Quando uma operação é modificada, a classe que contém a operação também o é, indiretamente. Esta regra de composição poderia incorretamente aumentar o desempenho da abordagem. Por exemplo, é inútil saber que "a classe C é modificada sempre que o atributo A da classe C é modificado" (essa informação é obvia). Uma situação análoga ocorre em relação a pacotes e classes. Por essa razão, estruturas de composição foram evitadas durante a avaliação, focando somente em elementos do mesmo tipo: classes. Contudo, é importante frisar que a abordagem em si não é restrita a um determinado tipo de elemento. Finalmente, somente elementos de modelo que foram modificados ao menos uma vez no *conjunto de treinamento* e no *conjunto de avaliação* foram processados.

[26] Um fator de confusão (*confounding factor*) é um fator que dificulta a diferenciação entre o efeito proveniente de um fator e o efeito proveniente de outro fator (TRAVASSOS *et al.*, 2002).

7.4.2 Execução do estudo

Depois desta fase de preparação, o Odyssey-WI foi executado sobre o *conjunto de treinamento* para cada elemento de modelo. Os rastros de modificação detectados representam predições providas pela ferramenta em 5 de setembro de 2004. Essas predições foram comparadas com as modificações reais, contidas no *conjunto de avaliação*, visando verificar se as predições realmente ocorreram em algum momento entre 8 de setembro de 2004 e 1 de março de 2005. Os resultados são apresentados em termos de algumas medidas amplamente utilizadas em recuperação de informação (BAEZA-YATES e RIBEIRO-NETO, 1999): precisão, revocação, precisão-R (*R-precision*) e média harmônica. Precisão indica quanto as predições foram corretas e revocação indica quanto os rastros de modificação foram preditos. O restante das medidas provê uma noção de precisão depois que um número fixo de rastros de modificações foi detectado (precisão-R) e o melhor compromisso entre precisão e revocação para determinados valores de suporte e confiança (média harmônica).

A avaliação foi executada de acordo com os procedimentos de "Avaliação de Desempenho de Recuperação" descritos por BAEZA-YATES et al. (1999).

7.4.3 Análise dos resultados

Inicialmente, uma análise de precisão x revocação foi feita em relação aos rastros de modificação detectados. Neste contexto, a medida de precisão consiste na fração relevante dos rastros de modificação detectados. Por outro lado, a medida de revocação consiste na fração detectada dos rastros de modificação relevantes:

$$Precisão = \frac{|Relevantes \cap Detectados|}{|Detectados|} \qquad Revocação = \frac{|Relevantes \cap Detectados|}{|Relevantes|}$$

A ferramenta Odyssey-WI foi consultada para cada elemento de modelo no *conjunto de treinamento*. O resultado (rastros de modificação detectados) foi ordenado decrescentemente pelos valores de suporte e confiança (rastros de modificação mais relevantes em primeiro lugar). Depois disso, os valores de precisão e revocação foram calculados para cada rastro de modificação relevante, iniciando pelo topo do resultado ordenado. Um rastro de modificação é considerado relevante se o elemento de modelo rastreado também foi modificado em conjunto com o elemento de modelo consultado no *conjunto de avaliação*.

Em seguida, um procedimento de interpolação foi aplicado para determinar os valores de precisão para os 11 níveis padrões de revocação, que vão de 0 a 1, com

intervalos de 0,1. A função de interpolação estabelece a precisão em um dado nível de revocação como sendo o valor máximo de precisão entre o nível dado e o próximo nível de revocação, inclusive:

$$Precisão(Revocação[nível]) = \max_{Revocação[nível] \le Revocação \le Revocação[nível+1]} Precisão(Revocação)$$

Finalmente, os valores já calculados de precisão e revocação para cada consulta individual foram combinados via uma função de média. Esta função consiste em estabelecer um valor médio de precisão para cada nível de revocação:

$$\overline{Precisão(Revocação[nível])} = \sum_{consulta=1}^{|consultas|} \frac{Precisão_{consulta}(Revocação[nível])}{|consultas|}$$

Os resultados da análise de precisão x revocação são apresentados na Figura 7.13.a. Esta curva mostra que uma precisão em torno de 65% foi obtida em 10% de revocação. Em outras palavras, 2 em cada 3 predições do Odyssey-WI foram corretas para detectar os primeiros 10% dos rastros de modificação relevantes. Além disso, uma precisão em torno de 33% foi obtida em 50% de revocação. Finalmente, uma precisão em torno de 4% foi obtida em 100% de revocação. Esses resultados são notáveis se comparados com os resultados existentes, em relação à detecção de rastros de modificação em arquivos. Por exemplo, os resultados médios em relação à IDE Eclipse são precisão em torno de 30% (YING *et al.*, 2004) ou 26% (ZIMMERMANN *et al.*, 2004) em 15% de revocação. No caso do Odyssey-WI, foi atingido 60% de precisão em 15% de revocação.

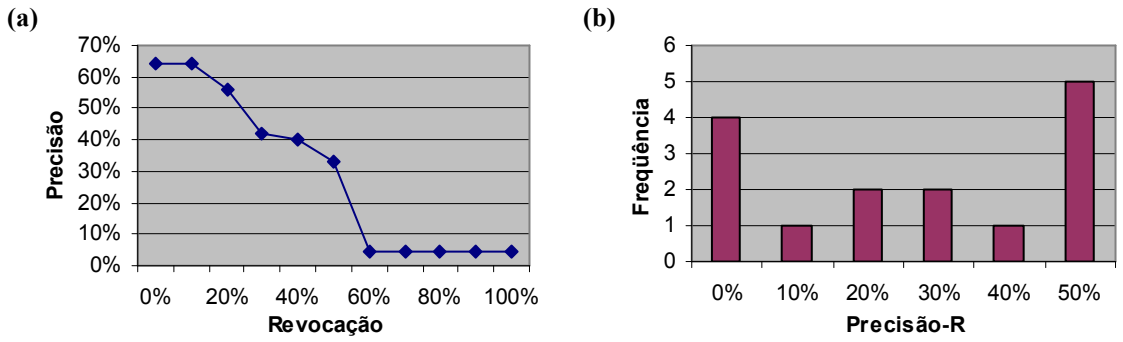


Figura 7.13: Curva de precisão x revocação (a) e histograma de precisão-R (b).

Independentemente da importância da curva de precisão x revocação média como estratégia padrão de avaliação, também é desejado visualizar os resultados em um único valor numérico. A medida precisão-R computa a precisão na posição R da lista ordenada de rastros de modificação detectados, onde R é o número total de rastros de modificação relevantes para uma dada consulta. Por exemplo, se um elemento

consultado tem um total de 5 rastros de modificações relevantes (R = 5), mas somente 1 desses elementos está contido nos 5 primeiros elementos detectados, então precisão-R = 20%. Após calcular a precisão-R para cada consulta individual, um histograma foi desenhado para exibir o número de consultas que pertence a intervalos específicos de precisão-R. O histograma, apresentado na Figura 7.13.b, mostra que 5 consultas conseguiram detectar metade dos rastros de modificação relevantes dentre os primeiros R rastros de modificação recuperados.

Finalmente, é importante conhecer os melhores valores de suporte e confiança, no contexto do projeto Odyssey, a serem utilizados na configuração do Odyssey-WI para produzir combinações ótimas de precisão e revocação. A medida de média harmônica pode ajudar nessa tarefa, fornecendo o melhor compromisso entre precisão e revocação para determinados valores de suporte e confiança:

$$MédiaHarmônica = \frac{2}{\frac{1}{Revocação} + \frac{1}{Precisão}}$$

A média harmônica de precisão e revocação é desenhada na Figura 7.14 para valores específicos de suporte e confiança. A Figura 7.14 mostra que o valor máximo da média harmônica, que indica o melhor compromisso entre precisão e revocação, é obtido para baixos valores de suporte e confiança. Esse tipo de análise deve ser feito para cada sistema alvo, visando apoiar a configuração de valores de suporte e confiança no Odyssey-WI.

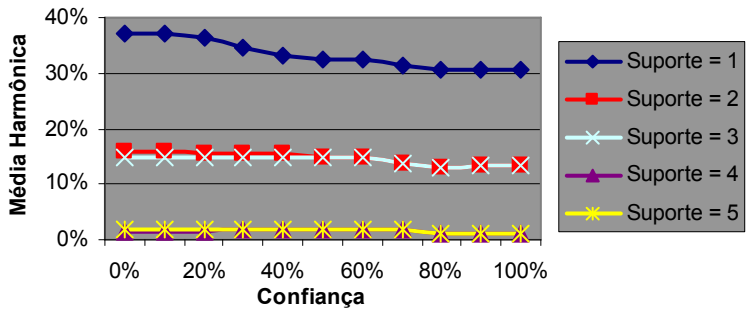


Figura 7.14: Média harmônica entre precisão e revocação em termos de suporte e confiança.

O sumário das medidas discutidas nesta seção é apresentado na Tabela 7.3 para cada consulta com suporte mínimo igual a 1 e qualquer confiança. A Tabela 7.3 exibe o número da consulta, o número de rastros de modificação relevantes, o número de rastros de modificação detectados, o número de rastros de modificação relevantes que foram detectados, e as medidas de precisão, revocação, precisão-R e média harmônica entre precisão e revocação.

Tabela 7.3: Sumário das medidas coletadas.

