Gerenciamento de Versões de Páginas *Web*

Marinalva Dias Soares

Orientação: 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 - Área de Ciências de Computação e Matemática Computacional.

USP - São Carlos

Outubro de 2000

Agradecimentos

A Deus, por não ter me deixado desistir diante de todas as dificuldades encontradas durante a realização deste trabalho.

Ao meu querido pai que, apesar de não estar mais entre nós, tenho certeza que está muito feliz por mim.

À minha mãe, que sempre me ajudou e me apoiou sem medir esforços durante toda a minha vida.

Ao meu marido Alexandre, por toda compreensão, paciência, incentivo e carinho dedicados a mim durante todo esse tempo que estamos juntos.

À professora Renata, minha orientadora, por ter acreditado em mim e, mesmo não estando presente por algum tempo, sempre me ajudou no que eu precisei.

Ao professor Dilvan, meu orientador substituto, pela atenção e contribuições dadas a este trabalho.

Ao professor João Benedito, pela força e incentivo que me deu para fazer esse mestrado assim que terminei a graduação.

À minha irmã Zenilda, pelo carinho e todo seu bom humor que nunca faltaram.

A todos os colegas e amigos que de alguma forma me ajudaram como Elaine, Taboca, Tatiana, Sr. Joaquim Mauro e sua esposa D. Conceição.

Aos funcionários do ICMC, em especial Beth, Laura, Marília e Paulinho por toda a atenção dada.

Ao CNPq, pelo apoio financeiro.

Ao ICMC - USP, pela oportunidade de estar aqui.

Sumário

Lista de Figuras

Resumo

Em um mundo computacional em constante evolução, a *Web* se apresenta como um ambiente caracterizado por um desenvolvimento acelerado de suas informações. Além das informações na *Web* sofrerem muitas mudanças e com extrema freqüência, os autores (ou desenvolvedores) das páginas enfrentam dificuldades nas suas atividades quando envolvem muitas pessoas trabalhando em paralelo no desenvolvimento de uma página ou de um conjunto de páginas. Diante desses problemas, este trabalho apresenta a ferramenta *VersionWeb* que foi desenvolvida. Os objetivos principais deste trabalho foram proporcionar que os internautas obtivessem as versões das páginas durante a navegação e fornecer um modo fácil de controle de versões de páginas da *Web* aos autores, através da própria *Web*.

Abstract

In the continually changing world of computing, the Web is an example of an environment where information evolves very rapidly. In addition to Web information that changes very much and very frequently, developers are faced with hard work when many people are involved in the parallel development of a set of related Web pages. In the face of such problems, a software tool, *VersionWeb*, was developed. The idea behind this tool is to make Web page version control available during browsing to users. The main goal of *VersionWeb* is to provide the developers with an easy way of controlling Web page versions, through the Web itself.

1. Introdução

1.1. Considerações iniciais

Diante da constante necessidade de algum suporte de versão de arquivos para equipes que trabalham em um desenvolvimento colaborativo de informações baseadas em computador, diversos estudos têm sido realizados há vários anos [Sommerville et al. 1998]. O primeiro uso desse suporte foi em sistemas de gerenciamento de versão como o SCCS (*Source Code Control System*) [Rochkind 1975] e o RCS (*Revision Control System*) [Tichy 1985] (esses sistemas serão descritos com mais detalhes no Capítulo 2) para desenvolvimento de *software*. Os objetivos principais desses sistemas foram reduzir os requisitos de armazenamento para múltiplas versões de *software* e controlar o trabalho cooperativo entre os autores de forma que nenhum sobrescrevesse a cópia do outro.

Esses sistemas evoluíram em vários outros sistemas mais sofisticados como ClearCASE [Leblang 1994] e Adele [Estublier e Casallas 1994] que dão suporte ao controle de processo, porém nem todos os desenvolvedores de *software* precisam de sistemas sofisticados. Muitos deles precisam apenas de sistemas que gerenciem as versões que são geradas dos arquivos para que não haja perda ou sobreposição de informações. O RCS e o SCCS satisfazem a essas necessidades de forma eficiente.

O processo de mudança é uma realidade que ocorre durante todo o ciclo de vida de um *software*. Como as organizações solicitam as mudanças de forma muito aleatória, todas as mudanças feitas devem ser registradas e efetuadas no sistema a um custo razoável, daí surge a necessidade delas serem gerenciadas [Sommerville 1995].

No contexto de *software*, o processo de gerenciamento de mudanças tem efeito quando o *software* e a documentação associada são colocados sob o controle de gerenciamento de configuração de *software* (SCM – *Software Configuration Management*). Durante o desenvolvimento de um *software* de grande escala, alterações descontroladas levam rapidamente ao caos, portanto, os procedimentos de gerenciamento de mudanças devem ser realizados para

garantir que seus custos e benefícios sejam analisados e que as mudanças sejam feitas de forma controlada. O controle de mudanças combina procedimentos humanos e ferramentas automatizadas para proporcionar um mecanismo eficiente de controle das alterações [Pressman 1997].

O controle de versão pode ser definido como o processo de organização, coordenação e gerenciamento de objetos em evolução [Hicks et al. 1998]. Em muitas áreas de aplicação, o processo de evolução do objeto pode ser caracterizado como uma série de refinamentos incrementais. Por exemplo, os desenvolvedores de *software* freqüentemente fazem mudanças em módulos de *software* quando erros são detectados, e os clientes raramente ficam satisfeitos com a primeira versão fazendo, então, com que várias revisões sejam produzidas antes de produzir uma versão final. Em cada um desses casos, objetos em desenvolvimento são alterados e atualizados de forma a produzir o próximo refinamento no processo evolucionário.

Uma versão de um sistema é uma instância que difere, de algum modo, de outras instâncias [Sommerville 1995; Vitali e Durand 1999]. Outras ferramentas de apoio ao controle de versões que são exemplos dessa abordagem, além do SCCS, RCS, ClearCASE e Adele, são aquelas que dão suporte, na forma de sistemas de gerenciamento de versão de arquivos, tais como: o CVS (*Concurrent Versions System*) [Cederqvist 1993; CVS 1999], COV [Lie et al. 1989], ICE [Zeller e Snelting 1995] entre outras.

Um sistema de controle de versão pode permitir o trabalho paralelo entre os desenvolvedores, exibir as diferenças entre várias versões subseqüentes de um artefato, recuperar versões anteriores e localizar modificações de um mesmo documento feitas por desenvolvedores diferentes.

1.2. Motivação

Um ambiente que sofre evolução de informações com extrema freqüência é a *World Wide Web* (WWW), ou simplesmente *Web*. A *Web* é, sem dúvida, uma das maiores fontes de informação que tem sido alvo de diversas pesquisas atualmente. Ela possibilita que pessoas em diferentes locais compartilhem informações de maneira rápida e eficiente, sendo que a colaboração entre pesquisadores é favorecida, pois a *Web*, por ser uma fonte de informação distribuída, facilita esse processo.

No ambiente WWW, os internautas freqüentemente se surpreendem ao visitar uma página e percebem que esta já não possui o mesmo conteúdo ou até mesmo que ela não existe mais; tudo isso é decorrente da rápida e natural evolução das informações na WWW [Sommerville et al. 1998].

Em adição, os desenvolvedores de páginas *Web* encontram dificuldades quando muitas pessoas estão envolvidas na construção em paralelo de uma mesma página ou de um conjunto de páginas relacionadas. Isso se deve do fato de que os desenvolvedores trabalham independentemente em suas próprias cópias, tendo como principal problema a integração dessas cópias em um hiperdocumento final [Sommerville et al. 1998]. Além disso, em geral, o volume de documentos envolvidos é significativamente grande e foge a um controle simples da evolução de suas cópias, pois pouco ou nenhum gerenciamento de informação é fornecido. As páginas são freqüentemente marcadas como "em construção" sem nenhuma informação sobre quando a construção começou ou quando vai terminar. Por isso os leitores têm que revisitar as páginas para verificar se elas já estão completas e disponíveis [Sommerville et al. 1998].

Esses problemas recaem basicamente em dois cenários típicos de formas de utilização de versionamento para páginas *Web:* **a)** criação de páginas de forma cooperativa **b)** gerenciamento das informações das páginas. Para o primeiro cenário, alguns requisitos são necessários, tais como:

- desenvolvimento independente de versões privadas da página;
- conhecimento de outras versões produzidas por outros desenvolvedores;
- facilidade de visualizar o que o outro desenvolvedor está fazendo;
- integração de diferentes versões de uma página para uní-las em uma só.

Quanto ao gerenciamento de informações das páginas, é preciso satisfazer alguns requisitos tais como:

- possibilidade de tornar as páginas "versionáveis" de forma simples, ou seja, colocá-las sob o controle de versão usando alguma ferramenta automatizada;
- várias versões de uma mesma página devem co-existir;
- mostrar a versão mais recente da página;
- permitir navegar pelas outras versões anteriores;

- fornecer facilidades para adicionar novas informações às páginas na medida em que se tornar necessário.

As várias formas de atuação nesses dois cenários típicos de versionamento para páginas *Web* mostram que um suporte ao controle de versão dos arquivos para os desenvolvedores, os quais trabalham em um desenvolvimento colaborativo, e um suporte à navegação por versões anteriores das páginas, por parte dos internautas, são alvos de investigação com muito interesse. Neste contexto, uma ferramenta que auxilie no trabalho cooperativo entre os desenvolvedores, no gerenciamento das diferentes versões de uma página *Web*, e forneça um mecanismo para visualização e recuperação da mesma por parte dos internautas se apresenta como um auxílio de grande utilidade.

1.3. Trabalhos Relacionados

Atualmente, as ferramentas existentes para controle de versões de arquivos e gerenciamento de mudanças, como o RCS e o SCCS por exemplo, têm ajudado desenvolvedores de *software* no controle das alterações. Os desenvolvedores, com o uso de tais ferramentas, economizam tempo e espaço físico requeridos para as diversas cópias criadas durante o processo de desenvolvimento. Em ambientes de autoria, especialmente falando da *Web*, foram encontradas algumas ferramentas tais como WebRC, V-Web e AIDE.

WebRC (*Configuration Management for a Cooperation Tool*) foi a primeira ferramenta baseada em cooperação projetada para a *Web* com suporte ao gerenciamento de configuração. Essa ferramenta auxilia a cooperação de grupos na *Web* com o conceito de *workspace* de cooperação [Fröhlich e Nedjl 1997]. No sistema WebRC, os usuários trabalham em *workspaces* separados, e o acesso ao *workspace* por um outro usuário pode ser feito através do *locking* de arquivos ou *downloading* dos arquivos para o seu próprio *workspace*.

O V-Web é um simples sistema de versionamento de páginas [Sommerville et al. 1998]. A principal dificuldade nesse sistema foi a necessidade de manter o acesso à página através da sua URL original da página não versionada (que não estava sob o controle de versão). No modelo de versão adotado em V-Web, uma página original não versionada é substituída por uma página V-Web. Esta contém o conteúdo da página original e uma referência para uma lista de versões da

página que está armazenada em um diretório juntamente com informações sobre cada uma de suas versões.

Esse sistema é muito utilizado em situações onde a informação evolui naturalmente através de uma série de versões, onde existe a necessidade de acessar versões anteriores da mesma forma que a versão atual e quando a informação é produzida com um esforço colaborativo por equipes de pessoas responsáveis pela criação da informação.

Neste sistema, a página original (não versionada) é substituída por uma página V-Web. A página V-Web possui basicamente 3 *frames* HTML: um *frame* exibe o conteúdo da página original, outro *frame* mostra o conjunto de versões da página e o último *frame* mostra um conjunto de opções que permite os desenvolvedores terem acesso às funcionalidades do V-Web como: adicionar uma versão à lista de versões, remover uma versão do conjunto de versões e criar uma página V-Web de uma página não versionada. Essas operações são feitas por *scripts* CGIs no servidor.

V-Web dá suporte para a autoria por grupos de autores (ou desenvolvedores). Os autores devem estar registrados em algum grupo para terem acesso à edição das páginas. Cada grupo tem um conjunto de páginas que lhe é específico. Se um autor fizer *login* como um membro do grupo, ele poderá ver o que os membros desse grupo estão fazendo, mas não poderá editar as páginas. Cada membro do grupo tem acesso a um conjunto de páginas que lhe é específico e pode editar somente esse conjunto.

Cada membro deve estar consciente do trabalho cooperativo e saber o que o outro está fazendo para facilitar a união de seus trabalhos em uma cópia final. Quando o grupo termina o trabalho e junta as cópias em uma única versão, essa versão é então disponibilizada para acesso público. Para isso, é necessária comunicação constante entre eles para notificar uns aos outros das atividades que estão sendo feitas para não haver inconsistências.

AT&T Internet Difference Engine (AIDE) é um sistema que fornece algum suporte de versão usando o sistema RCS [Douglis et al. 1998]. O objetivo principal do AIDE é permitir que os usuários (internautas e/ou desenvolvedores) vejam as diferenças entre páginas *Web* quando elas são atualizadas. Esse sistema permite que os usuários armazenem as páginas no sistema RCS e fornece facilidades para mostrar as diferenças entre as versões passadas e a versão atual da

página. Os usuários devem requisitar explicitamente ao sistema para mostrar essas diferenças de acordo com a data. O sistema também fornece uma facilidade automática para detectar quando páginas de interesse foram atualizadas e permite recuperar as versões mais recentes dessas páginas.

