Um controle de versões refinado e flexível
para artefatos de software

Daniel Cárnio Junqueira

Um controle de versões refinado e flexível para artefatos de software

Daniel Cárnio Junqueira

Orientador: ***Profa. Dra. Renata Pontin de Mattos Fortes***

Dissertação apresentada ao Instituto de Ciências Matemáticas e de Computação - ICMC-USP, como parte dos requisitos para a obtenção do título de Mestre em Ciências de Computação e Matemática Computacional.

USP - São Carlos

Dezembro/2007

Aos meus pais, à Mary e ao Henrique

Agradecimentos

A Deus, que torna tudo possível.

Aos meus pais, por terem me criado, educado, e colaborado com este trabalho de forma constante.

À Mary, pelo imenso apoio e paciência durante esta jornada; ao Henrique, por existir e iluminar nossas vidas de forma tão especial.

À Quel, que também teve boas doses de paciência.

À Tati, Rê, Lígia e Bia, pelo apoio que sempre deram em tudo o que fiz em minha vida, algumas vezes de perto, outras à distância – mas sempre apoiando de alguma forma.

Ao André e Berbert, que também sempre me incentivaram e apoiaram nesta jornada.

A todos os amigos de São Carlos.

Por fim, agradeço à Renata, que me orientou durante tanto tempo e tornou possível este trabalho.

Sumário

Lista de Figuras

Lista de Tabelas

Lista de Acrônimos

API	*Application Program Interface*
CRM	*Customer Relationship Management*
CSSC	*Compatibly Stupid Source Control*
DoD	*Department of Defense*
EBNF	*Extended Backus-Naur Form*
GCS	Gerenciamento de Configuração de Software
GPL	*General Public License*
GNU	*GNU's Not Unix*
IDE	Integrated Development Environment
IEEE	*Institute of Electrical and Electronics Engineers*
ISO	*International Standards Organization*
RCS	*Revision Control System*
SCCS	*Source Code Control System*
SGBD	Sistema Gerenciador de Banco de Dados
UEVM	*Unified Extensional Versioning Model*

Resumo

As atividades de controle de versões são consideradas essenciais para a manutenção de sistemas de computador. Elas começaram a ser realizadas na década de 1950 de forma manual. As primeiras ferramentas de controle de versões, que surgiram nos anos setenta, não evoluíram significativamente desde sua criação e, até hoje, o controle de versões de arquivos é geralmente realizado em arquivos ou mesmo módulos completos, utilizando os conceitos que foram lançados há mais de três décadas.

Com a popularização da utilização de sistemas computacionais, ocorreu um sensível aumento no número de sistemas existentes e, também, na complexidade dos mesmos. Além disso, muitas alterações ocorreram nos ambientes de desenvolvimento de software, e existe demanda por sistemas que permitam aos desenvolvedores ter cada vez mais controle automatizado sobre o que está sendo desenvolvido. Para isso, algumas abordagens de controle de versões refinados para artefatos de software foram propostas, mas, muitas vezes, não oferecem a flexibilidade de utilização exigida pelos ambientes de desenvolvimento de software.

Neste trabalho, é apresentado um sistema que visa a fornecer suporte ao controle de versões refinado e flexível para artefatos de software, tendo por base um modelo bem definido para representação das informações da estrutura dos arquivos que compõem determinado projeto de software, sejam eles código-fonte dos programas de computador, documentação criada em Latex, arquivos XML, entre outros. O sistema apresentado foi projetado para ser integrado com outras soluções utilizadas em ambientes de desenvolvimento de software.

Palavras-chaves: controle de versões, controle de versões refinado, gerenciamento de configuração de software

Abstract

Version control tasks are considered essential for the maintenance of computers systems. They have been done since beginning of 50's in a by hand manner. First tools, which were released in 70's, didn't evolve significantly since its creation, and, in general, version control systems still work with entire files or even modules of software, having the same concepts that were launched more than three decades ago.

With the popularization of computers systems there had been a sensible increase in the number of existing systems and also in the complexity of these systems. Besides that many changes have taken place in the software development environments, and there is demand for systems which allow developers to have more automated control about what is being developed. Regard to this demand some approaches of fine-grained version control have been proposed, but they usually do not provide the required flexibility for its use in the real software development environments.

In this work its presented a system which aims at providing support for flexible and fine-grained version control of software artifacts, using a well defined model to represent the logical structure of the files which compose a software project, independently of its type - they can be XML files, source-code files, Latex files and others. The system has been designed to be integrated with other software solutions used in software development environments.

Keywords: version control, fine-grained version control, software configuration management

Capítulo

1

Introdução

Controle de versões, no contexto de desenvolvimento de **artefatos de software**, refere-se à denominação para as técnicas e ferramentas[1] utilizadas para controlar a evolução de arquivos de computador. Os registros das primeiras atividades de controle de versões datam por volta de 1950, época em que eram realizadas de forma manual (BERLACK, 1991). Em 1972 foi lançado o sistema *Source Code Control System* (SCCS), que é considerada a primeira ferramenta de controle de versões (GLASSER, 1978).

Apesar das evoluções tecnológicas e das freqüentes alterações nos ambientes de negócio em que as técnicas de controle de versões são utilizadas (ESTUBLIER et al., 2005), a forma como as versões são controladas em sistemas de controle de versões não tiveram grandes alterações (CONRADI; WESTFECHTEL, 1998). Novos sistemas foram criados, mas, em geral, foram melhoradas as técnicas de armazenamento, controle de acesso, distribuição de repositórios, e integração com outras ferramentas para realizar os processos integrantes de Gerenciamento de Configuração de Software (GCS) – mas a forma como as versões de código fonte são controladas permanece quase inalterada há três décadas, e pouca ou nenhuma inteligência foi adicionada aos sistemas de controle de versões e mesmo aos processos de GCS (KEYES, 2004).

Pesquisadores têm investido em novas estratégias de controle de versões de forma a permitir um maior detalhamento das informações que estão sendo controladas, através da utilização de informações semânticas ou da estrutura dos documentos para fazer as atividades

[1] Neste trabalho, os termos "ferramentas de controle de versões" e "sistemas de controle de versões" serão utilizados de forma indistinta para denotar o conjunto de programas de computador que automatiza as tarefas para controle de versões

de controle das versões. Com esta abordagem, é possível disponibilizar informações mais precisas às pessoas envolvidas no processo de produção de software. Vale ressaltar que o termo original em inglês utilizado para este tipo de controle de versões é *fine-grained version control* (THOMAS; JOHNSON, 1988), cuja tradução utilizada neste trabalho é **controle de versões refinado** ou **controle de versões com granulosidade fina**[2].

Esta abordagem tem por objetivo fornecer melhor controle sobre os documentos ou arquivos de código fonte que estão no repositório de controle de versões, permitindo um mapeamento adequado da estrutura dos arquivos de computador no sistema de controle de versões (LIN; REISS, 1996; NGUYEN et al., 2004) e, conseqüentemente, melhores informações para que se realize o rastreamento mais detalhado da evolução dos softwares.

No entanto, apesar de alguns sistemas terem sido propostos e implementados para oferecer suporte às atividades de controle de versões refinado, os sistemas que foram desenvolvidos focaram na definição e implementação de modelos conceituais para o controle de versões refinado, não oferecendo suporte à flexibilidade das soluções desenvolvidas, não permitindo, dessa forma, a integração com ferramentas já existentes nos ambientes de desenvolvimento de software, nem permitindo aos desenvolvedores de software a utilização das ferramentas com as linguagens de programação com as quais já trabalham, pois muitas vezes o mecanismo de mapeamento entre a gramática do documento e o modelo conceitual utilizado pela ferramenta de controle de versões refinado é programado juntamente com a lógica do sistema inteiro (LIN; REISS, 1996; BENDIX et al., 1998) – não sendo, portanto, flexível – ou é de difícil utilização (MAGNUSSON; ASKLUND, 1996).

Assim, o objetivo deste trabalho foi desenvolver um sistema que permitisse a realização das atividades de controle de versões refinado, tendo enfoque nas características de flexibilidade do sistema, sendo considerados tanto o mapeamento entre a estrutura dos documentos e o modelo de dados implementado no sistema, como a possibilidade de integração do sistema com outras ferramentas, de forma a minimizar o impacto de sua adoção em ambientes de desenvolvimento de software já existentes, nos quais pode, então, ser dada liberdade aos desenvolvedores para trabalharem com as ferramentas que desejarem.

Através do estudo dos principais modelos propostos para a realização de atividades de controle de versões refinado, além do levantamento das principais características dos sistemas de controle de versão com repositórios centralizados e distribuídos, foi projetado o sistema *Phoca*. O sistema foi desenvolvido tendo em vista que o principal requisito não-funcional foi que o sistema deveria ser flexível.

Como resultado, foi implementado o sistema *Phoca* em linguagem Java e foi licenciado

[2] Os termos granulosidade e granularidade são encontrados na literatura da área de Engenharia de Software como sinônimos. Neste trabalho será utilizado o termo "granulosidade fina" para expressar um maior nível de detalhamento – enquanto "granulosidade grossa" refere-se a um menor nível de detalhamento.

sob a licença de código aberto[3] Apache versão 2. Esta licença foi escolhida por permitir que futuros desenvolvedores utilizem o código em suas aplicações ou desenvolvam novas aplicações a partir do código existente, e não precisam enviar as contribuições de volta para a comunidade. Este modelo tem provado sua eficiência, pois permite que empresas que não querem ou não podem divulgar particularidades de seus processos adotem tais software, adaptando-os às suas necessidades, sem a necessidade de divulgar suas alterações.

O sistema desenvolvido pode, então, ser utilizado em ambientes reais de desenvolvimento de software, seja em meio acadêmico ou corporativo, e pode beneficiar as diversas pessoas envolvidas com desenvolvimento de software – tanto os desenvolvedores, que passam a ter uma visão mais clara dos programas desenvolvidos e alterações realizadas, quanto gerentes, clientes internos e externos, orientadores em ambiente acadêmico, entre outros.

O restante desta dissertação está organizado da seguinte forma: no Capítulo 2 é apresentada uma revisão sobre os principais conceitos e sistemas de controle de versões, além das considerações sobre a relação existente entre as atividades de controle de versões e o processo de GCS. No Capítulo 3 é apresentada uma revisão da literatura sobre os principais sistemas de controle de versões refinado, sendo apresentado o resultado na forma de uma análise comparativa entre as soluções que já foram propostas. No Capítulo 4 é descrito o sistema *Phoca*, principal resultado deste trabalho. Por fim, no Capítulo 5 são apresentadas as principais conclusões sobre o trabalho realizado e são indicados possíveis trabalhos futuros.

Ao final do trabalho, é apresentado um Glossário, que contém a definição dos principais termos utilizados. Sempre que possível, os termos que podem ser encontrados no Glossário serão destacados em **negrito**.

[3] Neste trabalho, será utilizado o termo "código aberto" como tradução do termo original em inglês *open source*. Eventualmente, o termo projeto de software livre também será utilizado

Capítulo

2

Controle de Versões

Controle de versões, também conhecido por controle de revisões, refere-se à técnica de controlar a evolução de conteúdo criado ao longo do tempo, permitindo a recuperação de dados históricos, diferenças entre versões, e detalhes sobre a evolução de determinado conteúdo que tenha tido suas versões controladas (TICHY, 1982). Embora sua aplicação possa ser feita a qualquer tipo de conteúdo e sobre conteúdo gerado em qualquer mídia, neste capítulo será apresentado o controle de versões focado em arquivos de computador em geral, e em arquivos relacionados com a produção de software em particular. Neste trabalho serão utilizados os termos "controle de versões" e "controle de versões de software" para fazer referência às tarefas de controle da evolução de arquivos de computador. Na literatura encontram-se também outras definições, como controle de revisões de software, gerenciamento de código fonte (SCM - *Source Code Management*) e controle de código fonte (SCC - *Source Code Control*). Muitas vezes, também encontra-se o termo Gerenciamento de Configuração de Software aplicado como sinônimo de controle de versões. Neste trabalho, os termos controle de versões e Gerenciamento de Configuração de Software serão utilizados de forma distinta.

O controle de versões pode ser realizado completamente de forma manual. Existem relatos da realização de atividades de controle de versões manual de arquivos por volta da década de 1950. Segundo estes relatos, o código fonte de programas era armazenado no formato de cartões perfurados, e criavam-se padrões para escrever o nome do software, o autor daquela versão e qual o número que identificava a versão. Com isso, matinha-se um registro das versões de código fonte, testes associados a elas, resultados obtidos, entre outros, e, de forma sistematizada, era possível recuperar as versões desejadas conforme a necessidade (BERLACK, 1991).

Mesmo com o advento de mídias magnéticas, no começo fazia-se o controle manual de versões, mantendo-se cópias das diversas versões dos arquivos em disquetes separados. De certa forma, esse mecanismo apenas permitiu a redução do tamanho físico dos dados armazenados, mas manteve-se a mesma técnica de controle manual utilizada anteriormente (BERLACK, 1991; KEYES, 2004). A partir da década de 1970, foram desenvolvidas as primeiras ferramentas para realização automatizada das atividades de controle de versões. Inicialmente, o controle de versões era realizado de forma completamente centralizada em *mainframes*, com disco de dados compartilhado entre os usuários; em meados da década de 1980, começaram a ser utilizados, então, os computadores pessoais, e cada desenvolvedor podia trabalhar em uma área de dados não compartilhada. Da mesma forma que ocorreram estas evoluções na forma como os usuários trabalham com os computadores, as técnicas e ferramentas de controle de versões também evoluíram – embora alguns conceitos da primeira ferramenta tenham se mantido inalterados.

Neste capítulo são apresentados os principais conceitos relacionados a controle de versões, as principais ferramentas utilizadas, sua evolução, além do relacionamento entre controle de versões e GCS. O capítulo está organizado da seguinte forma: na Seção 2.1 são apresentados os principais conceitos relacionados à área de controle de versões; na Seção 2.2 são apresentadas as principais ferramentas de controle de versões e uma descrição de sua evolução; na Seção 2.3 são apresentadas questões atuais sobre a integração de controle de versões em outras ferramentas; na Seção 2.4 são apresentados alguns conceitos sobre a área de GCS e como as atividades de controle de versões estão integradas com a área; por fim, na Seção 2.5 são apresentadas as considerações finais pertinentes a este capítulo.

2.1. Principais conceitos de Controle de Versões

Para melhor entendimento, alguns conceitos e terminologia sobre controle de versões serão explicados nesta seção. Tais conceitos foram estudados a partir dos fundamentos encontrados em (IEEE Std 610, 1991), (TICHY, 1982), (AMBRIOLA; BENDIX; CIANCARINI, 1990).

Commit e Checkout

Commit (ou *checkin*) e *checkout* são dois dos principais conceitos de controle de versões. Pode-se dizer que a operação de *checkout* é análoga à operação de abrir um documento de texto num processador de textos tradicionais, e a operação de *commit* é análoga à operação de salvar este documento. Entretanto, o *commit* e o *checkout* referem-se a operações realizadas num sistema de controle de versões; portanto, os arquivos abertos e salvos possuem versões

associadas. *Checkout* envolve a recuperação de uma determinada versão do arquivo, enquanto *commit* envolve a criação de uma versão nova.

Para exemplificar estes conceitos, que são conceitos-chave na área de controle de versões, pode-se imaginar um usuário que pretende gravar as diferentes versões de um documento de texto no qual ele está trabalhando. Uma opção é ele renomear este arquivo a cada operação de salvar, escolhendo, por exemplo, os nomes "Documento_texto1.txt", "Documento_texto2.txt", e assim sucessivamente. Se este usuário utilizar um sistema de controle de versões, ele pode simplesmente realizar uma operação de *commit* a cada vez que quiser salvar uma versão diferente e uma operação de *checkout* a cada vez que quiser recuperar uma versão específica deste documento. Os sistemas de controle de versões, além de permitirem o acompanhamento da evolução do conteúdo, também realizam o armazenamento de forma aprimorada, utilizando menos espaço físico em disco do que seria utilizado se o usuário realizasse o controle de versões de forma manual, pois utiliza conceitos de cálculo de diferenças para montar a árvore de revisões.

Repositório de Versões

Para o funcionamento dos sistemas de controle de versões, é necessário que exista um local que armazene todas as versões dos arquivos sob controle de versões. O nome dado a este depósito de versões é repositório de versões ou, simplesmente, repositório. É no repositório que ficam armazenadas as árvores de versões. Os arquivos do repositório não são acessados diretamente pelos usuários, mas são copiados para a "área de trabalho" ou "cópia local" dos mesmos. Para copiar um arquivo do repositório para a cópia local são utilizados comandos específicos de cada sistema de controle de versões, mas, geralmente, o nome dos comandos é exatamente *checkout* e *commit*.

Existem dois tipos de repositórios de versões: repositórios centralizados e repositórios distribuídos.

Com repositórios centralizados, cada área de trabalho local contém apenas uma versão específica da árvore de versões do repositório central, e todos os usuários realizam as operações de controle de versões no repositório central.

Com repositórios distribuídos cada área de trabalho local possui um repositório acoplado, de forma que o usuário tem um repositório próprio para realizar o controle de versões. As operações realizadas sobre os arquivos são feitas no repositório local do usuário, e operações específicas dos repositórios distribuídos são utilizadas para sincronizar repositórios diferentes.

Sistema de Controle de Revisão Centralizado

Um Sistema de Controle de Revisão Centralizado é aquele em que existe um único repositório, centralizado. Neste tipo de sistema, cada cópia de trabalho local refere-se a uma única revisão de cada vez, e, para realizar qualquer atividade relacionada ao controle de versões dos arquivos, cada usuário precisa se conectar ao repositório central. Este é o sistema mais comum utilizado hoje. O CVS (CEDERQVIST et al., 2007) e o SubVersion (COLLINS-SUSSMAN; FITZPATRICK; PILATO, 2007) são exemplos de sistemas de controle de revisão centralizados. Na Figura 2.1 é exibida a configuração típica do repositório de um sistema de controle de revisão centralizado. Nela estão representados quatro usuários que possuem cópias de trabalho do repositório central, o qual possui as informações de todas as versões dos arquivos. Neste caso, diz-se que os usuários estão trabalhando de forma concorrente no repositório. Quando cada usuário realiza um *commit*, imediatamente todos os outros usuários são aptos a atualizar suas cópias de trabalho para visualizarem as alterações que o usuário que realizou *commit* enviou.

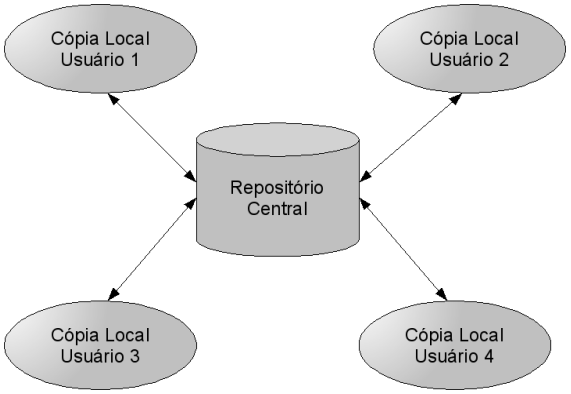

Figura 2.1. Exemplo de configuração típica do repositório para sistema de controle de revisão centralizado

Sistema de Controle de Revisão Distribuído

Um Sistema de Controle de Revisão Distribuído é aquele que utiliza repositórios distribuídos. Nestes sistemas, cada usuário possui todo um repositório local, e a integração de seu trabalho com o de outros usuários ocorre através de sincronização entre repositórios. Nos Sistemas de Controle de Revisão Distribuídos podem ser implementados alguns repositórios centrais. Os sistemas distribuídos são adotados há muitos anos para o desenvolvimento do ***kernel*** do sistema operacional **Linux**, e grandes projetos de código aberto têm migrado recentemente para sistemas deste tipo – como, por exemplo, projeto **Mozilla**. O Git (GARZIK, 2005), o Bazaar (BLACKWELL et al., 2006) e o Mercurial (MERCURIAL, 2007) são exemplos de sistemas de controle de revisão distribuídos.

Nos sistemas de controle de revisão distribuídos é possível configurar diversos tipos de topologias – pode-se permitir, por exemplo, que todos os usuários realizem sincronização entre si, ou com alguns repositórios centrais. Na Figura 2.2 é apresentado um exemplo de possível configuração de repositórios distribuídos. Nela são apresentados quatro usuários e seus respectivos repositórios locais; cada um deles pode obter as versões dos demais (todo o histórico de versões), através das operações de sincronização de repositórios.

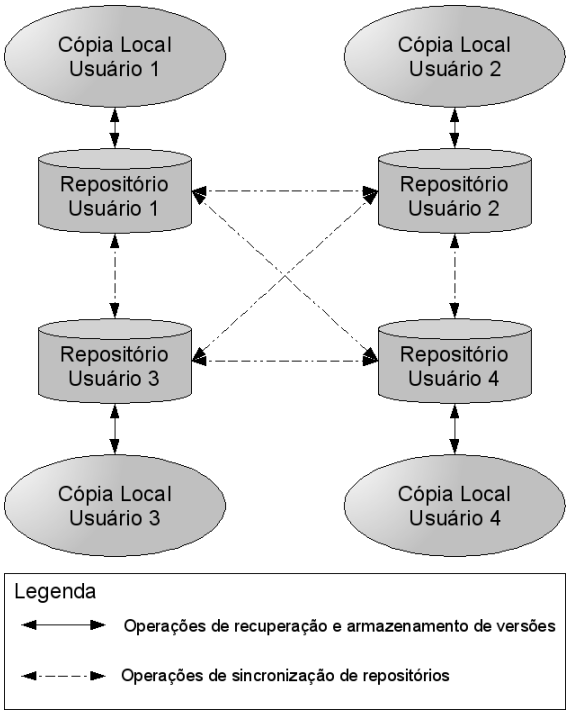

Figura 2.2. Exemplo de configuração de repositórios distribuídos em sistema de controle de revisão distribuído

Cópia de trabalho

A cópia de trabalho, também conhecida por **área local** ou **cópia de trabalho local**, refere-se a uma área do desenvolvedor na qual ele trabalha com uma versão específica de um arquivo que está sob controle de versões no repositório. No caso de repositório centralizado, a cópia de trabalho sempre contém apenas uma versão do repositório central e, sempre que alguma operação de controle de versões necessitar ser realizada, é necessário estabelecer conexão com o repositório central – o que muitas vezes significa que o usuário precisará ter acesso à internet ou intranet da empresa. Com sistemas de controle de revisão distribuídos, a cópia de trabalho possui a versão atual em que o usuário está trabalhando e também possui um repositório acoplado, que contém todo o histórico de versões – neste caso, as operações de controle de versões podem ser feitas localmente, ou seja, mesmo que o usuário não tenha acesso à internet ou intranet da empresa ou universidade onde o repositório central esteja

hospedado.

Merge e Conflito

Muitos sistemas de controle de versões permitem que os usuários trabalhem de forma concorrente nos arquivos, ou seja, mais de um usuário pode estar editando um mesmo arquivo ao mesmo tempo. O propósito dos sistemas de controle de versões é justamente permitir este trabalho concorrente, que pode aumentar a produtividade e permitir que diversos desenvolvedores atuem sobre arquivos em comum, cada um em uma área diferente.

Quando o trabalho dos usuários é realizado em regiões diferentes dos arquivos (linhas diferentes), uma solução adotada por muitos sistemas de controle de versões é a realização de *merge* ou intercalação das alterações. Estes algoritmos calculam a diferença de linhas e criam uma versão final de um arquivo comum com as alterações dos dois desenvolvedores. Na Figura 2.3 é ilustrada uma situação em que a intercalação dos arquivos foi realizada com sucesso pelo sistema de controle de versões. Nesta figura, observa-se que os dois usuários realizaram um *checkout* no mesmo instante de tempo, e cada um passou a ter a versão 1 em sua cópia de trabalho. Na mesma versão, cada um realizou a alteração de uma linha do arquivo, tendo o usuário 1 alterado a linha 1 e o usuário 2 alterado a linha 2. Posteriormente, o primeiro usuário realizou um *commit* antes do segundo, gerando, portanto, a versão 2 no repositório. Neste instante, o segundo usuário estará "desatualizado" em relação à versão do repositório central. No momento em que o segundo usuário tenta fazer um *commit*, ele será notificado de que precisa fazer uma atualização, pois está inconsistente em relação ao repositório. Neste instante, o usuário pode atualizar sua versão local e, então, é realizada a operação de *merge*, que integra as alterações dos dois usuários. Então, o segundo usuário pode proceder com o *commit*, criando a versão 3 no repositório – que contém as alterações dos dois usuários.

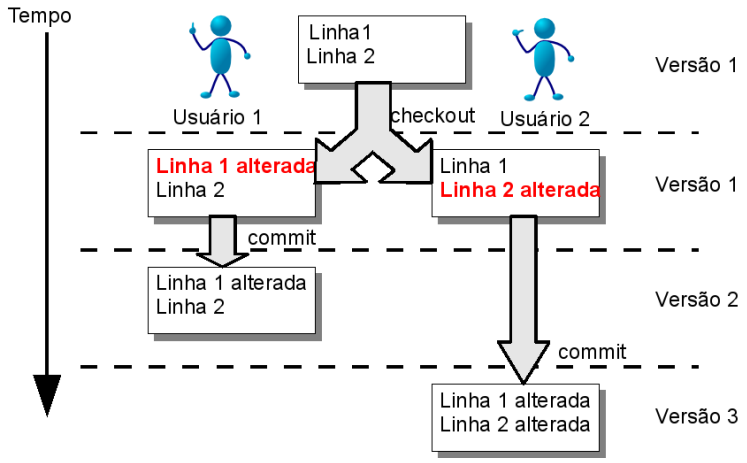

Figura 2.3. Exemplo de *merge* bem sucedido

Entretanto, muitas vezes ocorre que os desenvolvedores alteram regiões comuns dos arquivos. Nestes casos, podem ocorrer situações de conflito. Na Figura 2.4 é ilustrado um exemplo em que ocorre conflito, no qual não é possível realizar a operação de *merge* automaticamente. Na situação ilustrada nesta figura, os dois usuários envolvidos realizam *checkout* da versão 1, e cada um altera a linha 1, de formas diferentes. O primeiro usuário realiza um *commit* bem sucedido e cria a versão 2 no repositório. No momento em que o usuário 2, que estará com sua versão desatualizada em relação à versão do repositório, tentar fazer seu *commit* será criada uma situação de **conflito**. Nestes casos, não é possível realizar a operação de *merge* de forma automática, pois é necessário tomar uma decisão sobre qual será o conteúdo que deverá prevalecer na linha 1. Estas são as situações conhecidas como **conflito**, que só é possível ocorrer em sistemas que permitem o desenvolvimento concorrente dos usuários. Nestes casos, faz-se necessária a intervenção dos usuários envolvidos.