Consulta	Relevantes	Detectados	Rel. ∩ Det.	Prec.	Revoc.	Prec.-R	M. Harm.
1	3	3	0	0%	0%	0%	0%
2	12	5	3	60%	25%	25%	35%
3	10	7	2	29%	20%	20%	24%
4	6	3	3	100%	50%	50%	67%
5	6	3	3	100%	50%	50%	67%
6	5	3	2	67%	40%	40%	50%
7	2	3	2	67%	100%	50%	80%
8	11	0	0	0%	0%	0%	0%
9	11	2	2	100%	18%	18%	31%
10	4	2	2	100%	50%	50%	67%
11	4	2	2	100%	50%	50%	67%
12	12	4	3	75%	25%	25%	38%
13	3	12	1	8%	33%	0%	13%
14	0	19	0	0%	0%	0%	0%
15	9	19	3	16%	33%	11%	21%

7.4.4 Considerações finais sobre a avaliação do Odyssey-WI

Durante a fase de preparação do ambiente, um processo de engenharia reversa foi executado para converter cada configuração do código do ambiente Odyssey em modelos UML. Após essa conversão, os modelos UML foram armazenados no Odyssey-VCS. As 160 configurações de código fonte originais, utilizadas para criar o *conjunto de treinamento*, e as 104 configurações de código fonte originais, utilizadas para criar o *conjunto de avaliação*, foram convertidas, respectivamente, em 38 e 34 configurações de modelos UML após a engenharia reversa.

Isto ocorre porque alguns *check-ins* de código fonte não representam modificações nos modelos UML. Por exemplo, quando o código de um método é modificado, essa modificação pode manipular somente classes já importadas pela classe que está sendo modificada. Neste caso, não estão sendo adicionadas ou removidas dependências no modelo UML. Consequentemente, não é gerada uma nova configuração do modelo UML. No caso deste estudo, o modelo UML foi alterado a cada 3,66 modificações no código fonte. Isto significa que existe menos informação histórica disponível para um dado período de tempo no caso de modelos. Como conseqüência, somente modificações que realmente afetaram a estrutura de alto nível do sistema foram consideradas. Isto pode explicar as diferenças de precisão e revocação do Odyssey-WI se comparado com abordagens existentes, focadas em arquivos. Além disso, a abordagem Odyssey-WI foi idealizada para operar durante as fases de análise e projeto, quando ainda não existe código fonte disponível. Neste cenário, a detecção de rastros de modificação só pode ser executada sobre artefatos de alto nível de abstração (e.g.: modelos UML).

7.5 Considerações finais

Este capítulo apresentou uma avaliação do Odyssey-SCM, com foco nas subabordagens Odyssey-VCS, Odyssey-BRD e Odyssey-WI. Esta avaliação teve ênfase em aspectos de desempenho e escalabilidade. Os resultados iniciais são promissores, contudo, avaliações complementares sob outras dimensões do problema se tornam fundamentais caso esse protótipo acadêmico venha a ser transformado em produto.

No caso do Odyssey-VCS, foi identificada a necessidade de continuidade em pesquisas buscando formas eficientes de representação e transformação de dados semi-estruturados em estruturas orientadas a objetos. O maior gargalo identificado foi o processamento de arquivos XMI contendo modelos UML grandes. Tanto a leitura quanto a reconstrução desses arquivos demandam uma carga de processamento que inviabiliza a utilização prática da ferramenta.

Em relação ao Odyssey-BRD, as políticas atuais do ArchTrace se mostraram capazes de evoluir as ligações de rastreabilidade do Odyssey por um longo período de tempo. Entretanto, essas políticas não foram suficientes quando houve a remoção equivocada e posterior restabelecimento de diretórios, situação comum em projetos menos estáveis. Desta forma, novas políticas poderiam ser projetadas para tratar a inserção de elementos novos.

Quanto ao Odyssey-WI, os resultados obtidos foram superiores aos existentes, que tratam de código fonte. Uma explicação possível é o fato de modelos UML serem menos suscetíveis a pequenas modificações, se comparados com código fonte, somente considerando modificações que realmente afetam a estrutura de alto nível do sistema.

A avaliação sobre o Odyssey-CCS, rodando o processo Odyssey-SCMP, foi postergada para trabalhos futuros devido à necessidade de ambientes reais de DBC para a sua execução. Diferentemente das demais subabordagens, o Odyssey-CCS e o Odyssey-SCMP lidam com processos específicos, que influenciam na forma de trabalho dos engenheiros de software. Por esse motivo, qualquer tipo de estudo retrospectivo sobre repositórios prévios à existência do processo é inviabilizado.

Capítulo 8 – Conclusão

8.1 Epílogo

O DBC, apesar de ser um paradigma novo, vem se mostrando importante para o tratamento do crescente aumento de complexidade no processo de desenvolvimento de software. Dentre as suas principais características, estão a utilização de componentes, interfaces e conectores para descrever sistemas, a substituição de componentes e a reutilização como atividade central do processo de desenvolvimento.

Contudo, a evolução de software em cenários de DBC se mostra um desafio ainda maior se comparado com os cenários convencionais de desenvolvimento de software. Fatores como a existência de diversas equipes no ciclo produtivo de componentes, a necessidade de adaptar os componentes para consumidores específicos e o volume de itens a serem gerenciados fazem com que a identificação dos responsáveis pela implementação de modificações e o próprio controle sobre essas modificações sejam dificultados.

A GCS vem sendo utilizada no apoio à evolução de sistemas convencionais, gerando resultados positivos em termos de aumento de qualidade e produtividade. Esses resultados podem ser ainda melhores em cenários onde a demanda por controle é vital, como é o caso do DBC.

Entretanto, as técnicas de GCS utilizadas para sistemas convencionais não são adequadas para o cenário de DBC. Nos sistemas convencionais, a GCS é muito focada em arquivos, que armazenam o código fonte dos sistemas. Já no cenário de DBC, existe uma preocupação ainda maior com os modelos que especificam e projetam a arquitetura de um domínio de aplicação como um todo e dos componentes específicos desse domínio.

Visando minimizar esse problema, este trabalho de pesquisa apresentou uma infra-estrutura, denominada Odyssey-SCM, que faz uso de processos e técnicas de GCS projetadas especialmente para o cenário de DBC.

8.2 Contribuições

Esta tese apresentou os resultados de um trabalho de pesquisa que visa a aplicação de técnicas de GCS em cenários de DBC. Como discutido anteriormente, este

trabalho envolveu, além desta tese de doutorado, outras três dissertações de mestrado. As principais contribuições do trabalho de pesquisa como um todo são:

- Estudo das áreas de DBC e GCS, visando identificar as limitações existentes no apoio de GCS dado ao DBC;
- Definição de uma arquitetura integrada para GCS, visando prover apoio a cenários de DBC;
- Definição de um processo de GCS voltado para questões específicas do DBC;
- Especificação e implementação de um sistema de controle de modificações flexível, que possibilita a configuração do processo e das informações a serem coletadas durante a execução do processo;
- Especificação e implementação de um mapa de reutilização, que possibilita a coleta e consulta sobre informações contratuais de reutilização de componentes;
- Especificação e implementação de um sistema de controle de versões que atua em granularidade fina sobre modelos UML, possibilitando a configuração das unidades de versionamento e comparação para cada elemento de modelo;
- Especificação e implementação de mecanismos para definição de ligações de rastreabilidade entre elementos arquiteturais e código fonte, juntamente com nove políticas que automatizam a evolução dessas ligações de rastreabilidade em resposta a modificações na arquitetura ou na sua implementação;
- Especificação e implementação de mecanismos para a propagação de comandos de GCS dos níveis arquiteturais para o código fonte, por meio do uso das ligações de rastreabilidade previamente estabelecidas;
- Especificação e implementação de uma infra-estrutura para carga de componentes por demanda no ambiente de produção, sem que seja necessário reiniciar o ambiente de produção;
- Especificação e implementação de um mecanismo para detecção de rastros de modificação entre elementos de modelo UML, via aplicação de mineração de dados;

- Contextualização dos rastros de modificação minerados por meio da coleta de informações oriundas da integração dos sistemas de controle de versões e de controle de modificações; e
- Avaliação do desempenho do sistema de controle de versões, do mecanismo de evolução de ligações de rastreabilidade e do mecanismo para detecção de rastros de modificação.

Dessas contribuições globais, as seguintes contribuições dizem respeito especificamente a esta tese de doutorado:

- Estudo das áreas de DBC e GCS, visando identificar as limitações existentes no apoio de GCS dado ao DBC;
- Definição de uma arquitetura integrada de GCS, visando prover apoio a cenários de DBC;
- Definição de um processo de GCS voltado para questões específicas do DBC;
- Especificação dos requisitos de alto nível para cada sistema individual, visando atender às necessidades estabelecidas na arquitetura integrada de GCS;
- Especificação e implementação de componentes de apoio específicos para os sistemas individuais (e.g.: componente de execução de processos SPEM do sistema de controle de modificações, algoritmo de junção de elementos de modelo UML do sistema de controle de versões, componente de mineração de dados do mecanismo de detecção de rastros de modificação e componente de importação e exportação de XMI no ambiente Odyssey);
- Especificação e implementação de mecanismos para definição de ligações de rastreabilidade entre elementos arquiteturais e código fonte, juntamente com nove políticas que automatizam a evolução dessas ligações de rastreabilidade em resposta a modificações na arquitetura ou na sua implementação;
- Especificação e implementação de mecanismos para a propagação de comandos de GCS dos níveis arquiteturais para o código fonte, por meio do uso das ligações de rastreabilidade previamente estabelecidas;
- Especificação e implementação de uma infra-estrutura para carga de componentes por demanda no ambiente de produção, sem que seja necessário reiniciar o ambiente de produção; e

– Avaliação do desempenho do sistema de controle de versões, do mecanismo de evolução de ligações de rastreabilidade e do mecanismo para detecção de rastros de modificação.

8.3 Limitações

A partir de uma análise crítica sobre a abordagem proposta e sua implementação, puderam ser identificadas diversas limitações. Algumas dessas limitações, que se relacionam com decisões tomadas durante o desenvolvimento da abordagem, são detalhadas nesta seção.