Neste sistema, os usuários devem: **a)** especificar a URL para localização das mudanças, incluindo o grau de recursividade de *links* para outras páginas **b)** especificar que a nova versão da URL que está sendo localizada deve ser salva **c)** ver de forma textual as diferenças entre duas versões **d)** ou ver um grafo mostrando a estrutura de uma página, incluindo o estado das páginas a que ela se refere.

O sistema AIDE consiste de alguns componentes, incluindo um *Web-crawler* que detecta alterações nos arquivos, um arquivo contendo as versões da página, uma ferramenta chamada *HtmlDiff* que mostra as alterações entre duas versões de uma página, e uma interface gráfica para visualizar o relacionamento entre as páginas [Douglis et al. 1998]. Os usuários interagem com o sistema via formulários HTML, os quais passam os pedidos para um *script* CGI e este realiza operações como salvar e recuperar versões de uma página.

O estudo da maioria das funcionalidades dessas ferramentas possibilitou uma visão geral das características relacionadas às tarefas de controle de versão na *Web* que se apresentam na literatura. Uma breve comparação entre a ferramenta desenvolvida neste projeto e as ferramentas descritas será apresentada no Capítulo 6, juntamente com as Conclusões deste trabalho.

1.4. Objetivos

O objetivo deste trabalho de mestrado foi desenvolver uma ferramenta para auxiliar no gerenciamento de versões de páginas *Web* por meio da própria *Web*, apoiando os desenvolvedores de uma página no trabalho colaborativo, sem que haja perda ou sobreposição acidental de informações, além de possibilitar aos internautas visualizarem diferentes versões de uma mesma página e localizar as diferenças entre elas. A ferramenta foi denominada *VersionWeb*.

Para o desenvolvimento da ferramenta proposta foi utilizado o CVS, um sistema de versões concorrentes que oferece recursos para as operações básicas do controle de versão, permite o trabalho cooperativo através da rede e acesso simultâneo sobre os arquivos por vários

desenvolvedores. Além disso, como parte de revisão bibliográfica, foram estudados também alguns modelos de versão de *software* para SCM (*Software Configuration Management*) juntamente com alguns conceitos básicos relacionados ao controle de versão tais como: definição de uma versão, como são geradas as diversas versões (ou revisões) de um sistema (ou de um *software*), como são armazenadas todas essas versões em um espaço mínimo de armazenamento, geração de *branches*, além de vários outros conceitos relacionados ao controle de versão.

Em adição, para viabilizar o desenvolvimento da ferramenta, foram estudados alguns mecanismos para programação na *Web* tais como CGI (*Common Gateway Interface*), Java, Applets, Servlets e JSP.

1.5. Estrutura da Dissertação

Esta dissertação está organizada da seguinte forma: no Capítulo 2 são apresentados alguns modelos de versão de *software* para SCM, descrevendo o espaço do produto e o espaço da versão. Ou seja, como a estrutura de um produto de *software* e as diferentes versões de seus componentes podem estar organizadas. No Capítulo 3 é apresentado o CVS, *software* utilizado para dar suporte ao gerenciamento de versões das páginas construídas pelos desenvolvedores. No Capítulo 4 são apresentados os mecanismos para programação na *Web* que foram estudados com o objetivo de auxiliar o projeto adequado da ferramenta *VersionWeb* desenvolvida. No Capítulo 5 são descritos a *VersionWeb*, suas funcionalidades e alguns resultados de testes obtidos. No Capítulo 6 são apresentadas as conclusões deste trabalho e algumas sugestões para trabalhos futuros.

2. Modelos de Versão de *Software* para SCM

2.1. Considerações iniciais

O termo versão pode ter várias definições, pois seu significado depende muito do contexto e do modelo de versão adotado. Porém, o leitor ou qualquer desenvolvedor de *software* pode ter um entendimento intuitivo do que significa "versão", desde que considere que uma versão não pode existir por si só, ela deve ser vista ou entendida como sendo uma revisão de algo originalmente concebido. Para desenvolvedores de *software*, versão pode ser definida como uma instância concreta de algum objeto ou artefato de *software* [Munch 1995].

É muito comum, no decorrer do desenvolvimento de um *software*, efetuarem-se mudanças para melhoria do *software* ou para correção de seus componentes e relacionamentos entre eles. Portanto, diante dessas mudanças, a necessidade de manter uma cópia de tudo que é feito e modificado torna-se importante, pois as alterações descontroladas podem gerar erros e perdas de informações. Para isso, essas cópias devem ser gerenciadas e controladas de forma que seja possível a recuperação das mesmas quando necessário.

Neste Capítulo são descritas as principais funcionalidades do SCM (*Software Configuration Management*) que ajudam a gerenciar a evolução de sistemas de *software* durante o seu processo de desenvolvimento, alguns modelos de versão de *software* para SCM e termos relacionados como espaço do produto, espaço da versão e suas formas de representação, e exemplos de ferramentas automatizadas para o controle de versão. A Seção seguinte descreve a atuação do SCM no processo de desenvolvimento de um *software*.

2.2. SCM

O SCM, ou *Software Configuration Management*, é uma atividade abrangente aplicada em todo o processo de engenharia de *software* que é responsável por gerenciar a evolução de sistemas de *software* grandes e complexos [Pressman 1995]. O SCM identifica, controla, faz a auditoria e relata as modificações que inevitavelmente ocorrem quando o *software* está sendo desenvolvido e mesmo depois que ele é distribuído aos clientes.

O termo "configuração" pode ser definido como sendo um conjunto de revisões, onde cada revisão vem de um grupo de revisões diferente, e as revisões que compõem uma configuração são selecionadas de acordo com um certo critério ou regras de seleção [Tichy 1985]. Um grupo de revisões, por sua vez, é um conjunto de documentos texto, chamado revisões, onde cada revisão evolui uma da outra. Assim, a configuração de um *software* compõe-se de um conjunto de objetos inter-relacionados denominados itens de configuração ou simplesmente itens.

A importância do SCM tem sido amplamente reconhecida e refletida, em particular, no CMM (*Capability Maturity Model*). Desenvolvido pelo *Software Engineering Institute* (SEI), o CMM define níveis de maturidade de forma a avaliar o processo de desenvolvimento de *software* nas organizações [Conradi e Westfechtel 1998].

O SCM pode ser visto como uma disciplina de apoio ao gerenciamento e desenvolvimento de *software*. No caso de apoio ao gerenciamento, o SCM gerencia o controle de alterações dos produtos de *software* fazendo a identificação dos componentes do produto e suas versões, o controle de alterações (pelo estabelecimento de procedimentos a serem seguidos quando uma alteração é realizada), a contabilidade de *status* (registrando os *status* dos componentes e pedidos de alteração), a análise e a revisão (garantia de qualidade das funções dos componentes para preservar a consistência do produto) [Kilpi 1997]. Já no caso de apoio ao desenvolvimento, o SCM fornece funções que auxiliam os programadores na realização coordenada de alterações nos produtos de *software*. Além disso, o SCM auxilia os desenvolvedores com a composição dos produtos de *software* a serem controlados registrando, assim, suas revisões e variantes (esses dois termos serão tratados com mais detalhes na sub-seção 2.4.1), mantendo a consistência entre componentes dependentes, reconstruindo configurações de *software* previamente gravadas, construindo objetos derivados (código compilado e executável) de seus fontes (programa texto) e construindo novas configurações baseadas nas descrições de suas propriedades.

Os objetos (ou componentes) de *software* e seus relacionamentos constituem o espaço do produto, já as suas versões estão organizadas no espaço da versão. Existem vários modelos de versão de um *software*, mas cada um deles é caracterizado pela maneira como o espaço da versão está estruturado, pela decisão de quais objetos serão controlados e pela forma que a reconstrução de velhas e a construção de novas versões serão realizadas [Conradi e Westfechtel 1998]. A

Seção seguinte descreve o espaço do produto de um *software* e os relacionamentos entre seus componentes.

2.3. Espaço do Produto

O espaço do produto descreve a estrutura de um produto de *software* a ser controlado e pode ser representado por um grafo, no qual os nós correspondem aos objetos de *software* e as arestas correspondem aos seus relacionamentos. Os diversos modelos de versão diferem em relação ao espaço do produto que adotam e, essas diferenças, referem-se aos tipos de objetos de *software* e seus relacionamentos e à granularidade de representações de objeto [Conradi e Westfechtel 1998].

2.3.1. Objetos de *Software*

Um objeto de *software* contém o resultado de atividades de desenvolvimento ou de operações de manutenção. Um sistema SCM deve gerenciar todos os tipos de objetos criados durante o ciclo de vida do *software*, incluindo especificação de requisitos, projeto, documentação, codificação do programa, plano de testes, etc. A identificação de objetos é uma função essencial fornecida pelo SCM, na qual cada objeto de *software* possui um identificador (OID - *object identifier*) que serve para identificá-lo unicamente dentro de um determinado contexto.

Os objetos de *software* podem ser vistos como unidades de granularidade grossa que estão estruturadas internamente, ou seja, por exemplo, um módulo de um programa pode ser composto de declarações de variáveis e uma documentação pode consistir de seções e parágrafos (os quais representam unidades de granularidade fina). Portanto, um objeto de *software* é composto por mais de uma unidade de granularidade fina.

Os objetos de *software* podem ter diferentes representações, dependendo dos tipos de ferramentas que trabalham sobre eles. Em ambientes *toolkit*, os objetos de *software* são armazenados como arquivos texto [Rochkind 1975], enquanto que em ambientes orientados a estrutura, os objetos são armazenados como estrutura de árvores ou grafos [Nagl 1996]. Mas, independentemente do tipo de representação escolhida para os objetos de *software*, pode ser feita uma distinção entre modelos de domínio independente e modelos de domínio específico. Para os modelos de domínio independente não importa os tipos de objetos de *software* a serem mantidos, ou seja, todos os objetos criados durante o ciclo de vida do *software* são submetidos ao controle

de versão [Tichy 1985]. Por outro lado, os modelos de domínio específico são projetados para tipos específicos de objetos (por exemplo, tipos abstratos de dados em especificações algébricas) [Ehrig et al. 1989].

A seguir, são descritos os tipos de relacionamentos entre objetos, inclusive o que possibilita a representação de níveis de granularidade.

2.3.2. Relacionamentos

Os objetos de *software* são conectados por dois tipos de relacionamentos [Conradi e Westfechtel 1998]:

- **relacionamentos de composição** - são usados para organizar os produtos ou objetos de *software* de acordo com sua granularidade. Por exemplo, um produto de *software* pode ser composto de subsistemas, e estes podem consistir de módulos. Objetos que são decompostos são chamados **objetos compostos** ou **configurações** (objetos com granularidade grossa). Já os objetos que residem em nível de composição hierárquica não decomponível são denominados **objetos atômicos**. Um objeto de *software* "atômico" pode estar ainda estruturado internamente, ou seja, ele pode ter um conteúdo de granularidade fina. A raiz (*root*) de uma composição hierárquica é chamada de **produto**;
- **relacionamentos de dependência** - estabelecem conexões direcionadas entre objetos que são ortogonais aos relacionamentos de composição. Essas conexões incluem, por exemplo, dependências de ciclo de vida entre especificações de requisitos, projetos e implementações de módulos, importam ou incluem dependências entre módulos, geram dependências entre código compilado e código fonte. A fonte e o destino de uma dependência correspondem a um **objeto dependente** e a um **objeto mestre**, respectivamente. Uma dependência implica que o conteúdo de um objeto dependente deve estar consistente com o objeto mestre, pois quando o objeto mestre for modificado, o objeto dependente também deve ser alterado.

Em adição, os objetos de *software* também podem ser classificados em **objetos fontes** e **objetos derivados**. Um objeto fonte é criado pelo usuário através de ferramentas interativas (por exemplo, editores de texto ou editores gráficos). Já um objeto derivado é criado automaticamente por alguma ferramenta (por exemplo, um compilador). A classificação dos objetos de *software*

como fonte ou derivado depende, também, da ferramenta disponível. Além disso, os objetos de *software* podem ser parcialmente derivados e parcialmente construídos manualmente. Por exemplo, o esqueleto do corpo de um módulo de *software* pode ser criado automaticamente e, posteriormente, preenchido por um programador.

O processo de criação de objetos derivados a partir de objetos fontes e de outros objetos derivados é chamado **sistema de construção**. As ações a serem realizadas são especificadas por regras de construção, e a ferramenta que estiver sendo utilizada deve garantir que os passos de construção correspondentes a essas regras sejam executados na ordem correta.

Para melhor entender os conceitos do espaço do produto descritos, a sub-seção seguinte ilustra algumas formas de representação do espaço do produto e os seus relacionamentos.