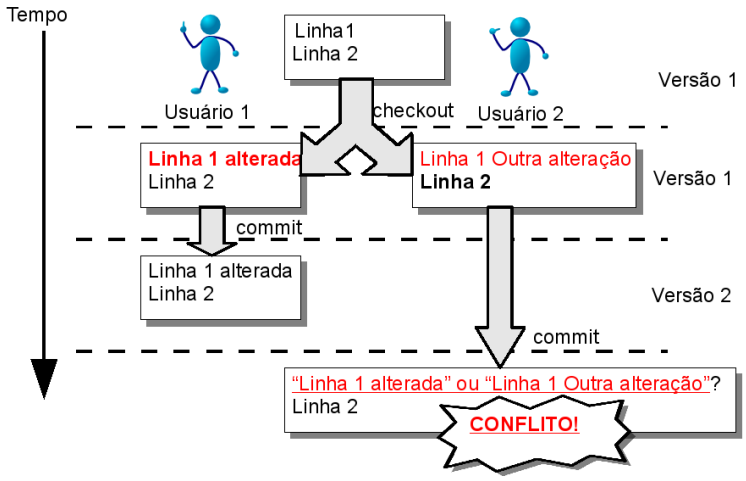

Figura 2.4. Exemplo de situação de conflito - não é possível realizar a operação de *merge*

Árvore de revisões

Árvore de revisões ou de versões refere-se a uma estrutura lógica que mapeia todas as versões armazenadas no repositório para determinado arquivo ou conjunto de arquivos. A analogia é feita com uma árvore porque existe um "tronco" e "ramos" ou "ramificações". Na Figura 2.5 é mostrado um exemplo de árvore de revisões. As versões 1, 2, 3, 4 e 5 formam o tronco da árvore; as versões 3.1, 3.2 e 3.3 formam um ramo e a versão 2.1 forma outro ramo da árvore. As versões 5, 3.3 e 2.1 são chamadas de "cabeças" de seus respectivos ramos, sendo a 5 a cabeça principal.

Qualquer alteração nas versões "cabeça" da árvore irá gerar uma nova cabeça, que substituirá e será seqüencial à anterior. As alterações nas versões que não são cabeça só podem ser salvas como novos ramos. Para exemplificar, vemos que na Figura 2.5 as próximas versões

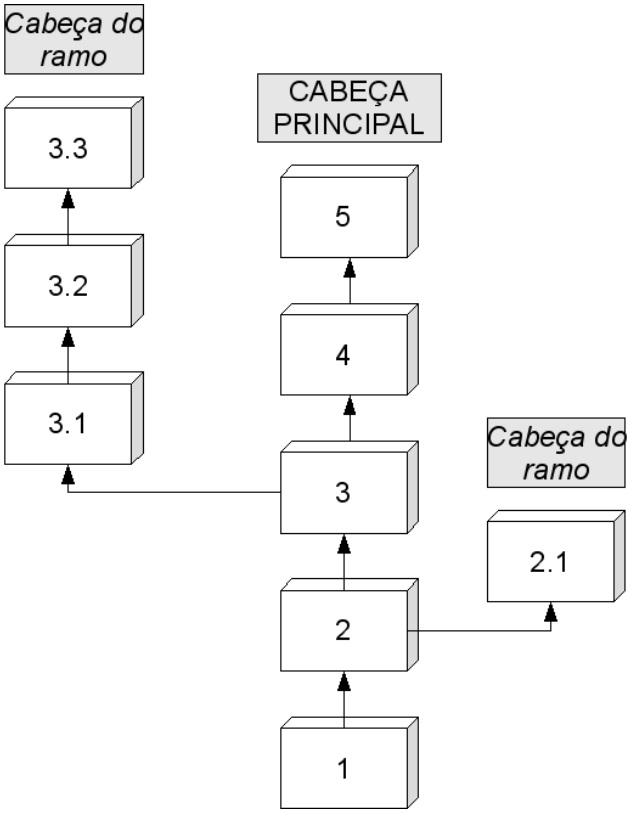

Figura 2.5. Exemplo de árvore de versões

possíveis sem a criação de novos ramos são 3.4, 6 e 2.2. Caso qualquer outra versão seja editada, será necessário criar um novo ramo.

A árvore de versões pode ser criada de duas formas, dependendo do sistema de controle de versões utilizado: árvore de versões por arquivos, na qual cada arquivo tem sua versão controlada individualmente e qualquer combinação de diferentes versões de arquivos deve ser feito manualmente – portanto, *releases* de produtos devem ser sempre marcadas no repositório –, ou árvore de revisão por repositório, que é utilizada pelos sistemas que utilizam estratégia de numeração de versões global. Nestes casos, todo o conjunto de arquivos do repositório recebe um número único, que corresponde àquela combinação de todos os arquivos em suas determinadas versões em certos instantes do tempo.

2.2. Evolução das Ferramentas de Controle de Versões

Nesta seção é apresentada a evolução cronológica das ferramentas de controle de versões. São mostrados os principais sistemas e as principais datas relacionadas a controle de versões. Diversas ferramentas comerciais e de código aberto foram lançadas, principalmente ao longo da década de 1990, mas só serão exibidas aquelas que trouxeram, de alguma forma, alguma inovação. Na Figura 2.6 é exibida a linha do tempo da evolução das ferramentas de controle de versões. Seu início é mostrado em 1960 e a marca mais recente data de 2005,

com as ferramentas Bazaar (BLACKWELL et al., 2006), Git (GARZIK, 2005) e Mercurial (MERCURIAL, 2007).

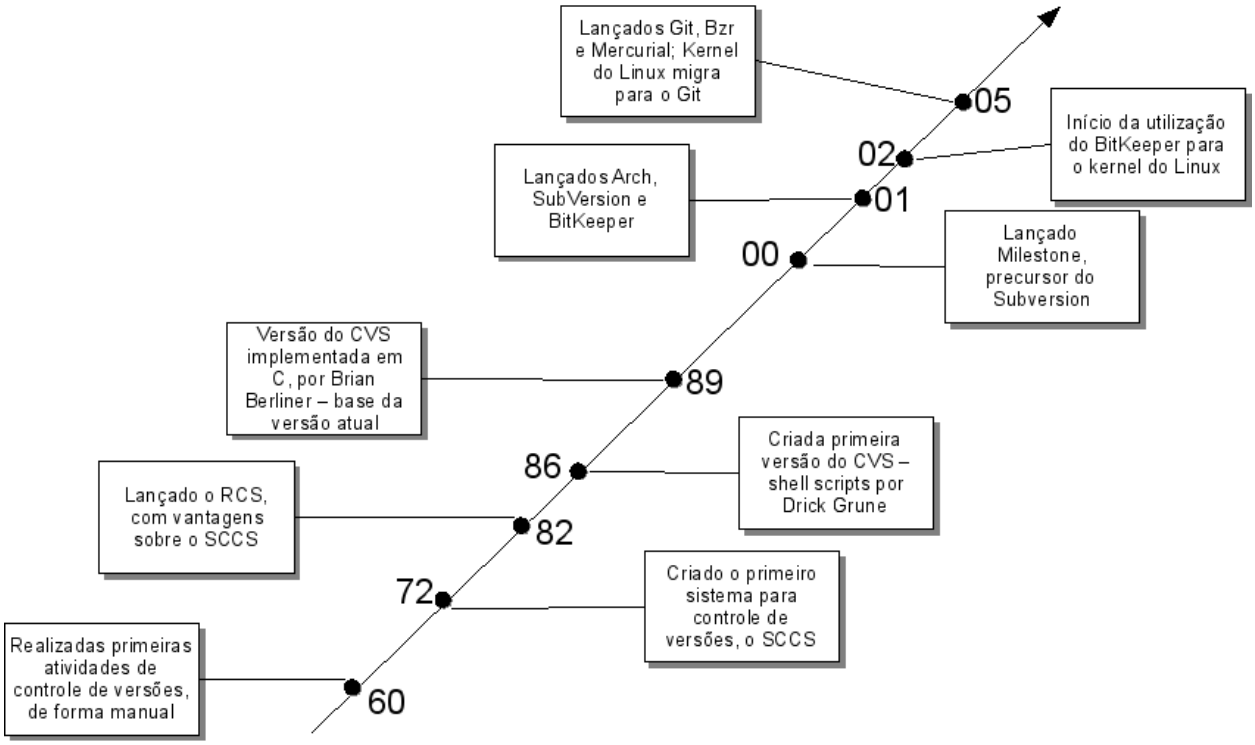

Figura 2.6. Linha do tempo das ferramentas de controle de versões

Na década de 1970 surgiram as primeiras ferramentas disponibilizadas comercialmente para suportar as atividades de controle de versões. É nesta época que se tem conhecimento da implementação da primeira ferramenta de controle de versões, o SCCS (ROCHKIND, 1975). Ele foi criado em 1972 por Marc Rochkind, nos laboratórios da **Bell Telephone Laboratories** (ROCHKIND, 1975; GLASSER, 1978). O SCCS foi rapidamente portado para o sistema operacional Unix e passou a ser distribuído juntamente com o *Programer's Workbench* (PWB) da **AT&T**. Ele oferece aos usuários comandos para controle de versões, permitindo operações sobre arquivos, como colocar um arquivo sob controle de versões, exibir diferenças entre versões de um arquivo, imprimir determinada versão de um arquivo, entre outras. Atualmente, o SCCS é distribuído com sistemas Unix comerciais, principalmente com os sistemas System V[1] (BOLINGER; BRONSON, 1995). Além disso, foi feita uma implementação *open source* deste sistema em 1998, chamada *Compatibly Stupid Source Control* (CSSC), mas que, segundo o próprio autor, deve ser utilizada apenas para a recuperação de dados de repositórios legados que tenham sido criados com SCCS [2].

[1] System V é uma versão do sistema operacional Unix, desenvolvida pela AT&T. Várias empresas, como SunOS e SCO utilizam partes do sistema System V em suas distribuições

[2] Disponível em: `http://cssc.sourceforge.net`

O ciclo básico de funcionamento do SCCS consiste em realizar primeiramente uma operação para inicializar um arquivo para que seja controlado pelo SCCS. Posteriormente, o sistema oferece ao usuário as opções de editar um arquivo (*"get -e"*) e salvar as versões quando achar necessário (*"delta"*). Neste sistema foi utilizado o conceito de "deltas positivos", de forma que a primeira versão colocada sobre controle é armazenada integralmente, e, a cada nova versão criada, armazena-se as diferenças entre elas, em termos de diferenças de linhas - caso um único caractere de uma linha toda seja alterado, o SCCS armazena a informação de que toda a linha foi excluída de uma versão e a linha alterada foi incluída na versão nova (ROCHKIND, 1975). Deve-se notar que, com a utilização de deltas positivos, a recuperação de qualquer versão posterior à primeira exige a reconstrução da versão solicitada, e, quanto mais recente a versão a ser recuperada, maior o processamento necessário. Na Figura 2.7 é exemplificado o controle de versões de arquivos no SCCS. Nela é possível observar que apenas a primeira versão de um arquivo é armazenada integralmente; para a recuperação das outras versões, é necessário aplicar as diferenças entre as diferentes versões em relação à versão original.

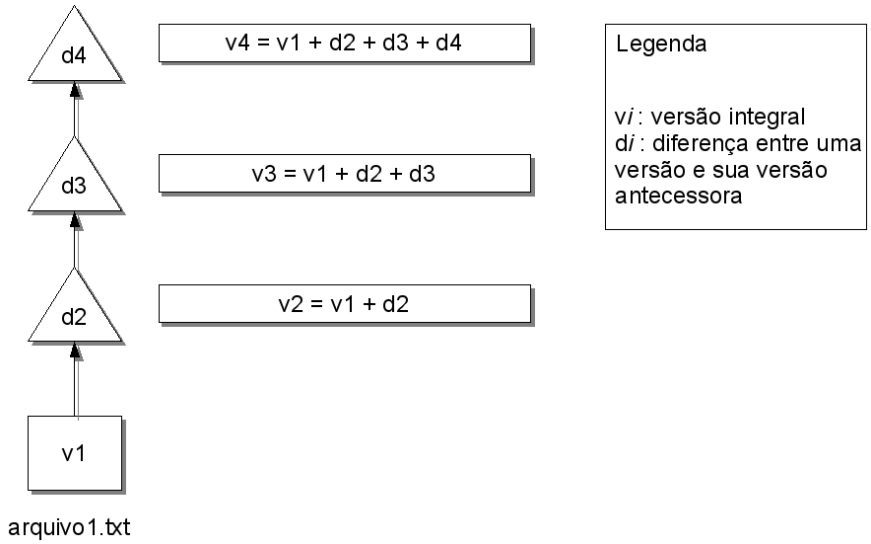

Figura 2.7. Exemplo de árvore de revisões utilizada pelo SCCS

O ambiente no qual o SCCS era utilizado difere bastante dos sistemas atuais. O sistema foi implementado para rodar em ***mainframes*** – IBM 370 rodando **OS** e **PDP 11** rodando Unix. Neste cenário, cada desenvolvedor envolvido num projeto tinha acesso ao *mainframe*, e o sistema de arquivos era compartilhado – utilizavam-se **terminais *slaves***. Portanto, o SCCS era implantado no servidor e os usuários podiam inicializar os "módulos" dos softwares desenvolvidos – cada módulo correspondendo a um arquivo de código fonte. Podia-se, então, dar permissão de edição aos módulos para cada usuário ou grupo de usuários, através de comandos específicos do SCCS. Para evitar problemas de edição concorrente de um mesmo arquivo, o SCCS possuía mecanismos de *lock* implementados (ROCHKIND, 1975).

Dez anos após o lançamento do SCCS, foi lançada a primeira versão do *Revision Control System* (RCS), em 1982, de autoria de Walter Tichy (TICHY, 1982). O projeto do RCS foi inspirado no SCCS, incorporando algumas melhorias sobre ele, incluindo uma inteface com o usuário facilitada e melhoria no mecanismo de armazenamento das versões para recuperação mais fácil e rápida de informações (RCS, 2007). Além disso, o projeto foi lançado sob uma licença de código aberto – atualmente é distribuído sobre a versão 2 da *General Public License* (GPL). Existem versões para Unix, Windows e Mac OS X (RCS, 2007).

O RCS consiste de um conjunto de scripts que devem ser invocados via linha de comando. É um sistema bastante semelhante ao SCCS, que permite aos usuários controlarem as versões de seus arquivos. Este sistema não utiliza o conceito de repositórios – de forma semelhante ao SCCS. Geralmente sua utilização ocorria em ambientes de desenvolvimento que utilizavam *mainframes*, nos quais existiam os discos de dados compartilhados. Desta forma, todos os usuários podiam trabalhar simultaneamente sobre o código fonte de um determinado programa. O RCS realiza as atividades de rastreamento de autoria das modificações através da infra-estrutura provida pelo sistema operacional, utilizando os nomes de usuários autenticados na máquina como autores das modificações. Além disso, ele auxilia os desenvolvedores ao permitir o trabalho concorrente e oferecer dois mecanismos distintos para lidar com desenvolvimento concorrente: utilização de ***locks***, impedindo que mais de um usuário edite um arquivo ao mesmo tempo, ou realização da operação de *merge* e resolução de conflitos quando necessário (TICHY, 1985).

Quanto ao mecanismo de armazenamento, o RCS introduziu o conceito de "delta negativo" ou "delta reverso". Este mecanismo consiste em manter no arquivo de controle do RCS a última versão existente – conhecida por HEAD – de forma integral, e armazenar as diferenças para as versões anteriores no tronco principal. Os deltas positivos são então utilizados para se obter as versões das ramificações. Na Figura 2.8 é exibido um exemplo de árvore de revisões armazenada para determinado arquivo. Nela é possível observar que a última versão do tronco principal é armazenada integralmente; para se calcular uma versão anterior, por exemplo a 1.3, basta subtrair a diferença entre a 1.3 e a 1.4. Utilizando esta abordagem, as versões mais antigas exigem mais processamento, o que, geralmente, é uma situação conveniente, pois a maioria das operações concentra-se em versões recentes.

Apesar de ter evoluído em relação ao SCCS, permitindo, de certa forma, a edição concorrente de arquivos e funcionando com um desempenho superior ao do SCCS, o RCS ainda era um sistema que trabalhava exclusivamente sobre arquivos na área local de determinado usuário, e que não podia sequer tratar todos os arquivos de um diretório de forma conjunta. Para resolver os problemas do RCS, foi realizada uma implementação de seus *scripts* para que pudessem trabalhar de forma otimizada e sobre um repositório central, que foi chamada de CVS, de *Concurrent Versions System*, ou Sistema de Versões Concorrentes (CEDERQVIST et al., 2007). A primeira versão do CVS foi lançada em 1986, e era apenas um conjunto

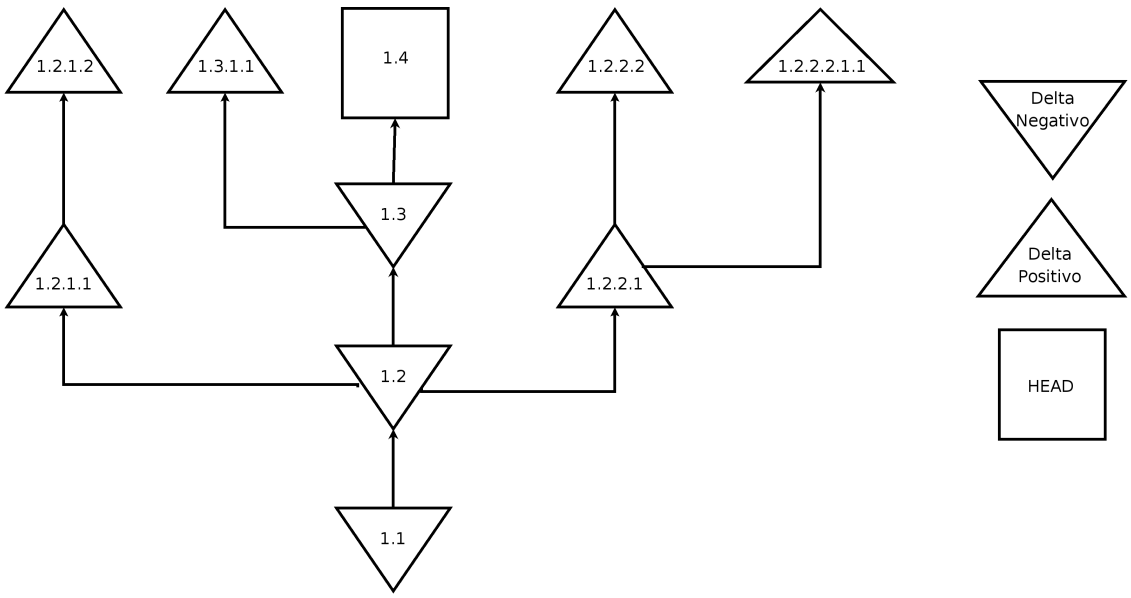

Figura 2.8. Árvore de revisões em cada ítem armazenado no repositório, utilizando deltas (adaptado de (TICHY, 1982))

de *scripts* que, por trás, utilizavam o RCS. Em 1989 foi lançada uma versão do CVS desenvolvida em linguagem C, que reimplementava os *scripts*.

As principais melhorias do CVS em relação ao RCS foram:

- **Repositório centralizado, local ou remoto:** com CVS, os arquivos sob controle de versões passaram a ser organizados em módulos, e estes passaram a ser armazenados em repositórios. Os repositórios de código podem ser locais ou remotos com utilização de CVS, permitindo, assim, a efetiva utilização do sistema em ambientes multi-usuário. A edição e visualização dos arquivos passou a ser feita nas máquinas dos usuários (em suas cópias de trabalho), e o acesso direto ao repositório – que no RCS refere-se aos arquivos que contém as informações de todas as versões – não é mais utilizado, mas, pelo contrário, é realizado através de comandos disponibilizados pelo sistema – *commit* e *checkout*. No RCS os comandos de *commit* e *checkout* são realizados localmente, mas, no CVS, envolvem o repositório central.

- **Usuários do CVS:** com RCS, caso se desejasse utilizar controle de versões em grupo, era preciso criar um diretório para um grupo num servidor, e todo o grupo precisava ter conta de usuário neste servidor, ou seja, o controle de acesso aos arquivos sob controle de versões era realizado através das permissões do sistema de arquivos no servidor. Com a utilização de CVS, passou a ser possível a autenticação de usuários através de métodos de autenticação próprios do CVS, num mecanismo conhecido por pserver. Com isso, os usuários com conta no CVS não precisavam ter acesso a um terminal do servidor. Além disso, a conta de usuário dá direito ao mesmo de realizar *checkout* dos

arquivos do repositório, copiando alguma versão dos mesmos para sua área local, e nunca editando diretamente o repositório.

- **Resolução de conflitos:** CVS permite que vários usuários tenham cópias locais de arquivos iguais, e, ao ser realizado commits, verifica se ocorreram conflitos. Caso ocorram conflitos, ele avisa ao usuário, gera um arquivo de conflito (arquivo resultado de um merge mal sucedido), o usuário ajusta manualmente o arquivo em sua área local e o envia de novo para o servidor, não perdendo o trabalho que foi feito em paralelo com alterações de outros usuários

O CVS foi utilizado por muitos anos, e tornou-se bastante popular, sendo o sistema de controle de versões mais utilizado na comunidade de software e também em muitas empresas. Muitas de suas falhas eram conhecidas e criticadas, mas poucas alternativas eficazes existiam. Dentre suas falhas, destaca-se o fato de utilizar um algoritmo de cálculo de diferenças que não é capaz de trabalhar com arquivos binários – o que poderia levar, inclusive, a perda de dados de repositórios no caso de não ser sinalizado ao CVS que o arquivo era binário – e a falta de suporte para operações sobre arquivos e diretórios – como, por exemplo, renomear um arquivo ou mesmo movê-lo para outro diretório que também estivesse no repositório.

Após muitos anos de utilização do CVS, foi proposto um projeto novo, que foi divulgado como uma versão melhorada do CVS. Isso ocorreu porque o CVS não implementa diversas funcionalidades que eram consideradas importantes, como, por exemplo, a operação de renomear um arquivo ou qualquer operação sobre diretórios. Estes problemas do CVS foram herdados do RCS e até hoje eles existem no sistema. Este sistema foi lançado em 2001 com o nome de SubVersion (COLLINS-SUSSMAN; FITZPATRICK; PILATO, 2007). Paralelamente ao SubVersion, foram lançados outros dois sistemas no mesmo ano: o Arch (MOFFITT, 2004) e o BitKeeper (KROAH-HARTMAN, 2002).

O sistema SubVersion apresentou como diferenciais em relação ao CVS o fato de poder trabalhar sob diretórios, além da estratégia de numeração de versões global por repositório, que o torna um sistema mais próximo das atividades de GCS. Além disso, SubVersion passou a utilizar uma versão melhorada do algoritmo de cálculo de diferenças, que podia trabalhar sobre arquivos em formato texto ou binários, sem causar inconsistências no repositório – problema que o CVS possui até hoje.

No entanto, os outros dois sistemas foram aqueles que criaram as principais inovações na área de controle de versões. Tanto o Arch quanto o BitKeeper eram sistemas com repositórios distribuídos, o primeiro desenvolvido como um projeto de código aberto e o segundo como um produto comercializado pela empresa BitMover. O Arch, em sua versão original, possuía um grande número de falhas, principalmente decorrentes da estratégia utilizada para nomear os arquivos sob controle de versões em cada repositório local de um usuário; além disso, sua performance era bastante inferior ao BitKeeper.

A partir de 2002, o BitKeeper passou a ser utilizado para o desenvolvimento do kernel do Linux. No entanto, em abril de 2005 a permissão concedida pela BitMover de utilização gratuita do sistema para o kernel do Linux e projetos de código aberto foi suspensa, e foram então lançados três sistemas para controle de versões, todos inspirados nas idéias do Arch e do BitKeeper: o Bazaar (BLACKWELL et al., 2006), que é uma reimplementação do Arch, o Git, projeto liderado por Linus Torvalds na época e que hoje é utilizado apara controle de versões do kernel do Linux (GARZIK, 2005), e, por fim, o Mercurial, bastante influenciado pelo BitKeeper (MERCURIAL, 2007).

Atualmente, estes são os sistemas de controle de versões mais novos que existem. Suas funcionalidades são semelhantes às dos sistemas BitKeeper e Arch, mas todos eles são projetos de código aberto, diferentemente do BitKeeper.

Na próxima seção, serão apresentadas algumas abordagens de integração das ferramentas de controle de versões com outras ferramentas, principalmente ferramentas *web*.

2.3. Integração de Controle de Versões com outras ferramentas

Nos últimos anos, tem-se visto a utilização de controle de versões integrado a diferentes tipos de ferramentas: além da integração que é feita para a realização de atividades inerentes ao processo de GCS – como a integração com ferramentas de *Customer Relationship Management* (CRM) e rastreamento de **bugs** –, também observou-se a criação de propostas cujo objetivo é agregar valor a sistemas mais tradicionais, geralmente sistemas web, a partir da integração de tais sistemas com ferramentas de controle de versões.

No ano de 2000, Soares propôs o sistema VersionWeb (SOARES; FORTES; MOREIRA, 2000; MOREIRA; SOARES; FORTES, 2002), cujo objetivo era permitir que os visitantes de sites da web pudessem visualizar versões anteriores das páginas disponibilizadas. Este sistema foi evoluído pelo autor do presente trabalho para que pudesse funcionar como um cliente web completo para o CVS e, posteriormente, foi evoluído para oferecer suporte a repositórios CVS e SubVersion, agregando novas funcionalidades a estes sistemas – principalmente de controle de acesso por arquivos e diretórios, e com a utilização de grupos – que não estavam disponíveis nestes sistemas, a não ser que o controle fosse feito diretamente pelo sistema de arquivos. O sistema original foi desenvolvido em linguagem C, e sua evolução foi feita em linguagem Python (JUNQUEIRA; FORTES, 2004). Outros sistemas para visualização de repositórios também foram criados, e merece destaque o ViewVC[3], que atualmente também suporta repositórios CVS e SubVersion, mas apenas para navegação, não sendo possível realizar operações de escrita nos repositórios.

[3] Site oficial do projeto: `http://www.viewvc.org`

Além destas abordagens, observa-se também a utilização de controle de versões integrada a outras aplicações *web*, como o sistema de edição colaborativa *Mediawiki*[4] (LERNER, 2006), que é o sistema utilizado pela Wikipedia, maior enciclopédia colaborativa (RIEHLE, 2006). O trabalho de Silva (SILVA, 2005) também aponta as vantagens da utilização de controle de versões em sistemas web de edição colaborativa.

Em 2006 foi apresentada uma implementação de integração de controle de versões com a CoTeia, utilizando o paradigma de programação orientada a aspectos (LEMOS et al., 2006). A CoTeia é um sistema *web* de edição colaborativa desenvolvido e mantido pelo ICMC. Este trabalho teve como um de seus principais autores o autor desta dissertação, e foi um resultado parcial do trabalho aqui apresentado.