8.3.1 Integração entre Odyssey-CCS e Odyssey-VCS

Atualmente, o Odyssey-SCM tem uma funcionalidade que permite ao engenheiro de software informar sobre qual modificação ele está trabalhando. A partir daí, todo e qualquer *check-in* fica associado à modificação informada. Desta forma, todas as versões de ICs adicionadas ou modificadas nesses *check-ins* ficam indiretamente relacionadas com a modificação.

Como discutido anteriormente, esse tipo de funcionalidade permite a execução de consultas bilaterais entre os sistemas de controle de modificação e de controle de versão. Por exemplo, é possível descobrir, para uma dada modificação, quais ICs foram efetivamente alterados. Por outro lado, é possível descobrir, para uma dada versão de IC, qual a razão da sua criação. Por exemplo, o Odyssey-WI faz uso dessa funcionalidade para contextualizar os rastros de modificação detectados.

Contudo, o Odyssey-SCM permite que qualquer engenheiro de software informe a qualquer momento que está trabalhando sobre uma dada modificação. O ideal seria restringir essa funcionalidade tanto em termos do momento quanto em termos de acesso. Como o Odyssey-CCS tem informações sobre as atividades em andamento e sobre os engenheiros de software autorizados a executar essas atividades, seria interessante só permitir *check-ins* em atividades demarcadas como “implementação” e verificar se o engenheiro de software que está efetuando o *check-in* está realmente autorizado a executar a atividade. Desta forma, a integração entre o Odyssey-CCS e o Odyssey-VCS passaria do nível de auditoria para controle.

8.3.2 Integração entre Odyssey-BRD e Odyssey-VCS

Atualmente, o Odyssey-BRD, mais especificamente o ArchTrace, permite que elementos arquiteturais definidos em xADL sejam relacionados com código fonte, persistidos no Subversion. Apesar do uso de xADL e Subversion, a arquitetura do ArchTrace é flexível, possibilitando a inclusão de outras ADLs e outros sistemas de controle de versão.

Contudo, o uso do Subversion implica versionamento de arquivos e traz todas as deficiências discutidas anteriormente. Apesar de ser muito importante a rastreabilidade entre arquitetura e código fonte, seria desejável que essa rastreabilidade também contemplasse elementos de análise e projeto em granularidade fina.

Para contornar essa limitação, poderia ser definido um novo conector de repositório no ArchTrace, que acessasse o Odyssey-VCS e mapeasse os elementos de modelo UML como possíveis alvos de rastreabilidade a partir dos elementos arquiteturais. Desta forma, a partir de um dado elemento arquitetural, seria possível acessar todos os elementos de modelo UML e códigos fonte utilizados na sua implementação.

Além disso, sob uma perspectiva complementar, caso o Odyssey-VCS passasse a fazer uso de UML 2.0, poderia também ser construído um conector arquitetural para o Odyssey-VCS. Assim, o ArchTrace teria como opções de ADL tanto a xADL quanto a UML 2.0, e como opções de repositório tanto o Subversion quanto o Odyssey-VCS.

8.3.3 Evolução da especificação UML

Em 2002, quando este trabalho de pesquisa foi iniciado, a versão corrente da especificação UML era a 1.4. Por essa razão, essa foi a versão adotada pelo Odyssey-SCM. Contudo, devido à própria imaturidade da linguagem, algumas outras versões foram disponibilizadas nesses quatro anos. A versão 1.5 trouxe apenas modificações superficiais, mas a versão 2.0 introduziu, dentre outras novidades, diagramas de componentes.

Apesar disso, o Odyssey-SCM permaneceu utilizando a versão 1.4 da especificação UML por duas razões principais: (1) como o Odyssey-SCM é composto de diversos sistemas, a migração para UML 2.0 deveria ser estudada a fundo para não gerar efeitos colaterais, visto que a especificação 2.0 da UML ainda não foi completamente finalizada, e (2) o repositório MOF utilizado (i.e., MDR) ainda não permite a utilização de meta-modelos que seguem o MOF 2.0, que é o caso da UML

2.0. A solução adotada, até então, é o mapeamento de componentes para pacotes ou classes, juntamente com estereótipos para essa identificação.

Essa imaturidade da especificação UML leva a problemas adicionais de compatibilidade. Somente a utilização de XMI como mecanismo para troca de informações não garante total compatibilidade entre ferramentas CASE. A especificação XMI, assim como a especificação UML, tem diversas versões. A implementação atual do Odyssey-SCM está compatível com as versões 1.1 e 1.2 da especificação XMI e com a versão 1.4 da especificação UML.

Vale ressaltar que a versão 1.4.2 da especificação UML foi recentemente transformada em ISO (ISO, 2005). Além disso, apesar da UML 2.0 já existir há algum tempo, nem toda a sua especificação está em estado final. A especificação da UML 2.0 é composta de quatro partes, onde somente duas estão finalizadas (*infrastructure* e *superstructure*) e outras duas estão em fase de finalização (OCL e *diagram interchange*).

8.3.4 Deficiências do repositório MOF adotado

O Odyssey-SCM adotou o MDR como repositório, principalmente pela sua compatibilidade com os padrões existentes (i.e., MOF, XMI e JMI) e por ser fruto de um projeto de código livre liderado pela Sun Microsystems, o NetBeans. Essa escolha foi bem sucedida no seu propósito principal, que era prover uma infra-estrutura uniforme para os diversos subsistemas do Odyssey-SCM.

Contudo, o MDR tem se mostrado deficiente em diversos aspectos. Em primeiro lugar, o seu desempenho fica bem abaixo do aceitável para viabilizar a utilização em ambiente real do Odyssey-SCM. Além disso, o seu repositório não apresenta a mesma robustez encontrada em sistemas de gerenciamento de banco de dados relacionais. Finalmente, o Projeto MDR, que é responsável pela construção do repositório MDR dentro do Projeto NetBeans, parece ter parado no tempo, tanto em termos da lista de discussão, que quase não tem mais atividade, quanto em termos de liberações do produto.

Caso essa decisão tivesse que ser tomada hoje, possivelmente o EMF (ECLIPSE FOUNDATION, 2006b), produto similar construído no contexto do Projeto Eclipse, seria adotado. Vale notar que o próprio Projeto Eclipse tem uma iniciativa de implementar o meta-modelo UML 2.0 fazendo uso de EMF (ECLIPSE FOUNDATION, 2006c).

8.3.5 Restrições para a carga de componentes por demanda

A implementação atual da Carga Dinâmica faz uso de um descritor de componentes para calcular as dependências de um dado componente quando esse componente é selecionado para implantação no ambiente alvo. Esse cálculo de dependências ocorre de forma recursiva, buscando por todos os componentes que são necessários para que o componente selecionado funcione corretamente.

Contudo, a implementação atual da Carga Dinâmica leva em consideração somente dependências, não possibilitando a declaração de restrições. A utilização de restrições permitiria estabelecer as incompatibilidades entre versões específicas de componentes. Desta forma, o cálculo de dependências poderia, além de indicar outros componentes que devem ser instalados em conjunto, também indicar componentes que devem ser removidos.

8.4 Trabalhos Futuros

A realização desse trabalho de pesquisa levou à construção de uma infra-estrutura de GCS para cenários de DBC. Essa intra-estrutura abre novas perspectivas de pesquisa, que podem ser exploradas em trabalhos futuros. Alguns desses trabalhos futuros são detalhados nesta seção.

8.4.1 GCS embutida na metáfora de DBC

No Odyssey-BRD, mais especificamente no ArchTrace, os comandos de *check-in* e *check-out* foram fornecidos no nível de arquitetura. Assim, quando o engenheiro de software aciona um desses comandos sobre um elemento arquitetural, o ArchTrace propaga o comando por meio das ligações de rastreabilidade, permitindo que o comando execute sobre o código fonte relacionado ao elemento arquitetural.

Esse tipo de postura pode ser adotado, também, em relação a outros comandos de GCS. Por exemplo, atividades inteiras de construção, empacotamento e liberação podem ser automatizadas no nível de arquitetura, e propagadas para código fonte via consultas às ligações de rastreabilidade existentes no ArchTrace.

8.4.2 Verificação de componentes

A verificação de componentes após cada *check-in* pode ser decomposta em três níveis: (1) detecção de conflitos, (2) detecção de quebras e (3) detecção de quebras lógicas. As ferramentas convencionais de controle de versão atuam no nível de detecção

de conflitos. Contudo, devido a uma unidade de comparação inadequada, vários conflitos passam despercebidos e só são detectados quando o sistema é compilado. A compilação do sistema é capaz de encontrar conflitos como, por exemplo, quando dois engenheiros de software editam linhas diferentes que manipulam um mesmo elemento. Nessas situações o sistema pode parar de compilar, e será possível detectar os engenheiros de software envolvidos nas modificações suspeitas da quebra. Em um cenário ainda mais complexo, o conflito passa despercebido inclusive pela compilação, contudo, quando a bateria de testes é executada, é possível detectar que algo indesejável ocorreu nas últimas modificações.

Desta forma, seria possível implementar uma política adicional ao ArchTrace para a verificação dos *check-ins* nesses três diferentes níveis. Com isso, em função do resultado dessa verificação, poderia ser atribuído um estado de qualidade à ligação de rastreabilidade estabelecida, permitindo, indiretamente, atribuir níveis de qualidade para as versões existentes dos componentes.

8.4.3 Visualização das informações

O estabelecimento de uma infra-estrutura de GCS como o Odyssey-SCM faz com que uma grande quantidade de informações seja coletada e armazenada. Essas informações são usualmente apresentadas aos engenheiros de software na forma de relatórios, seja na íntegra ou sumarizadas por meio de técnicas estatísticas.

Contudo, formas alternativas para apresentar grandes quantidades de informação constituem uma área de pesquisa em ascensão, principalmente devido à Internet. Usualmente, são utilizadas técnicas de coloração, agrupamento de informações e de desenho em duas ou três dimensões para facilitar a compreensão da informação apresentada. O próprio uso dessas técnicas em repositórios de controle de versões não é algo novo (JONES *et al.*, 2002; FROEHLICH e DOURISH, 2004). Entretanto, a aplicação dessas técnicas sobre repositórios integrados de GCS pode prover contribuições adicionais às já obtidas com repositórios de controle de versões.