2.3.3. Representações do espaço do produto

As **Figuras 2.1 a 2.4** ilustram diferentes representações de um simples produto de *software*, denominado **foo**, o qual está implementado na linguagem de programação C. A **Figura 2.1** mostra os módulos de foo e suas dependências. Pode-se observar que o módulo principal (**main**) é dependente dos módulos **a** e **b**, e esses dependem do módulo **c**. As diferentes representações para o produto foo estão ilustradas nas **Figuras 2.2, 2.3 e 2.4**.

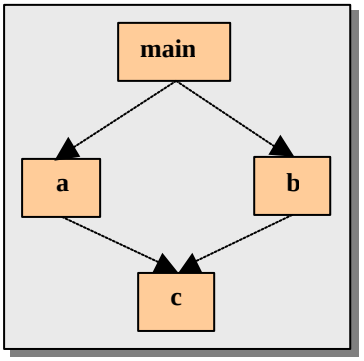


Figura 2.1 - Módulos do *software* foo e suas dependências [Conradi e Westfechtel 1998]

Na **Figura 2.2**, foo está armazenado no sistema de arquivos e cada módulo é representado por múltiplos arquivos. Os sufixos **.h**, **.c**, **.o** e **.exe** indicam cabeçalho de arquivo, corpo de arquivo, código compilado e executáveis, respectivamente. As dependências entre eles e as regras de construção estão armazenadas em um arquivo texto (neste caso, em um arquivo *make*).

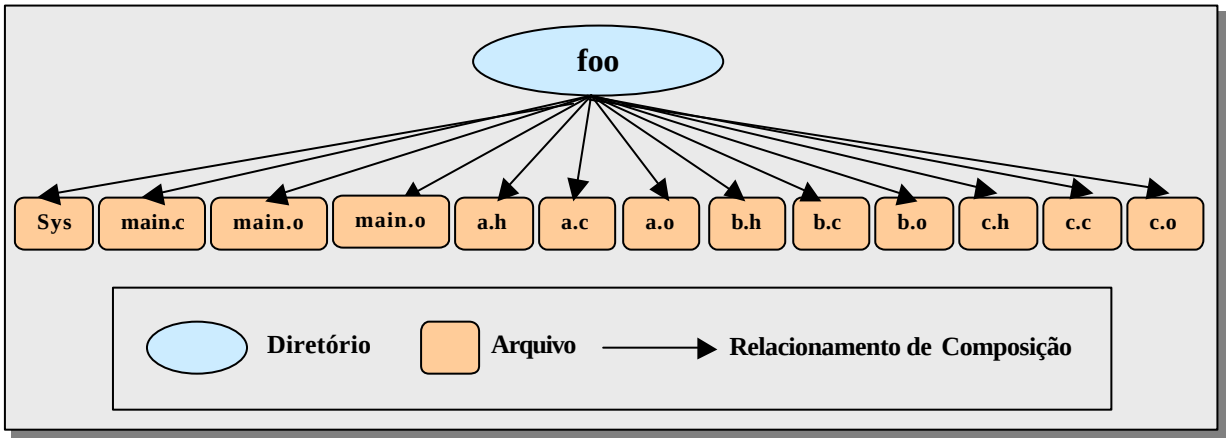


Figura 2.2 - Relacionamentos de composição de foo [Conradi e Westfechtel 1998]

Na **Figura 2.3**, assim como na representação do sistema de arquivos, há uma árvore de composição, onde os nós folhas dessa árvore correspondem a arquivos únicos. Entretanto, as dependências não estão armazenadas em um arquivo texto separado como no caso anterior (**Figura 2.2**), mas a árvore de representação é acrescida com os relacionamentos refletindo a inclusão de dependências entre os arquivos dos módulos do produto.

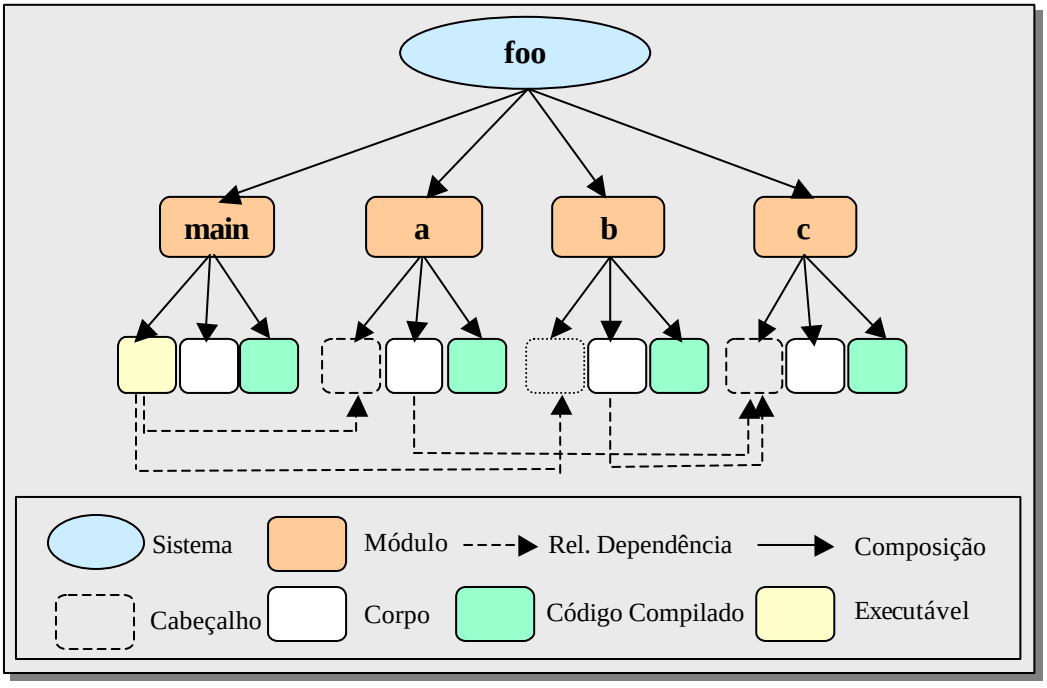


Figura 2.3 - Relacionamentos de composição e de dependência de foo [Conradi e Westfechtel 1998]

Já a **Figura 2.4** representa os arquivos pertencentes a um mesmo módulo em um objeto, e somente o relacionamento de dependência é usado.

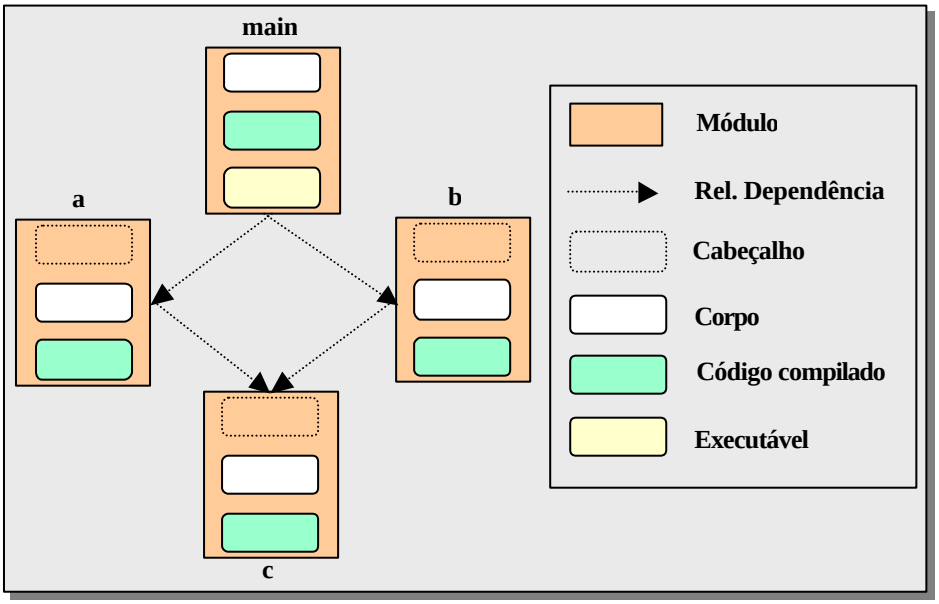


Figura 2.4 - Arquivos de um módulo representados como um objeto [Conradi e Westfechtel 1998]

Dependendo da complexidade do *software* e da freqüência com que as mudanças ocorrem, as diferentes formas de representação do espaço do produto apresentadas podem se tornar interessantes para os desenvolvedores pois, através delas, é possível observar os relacionamentos de composição do produto e os relacionamentos de dependência entre os componentes desse produto. Essas representações permitem analisar os efeitos causados por uma mudança em um dos componentes de forma mais rápida. Além disso, pode-se observar que algumas representações são mais compactas e outras mais detalhadas.

Na forma de representação apresentada na **Figura 2.1**, por exemplo, é possível saber quais são os módulos que compõem o *software* **foo** e as dependências entre eles. Mas isso ainda não é suficiente, pois cada módulo pode ainda ser composto por outros. Já na **Figura 2.2** é possível ver quais arquivos compõem cada módulo apresentado na **Figura 2.1**, mas, por outro lado, não apresenta claramente nenhum relacionamento de dependência entre os módulos e/ou entre os arquivos pertencentes a esses módulos (o relacionamento de dependência está descrito somente no arquivo texto Sys), o que não é suficiente para se saber visualmente se uma determinada alteração feita em um módulo afetará outros componentes do *software*. Já a **Figura 2.3** ilustra uma representação mais completa tanto dos relacionamentos de composição como dos relacionamentos de dependência. Essa representação pode auxiliar o desenvolvedor a analisar quais arquivos serão afetados por uma determinada mudança, corrigir erros (através dos

relacionamentos de dependência é possível detectar a origem de um erro), etc. A **Figura 2.4** é equivalente à **Figura 2.1**, porém, a primeira difere da segunda por representar os componentes de cada módulo agrupados em um objeto.

Podemos ainda concluir que, quanto mais complexo for o *software* (maior quantidade de relacionamentos de composição e de dependência), mais complexa será a sua representação. Quanto maior o número de componentes e dependências entre seus componentes, o grafo de representação só tenderá a crescer tanto em largura como em profundidade, além do emaranhado de relacionamentos de dependência que pode ser gerado. Logicamente, a complexidade do grafo vai depender da forma de representação usada. Não se pode descartar a hipótese de poder usar todas as formas de representação pois, dependendo da análise necessária, basta selecionar a forma mais adequada.

Devido às alterações efetuadas no *software*, este pode ter várias versões, tornando-se importante analisar, além do espaço do produto, o espaço da versão juntamente com as suas diferentes formas de representação. A Seção 2.4 descreve esses conceitos.

2.4. Espaço da Versão

Um modelo de versão define os itens a serem controlados, as propriedades comuns compartilhadas por todas as versões de um item, os deltas e a maneira com que os conjuntos de versões estão organizados (introduzindo dimensões de evolução tais como revisões e variantes) [Conradi e Westfechtel 1998]. Além disso, um modelo de versão define se uma versão é caracterizada pelo estado que ela representa ou pelas alterações relacionadas a alguma *baseline*[1], seleciona uma representação adequada para o conjunto de versões (grafos, por exemplo), e fornece também operações para recuperação e construção de versões.

Uma versão **v** representa o estado de um item **i**, sendo que a versão **v** é caracterizada pelo par **v = (ps, vs)**, onde **ps** denota um ponto (ou um componente) no espaço do produto e **vs** denota um estado (ou uma versão) no espaço da versão. Um item versionado é qualquer item que pode ser colocado sob o controle de versão (ou *versioning* – que pode ser definido como o gerenciamento

[1] No contexto da engenharia de *software*, uma *baseline* pode ser definida como um marco de referência no desenvolvimento de um *software*, que é caracterizado pela entrega de um ou mais itens de configuração e pela aprovação desses, obtida por uma revisão técnica formal [Pressman 1995].

de versões de objetos [Munch 1995]), incluindo, por exemplo, arquivos e diretórios de sistemas baseados em arquivos, objetos armazenados em base de dados orientada a objetos, entidades, relacionamentos, etc. Para os itens que são submetidos ao controle de versão, mais de um estado pode ser mantido, ao contrário daqueles que não são controlados, pois neste caso somente um estado é mantido. O controle de versão pode ser aplicado em qualquer nível de granularidade, abrangendo desde o produto de *software* até as linhas de texto de um determinado arquivo.

Para um item que está sendo controlado, cada versão deve ser unicamente identificada através de um identificador de versão (VID - *version identifier*). Vários sistemas SCM geram automaticamente um número para cada versão e oferecem nomes simbólicos (definidos pelo usuário) que podem servir como chave primária de identificação. Todas as versões de um item compartilham propriedades comuns, onde cada uma dessas propriedades pode ser representada por relacionamentos ou atributos que não são alterados. Entretanto, a decisão de quais propriedades serão compartilhadas pelas versões depende do modelo de versão adotado e também da forma como esse modelo foi adaptado para uma certa aplicação.