Atualmente, técnicas de controle de versões estão também implementadas nas ferramentas de edição de documentos disponibilizadas pelo Google – a Google Docs and Spreadsheets (JAZAYERI, 2007). Estes esforços mostram a importância da atividade de controle de versões, que está sendo adaptada a novas formas de utilização dos computadores – mais especificamente a internet.

2.4. Gerenciamento de Configuração de Software e Controle de Versões

Gerenciamento de Configuração de Software é a área responsável por controlar a evolução de um software, realizando acompanhamento de todas as alterações que ocorrem com o mesmo durante todo o seu ciclo de vida (KEYES, 2004). Esta área é bem mais abrangente que o controle de versões em si, e preocupa-se não apenas com os arquivos de código fonte, mas, e principalmente, com questões sobre quais as distribuições do software que estão implantadas em cada cliente, rastreamento de bugs e problemas, processo de gestão de mudanças, entre outros. Na Figura 2.9 é apresentado um diagrama com as principais atividades que estão envolvidas no processo de GCS. Estas atividades são (HASS, 2003):

- Identificação: envolve a atividade de identificar os itens que deverão integrar a configuração de um software.
- Controle de Mudanças: refere-se ao acompanhamento dos itens de configuração durante todo seu ciclo de vida. envolve a definição de procedimentos para solicitações de mudanças e sua respectiva aprovação por comitês gestores de mudança, análise de impacto das mudanças, entre outros

[4] Site oficial do projeto: `http://www.mediawiki.org`

- Verificação do Estado: relacionado à descrição do estado da configuração. Envolve a verificação de quais itens de configuração estão num ambiente de produção, por exemplo

- Auditoria: tem o objetivo de validar se tudo o que foi planejado está realmente sendo cumprindo. Envolve atividades como escolha de itens que devem ser auditados e forma de executar a auditoria

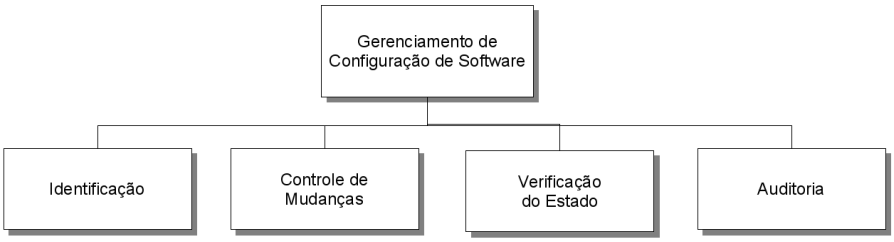

Figura 2.9. Diagrama das atividades de GCS

Como é possível observar, a Gerência de Configuração de Software é bastante abrangente e envolve diversas atividades, geralmente envolvendo diversas equipes dentro das empresas. As atividades de controle de versões, quando bem planejadas e bem executadas em ambiente de desenvolvimento, podem contribuir com as tarefas de gerência de configuração a partir do momento que permitem maior controle sobre o que está sendo produzido. Com isso, também auxiliam a facilitar o rápido rastreamento de erros e sua respectiva correção, permitindo a realização de relatórios precisos sobre o erro existente e a solução adotada, o que pode agilizar o processo de Controle de Mudanças em GCS.

2.5. Considerações Finais

Neste capítulo foram apresentados os principais conceitos relacionados a controle de versões, além de uma revisão das principais ferramentas existentes e suas funcionalidades. Também foram apresentadas algumas considerações sobre GCS e seu relacionamento com a área de controle de versões.

Foi possível observar que a utilização das ferramentas de controle de versões é muito importante para permitir o acompanhamento da evolução de sistemas de computador, incluindo os arquivos de **código fonte** e toda a documentação correspondente. Entretanto, observa-se que a implementação dos sistemas leva em consideração a lógica inerente aos programas de computador, mas os sistemas de controle de versões tradicionais utilizam como unidades de controle de versões as estruturas lógicas dos sistemas de arquivos, que são os arquivos e diretórios.

A disciplina de GCS é responsável por acompanhar a evolução e desenvolvimento de um software durante todo o seu ciclo de vida, acompanhando principalmente os problemas e correções identificados no software. A partir do momento que são realizadas atividades de controle de versões com maior detalhamento, é possível que sejam agilizados alguns processos da área de GCS e, também, que sejam minimizados os problemas ocorridos em ambiente de desenvolvimento, pois podem ser obtidos mais detalhes – e de forma mais rápida – sobre os produtos de software sendo desenvolvidos.

No próximo capítulo, serão apresentados os principais conceitos relacionados a controle de versões refinado.

Capítulo

3

Controle de Versões refinado

Controle de versões é uma atividade de suporte importante para Gerência de Configuração de Software. A maioria dos sistemas de controle de versões mantém um registro das mudanças ("deltas") entre as diferentes versões dos arquivos sob controle de versões, o que provê a base para rastrear a evolução de um sistema durante seu ciclo de vida. Entretanto, os sistemas de controle de versões geralmente realizam o controle de versões sobre os arquivos num nível de granulosidade muitas vezes inadequado, geralmente em documentos inteiros ou módulos inteiros (LINDSAY; LIU; OWENTRAYNOR, 1997). Mesmo os sistemas de controle de versões mais populares atualmente, como SubVersion e CVS – de repositório centralizado – e o Bazaar e o Mercurial – de repositório distribuído – ainda seguem este paradigma: a granulosidade de versões em níveis de arquivos e módulos.

Pesquisadores têm investido em novas estratégias de controle de versões de forma a permitir um controle de versões mais **refinado**, utilizando informações semânticas ou da estrutura dos documentos para fazer as atividades de controle das versões. Com esta abordagem, é possível disponibilizar informações mais detalhadas aos envolvidos no processo de produção de software, possibilitando inclusive a adoção de práticas mais eficazes de GCS (LIN; REISS, 1996).

Esta abordagem tem por objetivo fornecer melhor controle sobre os documentos ou arquivos de código-fonte que estão no repositório de controle de versões, permitindo um mapeamento mais adequado da estrutura dos arquivos de computador no sistema de controle de versões. O problema existente foi bem expressado no trabalho de Nguyen, Munson e Thao (NGUYEN et al., 2004):

> "A estrutura interna de artefatos de software, especialmente códigos-fonte de programas, são muito importantes para os engenheiros de software desenvolverem e manterem softwares de alta qualidade. Entretanto, os sistemas de controle de versões e gerenciamento de configurações existentes geralmente tratam um sistema de software como um conjunto de arquivos em diretórios sobre um sistema de arquivos. Eles geralmente desconsideram a estrutura lógica de documentação e código-fonte de programas tratando-os como um conjunto de linhas para controle de versões. Além disso, é criado um empecilho para os desenvolvedores, pois o domínio de implementação (nível lógico) e o domínio de controle de versões (nível de arquivo) requerem modelos mentais diferentes."

Este problema também é ilustrado na Figura 3.1.

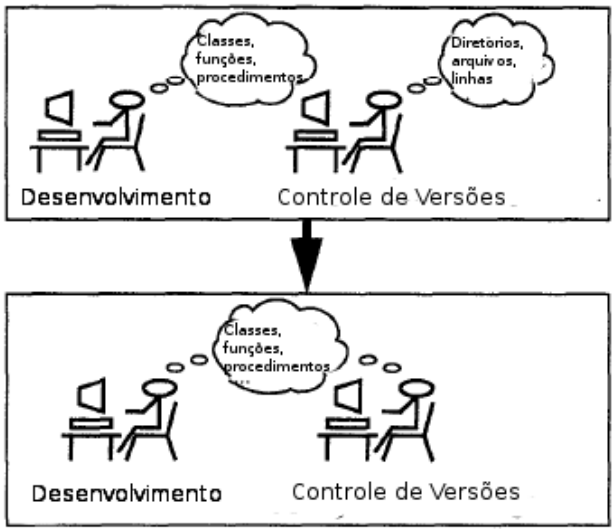

Figura 3.1. Controle de versões com granulosidade fina (adaptada de (LIN; REISS, 1996))

Abordagens para aplicar controle de versões refinado têm sido alvo de investigações desde o final da década de 1980. Em 1988, foi proposto o sistema Orwell, cujo objetivo era oferecer suporte para o controle de versões de sistemas desenvolvidos em ***Smalltalk***, permitindo o controle de versões de classes e métodos (THOMAS; JOHNSON, 1988). Em 1993, já havia sido identificado que os sistemas tradicionais "trabalham em arquivos completos que geralmente são unidades muito maiores do que é afetado por uma mudança única" e foi proposto um sistema para resolver o problema (MAGNUSSON; ASKLUND; MINöR, 1993), que posteriormente foi substituído pelo COOP/Orm (MAGNUSSON; ASKLUND, 1996).

Além do Orwell e do COOP/Orm, outros trabalhos foram propostos com o objetivo de permitir o controle de versões com granulosidade fina. Neste capítulo é apresentada uma análise comparativa dos trabalhos. Os sistemas que foram estudados foram o Molhado(NGUYEN; MUNSON; THAO, 2004), POEM(LIN; REISS, 1996), CoEd(BENDIX et

al., 1998), Coop/ORM(ASKLUND, 2002) e Stellation (CHU-CARROLL; WRIGHT; SHIELDS, 2002).

As características dos sistemas de controle de versões incluem o tipo de repositório, controle de acesso, existência ou não de **hooks**, protocolos utilizados, atomicidade das operações, possibilidade de realizar operações sobre diretórios, entre outras. Entretanto, três características são específicas dos sistemas que realizam controle de versões refinado: uma estrutura de dados que armazena as unidades que estão sob controle de versões, as regras que norteam o processo de criação de novas versões e a forma como é realizado o mapeamento da estrutura do arquivo para a estrutura de dados que representa os itens sob controle de versões. Em relação à forma como o mapeamento é realizado da estrutura do arquivo para a estrutura de dados que armazena a estrutura do arquivo, é importante analisar os mecanismos existentes para definir a gramática do arquivo e, também qual a flexibilidade oferecida por este mecanismo.

A partir do momento que as ferramentas de controle de versões refinado analisam a estrutura dos arquivos que estarão sob controle de versões, pode ser implementado o rastreamento de dependência entre arquivos de um repositório, desde que essa dependência seja declarada, de alguma forma, na gramática do arquivo. Por exemplo, no caso da linguagem de programação Java a palavra reservada "*import*" é utilizada para dizer que a classe sendo implementada depende de outra classe, que pode também estar no repositório. Na análise realizada foi verificada a existência de rastreamento de dependências deste tipo pelos sistemas de controle de versões refinado.

Outra característica que foi analisada foi a flexibilidade de utilização da ferramenta. Mesmo no século XXI muitas empresas não realizam as atividades de controle de versões no desenvolvimento de softwares; embora esta situação leve a uma perceptível queda na qualidade do processo de desenvolvimento de software – e conseqüente diminuição na qualidade dos produtos desenvolvidos (KEYES, 2004), a adoção de um processo de GCS, ou mesmo de ferramentas de controle de versões, não costuma ser fácil, principalmente devido ao fato de haver uma mudança na cultura de desenvolvimento de software da empresa (MICALLEF; CLEMM, 1996). Portanto, quanto maior a flexibilidade permitida pelas ferramentas utilizadas, maiores as chances de se ter a redução de impacto num ambiente de desenvolvimento de software, pois quanto maior a flexibilidade do sistema de controle de versões refinado, mais ele pode ser integrado às ferramentas já utilizadas no ambiente de desenvolvimento de software. Na análise realizada considerou-se que a flexibilidade de extensão da ferramenta deveria ser oferecida a partir de uma *Application Program Interface* (API) ou, em último caso, de ferramentas de linha de comando que pudessem ser utilizadas para a construção de APIs de integração com outras ferramentas.

A seguir, são apresentados os resultados que foram obtidos a partir da análise de ferramentas de controle de versões refinado, tendo destaque na análise: a estrutura de dados para armazenamento dos itens sob controle de versões, as regras de controle de versões, a maneira como é realizado o mapeamento, a forma como é feito o rastreamento de dependências entre os arquivos de um repositório e a flexibilidade na adoção da ferramenta, i.e., possibilidade de estender e adaptar outras ferramentas para que possam ser integradas ao sistema de controle de versões refinado.

Será apresentado o resultado da análise de cada uma das ferramentas individualmente e, no final da capítulo, será apresentada uma síntese da análise e considerações finais.

3.1. Molhado

O sistema Molhado (NGUYEN; MUNSON; BOYLAND, 2004) é apresentado como um sistema para GCS e controle de versões de **hiperdocumentos**. O foco do sistema é de fornecer informações detalhadas de hiperdocumentos.

Este sistema foi projetado utilizando um outro projeto, chamado Fluid, que implementa mecanismos genéricos para controle de versões de informações em nós genéricos, utilizando um padrão que foi batizado de *node-slot pattern*[1], além de fornecer meios para realizar o armazenamento destes nós e seu controle de versões. Portanto, a camada de controle de versões e armazenamento das informações no Molhado é feita utilizando o Fluid. Esta camada implementa mecanismos de armazenamento de dados primitivos, além de seu respectivo controle de versões. Como é descrito pelos autores do trabalho (NGUYEN; MUNSON; BOYLAND, 2004), "(Fluid) oferece uma combinação de um modelo de versões e um modelo de dados primitivos para permitir o controle de versões de muitos tipos de dados".

A estrutura de dados utilizada para representar, no repositório de controle de versões, os diversos itens que estão sendo controlados – que são "partes" de arquivos, com algum significado lógico – é baseada no padrão *node-slot*. De forma geral, o documento é organizado em forma de árvore, e suas partes são colocadas em nós genéricos, que podem conter referência para outros nós também genéricos.

Na Figura 3.2 é mostrada uma tela do sistema Molhado. Nesta tela é possível visualizar a funcionalidade de estrutura de projeto.

Molhado auxilia nas tarefas de GCS pelo fato do Fluid realizar o controle de projetos como um todo, realizando atividades semelhantes a "Gerenciamento de Produtos", segundo

[1] Foi mantido o termo original que denomina o padrão. A referência bibliográfica utilizada no artigo que cita o padrão era do tipo "em preparação", e não foram encontradas maiores informações sobre este padrão. Na página citada, do projeto Fluid – `http://fluid.cs.cmu.edu` –, também não foram encontradas referência ao padrão *node-slot*

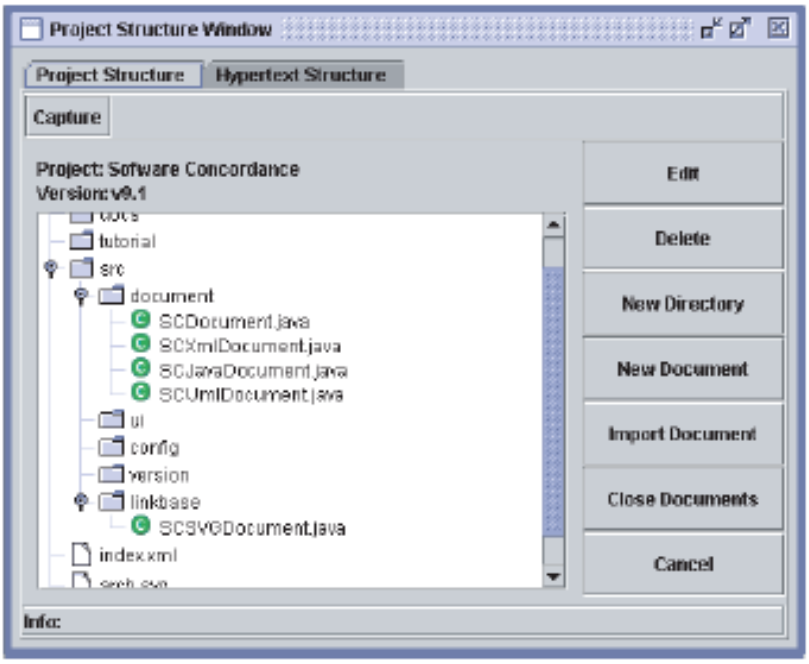

Figura 3.2. Tela típica do sistema Molhado

os autores do trabalho. No entanto, o estudo da ferramenta não deixa claro quais as funcionalidades disponíveis para que sejam feitas atividades de Gerenciamento de Produto.

Quanto às regras de controle de versões adotadas, observa-se a utilização do controle das versões dos nós e slots, e são realizadas operações de "snapshot" sobre toda a árvore de determinado projeto. Portanto, quando é realizada uma operação semelhante a *commit* para salvar determinada alteração, todo o projeto é armazenado na nova versão criada. Esta abordagem tem sido utilizada há bastante tempo nos sistemas de controle de versões.

O projeto Molhado não deixa claro quais são as regras de mapeamento utilizadas para que o conteúdo de um documento seja mapeado para a estrutura de dados apresentada – *node-slot pattern*. Este não é o foco do sistema, e a cada nova gramática de documentos que deve ser mapeada uma nova versão de partes do sistema deve ser reimplementadas.

O Molhado não disponibiliza APIs para que seja integrado a outras ferramentas.

3.2. POEM

O sistema POEM (LIN; REISS, 1996) – *Programmable Object-centered EnvironMent* – é constituído por um ambiente de desenvolvimento capaz de controlar as versões de itens menores do que arquivos, como classes e métodos. Ele é um ambiente de programação voltado para o desenvolvimento de programas na linguagem C++, e o mapeamento entre a gramática do documento e seu modelo de versões é programado internamente no sistema, não sendo possível redefenir estas regras. Na Figura 3.3 é apresentada uma tela padrão do sistema POEM.

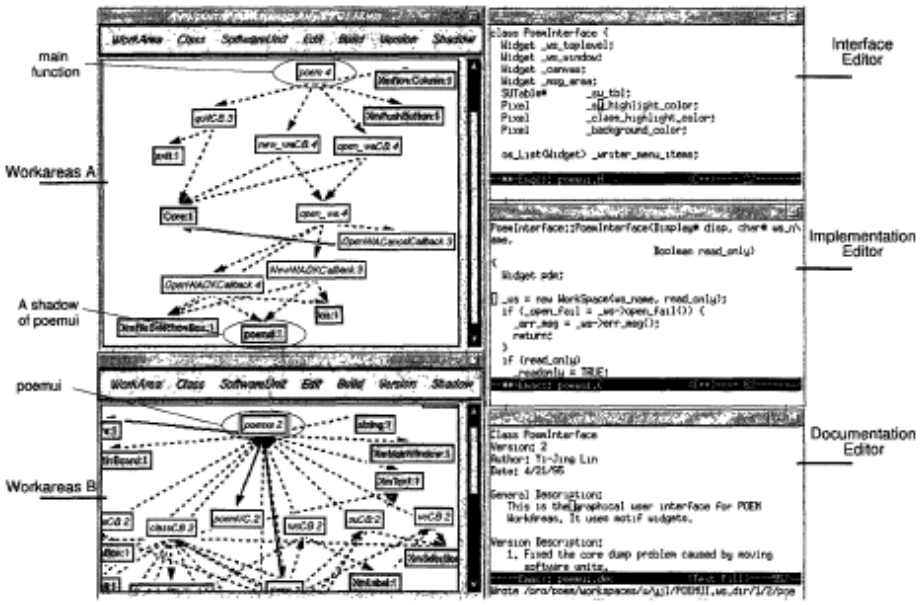

Figura 3.3. Tela típica do sistema Poem

O código-fonte dos arquivos em C++ é dividido em métodos. A estratégia utilizada para realizar as operações de controle de versões utiliza duas operações:

- *revise*, aplicada sobre um subsistema, que é uma parte do sistema. Um "subsistema" do POEM é formado por um conjunto de "partes", que são, geralmente, métodos relacionados.

- *snapshot*, aplicada sobre um subsistema - realiza uma operação semelhante à de *commit*, e é realizada sobre todas as partes atingidas pelo subsistema. Com isso, esta versão torna-se, então, imutável.

Através desta estratégia, geralmente cria-se muito mais versões dos itens do que realmente seria necessário, pois não aplica-se conceitos de propagação de mudanças e, além disso, são impostos limites à essa propagação, pois se um método que é parte de um documento faz parte de um subsistema ele pode ser alterado, mas a versão do documento que o contém não o será. Além disso, o POEM não considera os subsistemas definidos como unidades íntegras de software, que poderiam posteriormente ser utilizadas para comparações de mudanças em toda a unidade.

Em relação ao rastreamento de dependências, POEM permite que se defina a dependência entre os arquivos de forma manual, utilizando a interface gráfica da ferramenta. Além de permitir esta definição manual, POEM também oferece suporte às operações de **construção das aplicações**, e, quando seu mecanismo de construção é utilizado, as dependências definidas em sua interface têm prioridade sobre aquelas que são definidas num **Makefile** ou mesmo que podem ser descobertas pelo compilador pelas diretivas *include* do C++.

Quanto à possibilidade de estender, o POEM foi projetado como um ambiente de programação completo, e não disponibiliza qualquer API para que possa ser integrado em ferramentas já utilizadas em ambientes de desenvolvimento de software.

3.3. CoEd

O CoEd é apresentado pelos seus autores como um sistema de controle de versões refinado para documentos hierárquicos (BENDIX et al., 1998). No entanto, analisando as características da ferrmenta, observou-se que ela é uma ferramente focada um tipo específico de documentos hierárquicos, que são os textos em Latex.

O sistema utiliza uma arquitetura de repositório centralizado para organizar seus arquivos, e permite que múltiplos clientes acessem um mesmo repositório ao mesmo tempo. Os principais objetivos do projeto CoEd eram permitir que os autores de documentos textuais tivessem acesso às seguintes facilidades:

- visão geral do documento, incluindo sua estrutura e conteúdo
- controle de versões, incluindo versão atual e histórico de versões
- comunicação, permitindo, de certa forma, ciência sobre o que as outras pessoas estavam fazendo no texto

Na Figura 3.4 é apresentada uma tela do sistema, na qual é exibido um documento sendo editado – em Latex, a estrutura do documento, à esquerda, e a árvore de revisões, no canto superior esquerdo.

CoEd é um sistema implementado para o trabalho colaborativo, que separa em partes diferentes as várias partes de um texto em Latex que estejam escritas num arquivo único, ele não permite qualquer flexibilidade em relação à gramática do documento utilizado.

Em relação ao controle de versões, ele é realizado da seguinte forma: quando o dado de qualquer nó é alterado e salvo, o CoEd cria uma nova versão do nó que contém aquele dado (por exemplo, da Seção ou Capítulo que o contém) e realiza a propagação da mudança até a raiz do documento.

O sistema CoEd funciona de forma completamente dependente de sua interface gráfica, não tendo sido projetado para ser integrado em soluções que os usuários já podem estar acostumados a utilizar para escrever seus textos em Latex. Outro ponto negativo do projeto é que ele não suporta a linguagem Latex como um todo. Por exemplo, não é possível utilizar os comandos \include do Latex, i.e., os textos devem ser escritos utilizando sempre apenas um arquivo único.

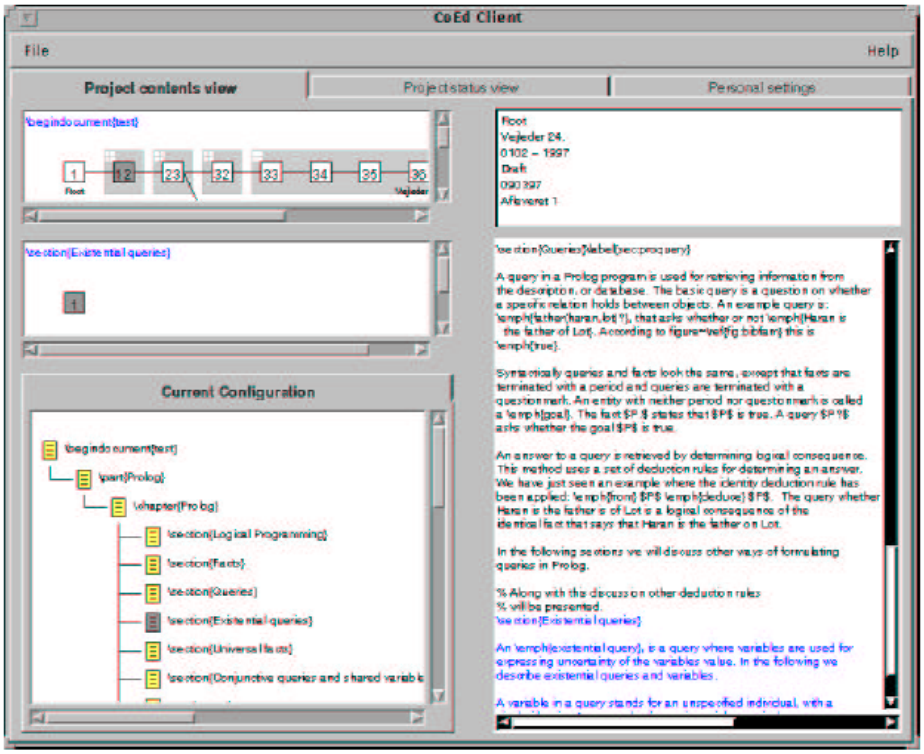

Figura 3.4. Tela da ferramenta CoEd, para edição de documentos Latex

3.4. COOP/Orm

COOP/Orm (MAGNUSSON; ASKLUND, 1996) é um sistema que foi inspirado no projeto Mjolner (KNUDSEN et al., 1994), um sistema para desenvolvimento de software orientado a objetos. Em 1993, Magnusson et al. (MAGNUSSON; ASKLUND; MINöR, 1993), cujos trabalhos são focados em desenvolvimento colaborativo de software e outras atividades relacionadas à cooperação, identificaram que "o controle de versões refinado é uma técnica crucial para o suporte da evolução de software por permitir o trabalho de times geograficamente distribuídos de engenheiros de software" (MAGNUSSON; ASKLUND; MINöR, 1993). Apesar de ter sido identificada esta melhoria em relação ao controle de versões refinado para projetos em que os desenvolvedores estão geograficamente distribuídos, Reis observou, dez anos depois, que os projetos de software livre – maiores exemplos dos projetos com desenvolvedores distribuídos – não utilizavam técnicas de controle de versões refinado, e, muitas vezes, nem mesmo realizavam qualquer atividade de controle de versões (REIS, 2003). Após terem realizado estas investigações iniciais, os pesquisadores envolvidos no grupo de Magnusson criaram o sistema COOP/Orm, cujo objetivo é auxiliar o desenvolvimento colaborativo de software, através da implementação de técnicas que permitem o controle de versões refinado, além de diversas outras características voltadas para desenvolvimento colaborativo – como um modelo de ciência do projeto, segundo o qual cada desenvolvedor envolvido pode estar sempre ciente do que está ocorrendo no projeto como um todo.