8.4.4 Simulações de estratégias de GCS

Na definição de processos de GCS, algumas premissas são tomadas como verdade. A partir dessas premissas, o processo é projetado visando obter o maior ganho de qualidade e produtividade possível. Todavia, essas premissas podem variar de organização para organização, levando a necessidades diferenciadas em relação aos

processos. Por exemplo, em função do tempo médio de testes e da sua variância, podem ser adotadas estratégias de verificação após cada *check-in*, após a finalização da modificação como um todo, ou somente antes da liberação de uma versão de produção do componente.

A análise dos fatores que podem influenciar na definição do processo de GCS (e.g.: duração de liberações, testes, auditorias, etc.), juntamente com simulações das diferentes configurações possíveis do processo para os valores coletados em uma organização específica, poderia ajudar na escolha da estratégia de GCS a ser adotada nessa organização. Por exemplo, a seleção de estratégias de ramificação (BERCZUK e APPLETON, 2002; WALRAD e STROM, 2002) pode ser influenciada pelo tipo de ciclo de vida utilizado e pela freqüência necessária de liberações, entre outros fatores.

8.4.5 Análise estatística de desempenho do processo de GCS

Após a execução de simulações para a adaptação do processo de GCS, esse processo deve ser implantado e medido, com o intuito de identificar possíveis melhorias e realçar os seus pontos fortes. Como resultado da aplicação de análises estatísticas sobre várias instâncias em execução do processo, é possível alimentar o próprio modelo de simulações, favorecendo instanciações mais precisas do processo no futuro. Além disso, decisões gerenciais passam a ser embasadas por números, aumentando a previsibilidade dos resultados obtidos por essas decisões.

8.4.6 Avaliações adicionais da abordagem

As avaliações do Odyssey-SCM focaram sobre os subsistemas Odyssey-VCS, Odyssey-BRD e Odyssey-WI. Além disso, essas avaliações focaram em aspectos relacionados ao desempenho do sistema em si (caso do Odyssey-VCS) e à acurácia da técnica adotada pelo sistema (caso do Odyssey-BRD e do Odyssey-WI).

Diferentes avaliações adicionais deveriam ser executadas. Em primeiro lugar, o próprio processo Odyssey-SCMP, implantado ou não no Odyssey-CCS, poderia ser avaliado por meio de projetos piloto em ambientes reais de DBC. Já o Odyssey-CCS poderia ser avaliado tanto em um ambiente de DBC quanto em um ambiente de desenvolvimento convencional, neste último caso, sem levar em consideração o mapa de reutilização. Finalmente, todo o Odyssey-SCM poderia ser avaliado sob outras perspectivas, além da de desempenho e acurácia.

Referências Bibliográficas

ABI-ANTOUN, M., ALDRICH, J., GARLAN, D*., et al.*, 2005, "Semi-Automated Incremental Synchronization between Conceptual and Implementation Level Architectures". In: *Working IEEE/IFIP Conference on Software Architecture (WICSA)*, pp. 265-268, Pittsburgh, PA, USA, November.

AGRAWAL, R., IMIELINSKI, T., SWAMI, A., 1993, "Mining Association Rules between Sets of Items in Large Databases". In: *ACM SIGMOD International Conference on Management of Data*, pp. 207-216, Washington, DC, USA, May.

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

ALDRICH, J., CHAMBERS, C., NOTKIN, D., 2002, "ArchJava: Connecting Software Architecture to Implementation". In: *International Conference on Software Engineering (ICSE)*, pp. 187-197, Orlando, USA, May.

ANTONIOL, G., CANFORA, G., CASAZZA, G*., et al.*, 2002, "Recovering Traceability Links between Code and Documentation", *IEEE Transactions on Software Engineering (TSE)*, v. 28, n. 10 (October), pp. 970-983.

ARANGO, G., 1988, *Domain Engineering for Software Reuse*, Ph.D. Thesis, University of California, Irvine, CA.

ASKLUND, U., BENDIX, L., 2002, "A Study of Configuration Management in Open Source Software", *IEE Proceedings - Software*, v. 149, n. 1 (February), pp. 40-46.

ATKINSON, C., BAYER, J., BUNSE, C*., et al.*, 2001, *Component-Based Product Line Engineering with UML*, Addison-Wesley.

ATKINSON, C., BAYER, J., LAITENBERGER, O*., et al.*, 2000, "Component-Based Software Engineering: The KobrA Approach". In: *3rd International Workshop on Component-based Software Engineering*, Limerick, Ireland, June.

BAEZA-YATES, R., RIBEIRO-NETO, B., 1999, *Modern Information Retrieval*, ACM Press.

BALL, T., KIM, J., PORTER, A.A., *et al.*, 1997, "If your version control system could talk". In: *Workshop on Process Modelling and Empirical Studies of Software Engineering*, Boston, MA, USA, May.

BANCILHON, F., DELOBEL, C., KANELLAKIS, P.C., 1992, *Building an Object-Oriented Database System, The Story of O2*, Morgan Kaufmann.

BARNSON, M.P., STEENHAGEN, J., WEISSMAN, T., 2003, *The Bugzilla Guide - 2.17.5 Development Release*, The Bugzilla Team.

BASILI, V.R., 1996, "The Role of Experimentation in Software Engineering: Past, Current, and Future". In: *International Conference on Software Engineering (ICSE)*, pp. 442-449, Berlin, Germany, March.

BECK, K., 1999, *Extreme Programming Explained: Embrace Change* Boston, MA, Addison-Wesley.

BERCZUK, S., APPLETON, B., 2002, *Software Configuration Management Patterns: Effective Teamwork, Practical Integration*, 1st. ed. Boston, MA, USA, Addison-Wesley Professional.

BIEMAN, J.M., ANDREWS, A.A., YANG, H.J., 2003, "Understanding change-proneness in OO software through Visualization". In: *International Workshop on Program Comprehension*, pp. 44-53, Portland, USA, May.

BITMOVER, 2006, "BitKeeper". In: http://www.bitkeeper.com, accessed in April 16.

BLOIS, A.P.T.B., 2006, *Uma Abordagem de Projeto Arquitetural Baseado em Componentes no Contexto de Engenharia de Domínio*, Tese de D.Sc., COPPE, UFRJ, Rio de Janeiro, RJ, Brasil.

BODOFF, S., ARMSTRONG, E., BALL, J., *et al.*, 2004, *The J2EE Tutorial*, 2nd. ed., Addison-Wesley Professional.

BOLINGER, D., BRONSON, T., 1995, *Applying RCS and SCCS*, 1st. ed., O'Reilly.

BOOCH, G., 1993, *Object-Oriented Analysis and Design with Applications*, 2nd. ed., Addison-Wesley Professional.

BOOTH, D., HAAS, H., MCCABE, F., *et al.*, 2005, "Web Services Architecture - W3C Working Group Note". In: http://www.w3.org/TR/ws-arch, accessed in April 16.

BORLAND, 2006a, "Borland InterBase". In: http://www.borland.com/interbase, accessed in July 8.

BORLAND, 2006b, "Borland JBuilder 2006". In: http://www.borland.com/jbuilder/, accessed in June 20.

BOSCH, J., 2000, *Design and Use of Software Architectures: Adopting and Evolving a Product-Line Approach*, Addison Wesley.

BRAGA, R.M.M., 2000, *Busca e Recuperação de Componentes em Ambientes de Reutilização de Software*, Tese de D.Sc., COPPE, UFRJ, Rio de Janeiro, RJ, Brasil.

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

BROWN, A.W., 2000, *Large Scale Component Based Development*, Prentice Hall PTR.

CEDERQVIST, P., 2003, *Version Management with CVS*, Free Software Foundation.

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

CHEN, P., CRITCHLOW, M., GARG, A.*, et al.*, 2003, "Differencing and Merging within an Evolving Product Line Architecture". In: *International Workshop on Product Family Engineering*, pp. 269-281, Siena, Italy, November.

CHIEN, S., TSOTRAS, V.J., ZANIOLO, C., 2001, "XML Document Versioning", *ACM SIGMOD Record*, v. 30, n. 3 (September), pp. 46-53.

CHRISSIS, M.B., KONRAD, M., SHRUM, S., 2003, *CMMI: Guidelines for Process Integration and Product Improvement* Boston, MA, Addison-Wesley.

CHRISTENSEN, H.B., 1999a, "The Ragnarok Architectural Software Configuration Management Model". In: *Annual Hawaii International Conference on System Sciences*, Maui, HI, USA, January.

CHRISTENSEN, H.B., 1999b, "The Ragnarok Software Development Environment", *Nordic Journal of Computing*, v. 6, n. 1 (Spring), pp. 4-21.

CHRISTENSEN, M.J., THAYER, R.H., 2002, *The Project Manager's Guide to Software Engineering's Best Practices*, IEEE Computer Society Press and John Wiley & Sons.

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

CM CROSSROADS, 2006, "Configuration Management Yellow Pages". In: http://www.cmyellowpages.com, accessed in July 8.

COBENA, G., ABITEBOUL, S., MARIAN, A., 2002, "Detecting Changes in XML Documents". In: *International Conference on Data Engineering (ICDE)*, pp. 41-52, San Jose, CA, USA, February.

COLLINS-SUSSMAN, B., FITZPATRICK, B.W., PILATO, C.M., 2004, *Version Control with Subversion*, O'Reilly.

CONRADI, R., WESTFECHTEL, B., 1998, "Version Models for Software Configuration Management", *ACM Computing Surveys*, v. 30, n. 2 (June), pp. 232-282.