Um item que está sob o controle de versão pode ser pensado como um recipiente para um conjunto **V** de versões. A funcionalidade do controle de versão é fortemente influenciada pela maneira como **V** está definido. Para isso, é importante diferenciar o controle de versão explícito (*extensional versioning*) e controle de versão implícito (*intensional versioning*). O controle de versão explícito significa que **V** está definido pela enumeração de seus membros: $\mathbf{V = \{v_1,...,v_n\}}$. Esse tipo de controle oferece suporte à recuperação de versões anteriores (requisito fundamental a qualquer modelo de versão) e cada versão é identificada por um único número. O usuário, ao interagir com o sistema SCM, pode recuperar qualquer versão v_i, realizar alterações na versão recuperada e, finalmente, submeter a versão modificada a uma nova versão v_{i+1}.

Por outro lado, o controle de versão implícito é aplicado quando a construção automática de versões consistentes torna-se necessária no espaço da versão. Em vez da enumeração de seus membros, o conjunto **V** de versão está definido pelo predicado: $\mathbf{V = \{v|c(v)\}}$. Neste caso, as versões estão implícitas e várias combinações podem ser construídas. O predicado **c** define a condição que deve ser satisfeita por todos os membros de **V** para produzir uma nova versão.

2.4.1. Propósitos de evolução: revisões, variantes e cooperação

O controle de versão é realizado com diferentes propósitos. Uma versão destinada a substituir seu predecessor é chamada revisão (versão serial). Além disso, uma revisão pode ser definida como sendo a forma mais simples de criar novas versões a partir de uma modificação na versão anterior. Dessa forma, as versões formam uma lista encadeada que pode ser referida como uma cadeia de revisões [Munch 1995]. Geralmente, uma nova revisão é gerada quando se torna necessário fazer alguma melhoria na versão anterior, prover algum aumento de funcionalidades, corrigir problemas, adaptar a versão anterior a algum outro ambiente, etc.

Por outro lado, versões que não substituem seu predecessor são chamadas variantes (versões paralelas). Por exemplo, variantes de estruturas de dados podem se diferenciar pelo consumo de armazenamento, eficiência de tempo de execução e operações de acesso. Já no que se refere ao produto de *software*, este pode ter variantes por suportar múltiplos sistemas operacionais e diferentes sistemas de janelas, por exemplo. Um outro exemplo pode ser o seguinte: considere que um simples programa é composto pelos componentes 1, 2, 3, 4 e 5. O componente 4 é usado somente quando o *software* é implementado usando monitores coloridos. O componente 5 é implementado para monitores monocromáticos. Neste caso, duas variantes de versão podem ser definidas: uma contendo os componentes 1, 2, 3 e 4; e outra contendo os componentes 1, 2, 3 e 5 [Pressman 1997].

Finalmente, as versões podem também ser mantidas para apoiar a cooperação. Neste caso, múltiplos desenvolvedores trabalham em paralelo em diferentes versões e cada desenvolvedor opera em um *workspace* que contém as versões criadas e usadas. Para isso, algumas políticas de cooperação são necessárias para controlar quando as versões devem ser exportadas de um *workspace* ou importadas para dentro de um *workspace* [Conradi e Westfechtel 1998].

Com base nos conceitos sobre modelos de versão de *software* descritos até aqui, a sub-seção seguinte ilustra algumas formas de representação desses modelos e cita alguns exemplos de ferramentas de gerenciamento de versões onde essas representações são aplicadas.

2.4.2. Representações do espaço da versão: grafos e grades de versão

Existem várias formas de representar o espaço da versão, mas a maioria dos sistemas SCM faz essa representação através de grafos. Um grafo de versão consiste de nós e arestas que correspondem às versões e seus relacionamentos, respectivamente.

No caso mais simples (organização unidimensional), um grafo de versão consiste de um conjunto de versões conectadas por relacionamentos de um único tipo chamado **sucessores**. Um grafo de versão deste tipo representa a evolução histórica de um item, onde "v_2 é sucessor de v_1" significa que "v_2 derivou de v_1" através de alguma alteração aplicada em v_1. Os grafos de versão podem ser representados de diferentes formas, como mostra a **Figura 2.5** . Na maioria dos casos, as versões podem estar organizadas em uma seqüência de revisões como está ilustrado na **Figura 2.5(a)**. Na representação através de árvores, **Figura 2.5(b)**, os sucessores de versões que não são folhas podem ser criados, por exemplo, de forma a manter as versões anteriores já distribuídas. A **Figura 2.5(c)** representa um grafo acíclico, onde uma versão pode ter múltiplos predecessores (isso ocorre quando é realizada uma operação *merging*).

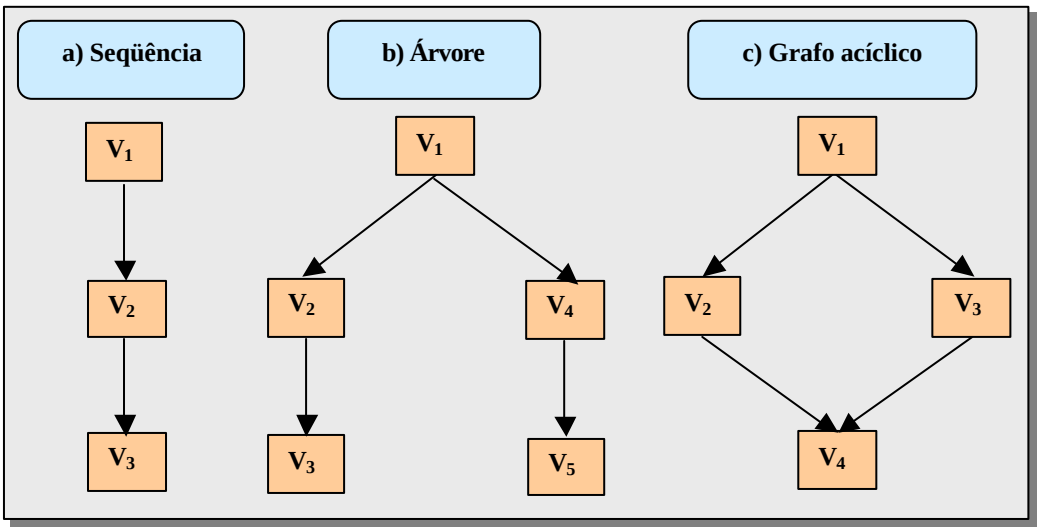


Figura 2.5 - Grafo de versão - organização unidimensional [Conradi e Westfechtel 1998]

Um grafo de versão em organização bidimensional é composto por *branches*, sendo que cada *branch* consiste de uma seqüência de revisões. Neste caso, no mínimo dois relacionamentos são necessários: **sucessores** (dentro de uma *branch*) e **descendentes** (entre as *branches*), como ilustra a **Figura 2.6**.

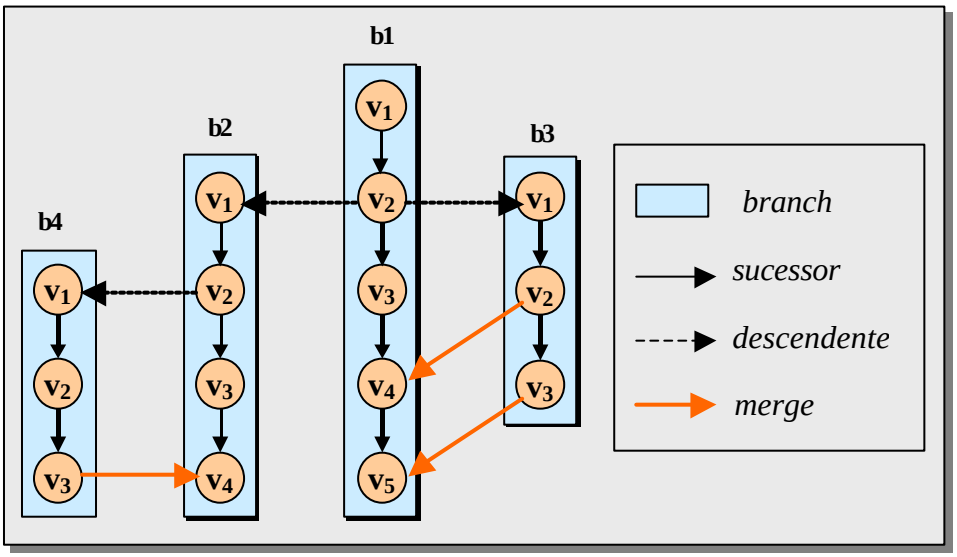


Figura 2.6 - Grafo de versão - organização bidimensional [Conradi e Westfechtel 1998]

Essa organização é aplicada, por exemplo, no RCS (*Revision Control System*) e no ClearCase [Leblang 1994], sendo que este vai além da organização do RCS pela representação de união no grafo de versão, ou seja, mudanças realizadas em uma *branch* podem ser propagadas para outra *branch* (através da operação *merging*). Essencialmente, isso resulta em um grafo acíclico, mas as *branches* não são unidas, em vez disso, cada uma continua a existir.

No caso de um número pequeno de variantes, a representação usando *branches* não apresenta problemas, mas para uma variação maior, essa representação não é conveniente devido ao grande número de *branches* necessário. Assumindo que cada dimensão seja modelada por um atributo com domínio $\mathbf{A_i}$, então o número de *branches* **b** é obtido pelo produto das cardinalidades do domínio: $\mathbf{B <= |A_1| ... |A_n}$.

Para melhor entendimento, as **Figuras 2.7** e **2.8** ilustram o *software* **foo** assumindo que este varia com relação ao sistema operacional (DOS, Unix, VMS – A_1 com cardinalidade 3), sistema de janelas (X11, SunView, Windows – A_2 com cardinalidade 3) e sistema de base de dados (Oracle, Informix – A_3 com cardinalidade 2). Se a representação para esse caso fosse através de *branches*, 18 *branches* seriam necessárias.

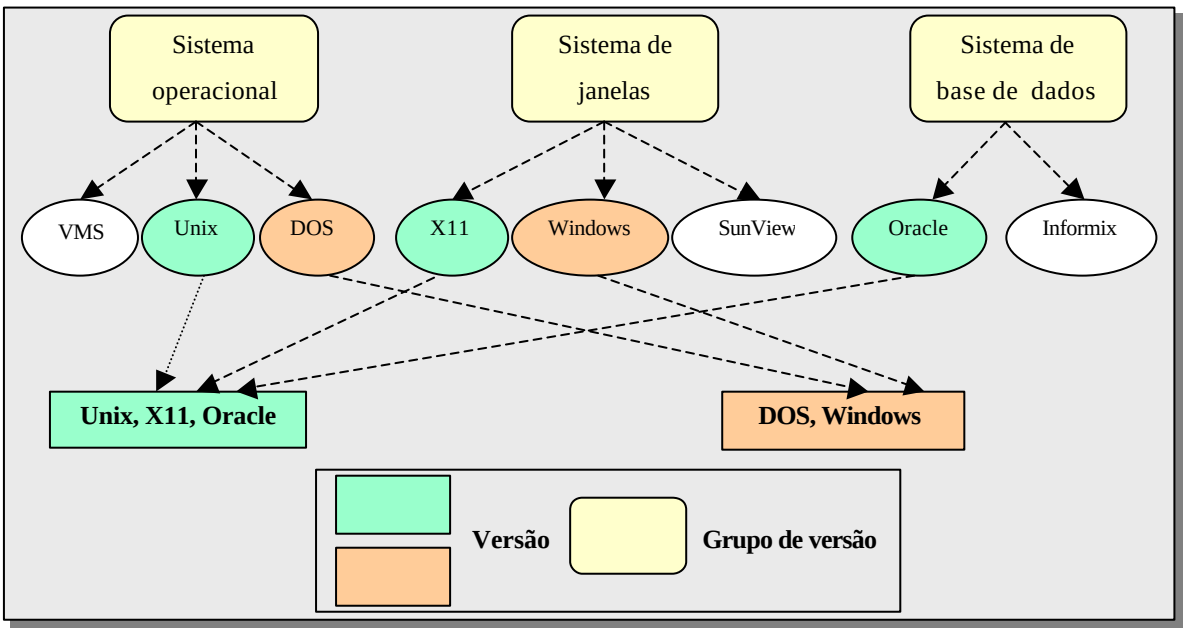


Figura 2.7 - Grafo de versão - variação n-dimensional [Conradi e Westfechtel 1998]

No entanto, este problema pode ser resolvido das seguintes formas:

- os grafos de versões podem ser generalizados de forma a suportar múltiplas variações. Na **Figura 2.7**, as versões estão organizadas em grupos de versões que são usados para construir hierarquias de classificação;
- alternativamente, as versões também podem estar organizadas em uma grade, ou seja, em um espaço n-dimensional onde as dimensões correspondem aos atributos variantes (**Figura 2.8**).