A descrição de uma versão atualizada do COOP/Orm foi utilizada para análise (ASKLUND, 2002). Apesar do foco do sistema ser a colaboração entre desenvolvedores, o conceito de controle de versões refinado é bastante explorado e é a partir dele que são oferecidas outras ferramentas que têm o objetivo de melhorar as experiências de colaboração. Para fornecer o controle de versões refinado, foi claramente definido um modelo para o controle de versões refinado, chamado *Unified Extensional Versioning Model* (UEVM). Apesar deste modelo ter sido apresentado anteriormente (ASKLUND et al., 1999), Asklund o apresentou com correções e atualizações, principalmente no que diz respeito à propagação de versões e preocupações adicionais com as características que permitem ao modelo de versões não sofrer a explosão combinatória (ASKLUND, 2002).

O UEVM, em relação ao controle de versões refinado, é constituído por um modelo de documentos e um modelo de versões, que foram bem definidos. Para o modelo de documentos assume-se que deve ser controlada a versão de forma refinada de documentos estruturados. Por documento estruturado entende-se qualquer arquivo de computador – mesmo arquivos em formatos binários, como imagens e vídeos. Entretanto, dependendo do formato do arquivo é possível mapear de forma diferente a gramática do arquivo no modelo de documentos. O modelo de documentos em si é composto por quatro nós genéricos, a saber: nós de documento, nós de composição, nós de ligação e nós de conteúdo.

Os nós de documento, por definição, são sempre mapeados em arquivos. Portanto, cada arquivo sempre corresponde a um documento específico. O conteúdo do documento, então, pode ser um nó de composição, um nó de conteúdo ou um nó de ligação.

Os nós de composição são os que permitem a organização da estrutura do arquivo. Estes nós podem ser mapeados em outros nós de composição, em nós de conteúdo ou em nós de ligação. A diferença do nó de composição em relação ao nó de documento é que o nó de composição é recorrente, ou seja, pode ser mapeado em outro nó de composição, permitindo a criação de estruturas tão profundas quanto forem necessárias.

Os nós de conteúdo são os que contêm o conteúdo final. São os nós-folha do modelo.

Os nós de ligação são aqueles que permitem realizar o mapeamento das ligações entre documentos e respectivo rastreamento de dependências. Os nós de ligação realizam o mapeamento para um conjunto de nome e versão do alvo. Desta forma, é possível verificar a dependência exata entre os arquivos, incluindo a versão em que se encontrava um relacionamento entre dois arquivos.

Para exemplificar as idéias aqui apresentadas, a Figura 3.5 exibe alguns exemplos de documentos mapeados no modelo de documentos do UEVM.

Além do mapeamento para o modelo de documentos, o UEVM define um modelo de versões, que consiste num conjunto de regras para controlar as versões dos nós do modelo de documentos. Este modelo é necessário porque tanto a estrutura quanto o conteúdo de um

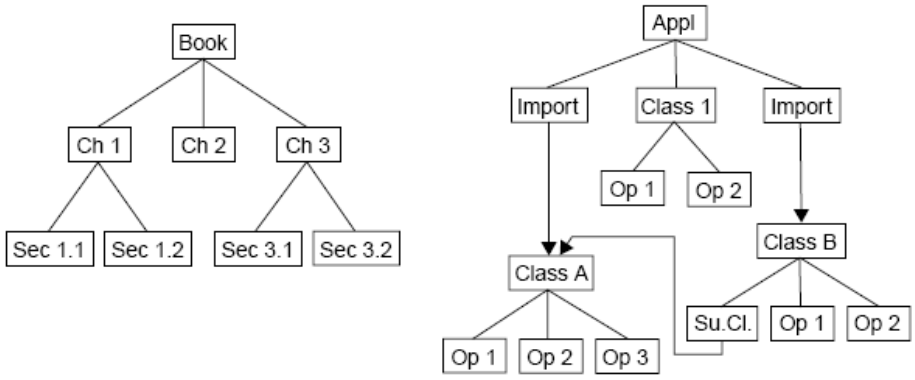

Figura 3.5. Exemplo de livro e programa de computador mapeados para o modelo de documentos do UEVM

documento irá evoluir no tempo, e todos os tipos de nó (composição, conteúdo e ligação) precisam ter suas versões controladas. Portanto, a criação de uma nova versão de cada nó acontecerá se acordo com as seguintes condições:

- Nós de documento terão uma nova versão criada sempre que for alterado qualquer dado do documento, seja de sua estrutura ou do conteúdo
- Nós de composição e conteúdo terão uma nova versão criada quando seus dados locais são alterados
- Nós de ligação terão uma nova versão criada quando tanto o "nome" ou a "versão" do documento alvo for alterada
- Nós de composição, além da regra já listada, também terão uma nova versão criada quando qualquer um de seus filhos for adicionado, removido ou alterado

É possível observar que a última regra listada é a regra que permite a propagação de mudanças. O objetivo deste modelo de versões é permitir que só sejam criadas novas versões de nós quando for estritamente necessário – agregando mais inteligência ao mecanismo de controle de versões refinadas. Desta forma pode-se rastrear de forma otimizada qual o impacto que determinada alteração tem no projeto como um todo.

O modelo de versões do UEVM foi implementado no COOP/Orm com sessões: os usuários, ao iniciarem uma sessão, criam um ramo próprio para que possam trabalhar – portanto, o sistema COOP/Orm possui uma arquitetura com repositórios mistos - é obrigatória a existência de um repositório central, mas cada desenvolver possui também um repositório local; além disso, durante as sessões dos usuários, eles podem realizar operações de sincronização dos repositórios entre si, sem intervenção do repositório central. A partir da abertura de uma sessão, um usuário pode editar livremente sua ramificação do repositório, que fica armazenada localmente em seu computador; durante o trabalho de edição, pode salvar diversas

versões intermediárias em seu repositório local e, no momento em que precisar sincronizar suas alterações com o repositório central, apenas a última versão salva localmente será sincronizada – portanto, não ocorre uma sincronização entre árvores, mas, sim, uma operação de *commit* no repositório central. Esta foi a abordagem utilizada pelo COOP/Orm para evitar a explosão combinatória das versões criadas. Na Figura 3.6 são exemplificados os conceitos de propagação de versões e sessões implementados pelo COOP/Orm. Nesta figura, observa-se que a alteração realizada no passo *b)* criou novas versões dos nós acima dele (nós de composição), conforme regra do modelo de versões. No passo *c)* foi realizada a alteração de outro nó – mas a nova versão do nó 1, que é comum aos dois nós alterados, foi mantida – ou seja, manteve-se o marcador da versão, que agora engloba as duas alterações realizadas. Quando esta mudança for submetida ao repositório central, mesmo se tiver sido realizada e salva em dois passos localmente pelo usuário, será entendida desta forma, que é otimizada, diminuindo o número de versões intermediárias que são desnecessárias para o modelo.

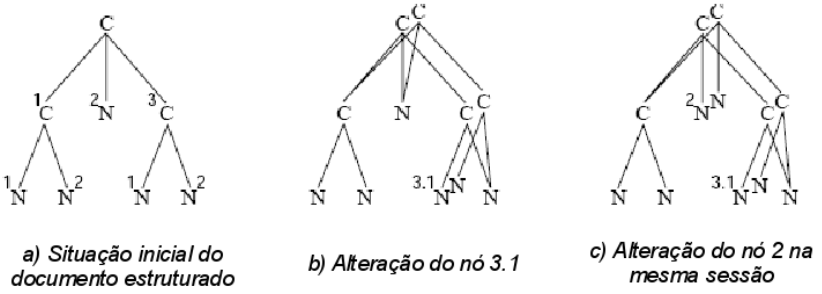

Figura 3.6. Exemplo de propagação de mudanças no modelo de versões do UEVM

Em relação ao mapeamento da gramática de documento para o modelo de versões, a ferramenta COOP/Orm suporta este mapeamento da seguinte forma: deve-se utilizar a notação *Extended Backus-Naur Form* (EBNF) para representar a gramática da linguagem utilizada pelo documento, conforme o nível de granulosidade desejado e, então, deve-se utilizar outro arquivo na notação EBNF para realizar o mapeamento entre os nós representados no arquivo da gramática da linguagem e os nós do modelo de documentos do UEVM. Este processo não foi automatizado por nenhuma ferramenta, de forma que os usuários do COOP/Orm precisam conhecer o modelo de documentos.

Quanto às facilidades de extensão e adaptação do sistema, o COOP/Orm não disponibiliza nenhuma forma para estender a ferramenta. Não são fornecidas APIs nem é possível executar as operações via linha de comando, sendo que a utilização do modelo implementado pelo COOP/Orm é dependente da ferramenta, não sendo possível realizar sua integração com outras IDEs ou ferramentas externas – por exemplo, ferramenta de auditoria de versões externa. Devido ao fato do sistema ter como foco principal as atividades de colaboração que não foram abordadas nesta análise, ela é bastante complexa e possui diversas outras funcio-

nalidades dependentes da atividade de controle de versões refinado, e torna-se compreensível a não implementação de APIs e outras formas de estender o sistema.

3.5. Stellation (Coven)

Stellation (CHU-CARROLL; WRIGHT; SHIELDS, 2002), originalmente chamado Coven (CHU-CARROLL; SPRENKLE, 2000), é um sistema que começou a ser desenvolvido em laboratórios de pesquisa da IBM com o objetivo de oferecer controle de versões refinado para arquivos de computador, principalmente código-fonte de programas.

O sistema analisa a estrutura lógica dos arquivos de computador e armazena itens lógicos – controle de versões mais refinado que arquivos. O menor nível documentado é o de métodos e atributos das classes.

Stellation foi disponibilizado como um plugin para a Integrated Development Environment (IDE) **Eclipse** em 2002, mas atualmente não encontra-se mais disponível. As informações sobre o sistema não têm sido atualizadas desde 2004.

Um dos conceitos utilizados pelo sistema é o de arquivos de código-fonte virtuais, que é o mapeamento feito dos arquivos no sistema. Este conceito assemelha-se à organização dos arquivos no sistema de arquivos, mas diferencia-se pelo fato de poder mapear unidades lógicas dentro dos arquivos - por exemplo, os métodos e atributos das classes.

No sistema Stellation não existe um método genérico que permita realizar o mapeamento da estrutura lógica do documento para o modelo de documentos do sistema. Os mapeamentos são bastante específicos, e têm o principal objetivo de criar agregações de itens de software – por exemplo, métodos – que estão, de alguma forma, relacionados. Todos os exemplos que foram encontrados do sistema estavam implementados na linguagem de programação Java, embora em sua documentação esteja especificado o suporte a programas desenvolvidos em C++. No entanto, o sistema não está mais disponível para download[2], o que dificultou a análise das questões de flexibilidade tanto do modelo de documentos quanto da possibilidade de extensão do sistema.

Em relação à extensão, não existe uma API documentada do sistema. Entretanto, enquanto esteve disponível era possível utilizar uma versão desenvolvida como ***plugin*** da IDE Eclipse ou uma versão que podia ser utilizada diretamente na linha de comandos. Ao oferecer a opção de utilização via linha de comandos, mesmo não existindo uma API formal, pode-se implementar adaptadores que invoquem os comandos disponibilizados. Embora seja uma alternativa mais flexível do que aquelas que não disponibilizam a opção de executar as operações via linha de comando, uma análise mais detalhada da flexibilidade da utilização desses comandos não pôde ser executada, pois as informações disponibilizadas não foram

[2] Tentativas de acesso realizadas durante todo o ano de 2007

suficientes. Para que uma solução via linha de comando seja flexível é necessário que ofereça regras padronizadas e bem documentadas de entrada e saída dos comandos, além de permitir a realização de todas as operações que seriam realizadas através de uma interface gráfica disponível – e, no caso do sistema Stellation, nenhuma das duas informações está disponível para avaliação.

3.6. Considerações Finais

O termo "controle de versões refinado e flexível para artefatos de software" refere-se ao controle de versões dos arquivos de computador que considera, além da estrutura de arquivos no sistema de arquivos, a estrutura interna dos arquivos sob controle de versões; por "flexível", entende-se as características dos sistemas de permitir alterar o nível de detalhamento utilizado na análise da estrutura. Portanto, a flexibilidade do sistema está relacionada com a possibilidade de se poder controlar as versões com um detalhamento diferente para arquivos que tenham a mesma estrutura.

A partir da análise das ferramentas para controle de versões refinado, levantou-se as principais características de cada uma no que diz respeito ao controle de versões refinado e flexível para os arquivos de computador. Na Tabela 3.1 é apresentado um resumo das características das ferramentas.

Tabela 3.1. Resumo das características das ferramentas analisadas em relação ao controle de versões de documentos estruturados

Ferramenta	Mapeamento	Rastreamento de Dependência*	IDE
Molhado	Implementação específica	Manual	Proprietária, incluída no sistema
POEM	Embutido para C++	Manual	Proprietária, incluída no sistema.
CoEd	Latex, não flexível.	Não disponível	Proprietária, incluída no sistema
COOP/Orm	Gramática EBNF da linguagem e gramática EBNF entre linguagem e modelo	Automática	Proprietária, incluída no sistema.
Stellation	Documentos XML realizam mapeamento específico	Manual	Plugin para Eclipse

*Verificação de dependência entre os arquivos de código-fonte.

Apesar de todas as propostas analisadas estarem relacionadas com atividades de controle de versões refinado para documentos de computador, a maior parte delas foi projetada para

atender um tipo específico de arquivo – por exemplo, Latex para o CoEd e C++ para o POEM. A única solução apresentada que é realmente flexível em relação à definição da estrutura dos documentos que podem ser controlados é o COOP/Orm, que possibilita o refinamento do controle de versões através da definição de duas gramáticas em formato EBNF. Quanto à extensibilidade, nenhuma das ferramentas analisadas oferece mecanismos para que seja integrada com outras ferramentas, e todas utilizam IDEs próprias.

No próximo capítulo, será apresentado o *Phoca*, que é constituído por uma implementação de um modelo de documentos, um modelo de versões, uma ferramenta que viabiliza sua utilização, uma API para que desenvolvedores possam utilizá-lo e adaptá-lo a outros sistemas, além de documentação que permitem sua correta utilização. Este sistema permite a utilização de controle de versões flexível para documentos estruturados.

Capítulo

4

Phoca

A partir das investigações sobre controle de versões de artefatos de software em geral que foram realizadas, identificaram-se pontos que poderiam ser melhorados no controle de versões refinado para os documentos estruturados. Os principais sistemas existentes possuem modelos de documentos e modelos de versões bem definidos e funcionais, tendo sido estes modelos o foco dos trabalhos desenvolvidos. Entretanto, por mais flexíveis que sejam estes modelos, não existe uma definição clara de como deve ser feito o mapeamento entre o conteúdo do arquivo de computador e o modelo de documentos que armazena este arquivo de forma detalhada num repositório para controle de versões. Além disso, os sistemas existentes possuem uma grande dependência em relação a ferramentas específicas e proprietárias, e sua adoção implicaria em grande impacto na cultura dos desenvolvedores, já habituados a determinadas IDEs que são otimizadas para as tarefas que realizam.

Com o objetivo de fornecer uma solução inicial para estas lacunas detectadas, foi desenvolvido o sistema *Phoca*. Propositalmente afirma-se que o sistema aqui descrito é uma solução inicial, pois ele foi implementado como um projeto de código aberto, que deverá ser aperfeiçoado e melhorado pelos interessados, conforme a necessidade. Além disso, a solução também é composta por documentação adicional, que, da mesma forma, deverá ser aperfeiçoada e adaptada às necessidades de cada utilizador ou grupo de utilizadores.

O projeto *Phoca* foi idealizado a partir da identificação das limitações nas ferramentas existentes para a realização das atividades de controle de versões flexível para documentos estruturados e, também, a partir da identificação de falta de ferramentas de código aberto disponíveis na comunidade – de todos os projetos estudados, apresentados no Capítulo 3, apenas o Stellation já foi disponibilizado como projeto de código aberto, mas não está mais

disponível.

Phoca é um sistema composto por um conjunto de APIs de código aberto que visam a possibilitar o controle de versões refinado e flexível para artefatos de software. O sistema foi licenciado sob a Apache versão 2[1].

Os sistemas de código aberto costumam ser desenvolvidos de forma que não seja estabelecido e seguido um processo formal de desenvolvimento. No entanto, no caso do *Phoca* foi utilizado um processo incremental, que é classificado por seus autores como um processo intermediário entre processos ágeis e tradicionais, chamado Iconix (ROSENBERG; STEPHENS; COLLINS-COPE, 2005; ROSENBERG; SCOT, 2001). Este processo define que alguns artefatos sejam gerados, tais como *mock ups* das interfaces gráficas, diagramas de casos de uso e diagramas de robustez – que é um diagrama sugerido pelo processo, que envolve a avaliação da compatibilidade entre os diferentes casos de uso. O Iconix é um processo iterativo, no qual deve ser adotada a prática de entregas pequenas e freqüentes ao cliente, além de reuniões periódicas para revisar o projeto. Neste projeto, o papel de cliente foi assumido pelo próprio autor do sistema e sua orientadora, além de outros alunos do grupo que se envolveram com o projeto. Os artefatos gerados durante a evolução do sistema foram armazenados num sistema de edição colaborativa via web e deverão ser distribuídos juntamente com o sistema. Na Figura 4.1 é apresentada a visão geral do processo Iconix.

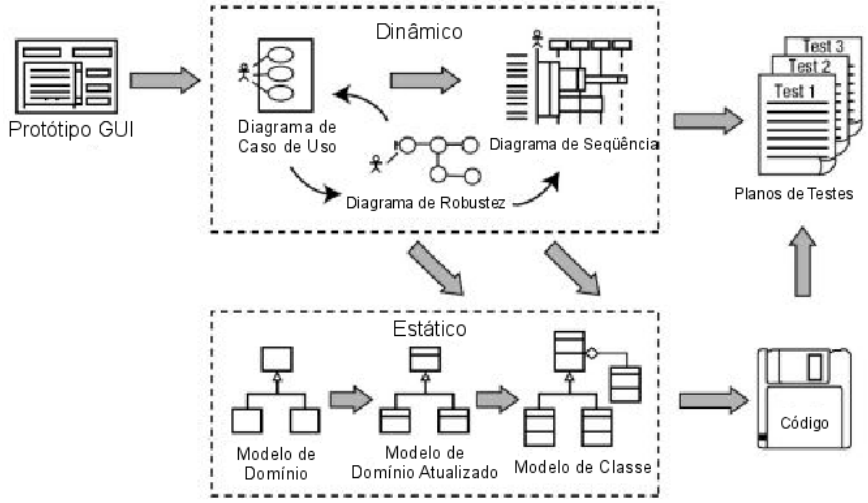

Figura 4.1. Visão geral do processo Iconix (ROSENBERG; STEPHENS; COLLINS-COPE, 2005)

Além da definição e implementação do sistema, também foi desenvolvida a documentação com dicas e procedimentos que visam a auxiliar na implantação de processos iniciais de controle de versões e GCS em ambiente corporativo – principalmente em pequenas e médias

[1] Disponível em `http://www.apache.org/licenses/LICENSE-2.0`

empresas.

Phoca é constituído por um conjunto de ferramentas e bibliotecas desenvolvidas em linguagem Java que podem ser configurados de forma a disponibilizar um ambiente de controle de versões refinado e flexível para artefatos de software. Ele está organizado numa arquitetura em camadas, que foi construída a partir de um conjunto básico de APIs implementadas. Quando possível, foram utilizadas soluções já existentes para compor a arquitetura geral do sistema.

Dentre seus diferenciais destacam-se o fato de ser um projeto de código aberto, que pode ser adaptado conforme as necessidades dos usuários; possuir uma API bem documentada que permite sua integração com outras ferramentas; ter o foco na facilidade de configuração das questões de controle de versões flexível para documentos estruturados, principalmente artefatos de software; ser um sistema modular, que pode ser integrado com outras soluções já existentes – por exemplo, sistemas de controle de versões tradicionais. Neste capítulo são descritos seu projeto e desenvolvimento, bem como os desafios e decisões tomadas.

4.1. Arquitetura do Sistema

O sistema *Phoca* foi desenvolvido com o objetivo de fornecer meios de se realizar atividades de controle de versões refinado em arquivos de computador, principalmente em projetos de software. Dentre suas características, destacam-se a flexibilidade de controle de versões para documentos estruturados e a existência de uma API que fornece meios de integração do sistema com outras ferramentas, possibilitando uma minimização no impacto da adoção deste sistema em ambientes reais de desenvolvimento de software. Na Figura 4.2 é exibida uma visão geral da arquitetura do sistema *Phoca*.

Na Figura 4.2 está representada uma configuração do sistema em que dois usuários estão acessando o repositório central e também estão realizando a sincronização entre si; a ilustração de uma situação com dois usuários foi escolhida para que pudesse ser melhor visualizada a interação permitida entre os usuários e o servidor e com os usuários entre si.

Os módulos representados como o *núcleo da aplicação* são módulos desenvolvidos em Java que implementam as funcionalidades principais do sistema, além de APIs que são utilizadas internamente pelo sistema e, opcionalmente, podem ser utilizadas por sistemas externos que queiram ter acesso às funcionalidades de controle de versões refinado e flexível disponibilizadas pelo *Phoca*.

Os módulos representados como *soluções externas / persistência* são constituídos por ferramentas de bancos de dados e repositório de controle de versões que são sistema já maduros e utilizados pela comunidade. Estes sistemas são distribuídos juntamente com o *Phoca*, mas podem ser alterados caso seja necessário.

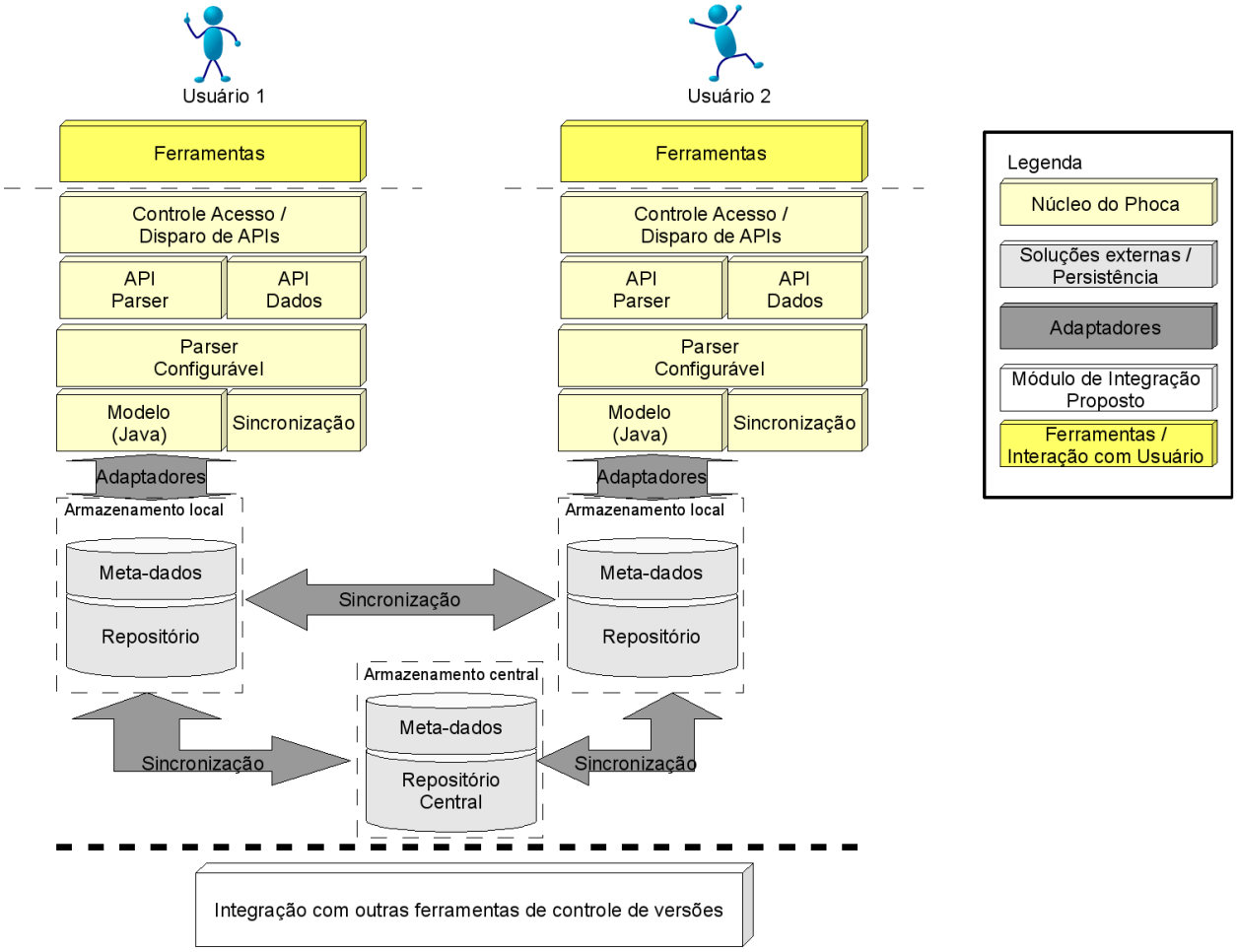

Figura 4.2. Visão geral da arquitetura do sistema *Phoca*

Os *adaptadores* são utilizados para mapear as funcionalidades das ferramentas externas para as chamadas realizadas pelo núcleo do sistema *Phoca*. Portanto, no caso de troca das soluções externas, também é necessário fornecer as novas classes que serão utilizadas para acessar as funções disponibilizadas por estas ferramentas; estas funções são relacionadas a funções de persistência e sincronização dos dados.

O módulo que está como *integração* foi proposto na arquitetura do sistema, mas não foi implementado no contexto deste trabalho. Sua principal funcionalidade é permitir a compatibilidade com ferramentas tradicionais de controle de versões, como SubVersion ou CVS.

As **ferramentas** são utilizadas para realizar interação com os usuários. É a partir deste módulo que torna possível a interação dos usuários com o sistema, para realizar as atividades de controle de versões, configuração do sistema, entre outras. Um conjunto básico de ferramentas é disponibilizado juntamente com o *Phoca*, mas novas ferramentas integradas a ambientes de desenvolvimento de software reais devem ser criadas de forma a ficarem integradas nestes ambientes.

Nas subseções que seguem serão explicadas as principais partes do sistema e suas funcionalidades e, por fim, serão apresentadas algumas considerações sobre o módulo de integração proposto.

4.1.1. Núcleo do sistema

O núcleo do sistema é composto por 6 módulos, essenciais ao funcionamento do sistema e que encapsulam a maior parte de sua lógica. A partir disso, já é possível observar que a lógica principal do sistema *Phoca* concentra-se em máquinas clientes, e não no servidor – na verdade, o servidor de forma isolada não tem qualquer funcionalidade.

É no núcleo do sistema que estão inseridos os modelos de documentos e de versões, além de todas as funcionalidades para a efetiva realização das atividades de controle de versões refinado. O núcleo do sistema, que foi construído de forma modular, é composto por um conjunto de módulos que interagem entre si. Os módulos das camadas mais abaixo fornecem serviços para os módulos acima.