CORREA, A.L., 2006, *Reestruturando Especificações de Restrições de Modelos Elaboradas em OCL*, Tese de D.Sc., COPPE, UFRJ, Rio de Janeiro, Brasil.

COSTA, M., 2002, *COMPAGENT: Uma Ferramenta para o apoio à Busca e Recuperação de Informações Orientadas a Domínio na Web*, Dissertação de M.Sc., COPPE, UFRJ, Rio de Janeiro, Brasil.

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

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

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

DANTAS, C.R., 2005, *Odyssey-WI: Uma Abordagem para a Mineração de Rastros de Modificação de Modelos em Repositórios Versionados*, Dissertação de M.Sc., COPPE, UFRJ, Rio de Janeiro, Brasil.

DANTAS, C.R., MURTA, L.G.P., WERNER, C.M.L., 2004, "Mineração de Rastros de Modificação de Modelos em Repositórios Versionados". In: *Workshop de Desenvolvimento Baseado em Componentes (WDBC)*, pp. 35-40, João Pessoa, PB, Brasil, setembro.

DANTAS, C.R., MURTA, L.G.P., WERNER, C.M.L., 2005, "Consistent Evolution of UML Models by Automatic Detection of Change Traces". In: *International Workshop on Principles of Software Evolution (IWPSE)*, pp. 144-147, Lisbon, Portugal, September.

DANTAS, C.R., MURTA, L.G.P., WERNER, C.M.L., 2006, "Odyssey-WI: Uma Ferramenta para Mineração de Rastros de Modificação em Modelos UML Versionados". In: *Simpósio Brasileiro de Engenharia de Software (SBES), Sessão de Ferramentas*, Florianópolis, SC, Brasil, outubro (a ser publicado).

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

em Componentes". In: *Workshop de Desenvolvimento Baseado em Componentes (WDBC)*, São Carlos, SP, Brasil, setembro.

DART, S., 1991, "Concepts in Configuration Management Systems". In: *International Workshop on Software Configuration Management (SCM)*, pp. 1-18, Trondheim, Norway, June.

DASHOFY, E., VAN DER HOEK, A., TAYLOR, R.N., 2001, "A Highly-Extensible, XML-Based Architecture Description Language". In: *Working IEEE/IFIP Conference on Software Architectures (WICSA)*, pp. 103-112, Amsterdam, Netherlands, August.

DASHOFY, E., VAN DER HOEK, A., TAYLOR, R.N., 2002, "An Infrastructure for the Rapid Development of XML-Based Architecture Description Languages". In: *International Conference on Software Engineering (ICSE)*, pp. 266-276, Orlando, FL, USA, May.

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

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

DOD, 1962, *AFSCM 375-1 - CM During the Development & Acquisition Phases*, Department of Defense.

DOD, 1971, *MIL Std 483 - CM Practices for Systems, Equipment, Munitions, & Computer Programs*, Department of Defense.

DOD, 1985, *DOD Std 2167A - Defense System Software Development*, Department of Defense.

DRAHEIM, D., PEKACKI, L., 2003, "Process-Centric Analytical Processing of Version Control Data". In: *International Workshop on Principles of Software Evolution (IWPSE)*, pp. 131-136, Helsinki, Finland, September.

ECLIPSE FOUNDATION, 2006a, "Eclipse 3.1". In: http://www.eclipse.org, accessed in June 20.

ECLIPSE FOUNDATION, 2006b, "Eclipse Modeling Framework (EMF)". In: http://www.eclipse.org/emf, accessed in August 21.

ECLIPSE FOUNDATION, 2006c, "EMF-based UML 2.x Metamodel Implementation". In: http://www.eclipse.org/uml2, accessed in August 21.

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

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

ESTRADA, A.F., 2003, *Un modelo de referencia para la Gestión de Configuración en la PYME de Software*, Tese de D.Sc., Instituto Superior Politécnico "José Antonio Echeverria", Havana, Cuba.

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

ESTUBLIER, J., 2001, "Objects Control for Software Configuration Management". In: *Conference on Advanced Information Systems Engineering (CAiSE)*, pp. 359-373, Interlaken, Switzerland, June.

ESTUBLIER, J., FAVRE, J.-M., MORAT, P., 1998, "Toward SCM / PDM integration?" In: *Software Configuration Management (SCM)*, pp. 75-95, Brussels, Belgium, July.

ESTUBLIER, J., LEBLANG, D., VAN DER HOEK, A., *et al.*, 2005, "Impact of Software Engineering Research on the Practice of Software Configuration Management", *ACM Transactions on Software Engineering and Methodology (TOSEM)*, v. 14, n. 4 (October), pp. 1-48.

FELDMAN, S.I., 1979, "Make - A Program for Maintaining Computer Programs", *Software - Practice and Experience*, v. 9, n. 4 (April), pp. 255-265.

FIGUEIREDO, S.M., 2004, *Gerência de Configuração em Ambientes de Desenvolvimento de Software Orientados a Organização*, Projeto Final de Curso, DCC/IM, UFRJ, Rio de Janeiro, Brasil.

FINEP, 2003, *RECOPE: Redes Cooperativas de Pesquisa, Relatório do Seminário de Avaliação Final do Programa RECOPE*, Financiadora de Estudos e Projetos, Rio de Janeiro, RJ, Brasil.

FLASHLINE, 2006, "Flashline Registry". In: http://www.flashline.com, accessed in July 25.

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

FOWLER, M., 2006, "Continuous Integration". In: http://www.martinfowler.com/articles/continuousIntegration.html, accessed in June 20.

FROEHLICH, J., DOURISH, P., 2004, "Unifying Artifacts and Activities in a Visual Tool for Distributed Software Development Teams". In: *International Conference on Software Engineering (ICSE)*, pp. 387-396, Edinburgh, UK, May.

FSF, 2002, *Comparing and Merging Files*, Free Software Foundation.

GAILLY, J.-L., ADLER, M., 2006, "zlib: A Massively Spiffy Yet Delicately Unobtrusive Compression Library". In: http://www.zlib.net, accessed in August 1.

GALL, H., JAZAYERI, M., KLÖSCH, R., *et al.*, 1997, "Software Evolution Observations based on Product Release History". In: *International Conference on Software Maintenance (ICSM)*, pp. 160-196, Bari, Italy, October.

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

GARG, A., CRITCHLOW, M., CHEN, P., *et al.*, 2003, "An Environment for Managing Evolving Product Line Architectures". In: *International Conference on Software Maintenance (ICSM)*, pp. 358-367, Amsterdam, Netherlands, September.

GENTLEWARE, 2006, "Poseidon for UML". In: http://www.gentleware.com, accessed in April 16.

GIMENES, I.M.D.S., HUZITA, E.H.M., 2005, *Desenvolvimento Baseado em Componentes: Conceitos e Técnicas*, 1a. ed. Rio de Janeiro, Brasil, Editora Ciência Moderna Ltda.

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

GOLDSTEIN, I.P., BOBROW, D.G., 1984, "A Layered Approach to Software Design". In: BARSTOW, D.R., SHROBE, H.E., SANDEWALL, E. (eds), *Interactive Programming Environments*, New York, NY, McGraw-Hill.

GOSLING, J., JOY, B., STEELE, G., *et al.*, 2005, *The Java Language Specification*, 3rd. ed., Addison-Wesley Professional.

GOTEL, O., FINKELSTEIN, A., 1994, "An Analysis of the Requirements Traceability Problem". In: *International Conference on Requirements Engineering (RE)*, pp. 94-101, Colorado, USA, April.

GUTWIN, C., GREENBERG, S., 2002, "A Descriptive Framework of Workspace Awareness for Real-Time Groupware", *Journal of Computer Supported Cooperative Work*, v. 11, n. 3, pp. 411-446.

HABERMANN, A.N., NOTKIN, D., 1986, "Gandalf: Software Development Environments", *IEEE Transactions on Software Engineering (TSE)*, v. 12, n. 12 (December), pp. 1117-1127.

HASS, A.M.J., 2003, *Configuration Management Principles and Practices* Boston, MA, Pearson Education, Inc.

HATCHER, E., 2001, *Automating the build and test process: Ant and JUnit bring you one step closer to XP nirvana*, IBM.

HATCHER, E., LOUGHRAN, S., 2004, *Java Development with Ant* Greenwich, CT, Manning Publications Company.

HEINEMAN, G.T., COUNCILL, W.T., 2001, *Component Based Software Engineering: Putting the Pieces Together*, Addison-Wesley Pub Co.

HERZUM, P., SIMS, O., 1999, *Business Component Factory: A Comprehensive Overview of Component-Based Development for the Enterprise*, John Wiley & Sons.

HNETYNKA, P., PLASIL, F., 2004, "Distributed Versioning Model for MOF". In: *Winter International Symposium on Information and Communication Technologies (WISICT)*, pp. 489-494, Cancun, México, January.

HUFFMAN HAYES, J., DEKHTYAR, A., OSBORNE, J., 2003, "Improving Requirements Tracing via Information Retrieval". In: *International Conference on Requirements Engineering (RE)*, pp. 138-147, Monterey, USA, September.

IBM RATIONAL, 2006, "Rational Rose XDE Modeler - Product Overview - IBM Software". In: http://www-306.ibm.com/software/awdtools/developer/modeler, accessed in July 8.

IEEE, 1987, *Std 1042 - IEEE Guide to Software Configuration Management*, Institute of Electrical and Electronics Engineers.

IEEE, 1990, *Std 610.12 - IEEE Standard Glossary of Software Engineering Terminology*, Institute of Electrical and Electronics Engineers.

IEEE, 1998, *Std 1219 - IEEE Standard for Software Maintenance*, Institute of Electrical and Electronics Engineers.

IEEE, 2004, *Std 1517 - IEEE Standard for Information Technology - Software Life Cycle Processes - Reuse Processes*, Institute of Electrical and Electronics Engineers.