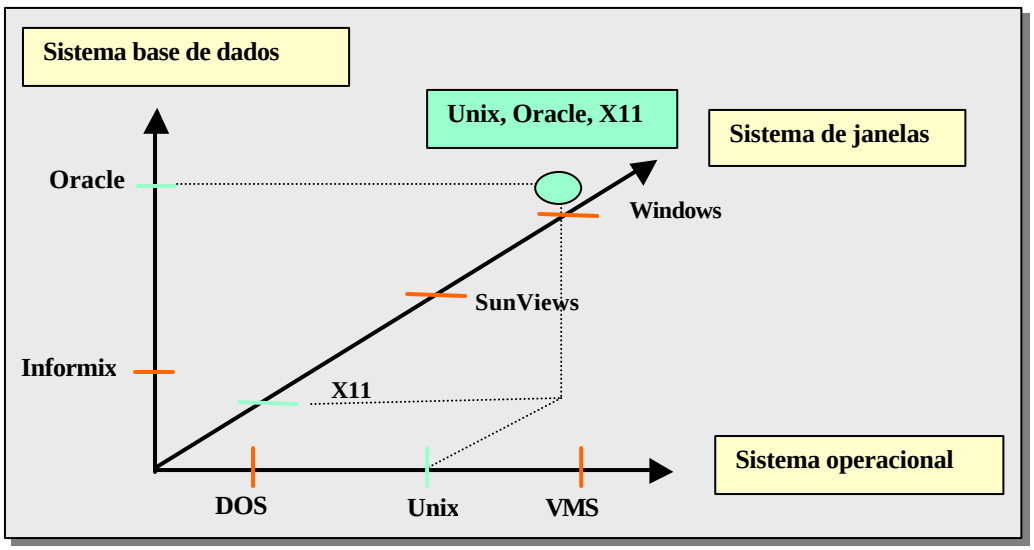


Figura 2.8 - Grade de versão - variação n-dimensional [Conradi e Westfechtel 1998]

As **Figuras 2.7** e **2.8** ilustram somente o espaço de variantes, assumindo que não há evolução ao longo do tempo.

2.4.3. Controle de versão baseado no estado e na alteração de um item

O controle de versão é baseado no estado quando o seu modelo se concentra no estado de um item. Neste caso, as versões são descritas em termos de revisões e variantes. Em modelos baseados na alteração, uma versão é descrita em termos de mudanças aplicadas a uma *baseline*. Dessa forma, as alterações estão associadas a um CID (*change identifier*).

O espaço da alteração (espaço da versão estruturado em termos de alterações) pode ser representado de diferentes formas, mas serão apresentadas aqui apenas duas. A **Figura 2.9** mostra uma matriz de representação onde as linhas e colunas correspondem a versões e alterações, respectivamente.

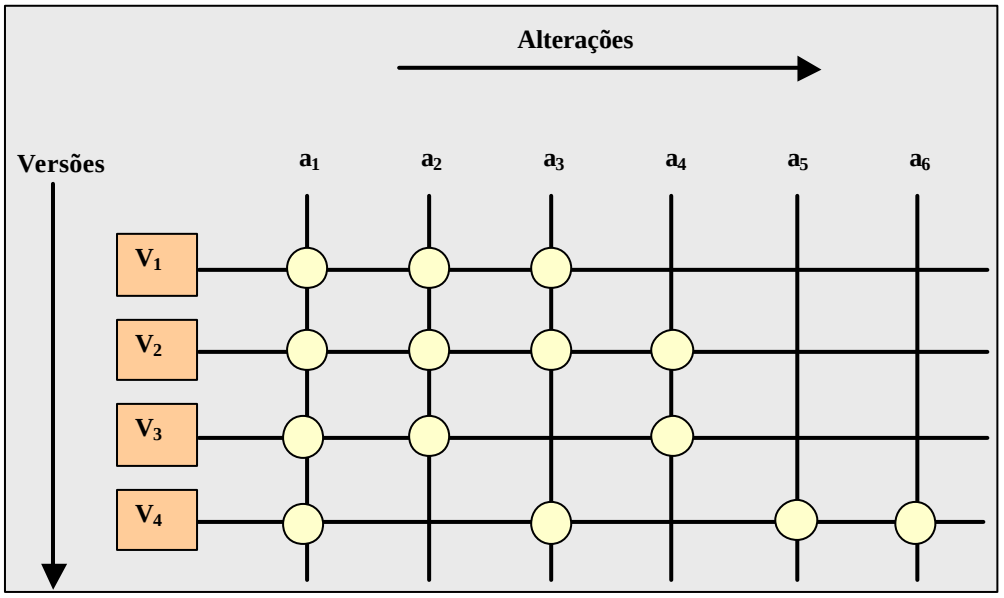


Figura 2.9 - Matriz de representação com alterações explícitas [Conradi e Westfechtel 1998]

A aplicação de uma alteração é indicada por um círculo em um ponto de interseção, ou seja, por exemplo, a versão 4 (V_4) de um item sofreu as alterações a_1, a_3, a_5 e a_6. Outra forma de representação está ilustrada na **Figura 2.10**, onde **b** representa uma *baseline*. Ambas as formas de representação expressam explicitamente as alterações aplicadas para produzir uma versão.

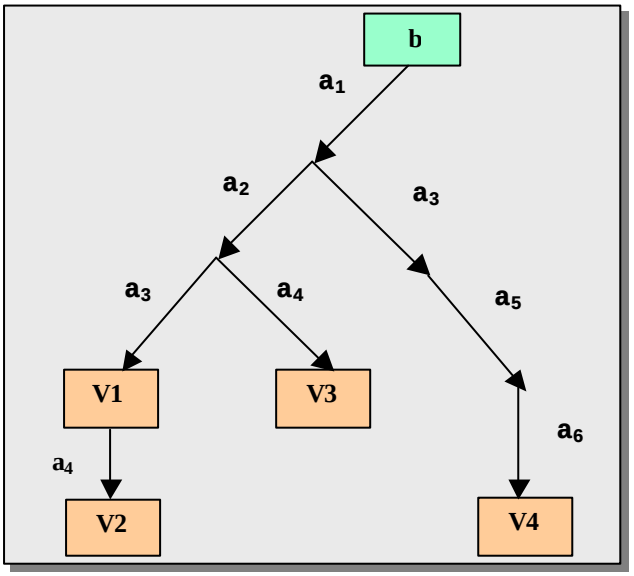


Figura 2.10 - Grafo de versão com alterações explícitas [Conradi e Westfechtel 1998]

As alterações são explicitadas somente no controle de versão baseado na alteração. Com base no que foi apresentado sobre espaço do produto e espaço da versão, é importante falar de alguma forma de integração de ambas as representações. Assim, a Seção seguinte descreve e ilustra algumas dessas formas de integração através dos grafos AND/OR.

2.5. Grafos AND/OR

Os grafos AND/OR (compostos por nós AND e OR) constituem um modelo genérico para representar a integração do espaço do produto e espaço da versão [Conradi e Westfechtel 1998]. Pode-se também fazer uma distinção entre arestas AND e arestas OR, que se originam de nós AND e nós OR, respectivamente. O grafo de um produto que não está sob o controle de versões pode ser representado por um grafo AND/OR consistindo, exclusivamente, de nós e arestas AND. Por outro lado, os objetos sendo controlados, e suas versões, são representados por nós OR e nós AND, respectivamente.

As **Figuras 2.11** a **2.13** ilustram uma classificação de modelos de versão de acordo com a ordem de seleção do produto e da versão:

- **produto primeiro** – é uma organização em que a composição do produto é selecionada primeiro e, depois, as versões dos componentes. A **Figura 2.11**

representa esse tipo de organização. Essa abordagem é seguida, por exemplo, pelo SCCS (*Source Code Control System*) e RCS;

- **versão primeiro** – neste caso, a versão do produto é selecionada primeiro e, depois, são selecionadas as versões dos componentes pertencentes àquela versão. Este exemplo está ilustrado na **Figura 2.12**;
- **intercalada** – para esse tipo de organização as seleções AND e OR são realizadas de forma intercalada, como ilustrado na **Figura 2.13**.

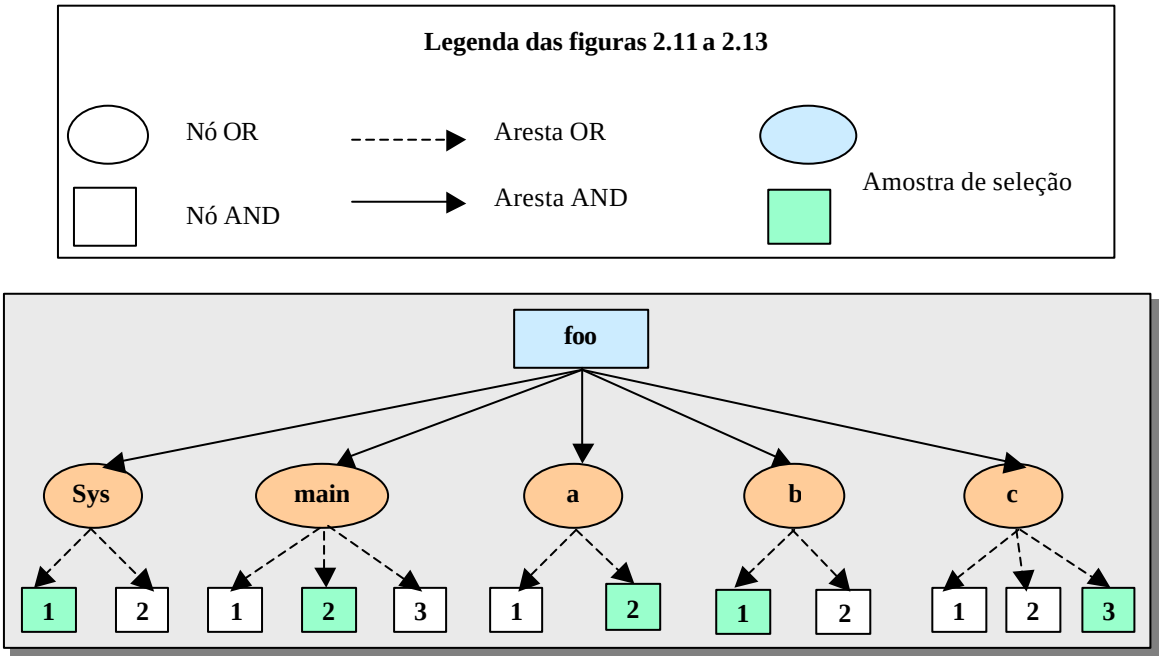


Figura 2.11 - Grafo AND/OR e ordem de seleção: produto primeiro [Conradi e Westfechtel 1998]

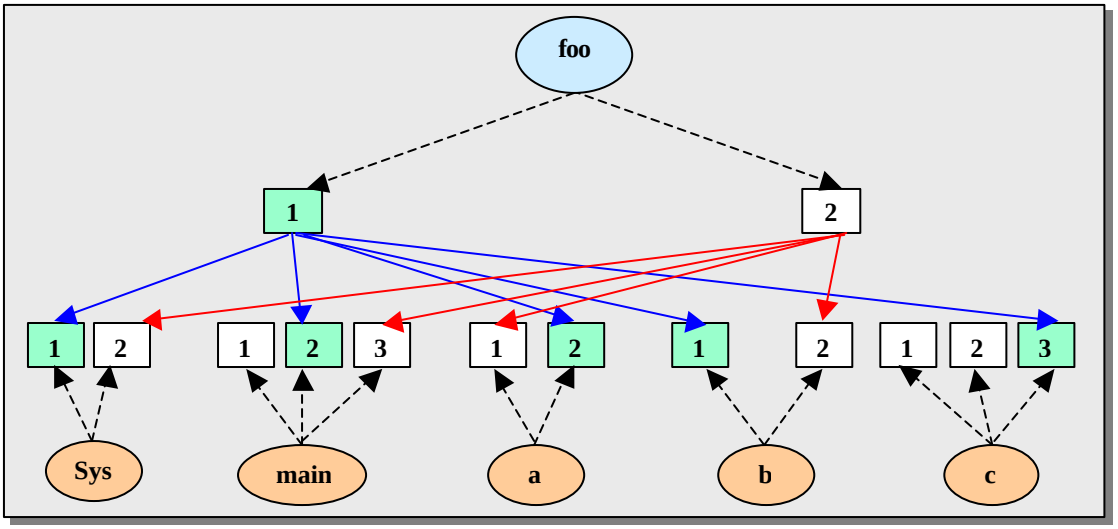


Figura 2.12 - Grafo AND/OR e ordem de seleção: versão primeiro [Conradi e Westfechtel 1998]

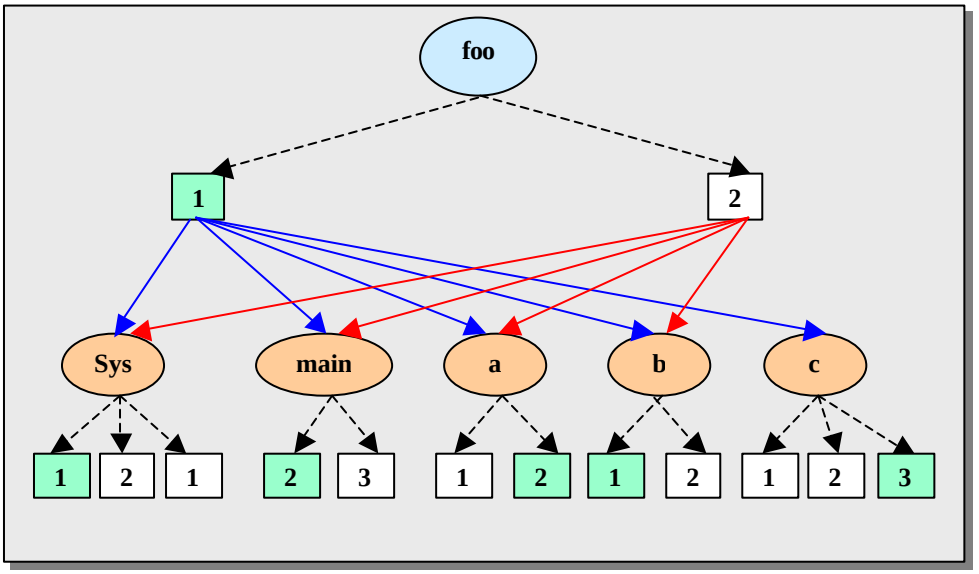


Figura 2.13 - Grafo AND/OR e ordem de seleção: intercalado [Conradi e Westfechtel 1998]

É importante observar que o versionamento para todas as representações mostradas anteriormente, foi considerado somente no nível de granularidade grossa, ou seja, o nível de composição do produto.