Módulo de Modelo

Este é o módulo localizado na base do núcleo do sistema – juntamente com o módulo de Sincronização. Neste módulo está implementado o modelo de documentos que foi utilizado, e a lógica necessária para o controle de versões. Os dados deste módulo são mapeados para a camada de persistência, que no *Phoca* foi implementada utilizando um banco de dados HSQLDB[2] e um repositório Mercurial (MERCURIAL, 2007). Na Seção 4.1.3 é explicado o mecanismo implementado para realizar o mapeamento do modelo nos sistemas de armazenamento.

Para o desenvolvimento do sistema *Phoca*, foi criado um modelo de documentos inspirado no modelo de documentos do UEVM, que foi apresentado na Seção 3.4. Este modelo prevê a existência de quatro tipos de nós, para os quais a estrutura do documento deve ser mapeada. Estes nós são:

- Nó de documento: mapeado para cada arquivo
- Nó de composição: pode conter vários nós filhos, sendo eles de composição ou de ligação
- Nó de conteúdo: mapeado para o conteúdo real do documento ou partes dele; cada nó de documento e de composição possui um nó de conteúdo associado; desta forma todo o conteúdo de um documento pode ser encontrado em nós de conteúdo, e a divisão de conteúdo nestes nós é feita conforme a definição dos nós de composição.

[2] Página oficial do projeto: `http://www.hsqldb.org`

- Nó de ligação: representa a ligação entre um arquivo de origem e um de destino. O arquivo de origem deve conter uma referência para o arquivo de destino. Este nó precisa armazenar as informações de nome e versão do arquivo de destino.

O diagrama de classes dos nós que representam o modelo de documentos do *Phoca* é apresentado na Figura 4.3.

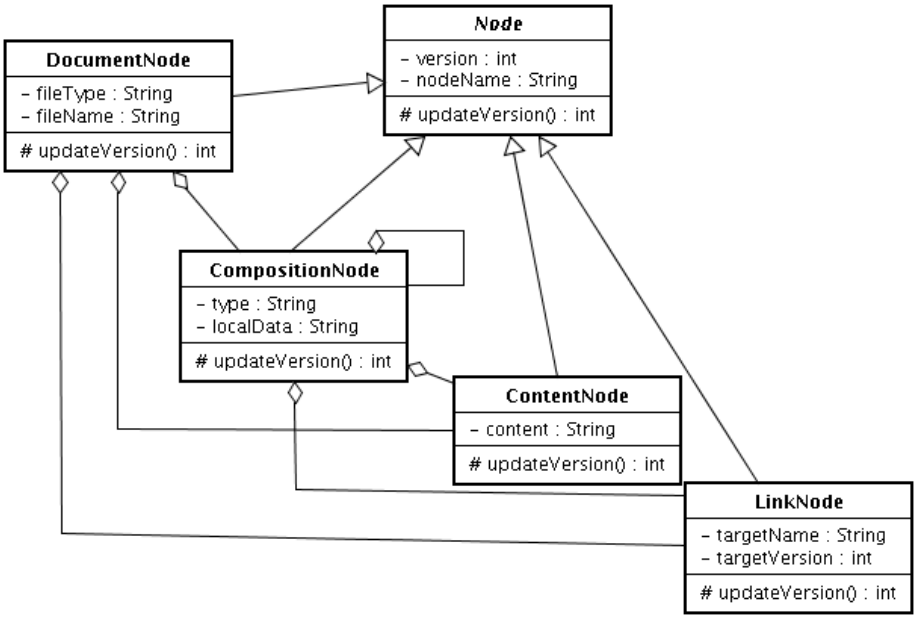

Figura 4.3. Diagrama de classes do modelo de documentos do *Phoca*

O módulo de Modelo implementa os quatro tipos de nós descritos, além de ter todas as regras associadas para calcular a alteração nos nós – conhecido como modelo de versões. O modelo de versões utilizado também foi inspirado no modelo de versões do UEVM. Na Figura 4.3 é possível observar o método *updateVersion*, que é utilizado para atualizar a versão dos nós, e que é responsável pelo mecanismo de sessões. O mecanismo de sessões foi criado para evitar a explosão combinatória, e, também, para permitir que os usuários enviem para o repositório central apenas versões significativas – o que pode variar conforme o local onde o sistema é utilizado, conforme as políticas de controle de versões. Segundo o modelo de versões, uma nova versão de cada tipo de nó é criada nas seguintes circunstâncias:

Regra 1: Nós de documento têm uma nova versão criada sempre que qualquer alteração é realizada no documento, seja na estrutura ou no conteúdo

Regra 2: Nós de composição e de conteúdo têm uma nova versão criada quando seus dados locais são alterados

Regra 3: Nós de ligação devem ter uma nova versão criada quando o nome ou a versão do documento de destino são alterados

Regra 4: Nós de composição devem ter uma nova versão criada quando um de seus nós filhos é adicionado, removido ou alterado (além da **Regra 2**)

O modelo de versões dita as regras que devem ser seguidas para a realização do controle de versões refinado. Estas regras estão bastante relacionadas com a alteração de dados locais de nós ou alteração de dados dos filhos de determinados nós – de qualquer forma, quer dizer que a versão de um nó pai se altera com a alteração de quaisquer de seus nós filhos. O sistema *Phoca* implementa estas regras com a utilização de encapsulamento dos dados – uma boa prática em programação orientada a objetos – e do padrão de projetos *Observer*[3] (GAMMA et al., 1995).

O "encapsulamento de dados" consiste em não permitir acesso direto aos atributos de uma classe por alguma classe externa. Ao invés disso, são disponibilizados métodos para atribuir e recuperar os valores destes atributos. Com isso, é possível que alguma validação seja realizada quando o valor de determinado atributo é alterado, por exemplo. No caso das classes do módulo de Modelo, todas as classes que representam objetos do modelo de domínio possuem encapsulamento de dados, mas nas classes dos nós de conteúdo e de composição é disparada uma mensagem para o método *updateVersion()* sempre que seus dados locais são alterados, ou seja, nos seguintes métodos:

- ContentNode: setContent(String newContent)
- CompositionNode: setName(String name)

Através da implementação destes métodos com disparo automático de mensagens para o método *updateVersion()* do próprio nó, implementa-se a regra 2. O nó de composição, no entanto, só pode ser alterado se um nó de conteúdo relacionado a ele (que contém sua definição e respectivo nome) for alterado.

Para a implementação das regras 1 e 4, foi utilizado o padrão de projetos *Observer*. Este padrão deve ser aplicado nos casos em que o estado de determinado objeto deve ser monitorado por outros objetos. Para atingir o objetivo, o padrão dita a implementação de objetos que podem ser observados e objetos que são observadores; os objetos do tipo observadores registram-se no objeto a ser observado, e quando determinado atributo do objeto observado é alterado, ele notifica a todos os seus observadores, que podem tomar uma ação em relação a isso – eles capturam dados como a origem do evento, o que foi que ocorreu e, a partir destes dados, decidem se precisam ou não tomar alguma decisão. No método *updateVersion()*, que é implementado por todos os tipos de nós, está programada a lógica que permite a um determinado nó determinar se sua versão deve ou não ser incrementada – implementando, assim, o mecanismo de sessões.

[3] Por ser um padrão de projeto (*design pattern*), o nome de referência do padrão foi mantido em inglês

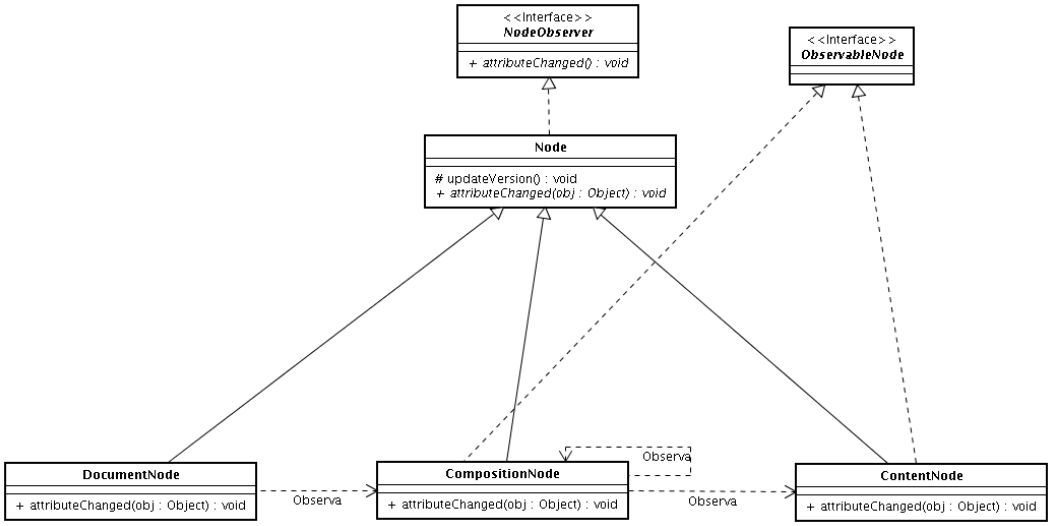

Figura 4.4. Diagrama ilustrando a modelagem do padrão *Observer*

Por fim, a implementação da regra 3 foi feita utilizando outros mecanismos de suporte. Isso ocorre porque foi dada flexibilização à forma como os usuários podem definir a regra no *meta-modelo do sistema*. Assim, um usuário pode escolher se a atualização das ligações ocorre automaticamente quando o objeto alvo de uma ligação sofre alteração de versões ou se o usuário deve, explicitamente, solicitar que a versão da ligação seja atualizada. Esta flexibilidade dada ao usuário é útil pois os projetos podem ter políticas próprias de atualização de dependências, que são diretamente relacionadas com as políticas de testes, por exemplo. Portanto, a atualização dos nós de versões foi implementada utilizando uma biblioteca externa, que, dependendo das configurações do meta-modelo, irão realizar uma varredura pelo modelo buscando por atualizações nas ligações.

Além do modelo de documento e modelo de versões, que estão implementados no módulo de Modelo do *Phoca*, também existe implementado um meta-modelo do sistema, que contém meta-dados utilizados pelo *parser* – que é responsável por analisar um documento e fazer seu respectivo mapeamento para o modelo de documentos – e também meta-dados utilizados pelo modelo de versões, mais especificamente para o controle de versões dos nós de ligação.

A técnica de mapeamento entre os documentos e o modelo de documentos utilizada é uma técnica de análise por expressões regulares aliada a pré-processamento e tratamentos de exceções. As técnicas de pré-processamento foram implementadas para que fosse realizado o rastreamento de uma forma "*top down*", ou seja, dos níveis mais altos para os mais baixos da estrutura. As técnicas de tratamento de exceções foram implementadas para facilitar a construção das expressões regulares, além de permitir, por exemplo, o empilhamento de símbolos (tokens).

O módulo de Modelo contém as classes que permitem configurar a forma como as expressões regulares serão utilizadas pelo módulo de *parser* configurável. Os meta-dados armazenados pelo meta-modelos são:

- Regras para os nós de documentos: num projeto ou repositório, podem existir diversos tipos de documentos diferentes, como, por exemplo, documentos de código-fonte desenvolvidos em Java, código-fonte de documentação em Latex, documentação em HTML, entre outros. Portanto, devem existir regras claras do mapeamento de cada tipo de arquivo para um tipo de nó de documento específico. A partir do tipo de documento, todas as outras regras serão direcionadas. A princípio, este mapeamento é feito utilizando-se a extensão do arquivo. Mais sobre este mapeamento na descrição do módulo de parser configurável e API de Parser.

- Regras de análise para cada tipo de nó dentro dos documentos: estas regras não incluem, obviamente, o próprio nó de documentos, que é mapeado pela regra definida anteriormente. Entretanto, um documento pode conter diversos tipos de nós, seguindo o modelo de documentos genérico que existe no sistema. A partir destas regras pode-se mapear, por exemplo, métodos em nós de composição e todo o conteúdo do método (incluindo sua assinatura) nos nós de conteúdo. São estas regras que definirão o nível de granulosidade do projeto.

- Regras de atualização de dependências: os nós de ligação não são atualizados segundo uma regra única. O parser é configurado de forma que, para cada projeto, tenha-se a opção de fazer ou não a atualização automática das ligações.

Estes dados são armazenados em classes do meta-modelo, que são mapeadas para o banco de dados e repositório de controle de versões. A atualização do meta-modelo pode ser feita sempre que necessário, e o sistema é capaz de armazenar a informação sobre qual versão do meta-modelo estava associada determinada versão de um arquivo – ou projeto – que está tendo sua versão controlada pelo sistema. Ressalta-se o fato de que não é necessário ter qualquer informação do meta-modelo para reconstruir determinado nó de informação, mas é necessário ter essas informações para realizar a análise do conteúdo e mapeamento do mesmo para o modelo de documentos.

Módulo de Sincronização

No *Phoca* é necessário realizar a sincronização entre o repositório central e o repositório local do usuário. Também é possível realizar sincronização direta entre dois usuários, mesmo que tenham armazenadas versões intermediárias de determinados nós. As possíveis operações de sincronização, que são implementadas no módulo de sincronização, são:

1. Abertura de sessão do usuário e respectiva recuperação de dados para repositório local
2. Fechamento de sessão do usuário e respectivo envio dos dados do repositório local para o repositório central
3. Sincronização entre usuários dos arquivos sendo editados em comum, para prever e evitar conflitos no repositório central
4. Sincronização para edição colaborativa, conjunta, em tempo real
5. Sincronização de ramificações entre os dois repositórios dos usuários, incluindo as versões intermediárias

As funcionalidades deste módulo são dependentes do tipo de repositório que estiver sendo utilizado na implantação do sistema e também das ferramentas disponíveis para os usuários. A implementação oficial do *Phoca* utiliza, na camada de persistência, um banco de dados HSQLDB e um repositório Mercurial, que fica disponível para cada usuário que instalar o cliente. Para o funcionamento completo do módulo de sincronização remota, é necessário que os adaptadores implementem as operações de sincronização de uma versão específica e de sincronização de ramificações diretamente no repositório – estas operações são, respectivamente, as operações de *merge* e *push* nos sistemas de controle de versões distribuídos.

A operação de edição colaborativa é realizada diretamente na camada de modelo do sistema, e exige a configuração de alguns parâmetros de rede para ser utilizada, que devem ser configurados através da API. A implementação desta operação deve ser feita através da utilização de alguma ferramenta gráfica de interação com o usuário, de forma que o estado do modelo seja capturado com freqüência e a camada de apresentação altere os dados visualizados pelo usuário quando necessário. As ferramentas que forem implementadas para disponibilizar as funcionalidades de sincronização devem implementar uma interface Java disponibilizada junto com o sistema que foi criada para este fim. No *Phoca* não foi implementada nenhuma interface gráfica para este tipo de sincronização, e foram realizados testes diretamente com as classes criadas para oferecer este tipo de operação.

Módulo de Parser configurável

Um dos principais diferenciais do sistema *Phoca* é a flexibilidade permitida pelo mesmo. Esta flexibilidade está associada tanto à flexibilidade do sistema – que pode ser integrado a diversas outras ferramentas através da disponibilização de suas APIs – como pela flexibilidade e facilidade de mapeamento de uma estrutura de documento no modelo de documentos do sistema.

Para que o mapeamento entre o conteúdo do documento e o modelo de documentos seja flexível, é parte integrante do sistema um meta-modelo de documentos, cujas classes estão no módulo de modelo. Estes dados de configuração do meta-modelo são utilizados pelo módulo de parser configurável, para que possa realizar a análise dos documentos e mapear para o modelo de documentos. Na Figura 4.5 é ilustrada a idéia geral do funcionamento do módulo de parser configurável. Nela pode ser observado que o módulo de parser configurável recebe como entradas um arquivo escrito numa linguagem específica e um meta-modelo de documento para esta linguagem, que contém as regras de mapeamento entre o arquivo e o modelo de documentos do sistema. O módulo de parser configurável utiliza então as informações do meta-modelo para realizar o mapeamento. Como saída deste processo, é fornecido o documento original mapeado para o modelo de documentos implementado.

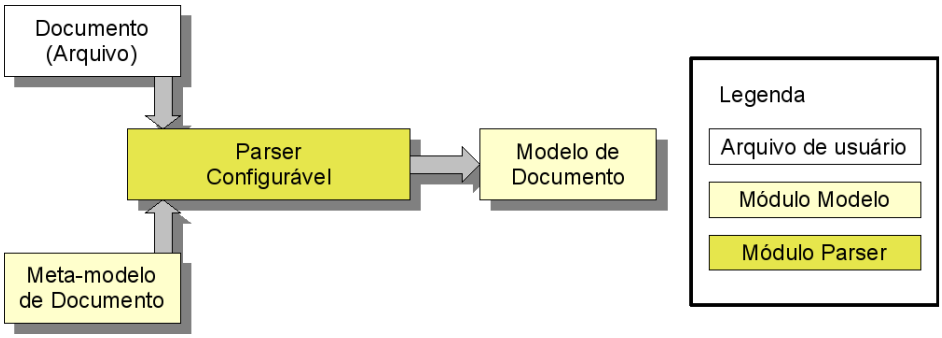

Figura 4.5. Ilustração do funcionamento do parser configurável

Na Figura 4.6 é exibido o modelo conceitual do meta-modelo do módulo de parser, e estão ilustrados alguns dos métodos modelados.

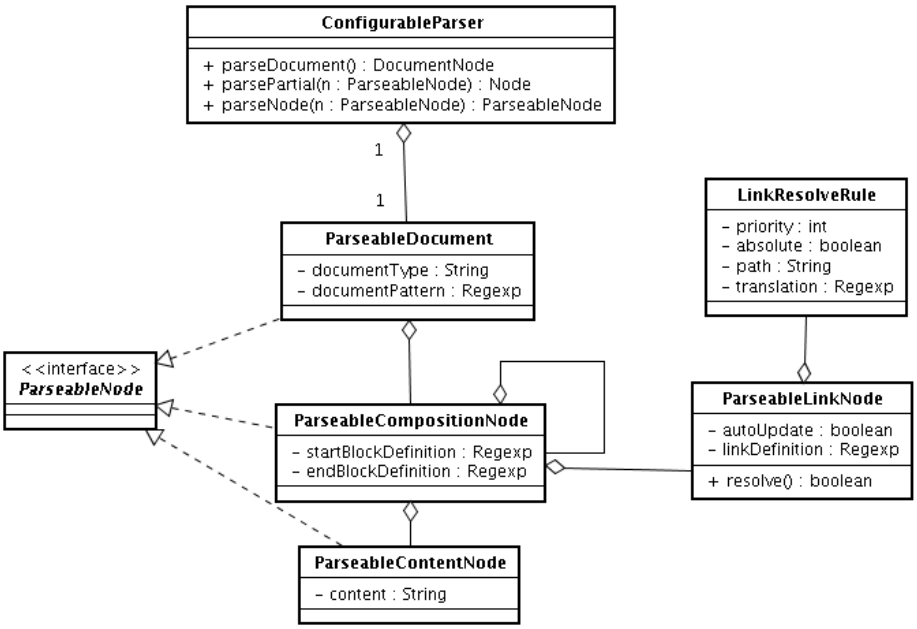

Figura 4.6. Diagrama de classes do meta-modelo utilizado pelo módulo de parser configurável

O modelo apresentado na Figura 4.6 está fiel à implementação que foi realizada, embora não seja necessário declarar um nó de composição para um documento. Um nó de composição é sempre alocado para um documento, e este tem pelo menos um nó de conteúdo associado. De forma indireta, tem-se o nó de conteúdo associado ao nó de documento.

A partir das regras definidas no meta-modelo, o parser realiza a análise do documento da seguinte forma: primeiramente é feito o mapeamento do arquivo num nó de documento, conforme as regras de mapeamento de arquivos para nós de documentos. Então todo o conteúdo é atribuído a um nó de conteúdo (associado ao documento). Após esta atribuição, o nó de conteúdo associado ao documento é submetido à nova análise, que procura pelos nós filhos do nó de documento. E a análise continua desta forma, sucessivamente, dos níveis mais altos para os mais baixos da estrutura declarada. Na Figura 4.7 é ilustrado um exemplo de funciomanento do *parser* para um documento Latex. Nesta figura, podemos observar que no passo 1 todo o texto foi atribuído ao nó de conteúdo do documento, no passo 2 foi feita a divisão entre preâmbulo e documento, e, no passo 3, o conteúdo do documento foi novamente analisado, sendo encontrados os nós de capítulo dentro do documento. Esta estratégia foi adotada após o estudo de alguns parsers disponíveis que não conseguiram satisfazer as necessidades do projeto, principalmente devido à complexidade. A maioria dos parsers e geradores de parsers disponíveis exige que seja utilizado uma série de regras para seu funcionamento, geralmente baseado na EBNF da gramática, e o objetivo deste trabalho era permitir uma configuração mais simples e flexível do mecanismo de análise dos documentos.

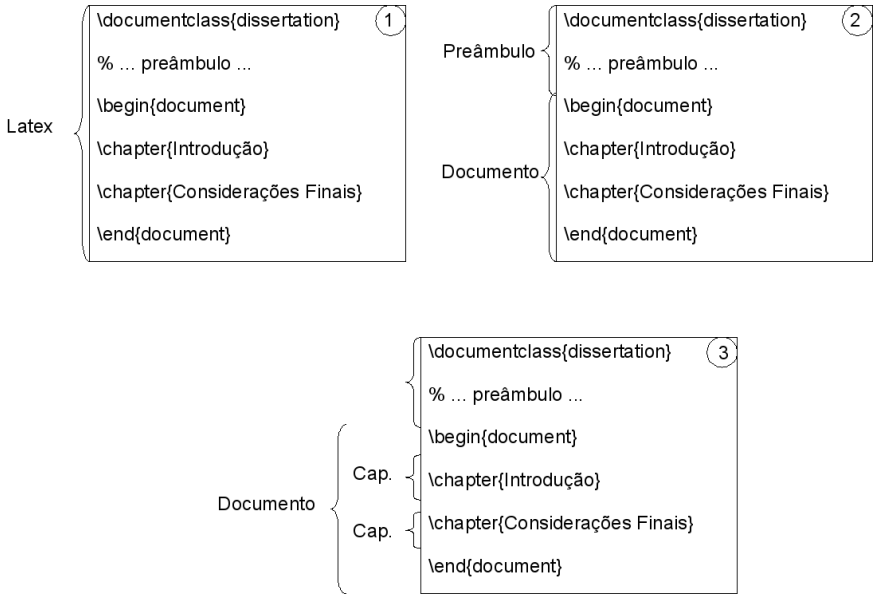

Figura 4.7. Ilustração da técnica de análise dos documentos

Outro detalhe que deve ser ressaltado é a classe "LinkResolveRule", que contém as regras para resolução de nós de ligação. Esta informação é utilizada para que o parser possa resolver os nós de ligação, i.e., encontrar o arquivo alvo deste nó de ligação. Cada definição de links

pode ter um conjunto de regras utilizadas para resolução, que podem ser caminhos relativos ou absolutos. Os caminhos relativos são calculados a partir da localização atual do arquivo, enquanto os caminhos absolutos são calculados a partir do diretório que corresponde à raiz do repositório. Portanto, os caminhos são sempre relativos: ou à posição do arquivo no projeto ou à raiz do projeto. Na classe que representa os nós de ligação existe um atributo que indica se este nó está resolvido ou não – em caso negativo, os atributos "nome" e "versão" do alvo não são inicializados, e é utilizado um comportamento padrão.

Módulo de API do Parser

O módulo de API do Parser é o módulo que disponibiliza as funcionalidades para a completa configuração do parser. A implementação deste módulo foi realizada com a utilização do padrão de projeto *Facade* (GAMMA et al., 1995). O padrão de projeto *Facade* deve ser utilizado quando é necessário fornecer uma funcionalidade complexa de forma abstrata. A classe externa (cliente) acessa o método implementado na classe *Facade*, e este método executa as operações complexas de forma transparente para a classe cliente. Na Figura 4.8 é ilustrada a idéia principal do padrão de projeto *Facade*.

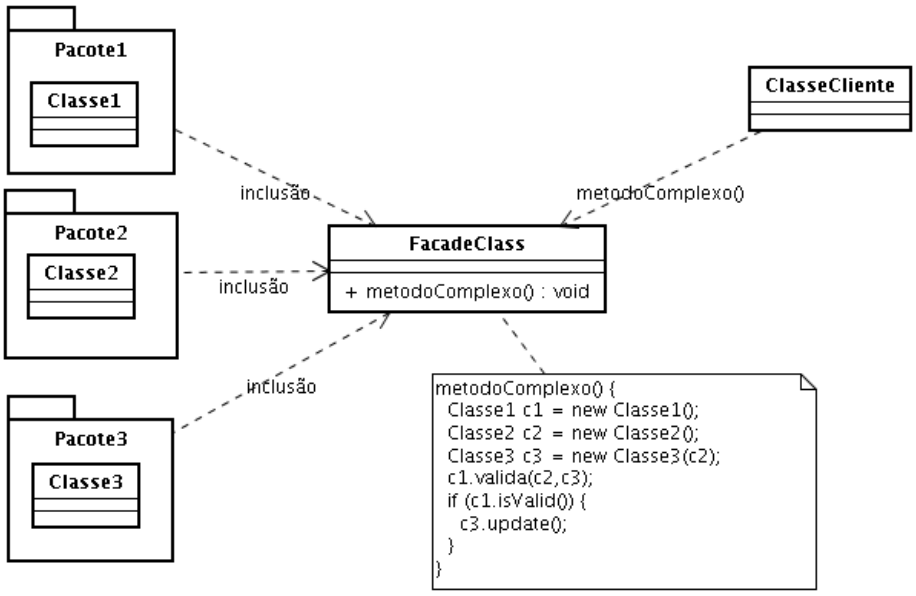

Figura 4.8. Ilustração do padrão de projeto *Facade*

As funcionalidades disponibilizadas pela API são:

- Inicialização de nova configuração de repositório: cria as classes necessárias para que seja criado um novo parser configurável a ser aplicado no repositório
- Listagem de parsers configurados: retorna uma lista de parsers configuráveis que já estão em operação no repositório.

- Configuração de novo documento: métodos utilizados para configurar os nós de documentos do meta-modelo
- Configuração de nó de composição: métodos utilizados para configurar os nós de composição do meta-modelo
- Configuração de nó de ligação: métodos utilizados para configurar os nós de ligação de meta-modelo
- Validação da configuração: métodos utilizados para validar a configuração. A validação pode ser realizada contra dados atuais do repositório ou contra dados de teste
- Atualização da versão do parser: após configurado um novo parser, sua versão pode ser atualizada. Caso ocorra a atualização da versão do parser, todos os documentos passam a funcionar com a nova versão. Existe a possibilidade de se realizar atualizações retroativas, que são úteis, por exemplo, nos casos em que a atualização é feita para corrigir determinado problema do parser – que pode ter gerado erros em versões anteriores já armazenadas no repositório. Entretanto, o comportamento padrão da atualização de versão do parser é atualizar apenas a partir do momento em que a nova versão foi incluída. Os meta-dados do repositório armazenam, entre outras informações, qual a versão do parser que está associada a cada versão de documento

Ressalta-se ainda o fato de que o parser não precisa ser utilizado para recuperar a versão original dos documentos, que pode ser obtida diretamente pelo módulo de Modelo. Assim, a API de Dados consegue acesso direto ao conteúdo original dos documentos, sem necessidade de utilização da API do Parser para isso; entretanto, toda a operação que envolve enviar novo conteúdo para o repositório exige a utilização do parser configurável para analisar o conteúdo enviado, mesmo que apenas um nó pequeno tenha sido editado.