IEEE, 2005, *Std 828 - IEEE Standard for Software Configuration Management Plans*, Institute of Electrical and Electronics Engineers.

ISO, 1995a, *ISO 10007, Quality Management - Guidelines for Configuration Management*, International Organization for Standardization.

ISO, 1995b, *ISO/IEC 12207 - Information technology - Software life cycle processes*, International Organization for Standardization.

ISO, 2001, *ISO/IEC 9126 - Software engineering - Product quality*, International Organization for Standardization.

ISO, 2004, *ISO/IEC 15504 - Information technology - Process Assessment*, International Organization for Standardization.

ISO, 2005, *ISO/IEC 19501 - Information technology - Open Distributed Processing - Unified Modeling Language (UML) Version 1.4.2*, International Organization for Standardization.

JACOBSON, I., GRISS, M., JONSSON, P., 1997, *Software Reuse: Architecture, Process and Organization for Business Success*, Addison-Wesley Professional.

JALOTE, P., 1999, *CMM in Practice: Processes for Executing Software Projects at Infosys* Boston, MA, Addison-Wesley.

JONES, J.A., HARROLD, M.J., STASKO, J., 2002, "Visualization of Test Information to Assist Fault Localization". In: *International Conference on Software Engineering (ICSE)*, pp. 467-477, Orlando, Florida, May.

JSF, 2006, "Jabber". In: http://www.jabber.org, accessed in August 16.

KANG, K., COHEN, S., HESS, J., *et al.*, 1990, *Feature-Oriented Domain Analysis (FODA) Feasibility Study*, CMU/SEI-90-TR-021, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, USA.

KAO, C.-L., 2006, "SVK". In: http://svk.elixus.org, accessed in June 20.

KICZALES, G., LAMPING, J., MENDHEKAR, A., *et al.*, 1997, "Aspect-Oriented Programming". In: *European Conference on Object-Oriented Programming (ECOOP)*, pp. 220-242, Jyväskylä, Finland, June.

KNETHEN, A., GRUND, M., 2003, "QuaTrace: A Tool Environment for (Semi-) Automated Impact Analysis Based on Traces". In: *International Conference on*

Software Maintenance (ICSM), pp. 246-255, Amsterdam, Netherlands, September.

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

KRUCHTEN, P., 2001, *The Rational Unified Process: An Introduction*, Addison-Wesley.

KRUEGER, C.W., 1992, "Software Reuse", *ACM Computing Surveys*, v. 24, n. 2 (June), pp. 131-183.

KWON, O., SHIN, G., BOLDYREFF, C.*, et al.*, 1999, "Maintenance with Reuse: An Integrated Approach Based on Software Configuration Management". In: *Asia Pacific Software Engineering Conference*, pp. 507-515, Takamatsu, Japan, December.

LARSSON, M., 2000, *Applying Configuration Management Techniques to Component-Based Systems*, Licentiate Thesis, Deparment of Information Technology, Uppsala University, Sweden.

LARSSON, M., CRNKOVIC, I., 2000, "Component Configuration Management". In: *Workshop on Component Oriented Programming*, Sophia Antipolis, France, June.

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

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

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

LINGEN, R., VAN DER HOEK, A., 2004, "An Experimental, Pluggable Infrastructure for Modular Configuration Management Policy Composition". In: *International Conference on Software Engineering (ICSE)*, pp. 573-582, Edinburgh, Scotland, May.

LOGICLIBRARY, 2006, "Logidex". In: http://www.logiclibrary.com, accessed in July 25.

LOPES, L.G.B., 2006, *Odyssey-CCS: Uma Abordagem para o Controle de Modificações no Contexto do Desenvolvimento Baseado em Componentes*, Dissertação de M.Sc., COPPE, UFRJ, Rio de Janeiro, Brasil.

LOPES, L.G.B., MURTA, L.G.P., WERNER, C.M.L., 2005, "Controle de Modificações em Software no Desenvolvimento Baseado em Componentes". In: *Workshop de Manutenção de Software Moderna (WMSWM)*, pp. 82-97, Manaus, AM, Brasil, novembro.

LOPES, L.G.B., MURTA, L.G.P., WERNER, C.M.L., 2006a, "Odyssey-CCS: A Change Control System Tailored to Software Reuse". In: *International Conference on Software Reuse (ICSR)*, pp. 170-183, Torino, Italy, June.

LOPES, L.G.B., MURTA, L.G.P., WERNER, C.M.L., 2006b, "Odyssey-CCS: Uma ferramenta flexível para o controle de modificações em software". In: *Simpósio Brasileiro de Engenharia de Software (SBES), Sessão de Ferramentas*, Florianópolis, SC, Brasil, outubro (a ser publicado).

LOPES, M.A.M., 2005, *MAIS: um Mecanismo para Apoio à Percepção aplicado a Modelos de Software Compartilhados*, Dissertação de M.Sc., COPPE, UFRJ, Rio de Janeiro, Brasil.

LORD, T., 2006, "GNU Arch". In: http://www.gnuarch.org, accessed in April 16.

LÜER, C., VAN DER HOEK, A., 2004, "JPloy: User-Centric Deployment Support in a Component Platform". In: *IFIP/ACM Working Conference on Component Deployment (CD)*, pp. 190-204, Edinburgh, UK, May.

MAIA, N., 2006, *Odyssey-MDA: Uma Abordagem para a Transformação de Modelos*, Dissertação de M.Sc., COPPE, UFRJ, Rio de Janeiro, Brasil.

MANGAN, M.A.S., 2006, *Uma abordagem para o Desenvolvimento de Apoio à Percepção em Ambientes Colaborativos de Desenvolvimento de Software*, Tese de D.Sc., COPPE, UFRJ, Rio de Janeiro, Brasil.

MANGAN, M.A.S., WERNER, C.M.L., BORGES, M.R.S., 2002, "Componentes para Colaboração Síncrona em um Ambiente de Reutilização de Software". In: *Simpósio Brasileiro de Engenharia de Software (SBES), Sessão de Ferramentas*, pp. 372-377, Gramado, RS, Brasil, outubro.

MARCUS, A., MALETIC, J.I., 2003, "Recovering Documentation-to-Source-Code Traceability Links using Latent Semantic Indexing". In: *International Conference on Software Engineering (ICSE)*, pp. 125-135, Portland, OR, USA, May.

MATTOSO, M., WERNER, C.M.L., BRAGA, R.M.M., *et al.*, 2000, "Persistência de Componentes num Ambiente de Reuso". In: *Simpósio Brasileiro de Engenharia de Software (SBES), Sessão de Ferramentas*, pp. 351-354, João Pessoa, PB, Brasil, outubro.

MATULA, M., 2005, "NetBeans Metadata Repository". In: http://mdr.netbeans.org, accessed in April 16.

MCT, 2002, *Qualidade e Produtividade no Setor de Software Brasileiro*, Ministério de Ciência e Tecnologia, Secretaria de Política de Informática, Brasília, DF, Brasil.

MEDICINENET, 2003, *Medical Dictionary*, 2nd. ed., Webster's New World.

MEDVIDOVIC, N., ROSENBLUM, D.S., 1997, "Domains of Concern in Software Architectures and Architecture Description Languages". In: *Conference on Domain-Specific Languages*, pp. 199-212, Santa Barbara, USA, October.

MEI, H., ZHANG, L., YANG, F., 2001, "A software configuration management model for supporting component-based software development", *ACM SIGSOFT Software Engineering Notes*, v. 26, n. 2 (March), pp. 53-58.

MELO JR., C.R.S., 2005, *MetricTool: Uma Ferramenta Parametrizável para Extração de Métricas de Projeto Orientados a Objetos*, Projeto Final de Curso, COPPE, UFRJ, Rio de Janeiro, Brasil.

MELO JR., C.R.S., MURTA, L.G.P., VASCONCELOS, A.P.V., *et al.*, 2004, "Infra-estrutura para Carga Dinâmica de Ferramentas no Ambiente Odyssey-Light". In: *Jornada de Iniciação Científica*, Rio de Janeiro, RJ, Brasil, novembro.

MEYER, B., 1991, *Eiffel: The Language*, 1st. ed., Prentice Hall.

MEYER, B., 1992, "Applying Design by Contract", *IEEE Computer*, v. 25, n. 10 (October), pp. 40-51.

MICROSOFT, 2006a, "COM: Component Object Model Technologies". In: http://www.microsoft.com/com, accessed in June 21.

MICROSOFT, 2006b, "Microsoft Office". In: http://office.microsoft.com, accessed in July 8.

MICROSOFT, 2006c, "Windows XP". In: http://www.microsoft.com/windowsxp, accessed in June 20.

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

MILLER, J., MUKERJI, J., 2003, *MDA Guide Version 1.0.1*, omg/2003-06-01, Object Management Group.

MORO, M.M., SAGGIORATO, S.M., EDELWEISS, N., *et al.*, 2001, "A Temporal Versions Model for Time-Evolving Systems Specification". In: *International Conference on Software Engineering and Knowledge Engineering (SEKE)*, pp. 252-259, Buenos Aires, Argentina, June.

MORO, M.M., ZAUPA, A.P., EDELWEISS, N., *et al.*, 2002, "TVQL - Temporal Versioned Query Language". In: *International Conference on Database and Expert Systems Applications (DEXA)*, pp. 618-627, Aix en Provence, France, September.

MOZILLA, 2006a, "Bonsai". In: http://www.mozilla.org/bonsai.html, accessed in June 20.

MOZILLA, 2006b, "Firefox". In: http://www.mozilla.com/firefox/, accessed in June 20.

MOZILLA, 2006c, "Tinderbox". In: http://www.mozilla.org/tinderbox.html, accessed in June 20.

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*, Dissertação de 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, Brasil.