2.6. Deltas

Para economizar espaço nos dispositivos de armazenamento secundários (como disco, por exemplo) a maioria das ferramentas de controle de versão armazena versões na forma de deltas. Delta é o conjunto de diferenças entre duas versões subseqüentes de um mesmo arquivo [Conradi e Westfechtel 1998]. Em outras palavras, delta pode ser definido como uma seqüência de comandos de edição que transformam uma *string* em outra [Tichy 1985].

Os sistemas de *software* consistem de muitos componentes e, durante o desenvolvimento e manutenção de um sistema, os componentes sofrem mudanças. Os componentes resultantes de transformações são chamados de versões, e essas podem ser distinguidas em dois tipos: variantes (versões paralelas) e revisões (versões seriais) [Bieliková 1999], já descritas na sub-seção 2.4.1. As versões de um componente de *software* são muitas vezes organizadas em uma árvore histórica (ancestral), a qual tem uma versão principal (*root*) que representa a primeira versão de um componente de *software*. A árvore histórica inicial é simples, consistindo somente de uma *branch* (ou ramificação), chamada tronco, mas com o prosseguimento do desenvolvimento, *branches* laterais podem surgir. As *branches* podem surgir nas seguintes situações:

- no desenvolvimento simultâneo entre múltiplos usuários;
- no desenvolvimento distribuído em vários *sites*;
- quando versões anteriores ainda precisam ser melhoradas (alteradas);
- quando há a necessidade de criar versões com propósitos alternativos.

Uma tarefa essencial do SCM é armazenar a árvore histórica de versões eficientemente e, para isso, várias técnicas têm sido propostas [Bieliková 1999]. A idéia principal para economizar espaço de armazenamento é a seguinte: se uma versão for derivada de uma outra versão (ou seja, um sucessor na árvore histórica) então as duas versões provavelmente têm uma grande parte em comum e um pequeno número de diferenças. Portanto, para economizar espaço, uma versão é armazenada completa e a outra em forma de delta, ou seja, armazena-se somente as diferenças de conteúdo. Durante a análise de armazenamento dos deltas, dois aspectos são importantes: **a)** como gerar um delta entre dois arquivos **b)** como aplicar o delta na árvore histórica de versão, ou seja, qual versão armazenar de forma completa e em qual versão aplicar o delta.

Atualmente, as técnicas de armazenamento de deltas mais conhecidas são o delta *reverse* e o delta *forward*. Para gerar um delta entre dois arquivos, um algoritmo para isolação da seqüência comum entre os dois é usado [Tichy 1985].

A técnica mais simples de delta é o delta *forward*, ou seja, as diferenças entre dois arquivos são criadas de forma que o delta seja aplicado para transformar a versão mais velha em uma mais nova. Os deltas *forward* são calculados entre a primeira e a segunda versão, a segunda e a terceira, e assim por diante, pois somente a primeira versão é armazenada completa, e as posteriores são armazenadas como deltas *forward* (como acontece, por exemplo, no SCCS). Já os deltas *reverse* são calculados a partir da versão mais recente até a versão requerida na árvore histórica de versões, pois, neste caso, a última versão é armazenada completa e as anteriores são armazenadas como deltas *reverse* (essa técnica é utilizada, por exemplo, no RCS).

No caso de somente versões mais recentes serem requeridas com mais freqüência, a técnica *forward* apresenta desvantagens em relação à técnica *reverse*, pois o tempo de reconstituição de uma versão mais recente é maior através do delta *forward* [Bieliková 1999].

No contexto de *software*, em termos de conteúdo, podemos distinguir entre três tipos de deltas [Conradi e Westfechtel 1998]:

- **delta simétrico** - um delta simétrico entre duas versões v_1 e v_2 consiste de propriedades específicas tanto de v_1 quanto de v_2 ($v_1 \setminus v_2$ e $v_2 \setminus v_1$, respectivamente, onde a "\" denota o conjunto de diferenças);
- **delta embutido** - neste caso, todas as versões são armazenadas de forma a sobrepor as propriedades comuns que são compartilhadas. A **Figura 2.14** mostra a diferença entre delta simétrico e delta embutido;
- **delta direcionado** - um delta direcionado é uma seqüência de operações de alterações $op_1 ... op_m$ que, quando aplicadas a uma versão v_1, produz uma versão v_2 (**Figura 2.15**). Esse tipo de delta é usado, por exemplo, no RCS.

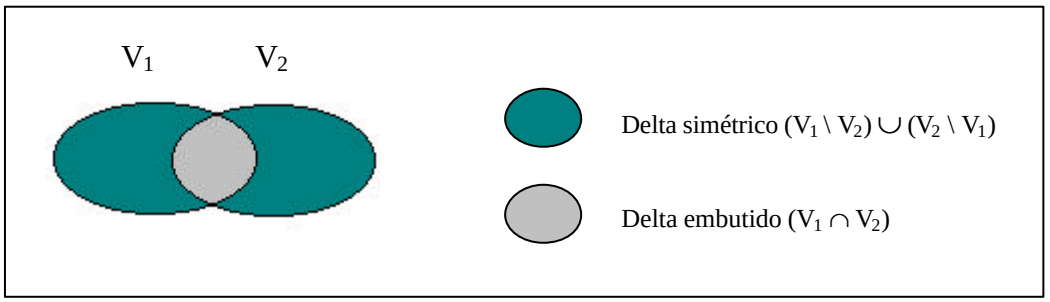


Figura 2.14 - Diferença entre delta simétrico e delta embutido

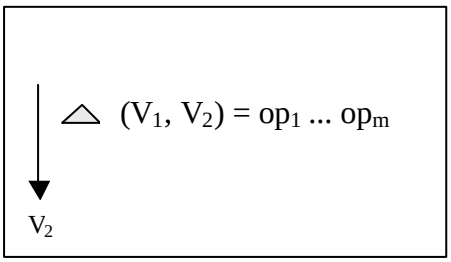


Figura 2.15 – Delta direcionado

O RCS, que é baseado em deltas direcionados, reconstrói versões de arquivos texto a partir da versão mais recente através da aplicação da técnica de delta *reverse* (no tronco principal do grafo) e deltas *forward* (nas ramificações).

2.7. Exemplos de ferramentas de gerenciamento de versão

No contexto de *software*, o gerenciamento de versão envolve o controle de grande quantidade de informação e garantia de que mudanças no sistema sejam registradas e controladas [Sommerville 1995]. Existem várias ferramentas disponíveis para apoiar esse processo, como SCCS, RCS e

CVS, por exemplo. Nas sub-seções seguintes são descritas apenas as ferramentas SCCS e RCS, pois o CVS será tratado no próximo Capítulo com mais detalhes, uma vez que este foi o sistema de gerenciamento de versões utilizado para dar suporte às funcionalidades básicas de controle de versão oferecidas na ferramenta *VersionWeb*. Um princípio comum a todas essas ferramentas (SCCS, RCS e CVS) é de que elas controlam um repositório de itens de configuração cujo conteúdo é imutável, ou seja, não pode ser alterado.

Para trabalhar sobre um item de configuração, deve ser feita uma cópia do mesmo (*checkout*) para um diretório de trabalho. Após o término das alterações, o item é então colocado de volta no repositório (*checkin*) e uma nova versão é criada. A maioria dos sistemas de gerenciamento de versão fornecem um conjunto básico de funções, como por exemplo: identificação de versão e *release*, controle de alterações, gerenciamento de armazenamento e registro histórico de mudanças [Sommerville 1995]. Normalmente, a primeira versão de um sistema é chamada 1.0 (*release*), e as versões subsequentes são chamadas 1.1, 1.2 e assim por diante, mas a qualquer momento pode-se desejar criar uma *release* 2.0 e o processo se inicia novamente em 2.1, 2.2 e assim por diante.

2.7.1. SCCS

O SCCS, ou *Source Code Control System*, foi desenvolvido em 1972 por Mac Rochkind como um sistema para controlar o desenvolvimento de código fonte [Bolinger e Bronson 1995]. O SCCS não é apenas para programadores usarem. Ele pode ser usado para qualquer arquivo texto e é especialmente útil quando se necessita manter, de forma confiável, mais de uma versão de um arquivo, modificar um arquivo freqüentemente e necessitar de uma forma confiável para recuperar ou visualizar versões anteriores.

Como já foi mencionado neste Capítulo, para economizar espaço de armazenamento a maioria das ferramentas de controle de versão armazena as versões na forma de deltas. O SCCS é um precursor do RCS e usa deltas intercalados para armazenar as versões de um arquivo [Tichy 1985]. De modo geral, um arquivo que contém deltas intercalados é particionado em blocos de linhas, onde cada bloco tem um cabeçalho que especifica a quais revisões ele pertence. Todos os blocos são ordenados de forma que uma só passagem pelo arquivo selecione todas as linhas pertencentes a uma dada revisão. Além disso, em geral, o tempo de reconstrução para todas as

revisões é o mesmo: todos os cabeçalhos devem ser vistoriados e os blocos associados devem ser igualmente copiados ou omitidos.

O SCCS armazena as alterações feitas em um arquivo chamado *s.arquivo*, onde *arquivo* é o nome do arquivo fonte que está sendo controlado. Esse arquivo é normalmente armazenado em um diretório chamado SCCS, que está abaixo do diretório de trabalho do usuário quando o comando sccs é usado. As alterações são gravadas nesse arquivo como blocos construtores, onde cada conjunto de alterações depende das revisões anteriores.

O SCCS tem grande aplicabilidade para tarefas de administração de sistema onde os arquivos para adição de usuários, tais como *passwd* e *nfs* são freqüentemente editados quando usuários são adicionados ao sistema de redes. Neste caso, o SCCS fornece uma trilha de auditoria satisfatória.

A principal diferença entre o RCS e o SCCS é o método de gerenciamento de armazenamento usado. O SCCS armazena a primeira versão de um sistema completa e, as posteriores, através de deltas. Já o RCS, armazena a versão mais recente completa e as anteriores através de deltas. O SCCS não faz operações de *merging*, não permite a criação de nomes simbólicos para as revisões e não permite uma entrada para comentários sobre um alteração que foi feita (*log message*). Por outro lado, uma limitação tanto do RCS como do SCCS é que ambos foram projetados para trabalharem somente com arquivos texto [Sommerville 1995].

2.7.2. RCS

O RCS, ou *Revision Control System*, foi desenvolvido por Walter F. Tichy na Universidade de Purdue em Indiana por volta de 1980, e parte do seu sistema usa programas que se originaram por volta de 1970 [Bolinger e Bronson 1995]. O RCS é um sistema de controle de revisão que auxilia no gerenciamento de versões geradas de arquivos durante o desenvolvimento de um *software* [Tichy 1985]. Ele gerencia revisões de documentos texto, em particular programas fonte, documentação e dados de teste. Além disso, o RCS automatiza o armazenamento e a recuperação de informações, faz a identificação de revisões e fornece mecanismos de seleção para composição de configurações. O RCS continua sendo o sistema de controle de versão mais popular em uso atualmente devido a sua simplicidade, eficiência e disponibilidade [Bolinger e Bronson 1995].

Geralmente, todos os sistemas de controle de versão oferecem alguma forma para os desenvolvedores ou gerenciadores manterem diferentes versões de documentos fontes. Na maioria das vezes, a capacidade de recuperar as versões originais dos programas e de manter as diferentes versões que são geradas é muito importante e, usando o RCS, os desenvolvedores podem fazer isso de forma eficiente.

A principal função do RCS é gerenciar grupos de revisão. Um grupo de revisão, como mencionado na Seção 2.2 do Capítulo 2, é um conjunto de documentos texto, chamado revisões, as quais evoluem uma da outra. Uma nova revisão é criada pela edição manual de outra revisão já existente. O RCS organiza as revisões em uma árvore ancestral (sistema de cópia de arquivos do mais velho para o mais novo). A revisão inicial é a raiz da árvore, e as conexões entre elas indicam de qual revisão uma outra foi gerada. A **Figura 2.16** ilustra o exemplo de uma árvore de revisões armazenada pelo RCS.