Módulo de API de Dados

O módulo de API de dados disponibiliza aos desenvolvedores de aplicações métodos para acesso aos dados armazenados no repositório; é esta API que fornece as principais funcionalidades de controle de versões. As principais funcionalidades deste módulo são:

- Listagem do repositório central: retorna a listagem do repositório em diferentes níveis de profundidade. O primeiro nível de profundidade corresponde a uma listagem da estrutura do repositório como arquivos e diretórios, e níveis de profundidade maiores permitem a recuperação imediata de informações detalhadas dos arquivos.
- Recuperação da estrutura de documentos: utilizada para verificar a estrutura de documentos específicos, em diferentes níveis de profundidade.

- Recuperação de conteúdo de nós: os nós de conteúdo são os nós folhas do modelo de documentos. A API de acesso aos dados possui funcionalidades para recuperação do conteúdo dos nós de conteúdo associados a determinado nó do documento.

- Salvamento de versões intermediárias – repositório local: utilizado para serializar os dados em memória para o banco de dados e repositório de controle de versões local.

- Recuperação de informações de meta-dados: todas as informações de meta-dados podem ser recuperadas por esta API. Os meta-dados envolvem a configuração do repositório, configurações de usuários, permissões.

- Sincronização de dados com outros usuários: o módulo de API de dados utiliza, de forma transparente para o utilizador da API, a utilização do módulo de sincronização, e fornece função de sincronização de dados com outros usuários, sem intervenção do repositório central.

- Resolução de conflitos: são fornecidos métodos para resolução de conflitos em edição concorrente.

Este módulo foi implementado utilizando o padrão de projeto *Facade* (GAMMA et al., 1995). A API fornece aos desenvolvedores comandos para realização das operações que são fornecidas pelos outros módulos do sistema. Por exemplo, a recuperação da estrutura de documentos envolve atividades do modelo, e o salvamento de versões intermediárias envolve os módulos de modelo, parser configurável, adaptador para acesso ao repositório, adaptador para acesso ao banco de dados.

Módulo de Controle de Acesso e Despacho de APIs

Este é o módulo disponibilizado para que ferramentas e classes externas utilizem as funções do *Phoca*. As próprias ferramentas disponibilizadas junto com o sistema utilizam este módulo para acessar as funcionalidades das APIs.

A implementação deste módulo é feita com a utilização de mecanismos de controle de acesso que permitem a verificação da permissão de usuários ao sistema e, principalmente, às funcionalidades disponibilizadas pelas APIs. Além da verificação de acesso, este módulo também disponibiliza as funcionalidades de todas as APIs que estão sob ele.

Para a implementação desta funcionalidade é utilizado o padrão de projeto Command (GAMMA et al., 1995), através do qual é possível fazer o mapeamento de diversos comandos de forma abstrata, utilizando encapsulamento de parâmetros para isso. Em relação à arquitetura do sistema, a utilização deste padrão de projeto assemelha-se à implementação direta das funcionalidades – portanto, as ferramentas que acessam as APIs "enxergam" um

módulo concentrador de funcionalidades. Entretanto, com a utilização do padrão de projeto aumenta-se a manutenibilidade do sistema, que pode ser estendido livremente – e, a cada extensão, basta registrar no banco de dados a existência de novas funcionalidades na API e quais os parâmetros necessários para que o módulo de Despacho de APIs consiga tratar as novas funcionalidades.

Além disso, o módulo de Despacho de APIs funciona realmente como um despachador, invocando o comando apropriado quando este estiver disponível e simplesmente avisando ao usuário que determinado comando não está disponível quando for invocado, pelo usuário, um método que não é disponibilizado por nenhuma das APIs.

Portanto, o ciclo de funcionamento deste módulo é:

1. Módulo é inicializado
2. APIs disponíveis são acopladas ao módulo – padrão de projeto Command
3. Ferramentas são acopladas ao módulo
4. A cada solicitação de funcionalidade pelas ferramentas, o módulo checa a permissão do usuário para aquela funcionalidade e, caso esteja correto, empacota a requisição para ser executada pela API correspondente
5. Comando é executado na API e resultado é enviado de volta ao módulo
6. Resposta é enviada à ferramenta que apresenta ao usuário

Para a implementação do controle de acesso foi utilizado o banco de dados HSQLDB e funções dos repositórios de controle de versões para que possam ser rastreados os dados de autoria.

4.1.2. Soluções externas

Além dos módulos do núcleo do sistema, vitais para sua funcionalidade, existem os módulos do sistema que utilizam soluções externas para funcionar. No sistema *Phoca* são utilizados sistemas externos para realizar o armazenamento dos dados.

Para o armazenamento de alguns meta-dados e armazenamento dos dados de controle de acesso, é utilizado um banco de dados relacional desenvolvido completamente em Java, que pode ser empacotado e distribuído junto com as aplicações, chamado HSQLDB. Este banco de dados foi escolhido devido ao fato de poder ser completamente integrado ao sistema, facilitando seu empacotamento e distribuição, e também por ser desenvolvido em Java, mesma linguagem em que o *Phoca* está implementado. Ele é distribuído sob uma licença de código

aberto, "*HSQLDB License*", que permite sua alteração e redistribuição em formato binário – compatível com a licença do *Phoca*.

Outra solução externa adotada é o sistema de controle de versões distribuído Mercurial, que é desenvolvido em Python e é licenciado sob a GPL. Foram estudados outros sistemas de controle de versões para serem colocados na versão oficial do *Phoca*, como o Bazaar-NG[4], Git[5] e SubVersion. Entretanto, o Mercurial foi o sistema escolhido por ter apresentado a melhor performance, melhores mecanismos de sincronização de repositórios e melhor compatibilidade com diferentes sistemas operacionais. O Git, que é utilizado para o controle de versões do kernel do Linux e ter o melhor desempenho geral, não é compatível com o sistema operacional Microsoft Windows – tal limitação comprometeria o projeto como um todo, que foi desenvolvido em linguagem Java e pode ser executado em diferentes sistemas operacionais. Vale ressaltar que grandes projetos de software livre, como o projeto Mozilla, após avaliarem os sistemas de controle de versões disponíveis, também migraram para o Mercurial (MERCURIAL, 2007).

4.1.3. Adaptadores

Devido à utilização de soluções externas, o sistema utiliza um mecanismo de adaptadores para poder realizar a integração com soluções diferentes. Devido à utilização de adaptadores, é possível que as soluções externas utilizadas sejam trocadas, desde que seja implementado um adaptador para a nova solução.

Este mecanismo é o mesmo que *frameworks* de persistência bastante conhecidos utilizam, como o Hibernate (ELLIOTT, 2004) para persistência em Java. Através da utilização de adaptadores, é possível implementar os sistemas com independência das ferramentas utilizadas – a utilização de tal abordagem também pode ser vistas em aplicações corporativas (GONG; GORTON; FENG, 2005).

Algumas partes do sistema *Phoca* podem ser adaptadas para que as funcionalidades de ferramentas externas sejam executadas corretamente. Este tipo de abordagem é o mesmo que é utilizado em *frameworks* de persistência, como o Hibernate, para Java.

Algumas interfaces são disponibilizadas pelo sistema. Nestas interfaces são declarados os métodos que são utilizados nas ferramentas externas – por exemplo, recuperação de metadados, que envolve uma consulta SQL no banco de dados relacional utilizado, e recuperação de determinada versão de um arquivo no repositório. Todos estes métodos são declarados nas interfaces apropriadas, e, ao se utilizar ferramentas externas, pode-se fornecer ao sistema os adaptadores necessários. Para isso, basta configurar nos arquivos de configuração *client.yaml* e *server.yaml* quais são as classes destes adaptadores. O núcleo do sistema, quando precisar

[4] Página oficial do projeto: `http://bazaar-vcs.org`
[5] Página oficial do projeto: `http://git.or.cz`

executar operações nas ferramentas externas, checará primeiro estes arquivos de configuração para encontrar definições diferentes das padrões; caso não seja encontrado, serão utilizados os adaptadores padrão implementados na ferramenta - para acesso ao HSQLDB e ao Mercurial.

Juntamente com o sistema *Phoca* é disponibilizada documentação detalhada sobre os adaptadores. Deve-se ter em mente que as alterações realizadas nos adaptadores – ou implementação de novos adaptadores – influenciam diretamente o funcionamento do sistema, podendo inclusive levar a situações de erro crítico.

4.1.4. Ferramentas e Interação com Usuário

No topo da arquitetura do sistema *Phoca* encontra-se o módulo de ferramentas. Durante o desenvolvimento do sistema, este foi o módulo que influenciou a maior parte das decisões tomadas, inclusive em relação ao projeto do sistema e sua divisão em módulos e camadas.

Neste módulo foram implementadas as ferramentas que permitem a interação dos usuários com o sistema. É também neste módulo que se encaixam novas ferramentas que forem desenvolvidas, e mesmo plugins e módulos para integração das funcionalidades do *Phoca* com ferramentas já utilizadas em ambientes de desenvolvimento de software, como, por exemplo, as IDEs Eclipse e NetBeans.

Duas ferramentas gráficas foram desenvolvidas para o sistema *Phoca*: um configurador de meta-modelos e um editor de documentos; tais ferramentas atendem aos casos de uso relacionados a configuração de meta-modelos e edição de documentos. Para os outros casos de uso, foram desenvolvidas ferramentas que devem ser utilizadas diretamente na linha de comandos. Esta abordagem foi utilizada justamente porque o foco do sistema é fornecer as APIs que permitem a realização do controle de versões refinado e flexível para artefatos de software, sem, no entanto, obrigar os usuários a utilizar ferramentas proprietárias ou uma IDE específica do sistema. Na Figura 4.9 são mostradas as telas do editor de documentos, à esquerda, e do configurador de meta-modelos, à direita.

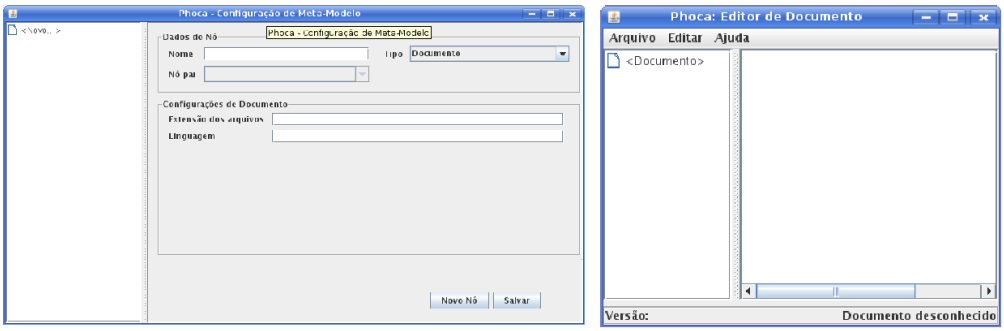

Figura 4.9. Configurador e editor de documentos do *Phoca*: interface gráfica

Para utilização via linha de comando foi implementada a ferramenta *phc*. Além dos

comandos tradicionais dos sistemas de controle de versões, esta ferramenta também implementa comandos específicos que são utilizados devido ao fato do *Phoca* ser um sistema de controle de versões refinado, como o comando para atualização de links, listagem de links existentes, recuperação de dados relacionados à estrutura de arquivo, compartilhamento de nó com outro usuário, alteração do meta-modelo de documento do projeto. A descrição dos principais comandos disponibilizados pela ferramenta é listada abaixo:

- **add**: adiciona um novo arquivo sob controle de versões. O arquivo é marcado para inclusão, e a inclusão definitiva é feita após a execução do comando *commit*
- **checkout**: recupera uma versão do repositório central, incluindo todos os nós existentes. Pode ser informada uma versão do repositório a ser recuperada, de qualquer uma das ramificações.
- **checkout structure**: recupera apenas a estrutura dos arquivos do repositório central. Pode ser informada uma versão do repositório a ser recuperada.
- **full checkout**: realiza uma sincronização com o repositório central, recuperando todas as versões armazenadas. Como parâmetros podem ser passadas a primeira e a última versão que devem ser recuperadas; este parâmetro deve ser numa ramificação específica. Se nenhum parâmetro for fornecido, todas as ramificações são recuperadas.
- **commit**: salva todas as alterações locais no repositório central, na ramificação correspondente.
- **save**: salva a versão intermediária de um arquivo. Não interage com repositório central.
- **load**: carrega a versão intermediária de um nó, podendo corresponder a um documento ou qualquer uma de suas partes.
- **sync**: realiza a sincronização de um nó ou de um determinado diretório com outro usuário. Deve ser passado o IP do usuário de destino com quem será feita a sincronização, além da freqüência de sincronização em segundos.
- **list metamodels**: lista os meta-modelos de documentos disponíveis no repositório central. Opcionalmente pode-se passar o tipo de arquivo a que o meta-modelo faz referência.
- **update metamodel**: atualiza a versão do meta-modelo de documento para um arquivo ou conjunto de arquivos específicos. A atualização é enviada ao repositório central. O usuário deve ter permissão para executar esta operação. Deve ser passado como parâmetro o tipo de arquivo e a versão desejada do meta-modelo.

- **update link**: atualiza a versão de um nó de ligação específico. Deve ser fornecido o id do link e a nova versão do arquivo alvo desejado (a atualização do arquivo ocorre via análise do documento)
- **list links**: lista nós de ligação existentes em determinado nó ou numa determinada porção do repositório
- **update all links**: atualiza todos os links de um nó ou de um diretório do repositório para a versão mais recente dos arquivos alvo
- **update all metamodels**: atualiza todos os meta-modelos de um determinado diretório do repositório para a versão mais recente

A utilização das ferramentas via linha de comando é suficiente para testar a aplicação, validando suas funções principais de controle de versões refinado e flexível. Entretanto, algumas funcionalidades que são oferecidas pelas APIs são difíceis de serem implementadas como ferramenta de linha de comando, como, por exemplo, a edição colaborativa de nós de documentos em tempo real.

Vale ressaltar que as APIs do sistema *Phoca* foram desenvolvidas utilizando o padrão de projeto *Observer*, o que facilita a construção de interfaces gráficas de usuário. Isso ocorre porque este padrão permite que atualizações no módulo de Modelo sejam capturadas automaticamente pela interface, que pode, então, atualizar a apresentação dos dados aos usuários.

4.1.5. Módulo Proposto - integração com ferramentas tradicionais de controle de versões

Por se tratar de uma ferramenta de controle de versões refinado e flexível, foi planejado um módulo para integração com ferramentas de controle de versões tradicionais. A função deste módulo é a de "emular" repositórios de outras ferramentas, permitindo aos usuários a execução de comandos como *checkout* e *commit* a partir de seus sistemas de controle de versões tradicionais. Entretanto, tais usuários não perceberiam nenhuma das vantagens do sistema de controle de versões refinado.

Embora o sistema *Phoca* utilize um repositório Mercurial, nem mesmo os comandos executados a partir de clientes do sistema Mercurial diretamente no repositório do *Phoca* retornam dados úteis aos usuários. Isso ocorre porque antes de serem armazenados no repositório os dados são convertidos para o modelo de documentos, e o modelo de documentos é mapeado para o repositório – conforme está explicado na descrição do módulo de armazenamento.

Para a compatibilidade do *Phoca* com outras ferramentas é necessário implementar um módulo de integração para cada sistema que se quiser emular. Por exemplo, se for realizada

a integração com Mercurial, precisa-se implementar um módulo que emule um repositório Mercurial. Para isso, é necessário implementar os comandos existentes nos repositórios Mercurial e, através das APIs fornecidas pelo sistema *Phoca*, realizar o mapeamento entre os dados armazenados no repositório e os dados que devem ser recuperados pelo comando fornecido pelo usuário. Na Figura 4.10 é ilustrada a idéia deste módulo de integração. Nesta figura, o usuário pretende copiar todo um repositório Mercurial – referenciado como *hg*, que é o comando utilizado para acessar o sistema – para uma área local em seu computador. Para isso, executa o comando *branch* e passa como argumento o endereço de um repositório Mercurial. Neste caso, o endereço do repositório utilizado pelo usuário é o endereço do repositório emulado pelo módulo de integração – portanto, não existe um repositório real neste endereço, mas o módulo de integração cria um repositório virtual, e todos os comandos executados neste repositório são então repassados ao módulo de integração. Ao receber o comando, o módulo de integração realiza as ações necessárias no sistema *Phoca*, construindo a estrutura que existira num repositório Mercurial real. Então esta estrutura é enviada ao usuário, no formato idêntico que seria enviado pelo sistema Mercurial servidor. A impressão do usuário será a de que realmente acessou um repositório Mercurial.

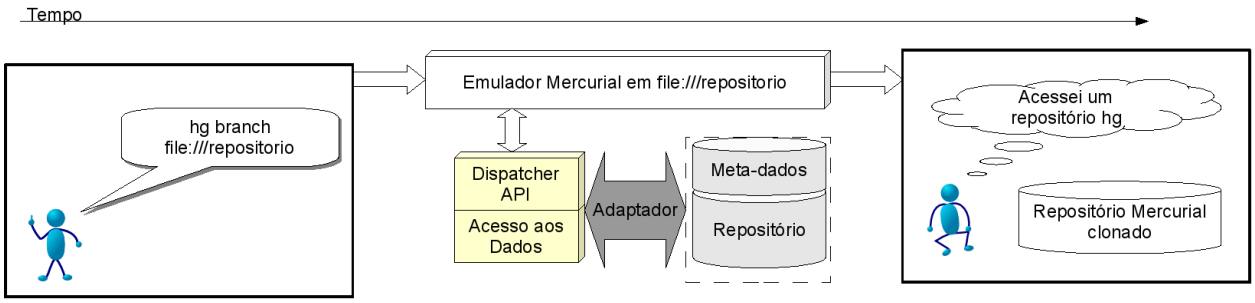

Figura 4.10. Exemplo do funcionamento do módulo de integração

A implementação deste módulo utiliza as mesmas funcionalidades de integração disponíveis para o desenvolvimento de qualquer outro tipo de ferramenta. Não foi necessário criar qualquer nova funcionalidade do sistema para que um módulo de integração com ferramentas externas pudesse ser projetado. O objetivo de manter este módulo desenhado na arquitetura do sistema é de mostrar que existe a possibilidade de se realizar a integração com ferramentas tradicionais de controle de versões e, também, mostrar que não é possível realizar o acesso direto ao repositório existente no *Phoca*, mesmo que seja utilizado o cliente do repositório utilizado no servidor do *Phoca*, pois os arquivos não são armazenados integralmente no repositório, e toda a nomenclatura é alterada.

Este módulo não foi implementado no sistema, embora tenha sido documentado todo o projeto de implementação, a partir do qual foi possível perceber em detalhes que não seriam necessárias alterações no sistema para que fosse implementado um módulo de integração com ferramentas de controle de versões tradicionais.

Deve ser destacado, ainda, que podem ser implementados e utilizados quantos módulos de integração com sistemas de controle de versões tradicionais quantos forem necessários, e que a utilização destes módulos não impede o funcionamento do sistema *Phoca*. Na Figura 4.11 é ilustrado um exemplo de como seria a utilização de vários módulos de integração com sistemas tradicionais em paralelo. Nesta figura, cada usuário tem a impressão de estar acessando um repositório de uma determinada ferramenta – portanto, o usuário SubVersion, que utiliza o sistema cliente do SubVersion, terá a impressão de que existe disponível um servidor SubVersion no endereço emulado pelo módulo, que foi denominado "repositório virtual SubVersion". Da mesma forma, cada usuário dos outros sistemas terá acesso a seus respectivos repositórios virtuais, nos endereços que serão emulados. O usuário do sistema *Phoca* poderá utilizar normalmente seu acesso ao repositório central e local. O repositório local de nenhum dos sistemas foi representado, mas o *Phoca*, o Mercurial e o Bazaar possuem repositórios locais, que funcionarão normalmente para os usuários. A diferença entre SubVersion (repositório central) e Bazaar (repositório distribuído), por exemplo, é que os comandos emulados do SubVersion irão disponibilizar a área de trabalho do usuário sem repositório local, e os comandos emulados do Bazaar irão disponibilizar a área de trabalho local acoplada a um repositório local do usuário, que é uma sub-árvore do repositório central.

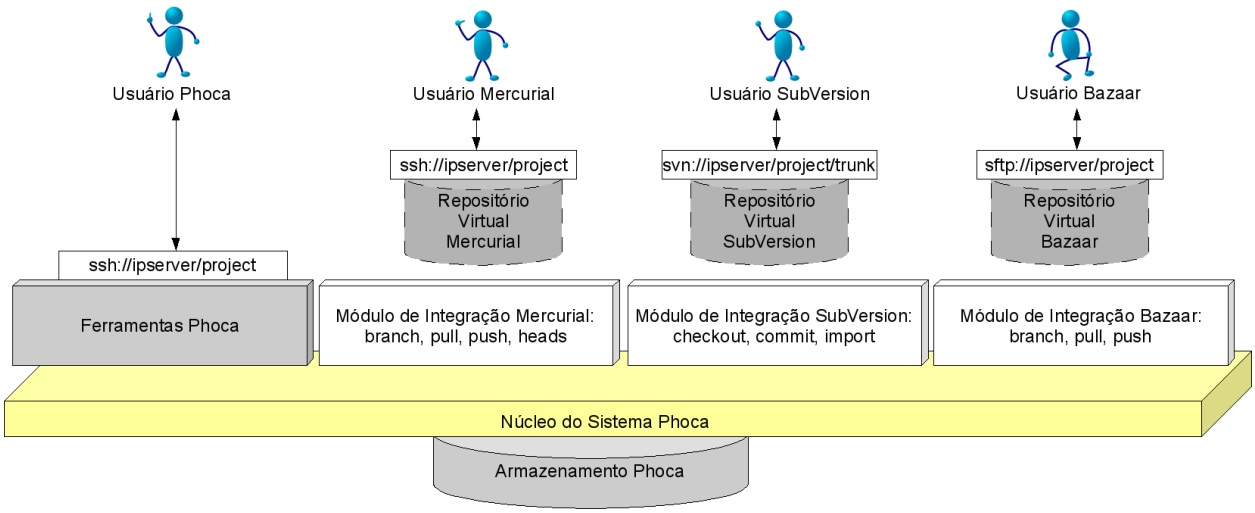

Figura 4.11. Exemplo de como seria a utilização de diversos módulos de integração com ferramentas tradicionais, além do uso do *Phoca*, em paralelo

4.2. Controle de Versões Refinado e Flexível - Estudos de caso

O *Phoca* foi projetado para fornecer controle de versões refinado e flexível para documentos de computador, principalmente artefatos de software. Para a realização dessas atividades são implementados o modelo de documentos, o modelo de versões e o meta-modelo de do-

cumentos. O meta-modelo de documentos é responsável pela granulosidade do controle de versões. É o meta-modelo que dita as regras de como será realizado o mapeamento – portanto, é ele quem configura os tipos de arquivos que podem ser controlados e, também, quão refinado será o mapeamento destes arquivos.

Para validação do sistema, foram realizados alguns estudos de caso,

4.2.1. Estudo de caso - Artefatos em Linguagem Java

Foi realizada, como estudo de caso, a configuração do meta-modelo para linguagem Java. Uma configuração trivial já pode realizar o controle de versões com algum refinamento, incluindo classes as classes dentro de nós de composição adequados.

Para realizar o mapeamento de arquivos de um projeto Java no sistema *Phoca* de forma a se ter a granulosidade até as classes, foi realizada a configuração dos seguintes nós:

- Nó de Documento: mapeado para os arquivos do projeto com extensão *.java*
- Nó de Composição "Cabeçalho": este nó foi configurado como sendo da primeira linha do arquivo até o início de outro nó do mesmo nível (i.e., outro nó cujo pai seja o nó de documento). O nó pai deste nó é o nó de documento, e o nome é estático - "Cabeçalho".
- Nó de Composição "Classe": o início deste tipo de nó foi configurado como sendo o texto "[public]? [abstract]? [final]? class .* [extends]? .* [implements]? .* ", seguido, obrigatoriamente, do marcador de início de bloco "{"; o final do bloco foi definido como sendo o marcador "}". Foi configurada a exceção de empilhamento dos caracteres "{" e "}" dentro do bloco, i.e., toda vez que um caracter "{" for encontrado, deve-se ignorar um "}" – com isso, podem ser definidos blocos com estes caracteres no corpo da classe, desde que haja uma correspondência correta entre eles. Este nó foi configurado como sendo filho do nó de documento, e seu nome é dinâmico, formado pela concatenação de "Classe " com a primeira palavra após "class".

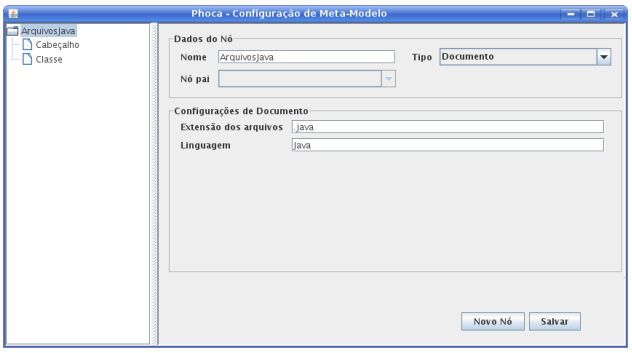

Figura 4.12. Tela de configuração do meta-modelo para documentos Java

A partir desta configuração já é possível importar arquivos Java no sistema *Phoca* e ter as partes do documento separadas em dois nós: um para o que foi chamado de cabeçalho do arquivo e outro com o corpo e definição da classe propriamente dito. Na Figura 4.12 é mostrada a tela do configurador de meta-modelos e na Figura 4.13 é mostrado um documento de exemplo e o mapeamento do conteúdo do arquivo no modelo de documento conforme o meta-modelo definido.