MURTA, L.G.P., BARROS, M., WERNER, C.M.L., 2000, "Token: Uma Ferramenta para o Controle de Alterações em Projetos de Software em Desenvolvimento", *Simpósio Brasileiro de Engenharia de Software (SBES), Sessão de Ferramentas*, pp. 383-386, João Pessoa, PB, Brasil, outubro.

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

MURTA, L.G.P., BARROS, M., WERNER, C.M.L., 2002b, "Charon: Uma máquina de processos extensível baseada em agentes inteligentes". In: *Workshop Ibero-*

americano de Engenharia de Requisitos e Ambientes de Software (IDEAS), pp. 236-247, Havana, Cuba, abril.

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

MURTA, L.G.P., DANTAS, C.R., OLIVEIRA, H.L.R.*, et al.*, 2005, "A Caminho da Manutenção de Software Baseada em Componentes via Técnicas de Gerência de Configuração de Software", *Revista de Tecnologia da Informação (RTInfo)*, v. 5, n. 2 (dezembro), pp. 45-62.

MURTA, L.G.P., OLIVEIRA, H.L.R., DANTAS, C.R.*, et al.*, 2004a, "Towards Component-based Software Maintenance via Software Configuration Management Techniques". In: *Brazilian Symposium on Software Engineering (SBES), Workshop on Modern Software Maintenance (WMSWM)*, Brasília, DF, Brasil, October.

MURTA, L.G.P., OLIVEIRA, H.L.R., DANTAS, C.R.*, et al.*, 2006a, "Odyssey-SCM: An Integrated Software Configuration Management Infrastructure for Model-driven Development", *Science of Computer Programming* (to be published).

MURTA, L.G.P., VAN DER HOEK, A., WERNER, C.M.L., 2006b, "ArchTrace: A Tool for Keeping in Sync Architecture and its Implementation". In: *Brazilian Symposium on Software Engineering (SBES), Tools Session*, Florianópolis, Brazil, October (to be published).

MURTA, L.G.P., VAN DER HOEK, A., WERNER, C.M.L., 2006c, "ArchTrace: Policy-Based Support for Managing Evolving Architecture-to-Implementation Traceability Links". In: *International Conference on Automated Software Engineering (ASE)*, Tokyo, Japan, September (to be published).

MURTA, L.G.P., VASCONCELOS, A.P.V., BLOIS, A.P.T.B.*, et al.*, 2004b, "Run-time Variability through Component Dynamic Loading". In: *Brazilian Symposium on Software Engineering (SBES), Tools Session*, pp. 67-72, Brasília, DF, Brazil, October.

MURTA, L.G.P., VERONESE, G.O., WERNER, C.M.L., 2001b, "MOR: Uma Ferramenta para o Mapeamento Objeto-Relacional em Java". In: *Simpósio Brasileiro de Engenharia de Software (SBES), Sessão de Ferramentas*, pp. 392-397, Rio de Janeiro, RJ, Brasil, outubro.

MYERS, E.W., 1986, "An O(ND) Difference Algorithm and its Variations", *Algorithmica*, v. 1, n. 2 (March), pp. 251-266.

MYSQL AB, 2006, "MySQL". In: http://www.mysql.com, accessed in August 16.

NEIGHBORS, J., 1980, *Software Construction Using Components*, Ph.D. Thesis, Department of Information and Computer Science, University of California, Irvine, CA.

NENTWICH, C., EMMERICH, W., FINKELSTEIN, A., *et al.*, 2003, "Flexible Consistency Checking", *ACM Transactions on Software Engineering and Methodology (TOSEM)*, v. 12, n. 1 (January), pp. 28-63.

NETBEANS COMMUNITY, 2006, "NetBeans 5.0". In: http://www.netbeans.org, accessed in June 20.

NGUYEN, T.N., MUNSON, E.V., BOYLAND, J.T., 2004, "The molhado hypertext versioning system". In: *Conference on Hypertext and Hypermedia*, pp. 185-194, Santa Cruz, USA, August.

OHST, D., KELTER, U., 2002, "A Fine-grained Version and Configuration Model in Analysis and Design". In: *International Conference on Software Maintenance (ICSM)*, pp. 521-527, Montreal, Canada, October.

OLIVEIRA, A.A.A.C.P., PRIMO, F.F., CRUZ, J.L., *et al.*, 2001, *Gerência de Configuração de Software - Evolução de Software sob Controle*, Instituto Nacional de Tecnologia da Informação (ITI), Ministério da Ciência e Tecnologia.

OLIVEIRA, H.L.R., 2005, *Odyssey-VCS: Uma Abordagem de Controle de Versões para Elementos da UML*, Dissertação de M.Sc., COPPE, UFRJ, Rio de Janeiro, Brasil.

OLIVEIRA, H.L.R., MURTA, L.G.P., WERNER, C.M.L., 2004, "Odyssey-VCS: Um Sistema de Controle de Versões Para Modelos Baseados no MOF". In: *Simpósio Brasileiro de Engenharia de Software (SBES), Seção de Ferramentas*, pp. 85-90, Brasília, Brasil, outubro.

OLIVEIRA, H.L.R., MURTA, L.G.P., WERNER, C.M.L., 2005, "Odyssey-VCS: a Flexible Version Control System for UML Model Elements". In: *International Workshop on Software Configuration Management (SCM)*, pp. 1-16, Lisbon, Portugal, September.

OLIVEIRA, R.F.D., 2006, *Formalização e Verificação de Consistência na Representação de Variabilidades.*, Dissertação de M.Sc., COPPE, UFRJ, Rio de Janeiro, Brasil.

OMG, 2001, *Unified Modeling Language (UML) Specification, version 1.4*, formal/01-09-67, Object Management Group.

OMG, 2002a, *Meta Object Facility (MOF) Specification, version 1.4*, formal/02-04-03, Object Management Group.

OMG, 2002b, *XML Metadata Interchange (XMI) Specification, version 1.2*, formal/02-01-01, Object Management Group.

OMG, 2005a, *Meta Object Facility (MOF) 2.0 Versioning and Development Lifecycle Specification, Final Adopted Specification*, ptc/05-08-01, Object Management Group.

OMG, 2005b, *Software Process Engineering Metamodel (SPEM) Specification, version 1.1*, formal/05-01-06, Object Management Group.

OMG, 2005c, *Unified Modeling Language (UML) Superstructure Specification, version 2.0*, formal/05-07-04, Object Management Group.

OMG, 2006a, *Object Constraint Language (OCL), Version 2.0*, formal/06-05-01, Object Management Group.

OMG, 2006b, "Object Management Group". In: http://www.omg.org, accessed in August 16.

OMMERING, R.V., LINDEN, F.V.D., KRAMER, J., *et al.*, 2000, "The Koala Component Model for Consumer Electronics Software", *IEEE Computer*, v. 33, n. 6 (March), pp. 78-85.

OREIZY, P., MEDVIDOVIC, N., TAYLOR, R.N., 1998, "Architecture-Based Runtime Software Evolution". In: *International Conference on Software Engineering (ICSE)*, pp. 177-186, Kyoto, Japan, April.

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

PINHEIRO, R., 2002, *COMPUBLISH: Um Sistema para a Publicação, Busca e Recuperação de Componentes de Software na Internet*, Dissertação de M.Sc., COPPE, UFRJ, Rio de Janeiro, Brasil.

PRESSMAN, R.S., 2005, *Software Engineering: A Practitioner's Approach*, 6th. ed., McGraw-Hill.

PRIETO-DIAZ, R., 1990, "Domain Analysis: An Introduction", *Software Engineering Notes*, v. 15, n. 2 (April), pp. 47-54.

REENSKAUG, T., 2003, "The Model-View-Controller (MVC): Its Past and Present". In: *Java Zone*, Oslo, Norway, September.

REISS, S.P., 2002, "Constraining Software Evolution". In: *International Conference on Software Maintenance (ICSM)*, pp. 162-171, Montreal, Canada, October.

RENDER, H., CAMPBELL, R., 1991, "An Object-oriented Model of Software Configuration Management". In: *International Workshop on Software Configuration Management (SCM)*, pp. 127-139, Trondheim, Norway, June.

RICHARDSON, D.J., WOLF, A.L., 1996, "Software Testing at the Architectural Level". In: *International Software Architecture Workshop (ISAW)*, pp. 68-71, San Francisco, USA, October.

ROCHA, A.R.C., AGUIAR, T.C., SOUZA, J.M., 1990, "TABA: A Heuristic Workstation for Software Development". In: *COMPEURO*, pp. 126-129, Tel Aviv, Israel, May.

ROCHE, T., WHIPPLE, L.C., 2001, *Essential SourceSafe*, Hentzenwerke Publishing.

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

ROSETI, M.Z., 1998, *Uma Proposta de Sistemática para Aquisição de Conhecimento no Contexto de Análise de Domínio*, Dissertação de M.Sc., COPPE, UFRJ, Rio de Janeiro, Brasil.

SCHNEIDER, R.L., 2001, *Um Sistema de Gerência Cooperativa de Configuração de Software*, Dissertação de M.Sc., COPPE, UFRJ, Rio de Janeiro, RJ, Brasil.

SCOTT, J.A., NISSE, D., 2001, "Software Configuration Management", *Guide to Software Engineering Body of Knowledge*, chapter 66, IEEE Computer Society Press.

SEI, 2002a, *Capability Maturity Model Integration (CMMI) Version 1.1 - Continuous Representation*, Carnegie Mellon University.

SEI, 2002b, *Capability Maturity Model Integration (CMMI) Version 1.1 - Staged Representation*, Carnegie Mellon University.

SELECT BUSINESS SOLUTIONS, 2006, "Select Component Manager". In: http://www.selectbs.com, accessed in July 25.

SETTIMI, R., CLELAND-HUANG, J., KHADRA, O.B., *et al.*, 2004, "Supporting Software Evolution through Dynamically Retrieving Traces to UML Artifacts".