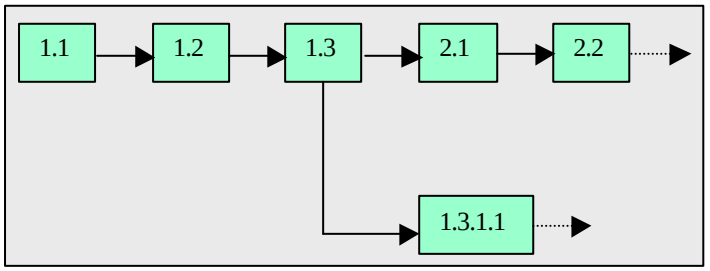


Figura 2.16 - Árvore ancestral de revisões com uma *branch* lateral [Tichy 1985]

Embora o RCS tenha sido originalmente voltado para programas, ele é muito usado para qualquer arquivo texto que é revisado freqüentemente e que revisões anteriores tenham que ser preservadas. O RCS não permite o acesso simultâneo à mesma revisão de um arquivo por parte dos desenvolvedores, pois somente uma pessoa por vez pode fazer *checkin* de uma revisão para gerar uma versão sucessiva. Quando uma pessoa faz *checkout* de uma versão ela obtém um *lock* para essa versão e outras pessoas só têm acesso a essa versão para leitura. Se por questões de emergência uma outra pessoa necessitar trabalhar sobre essa mesma versão, existem três alternativas [Tichy 1985]:

1. encontrar a pessoa que está com o *lock* e pedir para liberar;
2. fazer o *checkout* da revisão, modificá-la e fazer o *checkin* em uma *branch* para depois efetuar o *merging* das alterações;

3. quebrar o *lock*.

O RCS possui algumas vantagens como fácil administração, uso eficiente em sistemas simples e fornece o *locking* de arquivos. Isso impossibilita a concorrência, mas por outro lado evita perda e sobreposição de informações dos arquivos.

Os deltas aplicados pelo RCS são baseados em linha, o que significa que os únicos comandos de edição permitidos são "inserção" e "deleção" de linhas. Portanto, se um simples caractere em uma linha for alterado, considera-se que a linha inteira foi alterada.

Os arquivos RCS são chamados *arquivo,v*, ou seja, é o nome do arquivo fonte acrescido de ",v" no final ("v" de versão). Esse arquivo contém todas as revisões geradas, data, hora, autor e um comentário sobre a alteração que foi efetuada para cada revisão.

O RCS armazena o código fonte da versão mais recente de um sistema como uma versão (mestre), que é criada também a partir de uma versão mestre anterior [Tichy 1985]. Quando uma nova versão mestre é criada, a versão anterior é apagada e substituída por uma especificação das diferenças entre ela e a nova versão mestre, ou seja, pela especificação de deltas. Os deltas são com certeza bem menores que o código fonte de uma versão inteira do sistema. Este método de gerenciamento de armazenamento reduz o espaço em disco requerido pelo gerenciamento de versão. Em vez de armazenar todos os códigos fontes de todas as versões mestre, o RCS necessita armazenar somente uma versão mestre (a mais recente) e as outras versões são armazenadas como deltas *reverse* (reverso). Se uma particular versão de um sistema for requerida, é solicitado ao usuário do RCS o número da versão, a data ou o proprietário, e o RCS então aplica os deltas de forma a recuperar a versão requerida. A técnica de delta *reverse* é aplicada somente no tronco da árvore principal de versões, pois nas *branches* o RCS utiliza a técnica de delta *forward* (para frente).

Dessa forma, a recuperação de uma revisão em uma *branch* ocorre da seguinte forma: **a)** extrai-se a revisão mais recente do tronco principal (revisão mestre) **b)** aplica-se deltas *reverse* até que a revisão *fork* (ou seja, a revisão que deu origem à *branch*) da *branch* seja obtida **c)** aplica-se deltas *forward* até que a revisão desejada da *branch* seja alcançada. O RCS também tem a capacidade de efetuar *merging* de versões, e isso pode ser bastante útil quando mudanças independentes forem realizadas em um sistema por diferentes pessoas.

2.8. Considerações finais

Neste Capítulo foram descritas as principais funcionalidades do SCM durante o desenvolvimento de um produto de *software*, sua atuação no apoio ao gerenciamento das alterações realizadas durante todo o ciclo de vida do *software* e auxílio aos desenvolvedores. Foram apresentados também os principais conceitos da estrutura de um *software* e sua representação (que constitui o espaço do produto), bem como alguns modelos de versão de *software* e suas formas de representação.

Embora os conceitos de SCM sejam amplamente empregados em ambientes de engenharia de *software*, eles se aplicam também a ambientes de autoria. O alvo de interesse deste trabalho se concentrou especificamente no ambiente de autoria da WWW. Para atingir os objetivos deste projeto de mestrado tornou-se necessário a aplicação de muitos dos conceitos estudados, como controle de versões, auditoria, revisões, *branches*, dentre outros.

Foi visto também que o SCCS e o RCS possuem algumas limitações tais como: não fazem o gerenciamento de arquivos binários, não permitem acesso ao repositório através da rede e nem o trabalho paralelo entre os desenvolvedores. Essas limitações certamente impossibilitariam o uso de tais ferramentas para o desenvolvimento deste trabalho, pois as páginas *Web* podem envolver vários tipos de dados (não somente ASCII) e o trabalho paralelo entre os desenvolvedores (que podem estar em qualquer lugar do mundo) por meio da *Web* deve ser favorecido (com acesso simultâneo aos arquivos e acesso ao repositório através da rede). Essas limitações são superadas pelo CVS que será descrito no Capítulo seguinte.

Neste projeto, o espaço do produto e seus relacionamentos não foram considerados para o controle de versões, mas sim o espaço da versão para cada arquivo que compõe o produto. No nosso caso, o produto se refere a uma página HTML (ou um *site* ou um conjunto de páginas), que inclui diferentes tipos de objetos como imagens, outras páginas HTML, *links* para outros documentos, etc. Assim, se os arquivos que compõem a página estão sob o controle de versão, o tratamento e gerenciamento das alterações são realizados sobre esses arquivos. No próximo Capítulo é apresentado o CVS, *software* de controle de versão que serviu de base para o gerenciamento das páginas através da *Web*.

3. CVS

3.1. Considerações iniciais

O auxílio automatizado para o gerenciamento das diversas versões dos artefatos, durante o desenvolvimento e a evolução de um produto de *software*, é tido como imprescindível quando envolve equipes e trabalhos de maior porte. Existem várias ferramentas automatizadas para o gerenciamento de versões de arquivos fontes (já citadas no capítulo de Introdução desta dissertação), mas as mais conhecidas são o SCCS e o RCS, que já foram descritas no Capítulo anterior.

Atualmente, uma ferramenta que tem sido muito referenciada é o CVS. Este é um sistema de versões concorrentes que, além de gerenciar arquivos texto, gerencia também arquivos binários e possui algumas características adicionais ao RCS e ao SCCS as quais serão descritas na Seção seguinte. Além disso, o CVS executa em várias plataformas como, por exemplo, Unix, Windows 95, Windows NT, Macintosh e VMS.

Assim como o SCCS e o RCS, o CVS também permite a reconstrução de qualquer versão anterior de um determinado arquivo e localiza as modificações feitas por um determinado autor juntamente com a data e a hora em que foi feita a modificação. Além de ser uma ferramenta mais nova, os recursos do CVS possibilitam atender às necessidades funcionais de aplicativos na *Web*, tais como distribuição e concorrência. Dessa forma, neste capítulo será apresentado brevemente o estudo realizado sobre CVS, que fundamentou o projeto desenvolvido.

As seções seguintes descrevem brevemente as principais características do CVS, a estrutura do seu repositório, o uso de alguns comandos para operações mais comuns em sistemas de controle de versão, como usar o CVS com repositórios remotos e algumas comparações relacionadas ao RCS e ao SCCS, destacando as vantagens e desvantagens de cada um.

3.2. O sistema CVS

Assim como o RCS e o SCCS, o CVS é um sistema de controle de versão de código fonte bastante usado. O CVS foi originalmente desenvolvido por Dick Grune em 1986, e consistia de um conjunto de scripts shell do UNIX. Em 1989, ele foi projetado e codificado na linguagem de programação C por Brian Berlinger [Cederqvist 1993] e, mais tarde, Jeft Polk projetou os módulos de suporte à geração de *branches* no CVS.

Em grandes projetos de desenvolvimento de *software*, é comum que duas ou mais pessoas tenham a necessidade de modificar o mesmo arquivo ao mesmo tempo [Hung e Kunz 1992; Berliner 1990]. Com o RCS ou SCCS isso é impossível, pois os arquivos são bloqueados (*locked*) ao fazer o *checkout* para um diretório de trabalho, fazendo com que somente uma pessoa possa obter uma cópia do arquivo para escrita por vez.

Embora o bloqueio de arquivos seja desejável na teoria, isso traz conseqüências indesejáveis ao grupo de desenvolvedores, pois esse bloqueio causa a serialização do processo de desenvolvimento fazendo com que o restante do grupo fique esperando a pessoa que está com o arquivo fazer as alterações e efetuar o *commit*. Feito isso, o arquivo é liberado e, então, um outro desenvolvedor pode obter uma cópia do arquivo para escrita.

Com o sistema CVS, cada desenvolvedor pode obter uma cópia de uma versão do arquivo para escrita de todo o código fonte de um projeto para dentro do seu diretório de trabalho sempre que quiser. O CVS é um sistema de controle de versões que permite gravar o histórico de arquivos fonte [Cederqvist 1993; CVS 1999] e, como qualquer outra ferramenta de gerenciamento de configuração de *software* ou de controle de versões, ele possui um repositório central que armazena as cópias mestres de todos os arquivos que estão sob o gerenciamento de versões.

O CVS foi projetado para manter a localização de alterações feitas nos arquivos por grupos de desenvolvedores e é um *frontend* do RCS (ou seja, foi construído no topo do RCS) que fornece algumas características adicionais como: permitir o trabalho paralelo entre desenvolvedores através do acesso concorrente aos arquivos, dar suporte à geração de *releases* e evitar sobreposição ou perda de informações quando duas ou mais pessoas estão trabalhando no mesmo arquivo simultaneamente [Hung e Kunz 1992].

Dentre as funcionalidades básicas do CVS, podemos citar:

- mantém um histórico de todas as alterações feitas em cada árvore de diretório que ele gerencia; usando esse histórico, o CVS pode recriar estados anteriores da árvore, ou mostrar a um desenvolvedor quando, porque e por quem uma alteração foi feita;
- armazena e recupera versões anteriores de arquivos eficientemente;
- fornece controle de arquivos através da rede de forma transparente para grupos de desenvolvedores;
- suporta desenvolvimento paralelo, permitindo que mais de uma pessoa trabalhe em um mesmo arquivo ao mesmo tempo;
- fornece acesso seguro às árvores de diretórios de *hosts* remotos usando protocolos Internet.
- permite adicionar, remover e alterar arquivos e diretórios do repositório (hierarquia de diretórios);
- permite agrupar uma coleção de arquivos relacionados em módulos e, então, o módulo passa a ser gerenciado, em vez dos arquivos separadamente;
- tags simbólicas podem ser associadas a um conjunto específico de revisões;
- executa em várias plataformas como Unix, Windows 95, Windows NT, Macintosh e VMS [CVS 1999].

O CVS salva todas as informações de controle de versão em arquivos RCS armazenados em uma hierarquia de diretórios, chamada **repositório**, sendo que este é separado do diretório de trabalho do usuário. A Seção seguinte descreve a estrutura do repositório CVS e como os arquivos são armazenados nele.

3.3. O repositório CVS

A noção de repositório é fundamental para um sistema SCM, pois é nele que estão todas as cópias mestres dos arquivos que estão sob o controle de versão. O repositório CVS armazena uma cópia completa de todos os arquivos (no formato RCS) que estão sob o controle de versão e, normalmente, nenhum dos arquivos no repositório é acessado diretamente. Em vez disso, deve-se executar comandos CVS para obter uma cópia dos arquivos e então trabalhar nessa cópia.

Quando é feito um conjunto de alterações, deve-se submetê-las de volta ao repositório (*commit*) para que as alterações realizadas se tornem disponíveis às outras pessoas. O repositório então

contém as alterações realizadas sobre o arquivo e nele são gravadas apenas as modificações feitas, quando foram feitas, quem as efetuou, dentre outras informações. O repositório não é um subdiretório do diretório de trabalho ou vice versa; eles estão localizados separadamente em diretórios diferentes.

Há várias formas de informar ao CVS onde seu repositório está, ou seja, em qual diretório ele está armazenado. As formas mais usuais são:

- nomear o repositório explicitamente na linha de comando, com a opção *-d* (para diretório): *cvs -d /usr/local/cvsroot checkout yoyodyne/tc*; esse comando faz o *checkout* da árvore de diretório *yoyodyne/tc* que está armazenada no repositório CVS em */usr/local/cvsroot*;
- definir a variável de ambiente *$CVSROOT* para o caminho absoluto do repositório, */usr/local/cvsroot* neste exemplo. Usuários *csh* e *tcsh* devem ter a seguinte linha em seus arquivos *.cshrc* ou *.tcshrc*, respectivamente: *setenv CVSROOT /usr/local/cvsroot*. Já usuários *sh* e *bash* deverão ter as seguintes linhas em seus arquivos *.profile* ou *.bashrc*, respectivamente:

```
CVSROOT=/usr/local/cvsroot
export CVSROOT
```

Um repositório especificado com a opção *-d* sobrescreve a variável de ambiente *$CVSROOT*. Uma vez que se tenha feito *checkout* de algum arquivo ou módulo para o diretório de trabalho, o CVS saberá onde o repositório está (a informação é gravada no arquivo *CVS/Root* dentro do diretório de trabalho).