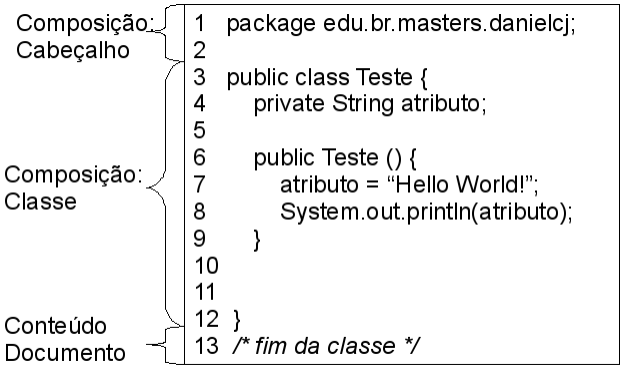

Figura 4.13. Mapeamento de documento Java para o modelo de documentos, conforme meta-modelo configurado

Como pode ser observado na Figura 4.13,o cabeçalho do arquivo foi mapeado corretamente para o nó de composição de cabeçalho – e o texto do cabeçalho, linhas 1 e 2, ficam armazenados num nó de conteúdo que está associado a este nó de composição. A definição e o corpo da classe, que correspondem às linhas 3 a 12 do arquivo, são mapeados n nó de composição chamado "Classe Teste", que é um nó de composição "Classe" no meta-modelo; da mesma forma que acontece com o cabeçalho, seu conteúdo é mapeado num nó de conteúdo associado a este nó de composição. Já a linha 13, que está fora do escopo da classe, é mapeada no nó de conteúdo associado ao documento. Ela não foi associada ao nó "Cabeçalho" porque, segundo a definição do meta-modelo, ele está no mesmo nível que o nó de classe (ambos são filhos do nó "Documento") e seu início ocorre na primeira linha do arquivo e término ocorre assim que outro nó de mesmo nível é declarado. Portanto, só pode existir um nó "Cabeçalho" por documento. Caso a definição do nó "Classe" tivesse sido feita considerando-o como um nó filho do nó "Cabeçalho", a linha 13 estaria associada a um nó de conteúdo do nó "Cabeçalho" – e o nó classe seria inserido no nó "Cabeçalho", o que, neste caso, não faz sentido.

Caso o arquivo utilizado tivesse mais de uma classe declarada, o que é possível de ser feito na linguagem Java, todas as classes seriam mapeadas para nós de composição "Classe".

O próximo passo na edição do meta-modelo é inserir a definição de dois nós importantes: nós de composição "método" e nós de ligação para os casos em que outras classes são importadas. Estes nós foram definidos no meta-modelo da seguinte forma:

- Nó de Composição “Método”: o início deste tipo de nó foi configurado como sendo o texto “(proteted,private,public)? (static)? (abstract)? (final)? (native)? (synchronized)? [.*![ ]] [.*![ ]] \(.*\) .* ”, seguido, obrigatoriamente, do marcador de início de bloco “{”; o final do bloco foi definido como sendo o marcador “}”. Foi configurada a exceção de empilhamento dos caracteres “{” e “}” dentro do bloco. O nó pai deste nó é o nó “Classe”, e seu nome é dinâmico, sendo formado pela concatenação do texto “Método” com o texto do nome do método (primeira palavra antes dos parênteses).

- Nó de Ligação “Import”: a sintaxe deste nó, que não é um nó de bloco, foi definida como sendo “import .*;”. A regra de resolução da ligação do arquivo considera que ela deve ser absoluta e ser feita através da substituição do caracter “.” por “/” no texto capturado por “.*” na expressão regular. Os nós de ligação foram configurados para realizar import manual, ou seja, o usuário deve notificar a intenção de atualizar o nó de ligação quando o arquivo de destino for atualizado.

A partir da definição destas novas regras, o sistema passa a funcionar realizando o mapeamento dos métodos e tentando resolver os imports. É importante observar que nos casos em que é feita a ligação com arquivos da biblioteca Java (e não arquivos do projeto) não é possível resolver a ligação e é feito o comportamento padrão – é criado o nó de ligação para que a estrutura esteja conforme definido, e não são atribuídos os dados de arquivo de destino e versão. Também é interessante notar que nos casos em que é importado um arquivo do projeto, o nó de ligação é salvo na última versão existente do arquivo no momento em que é declarado o *import*, mas se o arquivo de destino for atualizado, as versões recuperadas do repositório com dependências irão considerar a versão original do momento da criação do link, a não ser que seja feita de forma expressa a atualização do nó de ligação (comando *“updatelink idlink”*). Na Figura 4.14 é apresentado um exemplo de um arquivo Java que foi tratado por este meta-modelo.

Este estudo de caso demonstrou o funcionamento do sistema com a linguagem Java, mostrando o funcionamento de funcionalidades como declaração da sintaxe e resolução de ligações.

4.2.2. Estudo de caso - Documentos em Latex

A configuração do meta-modelo para tratar documentos de texto realizados em Latex foi realizada pois possibilita o controle de versões de documentação de software que seja desenvolvida em Latex – e que também faz parte dos artefatos de software que podem ser controlados – e, também, para que pudesse ser feita a utilização do sistema *Phoca* para a escrita deste documento, que foi utilizado para validar o meta-modelo para controle de

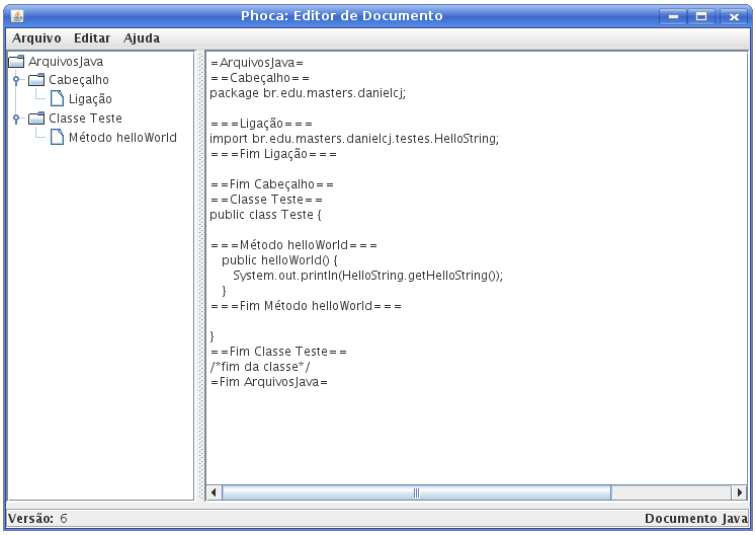

Figura 4.14. Ferramenta de edição de documentos, com um documento Java que utiliza um meta-modelos com definição de cabeçalho, classe, imports e métodos

versões de documentos em Latex. A utilização do controle de versões para Latex permitiu a identificação de algumas falhas no analisador e sua respectiva correção.

A estrutura dos documentos em Latex difere bastante da estrutura das linguagens de programação mais comuns. Isso ocorre porque os elementos da gramática de Latex que descrevem a estrutura geral do texto, como capítulos, seções, e subseções, não possuem delimitadores de fim. Enquanto em Java temos claramente a definição de abertura e fechamento de blocos, através dos caracteres “” e “”, respectivamente, em Latex o fechamento de um determinado bloco é marcado pelo início de outro bloco do mesmo nível. Portanto, o fechamento de uma seção não é declarado – ao ser iniciada outra seção ocorre o fechamento da seção atual. Na Figura 4.15 é ilustrado um exemplo de texto em Latex. Observa-se no exemplo apresentado que a linguagem possui blocos delimitados pelos comandos \begin e \end, e também blocos que devem ser iniciados com comandos específicos e finalizados com a ocorrência de outro comando igual – que delimita o início de um novo bloco no mesmo nível hierárquico – como os capítulos, por exemplo.

```
1  \documentclass {dissertation}
2
3  \title{Um controle de versões refinado e flexível para artefatos de software}}
4  \author{Daniel Cárnio Junqueira}
5
6  \begin{document} %início do documento
7
8  \maketitle
9
10 \chapter{Introdução} %início do capítulo
11
12 Texto da Introdução
13
14 \chapter{Controle de Versões} %início de outro capítulo
15
16 \section{Considerações Finais} %início de seção
17
18 \end{document} %fim de documento
```

Figura 4.15. Exemplo de documento em Latex

Na Figura 4.16 é apresentado o mapeamento desejado entre o texto do documento e modelo de documentos do *Phoca*. Na figura não estão indicados de forma explícita os nós de documento e de conteúdo: o nó de documento é todo o arquivo e os nós de conteúdo correspondem a todo o texto – o texto das menores unidades de composição são mapeados nos nós de conteúdo. Portanto, associado ao nó de composição "preâmbulo" existirá um nó de conteúdo com o texto do preâmbulo – linhas 1, 2, 3, 4 e 5; já o nó de composição "Documento" terá um nó de conteúdo associado, que conterá as linhas 6, 7, 8, 9 e 18; ele também conterá outros dois nós de composição, uma para cada capítulo. O nó de composição do capítulo "Introdução" conterá um nó de conteúdo associado, com todo o texto do capítulo – linhas 10, 11, 12 e 13. O nó de composição do capítulo "Controle de Versões" terá um nó de conteúdo associado, com as linhas 14 e 15, e um nó de composição para a seção "Considerações Finais". O nó de composição da seção "Considerações Finais", por sua vez, terá um nó de conteúdo associado, e este conterá o texto das linhas 16 e 17.

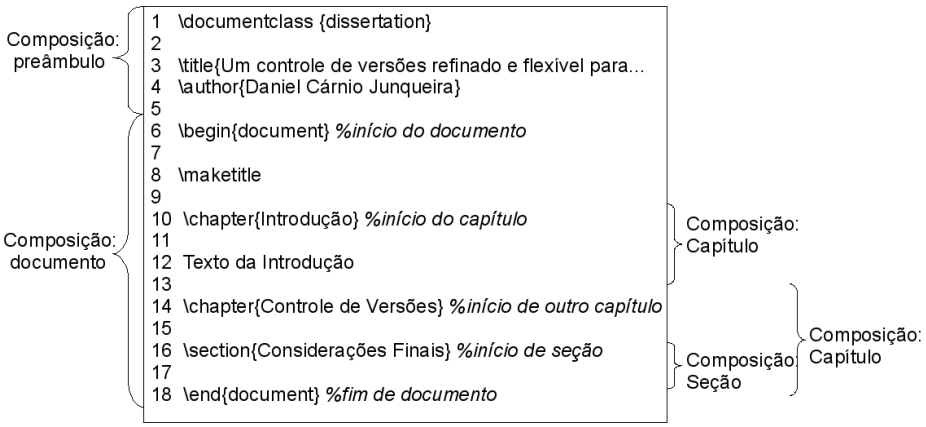

Figura 4.16. Mapeamento do documento Latex para o modelo de documentos

A configuração do meta-modelo para este caso foi realizada com a utilização da ferramenta de configuração de meta-modelo disponibilizada junto com o sistema *Phoca*, e foram criadas as seguintes configurações:

- Nó de Documento: mapeado para os arquivos do projeto com extensão *.tex*
- Nó de Composição "Preâmbulo": este nó foi configurado como sendo da primeira linha do arquivo até o início de outro nó do mesmo nível (i.e., outro nó cujo pai seja o nó de documento). O nó pai deste nó é o nó de documento, e o nome é estático - "Preâmbulo".
- Nó de Composição "Documento": o início deste tipo de nó foi configurado como sendo o texto "\begin{document}" e o final do bloco como sendo o texto "\end{document}". Este nó foi configurado como sendo filho do nó de documento, e seu nome é estático: "Documento".
- Nó de Composição "Capítulo": o início deste tipo de nó foi configurado como sendo o texto "\chapter{.*}". O nó pai deste nó é o "Documento", e seu nome é dinâmico, sendo formado pela concatenação do texto "Capítulo" com o texto entre as chaves do bloco de definição de início. Este bloco foi declarado como sendo com o fim igual ao início de outro nó de mesmo nível.
- Nó de Composição "Seção": o início deste tipo de nó foi configurado como sendo o texto "\section{.*}". O nó pai deste nó é o "Capítulo", e seu nome é dinâmico, sendo formado pela concatenação do texto "Seção" com o texto entre as chaves do bloco de definição de início. Este bloco foi declarado como sendo com o fim igual ao início de outro nó de mesmo nível.

A partir da configuração do meta-modelo de documento com as regras listadas foi possível importar documentos .tex no sistema *Phoca*. Entretanto, apenas com essas regras não era possível utilizar o comando *include* do Latex, que permite a inclusão de arquivos externos. Para contemplar tal regra, foi criado um nó de ligação mapeado para o texto *include*{.*}. A utilização deste comando no texto em questão ocorria apenas dentro do documento, e fora de capítulos ou seções, pois o projeto foi organizado de forma que existia um documento raiz que incluía todos os capítulos, e cada capítulo foi feito num arquivo individual. A utilização da ferramenta foi feita principalmente via linha de comando. Na Figura 4.17 é mostrada a tela da ferramenta de configuração de meta-modelo, utilizada para configurar o meta-modelo.

4.2.3. Proposta - Controle de Versões para Aplicações Cientes de Contexto

Como último estudo de caso será descrita uma proposta que foi realizada de adaptação do sistema para a captura de dados de contexto das aplicações cientes de contexto. Esta proposta ilustra, principalmente, as características de extensão do *Phoca*.

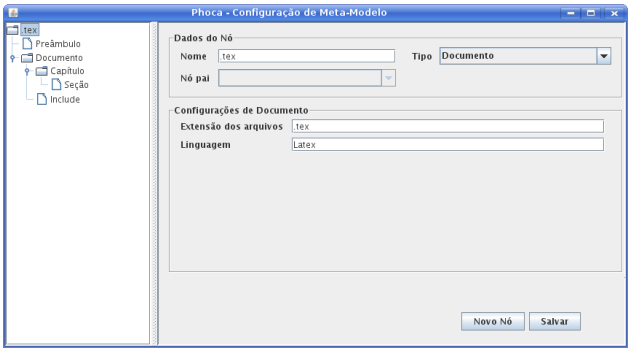

Figura 4.17. Tela da ferramenta de configuração de meta-modelo, exibindo a estrutura dos nós criados para Latex

A proposta foi projetada após a identificação da possibilidade de contribuição para a área de aplicações cientes de contexto em Computação Ubíqua. Aplicações cientes de contexto são aquelas que, de alguma forma, alteram seu comportamento com base em informações de contexto disponíveis – sejam elas informações do ambiente, de algum dispositivo onde a aplicação esteja rodando, de posição do usuário, entre outras (HARTER et al., 1999). Um dos problemas críticos neste tipo de aplicação é a grande quantidade de alterações importantes nas informações do contexto e no "modelo do contexto", ou seja, inserção ou remoção de sensores, ou mesmo alteração na forma como os dados são capturados (ABOWD, 1999).

Muitas vezes, as aplicações declaram um formato XML com a utilização de XML Schema, e passam a armazenar os dados em formato XML. Assim, o XML Schema passa a ter a definição do modelo das informações que podem ser capturadas e o arquivo XML resultante possui as informações do contexto. Entretanto, esta abordagem geralmente é implementada de forma que os dados históricos não são armazenados, tendo registrado apenas a última informação de como o contexto estava configurado e quais foram as informações capturadas (MANZATO; GOULARTE, 2005).

Com o objetivo de prover uma solução para o desenvolvimento de aplicações cientes de contexto em computação ubíqua, foi projetada uma proposta utilizando o sistema *Phoca*, que consiste em implementar um módulo que realize a configuração automática do meta-modelo a partir de informações disponibilizadas no XML Schema que define o modelo do contexto. Esta proposta, que não chegou a ser implementada, foi projetada utilizando apenas as APIs disponibilizadas pelo sistema *Phoca* para projetar a forma como os dados seriam armazenados e como o configurador de meta-modelo deveria se comportar. A única alteração que seria necessária seria a implementação do configurador de meta-modelo que fosse capaz de ler as informações do XML Schema e, com base nestas informações, invocar a API do Parser do sistema *Phoca*. O armazenamento e recuperação de dados poderiam ser feitos diretamente através da API de dados. Na Figura 4.18 é apresentado um esquema da abordagem proposta.

A vantagem desta proposta em relação ao armazenamento em bancos de dados é a flexibilidade do modelo de contexto existente. As abordagens baseadas em bancos de dados

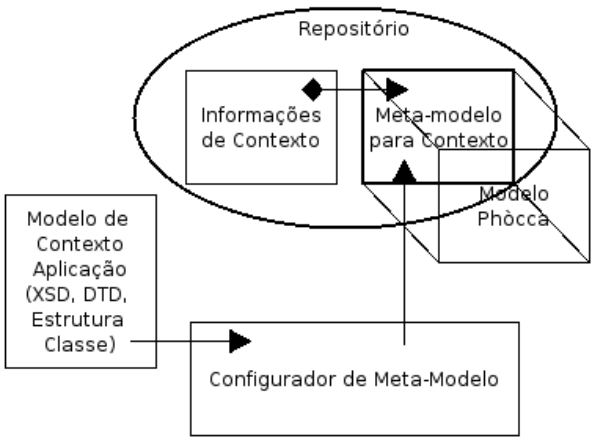

Figura 4.18. Esquema da abordagem proposta para controle de versões refinado e flexível para aplicações cientes de contexto

requerem alteração de tabelas quando novas informações de contexto precisarem ser capturadas e, quando determinada informação não for mais capturada, a estrutura da tabela não poderá ser alterada – por exemplo, removendo uma coluna – devido aos dados históricos que estarão armazenados. No sistema proposto, é possível armazenar o histórico tanto das informações quanto do modelo do contexto em que estas foram capturadas. Vale ressaltar que o *Phoca* realiza controle de versões do meta-modelo de documentos, associando cada versão do arquivo capturada à versão do meta-modelo em que o mesmo foi capturado.

4.3. Utilização do *Phoca* em ambiente real

Nesta seção serão apresentadas algumas informações em relação à utilização do *Phoca* em ambiente real.

Os seguintes passos devem ser realizados para utilização do *Phoca* num ambiente real: implantação do servidor, instalação da aplicação cliente, configuração da primeira versão do meta-modelo, documentação de políticas de controle de versões, configuração de permissões, treinamento de equipe, utilização e, por fim, como atividade opcional, integração do sistema no processo de GCS do ambiente em que é utilizado. Na Figura 4.19 são ilustrados os passos que devem ser realizados para utilização do *Phoca* em ambiente real.

Cada uma destas fases será apresentada a seguir.

Implantação do servidor

O *Phoca* utiliza, por padrão, o sistema de controle de versões distribuídas Mercurial. Sua implantação no servidor, no formato como foi criado, depende da instalação do sistema

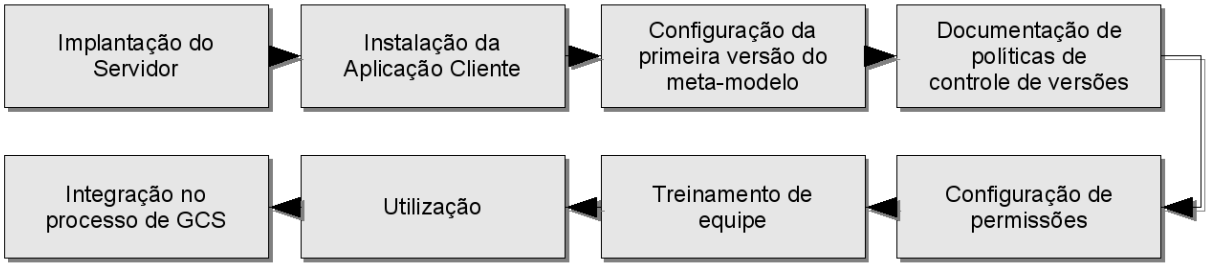

Figura 4.19. Passos a serem realizados num ambiente real de utilização do *Phoca*

Mercurial e do banco de dados HSQLDB. Na verdade, o Sistema Gerenciador de Banco de Dados (SGBD) utilizado no servidor não precisa ser necessariamente o HSQLDB e, por questões de performance e controle de acesso concorrente ao servidor, é indicada, inclusive, a utilização de outro SGBD, como o PostgreSQL.

A configuração do banco de dados pode ser feita a partir de um *script* distribuído juntamente com o *Phoca*. A configuração dos usuários do banco de dados e do repositório central ditam, basicamente, quais os usuários autorizados a acessar o sistema. Entretanto, configurações adicionais podem ser feitas para as operações de controle de acesso, e essas operações são feitas de duas formas:

- Aplicações cliente: a partir das aplicações cliente é possível realizar operações de controle de acesso sobre os dados e operações do servidor; no banco de dados central pode-se configurar os privilégios das contas, e pelo menos uma conta de administrador precisa ser criada para correto funcionamento do sistema
- Configuração manual via banco de dados: as permissões dos usuários podem ser realizadas diretamente no banco de dados central. Como apresentado anteriormente, ele possi meta-dados

O Mercurial – ou outro sistema de controle de versões que estiver sendo utilizado – deve ser capaz de oferecer acesso remoto para que os clientes possam utilizar. Além disso, o banco de dados central também precisa fornecer acesso remoto para as aplicações clientes.

A lógica do *Phoca* ainda está concentrada nas APIs utilizadas no cliente da aplicação – o módulo de sincronização é responsável pelas tarefas de comunicação com repositórios e bancos de dados locais e remotos. O servidor oferece apenas um repositório de controle de versões e um banco de dados.

Caso seja julgado importante após testes em ambiente de produção, futuras versões do *Phoca* devem conter classes da aplicação do lado do servidor.

Instalação da aplicação cliente

Todos os usuários do sistema precisam realizar o acesso ao servidor a partir da aplicação cliente. Embora o repositório central utilize um repositório de controle de versões, o acesso direto ao repositório não fornece dados úteis sobre o conteúdo que está sob controle de versões.

Uma recomendação é realizar a instalação da aplicação cliente diretamente no servidor, para que possa, então, ser utilizada para fins de configuração. A aplicação cliente permite acesso às APIs de configurações, criação de novos usuários, entre outras. Na verdade, as operações de criação de usuários podem ser realizadas diretamente no banco de dados e repositório, e em futuras versões do *Phoca* está prevista a criação de uma interface administrativa via web que deve permitir essas configurações.

Configuração da primeira versão do meta-modelo

O banco de dados do servidor deve armazenar, entre outras informações, as configurações do meta-modelo utilizado no sistema – módulo de Modelo. Para o correto funcionamento do sistema, a implantação deve ser feita já com a criação da primeira versão a ser utilizada de meta-modelo.

É essa informação que permitirá ao sistema realizar o controle de versões refinado.

Para configurar a primeira versão do meta-modelo, é necessário que tenha sido, primeiramente, criado o servidor e o usuário com privilégios de administrador. O script distribuído juntamente com o sistema já inclui a criação de um usuário administrador. A partir de então, deve-se realizar o *login* com o usuário criado, e realizar a configuração da primeira versão do meta-modelo.

A configuração do meta-modelo é parte essencial para o funcionamento do *Phoca*. É possível criar um meta-modelo genérico, no qual o nível de detalhamento do controle de versões não seja tão refinado e faça com que o sistema tenha um comportamento padrão para trabalhar com detalhamento de arquivos.

Documentação de políticas de controle de versões

Para a correta utilização do sistema, é necessário que sejam criadas as políticas de controle de versões. A definição de políticas é um dos passos estratégicos para a realização de GCS com sucesso, e, no caso de controle de versões, é nesta fase que devem ser documentadas todas as regras do sistema de controle de versões.

As políticas de controle de versões incluem políticas de acesso, procedimentos gerais de trabalho, momentos em que devem ser realizadas as operações de *commit*, como devem ser

realizadas as edições, entre outras. Alguns modelos de maturidade exigem que as empresas tenham documentadas regras claras para suas atividades de GCS para que se enquadrem mesmo nos níveis mais baixos de maturidade de processo (WEBER et al., 2004; CMMI Product Team, 2002).

O *Phoca* é distribuído com documentação que auxilia na criação das políticas de controle de versões.

Configuração de permissões

Após a configuração do meta-modelo, é necessário que sejam configurados usuários para utilizar o sistema. Os usuários do sistema podem ter diversos tipos de permissões: as tradicionais permissões de leitura e escrita, que podem ser utilizadas até a menor unidade de controle de versões configurada; permissões referentes às configurações do sistema, que são a configuração do meta-modelo, permissão de criação de novos usuários, de alteração de permissões de usuários.

Nesta versão do sistema, as permissões podem ser configuradas pela aplicação cliente ou diretamente no banco de dados, mas ainda não existe o conceito de "papéis" dos usuários. Esta é uma nova funcionalidade que está sendo planejada para versões futuras.

Treinamento de equipe

A fase de treinamento de equipe é incluída na documentação do projeto descrevendo a implantação por ser essencial à boa utilização do sistema. Muitas empresas brasileiras, de todos os portes, não realizam qualquer tipo de atividade de controle de versões (REIS, 2003). Portanto, é necessário, para a adoção do *Phoca*, que sejam realizadas atividades de treinamento programado.

Estas atividades devem ser realizadas para divulgar os aspectos técnicos de utilização do sistema e, também, as políticas de controle de versões criadas pela empresa. As políticas de controle de versões são específicas de cada empresa, e é definida de acordo com o processo de desenvolvimento adotado pela empresa.

Utilização

Após a realização das atividades listadas anteriormente, o sistema implantado pode ser utilizado em ambientes reais. A utilizaçao envolve a criação de repositórios adicionais necessários, importação de arquivos de projeto, acompanhamento da política de controle de versões, e realização das atividades definidas na política.

Durante a utilização do sistema, as alterações realizadas em permissões e no meta-modelo são registradas automaticamente, de forma que é possível acompanhar inclusive este histórico.

Integração no processo de GCS

Uma das maiores vantagens em se utilizar um sistema de controle de versões refinado e flexível é a possibilidade de integrar no processo de GCS. Com a integração da ferramenta de controle de versões no processo de GCS, pode-se permitir, por exemplo, que o comitê responsável pela aprovação de solicitações de mudanças tenha dados mais detalhados sobre a mudança que irá ocorrer, e qual seu impacto previsto, pois é possível visualizar, no caso do *Phoca*, as dependências existentes entre os diversos componentes dos sistemas – desde que o meta-modelo tenha sido configurado de forma a permitir este rastreamento.

Outras atividades do processo de GCS além do controle de mudanças também poderiam se beneficiar da integração com o *Phoca*. Por exemplo, se fosse totalmente integrado, o *Phoca* poderia fornecer informações detalhadas sobre as versões enviadas para análise de impacto, testada e implantada em produção, facilitando a atividade de auditoria da configuração.

Portanto, é possível realizar a integração do *Phoca* no processo de GCS. Juntamente com a distribuição do sistema existe uma documentação com sugestões de integração, e este documento, que deverá ser mantido de forma colaborativa, tende a fornecer novas formas de integração. Embora seja possível realizar a integração de sistemas que não realizam o controle de versões refinado, como o ClearCase (BELLAGIO; MILLIGAN, 2005), um sistema de controle de versões refinado e flexível permite uma melhor integração em diferentes fases do processo de GCS.