In: *International Workshop on Principles of Software Evolution (IWPSE)*, pp. 49-54, Kyoto, Japan, September.

SHAW, M., GARLAN, D., 1996, *Software Architecture: Perspectives on an Emerging Discipline*, Prentice Hall.

SHIRABAD, J.S., LETHBRIDGE, T., MATWIN, S., 2001, "Supporting Software Maintenance by Mining Software Update Records". In: *International Conference on Software Maintenance (ICSM)*, pp. 22-31, Florence, Italy, November.

SILVA, F.A., COSTA, R.V.C., EDELWEISS, N.*, et al.*, 2003, "Using the Temporal Versions Model in a Software Configuration Management Environment". In: *Brazilian Symposium on Software Engineering (SBES)*, pp. 302-317, Manaus, AM, Brasil, October.

SILVA, I.A.D., 2005, *GAW: Um mecanismo de percepção de grupo aplicado ao desenvolvimento de software.*, Projeto Final de Curso, DCC/IM, UFRJ, Rio de Janeiro, Brasil.

SMDS, 1994, *Aide de Camp Product Overview*, Software Maintenance & Development Systems.

SOFTEX, 2006a, *MPS.BR - Melhoria de Processo do Software Brasileiro - Guia de Avaliação (Versão 1.0)*, Associação para Promoção da Excelência do Software Brasileiro.

SOFTEX, 2006b, *MPS.BR - Melhoria de Processo do Software Brasileiro - Guia Geral (Versão 1.1)*, Associação para Promoção da Excelência do Software Brasileiro.

SPARX SYSTEMS, 2006, "Enterprise Architect". In: http://www.sparxsystems.com/products/ea.html, accessed in April 16.

SUN, 2006a, "Enterprise JavaBeans Technology". In: http://java.sun.com/products/ejb, accessed in June 21.

SUN, 2006b, "Javadoc Tool Home Page". In: http://java.sun.com/j2se/javadoc, accessed in June 21.

SVAHNBERG, M., GURP, J.V., BOSCH, J., 2005, "A Taxonomy of Variability Realization Techniques", *Software - Practice and Experience*, v. 15, n. 8 (July), pp. 705-754.

SZYPERSKI, C., 2002, *Component Software: Beyond object-oriented programming*, Addison-Wesley.

TECHTARGET, 2006, "WhatIs.com". In: http://www.whatis.com, accessed in June 20.

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

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

TEIXEIRA, H.V., MURTA, L.G.P., WERNER, C.M.L., 2001b, "LockED: Uma Ferramenta para o Controle de Alterações no Desenvolvimento Distribuído de Artefatos de Software". In: *Simpósio Brasileiro de Engenharia de Software (SBES), Sessão de Ferramentas*, pp. 380-385, Rio de Janeiro, RJ, Brasil, outubro.

TICHY, W., 1985, "RCS: a system for version control", *Software - Practice and Experience*, v. 15, n. 7 (July), pp. 637-654.

TIGRIS, 2006, "Scarab". In: http://scarab.tigris.org, accessed in July 8.

TORVALDS, L., 2006, "Linux". In: http://www.linux.org, accessed in June 20.

TORVALDS, L., HAMANO, J.C., 2006, "GIT". In: http://git.or.cz, accessed in June 20.

TRAVASSOS, G.H., 1994, *O Modelo de Integração de Ferramentas da Estação TABA*, Tese de D.Sc., COPPE, UFRJ, Rio de Janeiro, Brasil.

TRAVASSOS, G.H., GUROV, D., AMARAL, E.A.G.D., 2002, *Introdução à Engenharia de Software Experimental*, RT-ES-590/02, COPPE, UFRJ, Rio de Janeiro, Brasil.

VAN DER HOEK, A., 2004, "Design-Time Product Line Architectures for Any-Time Variability", *Science of Computer Programming*, v. 53, n. 3 (December), pp. 285-304.

VAN DER HOEK, A., CARZANIGA, A., HEIMBIGNER, D., *et al.*, 2002, "A Testbed for Configuration Management Policy Programming", *IEEE Transactions on Software Engineering (TSE)*, v. 28, n. 1 (January), pp. 79-99.

VASCONCELOS, A., 2004, *Uma Abordagem para Recuperação de Arquitetura de Software visando sua Reutilização em Domínios Específicos*, Exame de Qualificação, COPPE, UFRJ, Rio de Janeiro, Brasil.

VENUGOPALAN, V., 2002, *CVS Best Practices - Revision 0.6*, Free Software Foundation.

VERONESE, G.O., CORREA, A.L., JEZINNI, F., *et al.*, 2002, "ARES: Uma Ferramenta de Engenharia Reversa Java-UML". In: *Simpósio Brasileiro de*

Engenharia de Software (SBES), Sessão de Ferramentas, pp. 347-352, Gramado, RS, Brasil, outubro.

VERONESE, G.O., NETTO, F.J., 2001, *Uma Ferramenta de Auxílio à Recuperação de Modelos UML de Projeto a partir de Códigos Java*, Projeto Final de Curso, DCC/IM, UFRJ, Rio de Janeiro, RJ, Brasil.

VIEIRA, V., 2003, *ARIANE: Um Mecanismo de apoio à Percepção em Bases de Dados Compartilhadas*, Dissertação de M.Sc., COPPE, UFRJ, Rio de Janeiro, Brasil.

VINCENZI, A.M.R., MALDONADO, J.C., DELAMARO, M.E.*, et al.*, 2005, "Software Baseado em Componentes: Uma Revisão sobre Teste". In: GIMENES, I.M.D.S., HUZITA, E.H.M. (eds), *Desenvolvimento Baseado em Componentes: Conceitos e Técnicas*, 1a. ed., Rio de Janeiro, Brasil, Editora Ciência Moderna Ltda.

W3C, 1999, *HTML 4.01 Specification*, World Wide Web Consortium.

W3C, 2001, *XML Schema 1.0*, World Wide Web Consortium.

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

WALL, L., CHRISTIANSEN, T., ORWANT, J., 2000, *Programming Perl*, 3rd ed., O'Reilly Media.

WALLNAU, K., HISSAM, S., SEACORD, R., 2001, *Building Systems from Commercial Components*, Addison-Wesley Pub Co.

WALLS, C., RICHARDS, N., 2003, *XDoclet in Action*, Manning Publications.

WALRAD, C., STROM, D., 2002, "The Importance of Branching Models in SCM", *IEEE Computer*, v. 35, n. 9 (September), pp. 31-38.

WANG, Y., DE WITT, D.J., CAI, J., 2003, "X-Diff: An Effective Change Detection Algorithm for XML Documents". In: *International Conference on Data Engineering (ICDE)*, pp. 519-530, Bangalore, India, March.

WERNER, C.M.L., 1992, *Reutilização de Software no Desenvolvimento de Software Científico*, Tese de D.Sc., COPPE, UFRJ, Rio de Janeiro, Brasil.

WERNER, C.M.L., BRAGA, R.M.M., MATTOSO, M.L.Q.*, et al.*, 2000, "Infra-estrutura Odyssey: estágio atual". In: *Simpósio Brasileiro de Engenharia de Software (SBES), Sessão de Ferramentas*, pp. 366-369, João Pessoa, Brasil, outubro.

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), pp. 61-68, Las Vegas, USA, October.

WERNER, C.M.L., MANGAN, M.A.S., MURTA, L.G.P.*, et al.*, 2002, "Odyssey Share: Um Ambiente para o Desenvolvimento Cooperativo de Componentes". In: *Simpósio Brasileiro de Engenharia de Software (SBES), Sessão de Ferramentas*, pp. 444-449, Gramado, RS, Brasil, outubro.

WERNER, C.M.L., MATTOSO, M., BRAGA, R.M.M.*, et al.*, 1999, "Odyssey: Infra-estrutura de Reutilização Baseada em Modelos de Domínio". In: *Simpósio Brasileiro de Engenharia de Software (SBES), Sessão de Ferramentas*, pp. 17-20, Florianópolis, SC, Brasil, outubro.

WERNER, C.M.L., TRAVASSOS, G.H., ROCHA, A.R.C.*, et al.*, 1997, "Memphis: A Reuse Based OO Software Development Environment". In: *Technology of Object-Oriented Languages and Systems (TOOLS)*, pp. 182-191, Beijing, China, September.

WESTHUIZEN, C., VAN DER HOEK, A., 2002, "Understanding and Propagating Architectural Changes". In: *Working IEEE/IFIP Conference on Software Architecture (WICSA)*, pp. 95-109, Montreal, Canada, August.

WHITE, B.A., 2000, *Software Configuration Management Strategies and Rational ClearCase: A Practical Introduction*, Addison-Wesley.

XAVIER, J.R., 2001, *Criação e Instanciação de Arquiteturas de Software Específicas de Domínio no Contexto de uma Infraestrutura de Reutilização*, Dissertação de M.Sc., COPPE, UFRJ, Rio de Janeiro, Brasil.

YING, A.T.T., MURPHY, G.C., NG, R.*, et al.*, 2004, "Predicting Source Code Changes by Mining Change History", *IEEE Transactions on Software Engineering (TSE)*, v. 30, n. 9 (September), pp. 574-586.

ZHANG, L., MEI, H., ZHU, H., 2001, "A Configuration Management System Supporting Component-Based Software Development". In: *International Computer Software and Applications Conference (COMPSAC)*, pp. 25-30, Chicago, IL, USA, October.

ZHAO, J., YANG, H., XIANG, L.*, et al.*, 2002, "Change impact analysis to support architectural evolution", *Journal of Software Maintenance: Research and Practice*, v. 14, n. 5 (September), pp. 317-333.

ZIMMERMANN, T., WEISGERBER, P., DIEHL, S., *et al.*, 2004, "Mining version histories to guide software changes". In: *International Conference on Software Engineering (ICSE)*, pp. 563-572, Edinburgh, Scotland, May.