4.4. Considerações Finais

Os sistemas de controle de versões tradicionais realizam os procedimentos de controle de versões utilizando como menor unidade de versionamento os arquivos de computador, e realizam o cálculo das diferenças através do cálculo de diferença de linhas destes arquivos. Entretanto, este modelo muitas vezes não é capaz de prover detalhes sobre a evolução dos sistemas, não fornecendo mecanismos para rastreamento de ligações entre arquivos e não fornecendo maiores informações sobre a estrutura dos arquivos, que, muitas vezes, são utilizadas pelos desenvolvedores de software. Algumas soluções foram propostas para oferecer suporte ao controle de versões refinado para arquivos de computador. Entretanto, estas soluções não provêm uma forma eficaz e prática de configurar o mapeamento do nível de detalhamento que deve ser utilizado para realizar o controle de versões dos arquivos, sendo

que muitas vezes este mapeamento é completamente estático e direcionado para linguagens de programação específicas.

Neste capítulo foi apresentado o sistema *Phoca*, cujo objetivo é fornecer suporte para as atividades de controle de versões refinado e de forma flexível, permitindo aos usuários a definição das regras que devem ser utilizadas para configurar o nível de detalhamento do controle de versões desejado. Além disso, o sistema apresentado foi desenvolvido para ser uma solução que possa ser adaptada em outras ferramentas, permitindo, assim, que seja minimizado o impacto da adoção de uma ferramenta deste tipo em ambientes de desenvolvimento de software.

A utilização do processo Iconix para o desenvolvimento do sistema permitiu aos envolvidos uma grande interação possibilitou que durante as iterações do processo fossem percebidas e corrigidas falhas do sistema, como a falha relacionada à falta de regras que permitissem o acompanhamento de documentos Latex, que, numa próxima iteração, foi corrigido. Todos os artefatos gerados durante o desenvolvimento do sistema foram registrados e serão disponibilizados para continuidade do projeto como um projeto de código aberto que siga o mesmo processo.

Além do sistema desenvolvido – que é licenciado sob a licença Apache versão 2 e pode ser estendido por indivíduos ou empresas –, foi também desenvolvida documentação relativa à utilização e extensão do sistema, que é disponibilizada juntamente com o sistema em formato HTML.

Para que o sistema *Phoca* seja efetivamente utilizado em ambientes reais, é necessário que algumas ferramentas de apoio sejam criadas, a fim de fornecerem maior facilidade de implantação e utilização do mesmo. Além disso, é necessário que ele seja integrado às principais IDEs utilizadas. Planeja-se fazer projetos de código aberto para o desenvolvimento de um módulo para a IDE NetBeans e um *plugin* para a IDE Eclipse, o que facilitaria a divulgação e utilização do sistema em ambientes reais.

No próximo capítulo, serão apresentadas as conclusões do trabalho e as propostas de trabalhos futuros.

Capítulo

5

Conclusões

Em ambientes de desenvolvimento de software, as atividades de controle de versões são utilizadas para controlar a evolução dos sistemas de computador. Tais atividades, se bem realizadas, dão aos desenvolvedores a capacidade de acompanhar as alterações realizadas nos sistemas de forma detalhada, e, através de atividades como análise dos logs, é possível detectar as versões do sistema nas quais existe maior probabilidade de um bug ter sido introduzido, por exemplo.

Apesar das ferramentas de controle de versões existirem desde a década de 1970, época em que surgiu o SCCS, poucas alterações significativas foram realizadas. O RCS introduziu os primeiros conceitos de edição de arquivos concorrente, através da introdução de mecanismos de resolução de conflitos, e o CVS, com a introdução de repositórios centrais, permitiu aos desenvolvedores a utilização de um sistema com repositório central, que foi inspirado no RCS mas cujas capacidades de desenvolvimento concorrente foram muito evoluídas. Entretanto, o CVS, lançado em 1986, ficou por muito tempo sendo o sistema de controle de versões mais utilizado, inclusive no século XXI.

Entretanto, desde o lançamento do CVS a aplicação de sistemas computacionais evoluíram drasticamente, e atualmente é possível observar a utilização de softwares de computador para a realização de atividades cotidianas comum, como falar ao telefone e realizar compras. Com esta popularização da computação, também ocorreu um significativo aumento na quantidade e complexidade dos sistemas sendo desenvolvidos, e foram criados novos paradigmas de programação, como a programação orientada a objetos e programação de sistemas web.

Com a evolução e conseqüente aumento de complexidade dos sistemas, melhores técnicas de controle de versões dos sistemas fazem-se necessárias. Desde o início da década de 1990,

algumas abordagens foram propostas para oferecer controle de versões refinado para artefatos de software. Entretanto, essas abordagens são, de certa forma, rígidas, a partir do momento em que não permitem uma fácil definição da estrutura dos documentos que serão controlados, e, também, por não poderem ser integradas a outras ferramentas de desenvolvimento de software e gerenciamento de configuração de software.

Neste trabalho, foram estudadas as principais ferramentas de controle de versões e de controle de versões refinado para artefatos de software, e, a partir dos resultados obtidos, foi proposto o sistema *Phoca*, cujo objetivo era o de fornecer um controle de versões refinado e flexível para artefatos de software. Assim, as principais metas planejadas neste trabalho científico foram alcançadas e, conforme mostram as próximas seções, podem-se destacar contribuições e trabalhos futuros.

5.1. Principais contribuições

A primeira contribuição deste trabalho é a própria revisão do estado da arte relativo a sistemas e conceitos de controle de versões.

Este trabalho, ao apresentar um sistema que permite o controle de versões refinado e flexível para artefatos de software, visa a contribuir com a melhoria das atividades de controle de versões, que podem ser executadas de forma refinada, oferecendo melhor rastreamento e acompanhamento das mudanças realizadas nos softwares. Ao permitir que o sistema seja integrado com ferramentas de desenvolvimento de software já utilizadas nos ambientes de desenvolvimento, contribui com uma real adoção do sistema e com impactos minimizados.

Além disso, contribui com a definição de um modelo de documentos e sua implementação, com destaque para os nós de ligação entre documentos diferentes. As regras claras das técnicas de análise utilizadas e a distribuição do sistema sob uma licença de código aberto são também contribuições deste trabalho.

Por fim, ao disponibilizar documentação com procedimentos para implantação do sistema e dicas sobre a criação de processos de GCS, este trabalho contribui não apenas na área acadêmica, seu principal foco, mas também nos ambientes corporativos, principalmente empresas de pequeno e médio porte.

O presente trabalho possui limitações que proporcionam continuidades nas pesquisas iniciadas. Na próxima seção são apresentados os trabalhos futuros que foram identificados.

5.2. Trabalhos Futuros

Com a conclusão do presente trabalho, foram também identificadas algumas possibilidades de trabalhos futuros.

A realização de um experimento de avaliação com usuários finais para identificar pontos de melhoria na interação com o usuário e avaliar, formalmente, as reais vantagens implementadas pelo sistema nas atividades de controle de versões de artefatos de software é indicada como um trabalho futuro, que além de validar a utilização do presente trabalho com os usuários, pode fornecer informações úteis para seu aperfeiçoamento.

Outro possível trabalho futuro refere-se à criação de um assistente automático para a configuração do meta-modelo. Este assistente deve ser capaz de receber como entrada um arquivo de exemplo do usuário, ou o endereço de seu projeto no disco, e fornecer então sugestões de configuração de meta-modelo a partir do conteúdo dos arquivos existentes.

Além disso, precisam ser implementadas ferramentas para auxílio a implantação do servidor do *Phoca*, além de monitoramento e acompanhamento e atualização da ferramenta, pois a forma como ele deve ser implantado atualmente é bastante suscetível a falhas e pode comprometer sua utilização. Em relação a ferramentas, também devem ser implementados os módulos para utilização do sistema de forma integrada nas IDEs Eclipse e NetBeans.

Por fim, a criação de um módulo de relatórios gerenciais detalhados para o sistema e a criação de web services para recuperação de dados relacionados ao projeto seriam importantes para a melhor integração do *Phoca*com as atividades de GCS.

Referências

ABOWD, G. D. Software engineering issues for ubiquitous computing. In: *Software Engineering, 1999. Proceedings of the 1999 International Conference on*. Los Angeles, Estados Unidos: IEEE, 1999. p. 75–84.

AMBRIOLA, V.; BENDIX, L.; CIANCARINI, P. The evolution of configuration management and version control. *Software Engineering Journal*, v. 5, n. 6, p. 303–310, 1990. Disponível em: `http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=61744`.

ASKLUND, U. *Configuration Management for Distributed Development in an Integrated Environment*. Tese (Doutorado) — Lund University, Lund, Suécia, 2002.

ASKLUND, U. et al. The unified extensional versioning model. In: *SCM-9: Proceedings of the 9th International Symposium on System Configuration Management*. London, UK: Springer-Verlag, 1999. p. 100–122. ISBN 3-540-66484-X.

BELLAGIO, D. E.; MILLIGAN, T. J. *Software Configuration Management Strategies and Rational ClearCase: A Practical Introduction*. 2. ed. [S.l.]: IBM Press, 2005. ISBN 0321200195.

BENDIX, L. et al. Coed - a tool for versioning of hierarchical documents. In: *ECOOP '98: Proceedings of the SCM-8 Symposium on System Configuration Management*. London, UK: Springer-Verlag, 1998. p. 174–187. ISBN 3-540-64733-3.

BERLACK, H. R. *Software Configuration Management*. 2. ed. Estados Unidos: Wiley, 1991. (Wiley series in Software Engineering Practice).

BLACKWELL, J. et al. *Introduction to Bzr*. [S.l.], 2006. Disponível em http://bazaar-vcs.org/IntroductionToBzr.

BOLINGER, D.; BRONSON, T. *Applying RCS and SCCS*. 1. ed. EUA: O'Reilly, 1995.

CEDERQVIST et al. *Version Managment with CVS*. EUA, 2007. Disponível online em: http://ximbiot.com/cvs/manual/.

CHU-CARROLL, M. C.; SPRENKLE, S. Coven: brewing better collaboration through software configuration management. *SIGSOFT Softw. Eng. Notes*, ACM, New York, NY, USA, v. 25, n. 6, p. 88–97, 2000. ISSN 0163-5948.

CHU-CARROLL, M. C.; WRIGHT, J.; SHIELDS, D. Supporting aggregation in fine grained software configuration management. *SIGSOFT Softw. Eng. Notes*, ACM, New York, NY, USA, v. 27, n. 6, p. 99–108, 2002. ISSN 0163-5948.

CMMI Product Team. *Capability Maturity Model Integration (CMMI) Version 1.1*. Pittsburgh: SEI-CMU, 2002. Disponível on-line em: http://www.sei.cmu.edu/pub/documents/02.reports/pdf/02tr012.pdf.

COLLINS-SUSSMAN, B.; FITZPATRICK, B. W.; PILATO, C. M. *Version Control with Subversion*. Revisão 10945. EUA: O'Reilly, 2007. Livro online, disponível em http://svnbook.red-bean.com/svnbook/book.html.

CONRADI, R.; WESTFECHTEL, B. Version models for software configuration management. *ACM Comput. Surv.*, ACM, New York, NY, USA, v. 30, n. 2, p. 232–282, 1998. ISSN 0360-0300.

ELLIOTT, J. *Hibernate: A Developer's Notebook*. Estados Unidos: O'Reilly Media, Inc., 2004. ISBN 0596006969.

ESTUBLIER, J. et al. Impact of software engineering research on the practice of software configuration management. *ACM Trans. Softw. Eng. Methodol.*, ACM Press, New York, NY, USA, v. 14, n. 4, p. 383–430, October 2005. Disponível em: http://dx.doi.org/10.1145/1101815.1101817.

GAMMA, E. et al. *Design Patterns*. Addison-Wesley Professional, 1995. Hardcover. ISBN 0201633612. Disponível em: http://www.amazon.ca/exec/obidos/redirect?tag=citeulike09-20{\&}path=ASIN/0201633612.

GARZIK, J. *Kernel Hackers' Guide to git*. [S.l.], September 2005. Disponível em http://linux.yyz.us/git-howto.html. Disponível em: http://linux.yyz.us/git-howto.html.

GLASSER, A. L. The evolution of a source code control system. In: *Proceedings of the software quality assurance workshop on Functional and performance issues*. [S.l.: s.n.], 1978. p. 122–125.

GONG, P.; GORTON, I.; FENG, D. D. Dynamic adapter generation for data integration middleware. In: *SEM '05: Proceedings of the 5th international workshop on Software engineering and middleware*. New York, NY, USA: ACM, 2005. p. 9–16. ISBN 1-59593-204-4.

HARTER, A. et al. The anatomy of a context-aware application. In: *MobiCom '99: Proceedings of the 5th annual ACM/IEEE international conference on Mobile computing and networking*. New York, NY, USA: ACM, 1999. p. 59–68. ISBN 1-58113-142-9.

HASS, A. M. J. *Configuration Management Principles and Practice*. 1. ed. [S.l.]: Addisson Wesley, 2003. (The Agile Software Development Series).

IEEE Std 610. *IEEE standard computer dictionary, A compilation of IEEE standard computer glossaries*. New York, 1991.

JAZAYERI, M. Some trends in web application development. In: *FOSE '07: 2007 Future of Software Engineering*. Washington, DC, USA: IEEE Computer Society, 2007. p. 199–213. ISBN 0-7695-2829-5.

JUNQUEIRA, D. C.; FORTES, R. P. de M. Versionweb: A tool for open source software development support. In: *LA-WEBMEDIA '04: Proceedings of the WebMedia & LA-Web 2004 Joint Conference 10th Brazilian Symposium on Multimedia and the Web 2nd Latin American Web Congress*. Washington, DC, USA: IEEE Computer Society, 2004. p. 65–67. ISBN 0-7695-2237-8.

KEYES, J. *Software Configuration Management*. 1. ed. New York, NY, USA: Auerbach, 2004.

KNUDSEN, J. L. et al. *Object Oriented Software Development Environments: The Mjolner Approach*. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 1994. ISBN 0130092916.

KROAH-HARTMAN, G. Kernel korner: the kernel hacker's guide to source code control. *Linux J.*, Specialized Systems Consultants, Inc., Seattle, WA, USA, v. 2002, n. 101, p. 11, 2002. ISSN 1075-3583.

LEMOS, O. A. L. et al. Using aspect-oriented php to implement crosscutting concerns in a collaborative web system. In: *SIGDOC '06: Proceedings of the 24th annual ACM international conference on Design of communication*. New York, NY, USA: ACM, 2006. p. 134–141. ISBN 1-59593-523-1.

LERNER, R. M. Installing and customizing mediawiki. *Linux J.*, Specialized Systems Consultants, Inc., Seattle, WA, USA, v. 2006, n. 144, p. 3, 2006. ISSN 1075-3583.

LIN, Y.-J.; REISS, S. P. Configuration management with logical structures. In: *ICSE '96: Proceedings of the 18th international conference on Software engineering*. Washington, DC, USA: IEEE Computer Society, 1996. p. 298–307. ISBN 0-8186-7246-3.

LINDSAY, P.; LIU, Y.; OWENTRAYNOR. A generic model for fine grained configuration management including version control and traceability. In: *Software Engineering Conference, 1997. Proceedings. 1997 Australian*. Sydney, NSW: IEEE, 1997.

MAGNUSSON, B.; ASKLUND, U. Fine grained version control of configurations in coop/orm. In: *ICSE '96: Proceedings of the SCM-6 Workshop on System Configuration Management*. London, UK: Springer-Verlag, 1996. p. 31–48. ISBN 3-540-61964-X.

MAGNUSSON, B.; ASKLUND, U.; MINöR, S. Fine-grained revision control for collaborative software development. In: *SIGSOFT '93: Proceedings of the 1st ACM SIGSOFT symposium on Foundations of software engineering*. New York, NY, USA: ACM, 1993. p. 33–41. ISBN 0-89791-625-5.

MANZATO, M. G.; GOULARTE, R. Live video adaptation: a context-aware approach. In: *WebMedia '05: Proceedings of the 11th Brazilian Symposium on Multimedia and the web*. New York, NY, USA: ACM, 2005. p. 1–8.

MERCURIAL. *Site oficial do projeto Mercurial*. 2007. Disponível em `http://www.selenic.com/mercurial/wiki/`.

MICALLEF, J.; CLEMM, G. The asgard system: Activity-based configuration management. In: *ICSE '96: Proceedings of the SCM-6 Workshop on System Configuration Management*. London, UK: Springer-Verlag, 1996. p. 175–186. ISBN 3-540-61964-X.

MOFFITT, N. Revision control with arch: introduction to arch. *Linux J.*, Specialized Systems Consultants, Inc., Seattle, WA, USA, v. 2004, n. 127, p. 6, 2004. ISSN 1075-3583.

MOREIRA, D.; SOARES, M.; FORTES, R. P. M. Versionweb: A tool for helping web page version control. *Linux Journal*, 2002.

NGUYEN, T. N.; MUNSON, E. V.; BOYLAND, J. T. The molhado hypertext versioning system. In: *HYPERTEXT '04: Proceedings of the fifteenth ACM conference on Hypertext and hypermedia*. New York, NY, USA: ACM, 2004. p. 185–194. ISBN 1-58113-848-2.

NGUYEN, T. N. et al. Flexible fine-grained version control for software documents. In: *Proceedings of Software Engineering Conference, 2004. 11th Asia-Pacific*. [S.l.]: IEEE, 2004.

NGUYEN, T. N.; MUNSON, E. V.; THAO, C. Fine-grained, structures configuration management for web projects. In: *Proceedings of WWW2004*. New York, NY: ACM Press, 2004. p. 433–442.

RCS. *GNU Revision Control System*. 2007. Site oficial do projeto: `http://www.gnu.org/software/rcs/`. Visitado em outubro/2007.

REIS, C. R. *Caracterização de um Processo de Software para Projetos de Software Livre*. Dissertação (Mestrado) — ICMC, USP, São Carlos, Brasil, Fevereiro 2003.

RIEHLE, D. How and why wikipedia works: an interview with angela beesley, elisabeth bauer, and kizu naoko. In: *WikiSym '06: Proceedings of the 2006 international symposium on Wikis*. New York, NY, USA: ACM, 2006. p. 3–8. ISBN 1-59593-413-8.

ROCHKIND, M. J. The source code control system. *IEEE Trans. Software Eng.*, v. 1, n. 4, p. 364–370, 1975.

ROSENBERG, D.; SCOT, K. *Applying Use Case Driven Object Modelling with UML: An Annotated e-Commerce Example*. 1. ed. Estados Unidos: Addison Wesley, 2001.

ROSENBERG, D.; STEPHENS, M.; COLLINS-COPE, M. *Agile Development with Iconix Process - People, Process and Pragmatism*. 1. ed. Estados Unidos: Apress, 2005.

SILVA, S. R. Q. M. da. *Controle de Versões - um apoio à edição colaborativa na web*. Dissertação (Mestrado) — Universidade de São Paulo, São Carlos, 2005.

SOARES, M. D.; FORTES, R. P. de M.; MOREIRA, D. A. Versionweb: a tool for helping web pages version control. In: *International Conference on Internet Multimedia Systems and Applications*. Las Vegas, EUA: [s.n.], 2000. p. 275–280.

THOMAS, D.; JOHNSON, K. Orwell - a configuration management system for team programming. In: *OOPSLA '88: Conference proceedings on Object-oriented programming systems, languages and applications*. New York, NY, USA: ACM, 1988. p. 135–141. ISBN 0-89791-284-5.

TICHY, W. F. Design, implementation, and evaluation of a revision control system. In: *ICSE '82: Proceedings of the 6th international conference on Software engineering*. Los Alamitos, CA, USA: IEEE Computer Society Press, 1982. p. 58–67.

TICHY, W. F. RCS — a system for version control. *Software — Practice and Experience*, v. 15, n. 7, p. 637–654, 1985.

WEBER, K. et al. Modelo de referência para melhoria de processo de software: uma abordagem brasileira. In: SOLAR, M.; FERNáNDEZ-BACA, D.; CUADROS-VARGAS, E. (Ed.).

30ma Conferencia Latinoamericana de Informática (CLEI2004). [S.l.], 2004. p. 461–476. ISBN 9972-9876-2-0.

Glossário

Árvore Termo utilizado, pela comunidade de software livre, para designar um repositório. A árvore pode ser criada vazia, a partir de arquivos cujas versões não são controladas, ou duplicando, total ou parcialmente, um outro repositório. Neste caso, mantém-se a relação entre os repositórios (como se fosse uma nova ramificação do repositório origem), mas em repositórios separados e independentes. As alterações realizadas no novo repositório não afetam o repositório do qual ele se originou. Posteriormente, é possível enviar as alterações feitas de volta para a árvore original.

Artefato de Software
Item que compõe um software. Relaciona-se a arquivos de código fonte, bibliotecas e documentação.

AT&T *American Telephone and Telegraph*, companhia americana de telecomunicações.

Bell Telephone Laboratories
Laboratório formado a partir da consolidação de laboratórios de pesquisa e engenharia da AT&T. Formalmente conhecido por AT&T Bell Telephone Laboratories

Bug Erro encontrado no software. O termo "bug" também é utilizado na ferramenta Bugzilla para nomear pedidos de alteração ou relato de falhas ou erros de um software.

Código fonte
Conjunto de arquivos em formato texto, escritos em determinada linguagem de programação, que descrevem um programa de computador

Construção de Aplicação
Do inglês application build, significa compilar uma aplicação de forma automatizada, seguindo scripts para realizar os passos de compilação com outras tarefas associadas. A tarefa de construção de aplicação utiliza informações de

dependências entre as partes de um sistema para saber quais partes devem ser compiladas primeiro

Controle de versões refinado
Controle de versões realizado de forma mais detalhada que o tradicional. O controle de versões tradicionais utiliza conceitos de arquivos e diretórios, enquanto controle de versões refinado preocupa-se com itens internos aos arquivos, como parágrafaos em textos e métodos em código fonte de programas de computador

Documento XML
Arquivo escrito na linguagem XML. Um documento XML pode ser bem-formado e válido. Um documento bem-formado é aquele que obedece a todas as regras estabelecidas pela linguagem XML. Já um documento válido, além de ser bem-formado, precisa declarar um DTD ou um ou mais XSD aos quais seus elementos obedecem.

DTD DTD é um documento que descreve uma estrutura e o conteúdo de documentos XML, definindo assim um tipo de documento XML.

Eclipse Ambiente de desenvolvimento integrado desenvolvido como projeto de código aberto, apoiado por diversas empresas, destacando-se a IBM. É desenvolvido em linguagem Java, e pode ser utilizado para o desenvolvimento de software em diversas linguagens. Podem ser utilizados plugins para estender as funcionalidades.

GPL General Public License - Licença de código aberto que permite que o conteúdo de determinado software seja alterado, mas impõe determinadas regras quando isso é feito, como, por exemplo, a necessidade de se distribuir sempre o código fonte, e, também, de enviar as alterações de volta para o projeto original

HEAD Versão mais recente dos artefatos disponíveis em um sistema de controle de versão. É um termo usualmente empregado para controle de versão de arquivos de código fonte.

Hipertexto Sistema de organização da informação, no qual certas palavras de um documento estão ligadas a outros documentos, exibindo o texto quando a palavra é selecionada. Um exemplo típico são as páginas da World Wide Web

Hook Script que pode ser configurado para ser executado automaticamente no momento em que determinada ação é executada, como, por exemplo, o commit de um determinado arquivo num repositório. Assemelha-se aos triggers de bancos de dados

HTTP Protocolo de comunicação orientado a conexão e sem estado, construído acima do protocolo TCP.

Internet Rede IP de alcance mundial. Nasceu de um projeto acadêmico, financiado pela DARPA, alcançando fins comerciais décadas mais tarde. Hoje é a rede mais utilizada no mundo, interligando milhões de pessoas.

IP Protocolo de rede não confiável, sem conexão, baseado em pacotes. Geralmente utilizado pelos protocolos UDP e TCP, formando assim a base da Internet.

Java Linguagem de programação orientada a objetos desenvolvida pela *Sun Microsystem*.

Licença Permissão concedida pelo detentor do direito de cópia de um trabalho original. Essa permissão define os direitos e deveres dos pessoas que obtêm uma cópia do trabalho.

Mainframe Computadores de grande porte, geralmente utilizados para processamento de grande volume de informações e que podem ser acessados por diversos usuários simultaneamente

Makefile Arquivo que contém regras relacionadas a dependências entre as partes de um sistema, e que contém instruções sobre a ordem em que as partes do sistema devem ser geradas no momento em que forem compiladas. As informações são utilizadas por um software chamado make, que é capaz também de realizar a construção incremental de aplicações, na qual apenas as versões alteradas entre uma construção e outra são recompiladas

OS Sistema operacional que era utilizado nos computadores da IBM

PDP 11 Série de minicomputadores de 16 bits que eram comercializados pela DEC - Digital Equipment Corporation

Perl Linguagem de programação de propósito geral criada por Larry Wall.

PHP Linguagem de programação de propósito geral, voltada para o desenvolvimento de aplicações Web.

Ramo Termo utilizado, no contexto de controle de versões, pela comunidade de software livre, para denominar uma nova linha de desenvolvimento em um mesmo repositório, criada a partir de uma linha principal (por exemplo, a HEAD). As alterações realizadas nela não afetam a linha da qual ela se originou. Posteriormente, é possível unir as alterações feitas no ramo em sua linha original.

Refinado Controle de versões refinado é o controle de versões que considera detalhes da gramática ou estrutura dos documentos cujas versões são controladas

Site Local onde os recursos disponibilizados na WWW estão armazenados.

Smalltalk Linguagem de programação orientada a objetos, bastante popular nas décadas de 1980 e 1990

SOAP Tecnologia para comunicação entre objetos. O SOAP estabelece um protocolo baseado em XML cuja troca de dados é feita por chamadas remotas (RPC) ou mensagens (XML-RPC).

Terminal slave
Também conhecido por pseudo-terminais. Refere-se a "computadores" remotos que eram utilizados para acessar mainframes e que não passavam de um teclado e um monitor que eram associados a um terminal virtual no servidor.

URL Endereço de um determinado recurso na rede (geralmente a Internet).

Web Services
Conjunto de especificações e convenções para a construção de aplicações distribuídas complexas. Utiliza SOAP como meio de integração entre objetos, UDDI para serviços de registro, descoberta e nomes, e WSDL para descrever os serviços.

WWW A World Wide Web é o principal serviço disponibilizado na Internet. Ela forma um complexo sistema hipermídia distribuído pelos vários nós da rede, sistema este com fins pessoais, acadêmicos e comerciais. Atualmente ela é dirigida pela W3C, que cria normas e especificações (por exemplo, o HTML) que organizam o uso.

XLink Permite a criação de relacionamentos entre documentos XML.

XML A XML é uma linguagem de marcação derivada da SGML. Ela é utilizada na criação de documentos que são um importante meio para armazenamento e recuperação de dados em ambientes heterogêneos.

XML Schema
Cumpre papel semelhante ao DTD, mas acrescenta suporte a vários espaços de nomes em um mesmo documento, além de regras mais ricas para definição da estrutura e conteúdo de documentos XML.