O repositório é dividido em duas partes: *$CVSROOT/CVSROOT* contém os arquivos administrativos para o CVS. Os outros diretórios que estão em *$CVSROOT* contêm os módulos definidos pelo usuário. Um exemplo de uma estrutura do repositório CVS é ilustrada na **Figura 3.1**.

Os arquivos históricos (nome do arquivo fonte acrescido de ",v" no final) do CVS contêm informações suficientes para recriar qualquer revisão do arquivo, localizar o autor, data e hora da revisão, além dos comentários da alteração realizada para melhor identificar a razão da geração

daquela revisão. Todos os arquivos históricos são criados apenas para leitura e essa permissão não deve ser alterada devido às informações neles contidas para a recuperação de suas revisões. Os diretórios dentro do repositório podem ser habilitados para escrita por pessoas que tenham permissão para modificar os arquivos dos diretórios.

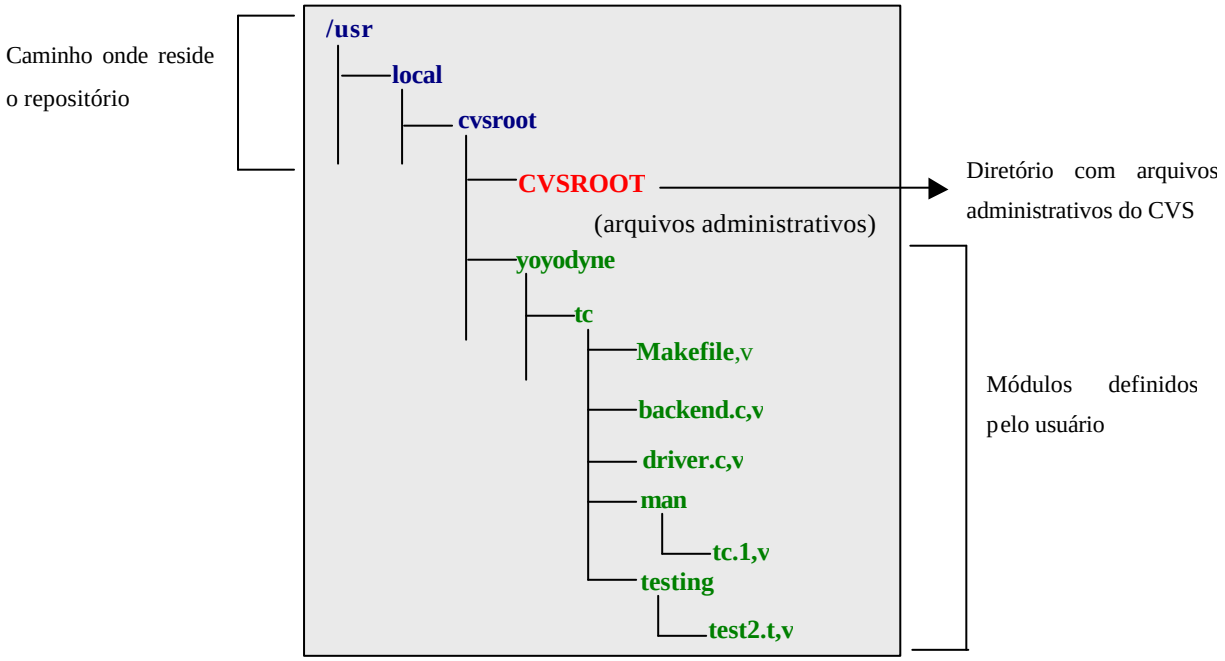


Figura 3.1 - Estrutura de um repositório CVS [Cederqvist 1993]

O CVS mantém as permissões de arquivo para novos diretórios que são adicionados dentro da árvore de diretórios, mas as permissões devem ser dadas manualmente quando um novo diretório tiver permissões diferentes do seu diretório pai.

Para criar um repositório CVS deve-se executar o comando *cvs init*. Esse comando definirá um repositório vazio no diretório especificado, como por exemplo: *cvs -d /usr/local/cvsroot init*. Após executado esse comando, o usuário poderá definir os módulos que irão residir no repositório e também importar projetos já existentes para o repositório. A Seção seguinte descreve como iniciar um projeto com o CVS.

3.4. Exemplo de uma simples sessão de trabalho com o CVS

Supondo que já exista um repositório CVS em */usr/local/cvsroot* criado pelo comando *cvs init* descrito na Seção anterior, a seguir são descritos os comandos mais comumente solicitados em CVS, tais como: colocar um diretório com arquivos no repositório, obter uma cópia de um

arquivo para alteração, colocar o arquivo alterado de volta ao repositório CVS, remover um diretório de trabalho e visualizar as diferenças entre duas revisões de um arquivo.

3.4.1. Criando uma árvore de diretórios no repositório CVS

Geralmente, quando o usuário começa a usar o CVS é provável que ele já tenha vários projetos ou arquivos que podem ser colocados sob o controle do CVS. Neste caso, a forma mais fácil é usar o comando *cvs import* para colocar esses projetos ou arquivos no repositório. Se os arquivos que o usuário deseja colocar sob o controle do CVS estão, por exemplo, no diretório *wdir*, e o usuário quer que esses arquivos apareçam em *$CVSROOT/yoyodyne/rdir*, os seguintes passos devem ser realizados:

1. Posicione dentro do diretório *wdir*: $ *cd wdir*
2. Com o comando *cvs import*, coloque os arquivos no repositório: *$cvs import -m "Imported sources" yoyodyne/rdir yoyo start*

Esse comando cria a estrutura de diretório *yoyodyne/rdir* com todos os arquivos e diretórios contidos em *wdir* abaixo na hierarquia do repositório em */usr/local/cvsroot*.

Para um novo projeto, a forma mais fácil é criar uma estrutura de diretórios vazia, tal como mostram os comandos abaixo, por exemplo:

```
$ mkdir tc
$ mkdir tc/man
$ mkdir tc/testing
```

Depois de fazer isso, pode-se usar o comando *import* para criar a estrutura de diretório correspondente dentro do repositório:

```
$ cd tc
$ cvs import -m "Created directory structure" yoyodyne/tc yoyo start
```

A *string yoyo* é um nome simbólico dado ao diretório criado. Para adicionar arquivos e novos diretórios à estrutura gerada, deve-se usar o comando *add* (esse comando e outros serão descritos na sub-seção 3.4.6).

3.4.2. Definindo um módulo

É comum trabalhar com o conceito de módulos no CVS. Geralmente, um módulo é definido para agrupar arquivos e diretórios relacionados, mas essa tarefa não é estritamente necessária. Para criar um módulo, os comandos abaixo são suficientes:

1. Obtenha uma cópia do arquivo modules que está no diretório *CVSROOT*:

   $ *cvs checkout CVSROOT/modules*
   $ *cd CVSROOT*

2. Edite o arquivo e insira uma linha que define o módulo. Para definir um módulo de nome *tc*, por exemplo, coloque a seguinte linha no arquivo modules: *tc yoyodyne/tc*. Faça o *commit* das alterações para o arquivo modules:

   $ *cvs commit -m "Added the tc module" modules*

3. Remova o diretório *CVSROOT*:

   $ *cd ..*
   $ *cvs release -d CVSROOT*

Depois de definir o módulo acima, pode-se fazer o *checkout* da árvore de diretório *yoyodyne/tc* usando o nome do módulo dado a ela, como por exemplo: *cvs checkout tc*.

3.4.3. Obtendo o fonte para edição

Como os arquivos dentro do repositório são imutáveis, para se trabalhar sobre eles é preciso obter uma cópia para o diretório de trabalho. O comando utilizado para essa operação é o *checkout*:

$ *cvs checkout tc*

Esse comando irá criar um novo diretório chamado *tc* no diretório corrente do usuário e dentro dele estão todos os arquivos e diretórios que se deseja trabalhar.

```
$ cd tc
$ ls
CVS  Makefile  backend.c  driver.c  frontend.c  parser.c
```

O diretório CVS é usado internamente pelo CVS e, normalmente, não se deve modificar ou remover nenhum dos arquivos contidos nele. Dentro desse diretório contém os seguintes arquivos:

- **Root:** este arquivo contém o caminho onde reside o repositório CVS.
- **Repository:** este arquivo contém o diretório dentro do repositório ao qual corresponde o diretório corrente. Ou seja, se o repositório está em: *:local:/usr/local/cvsroot* e o usuário fizer o *checkout* do módulo *yoyodyne/tc* (*cvs -d :local:/usr/local/cvsroot checkout yoyodyne/tc*), então **Root** conterá: *:local:/usr/local/cvsroot* e **Repository** conterá: */usr/local/cvsroot/yoyodyne/tc.*
- **Entries:** este arquivo lista os arquivos e diretórios no diretório de trabalho.

As informações contidas nos arquivos acima são utilizadas pelo CVS quando se faz o *commit*.

3.4.4. Colocando as alterações no repositório

Depois de fazer todas as alterações em um determinado arquivo, por exemplo *backend.c*, deve-se executar o comando *commit* para que as modificações sejam visíveis por outras pessoas:

```
$ cvs commit backend.c
```

Com o comando acima, o CVS abre um editor para permitir que o usuário entre com uma mensagem de alteração. Para evitar que um editor seja aberto, pode-se entrar com a mensagem na própria linha de comando: $ *cvs commit -m "Added na optimization pass" backend.c*. Depois de fazer todas as alterações necessárias e colocá-las de volta ao repositório, basta remover a cópia dos fontes no diretório de trabalho:

```
$ cd ..
$ cvs -d release tc
```

3.4.5. Visualizando diferenças

O CVS exibe diferenças somente de arquivos texto. Portanto, para visualizar as diferenças entre duas versões do arquivo *backend.c*, por exemplo, deve-se fazer o *checkout* do arquivo e proceder da seguinte forma:

```
$ cd tc
$ cvs diff -r1.1 -r1.2 backend.c
```

Este comando exibe as diferenças entre as versões 1.1 e 1.2 do arquivo *backend.c*.

3.4.6. Adicionando, removendo e renomeando arquivos e diretórios

Durante o desenvolvimento de um projeto, o usuário necessitará eventualmente adicionar, remover e renomear arquivos e diretórios. Portanto, para adicionar um novo arquivo ao repositório, é preciso seguir os seguintes passos:

1. o usuário deverá ter uma cópia do diretório, no qual deseja incluir o arquivo, no seu diretório de trabalho;
2. criar o novo arquivo dentro da cópia de trabalho do diretório;
3. usar o comando *cvs add arquivo* para dizer ao CVS que este novo arquivo é para ser colocado sob o controle de versão; se o arquivo contém dados binários, deve-se especificar a opção *-kb* (*cvs add -kb arquivo*);
4. usar o comando *cvs commit arquivo* para de fato colocar o arquivo no repositório; outras pessoas não poderão ver esse arquivo enquanto esse passo não for realizado.

Para adicionar um novo diretório ao repositório, os mesmos passos citados acima devem ser seguidos. Quanto à remoção de arquivos de um diretório, os seguintes passos são necessários:

1. remover o arquivo da cópia de trabalho do diretório (pode-se usar o comando *rm*, ou seja, *rm arquivo*) ;
2. usar o comando *cvs remove arquivo* para dizer ao CVS que o arquivo deve ser removido;
3. usar o comando *cvs commit arquivo* para de fato remover o arquivo do repositório.

Para renomear um arquivo, deve-se proceder da seguinte forma:

1. usar o comando *mv velho novo;*

2. usar o comando *cvs remove velho*;
3. usar o comando *cvs add novo*;
4. usar o comando *cvs commit -m "Renamed velho para novo" velho novo.*

Para renomear um diretório, os seguintes passos devem ser seguidos:

1. informar às pessoas que estão usando esse diretório que irá renomeá-lo;
2. renomear o diretório dentro do próprio repositório, e não na cópia de trabalho do diretório:

```
$ cd $CVSROOT/diretorio
$  mv velho novo
```

A Seção seguinte descreve como o CVS trabalha com os números de revisões dos arquivos.

3.5. Revisões

Toda vez que o usuário altera um arquivo e usa o comando *cvs commit* para efetivar as alterações no repositório, uma nova versão ou revisão do arquivo é gerada. O número de cada revisão é gerado automaticamente pelo CVS e de forma seqüencial tal como 1.1, 1.2, 1.3 e assim por diante. Cada versão de um arquivo tem um único número de revisão. Por *default*, o número de revisão de um arquivo é 1.1 e, a cada revisão sucessiva, é dado um novo número pelo incremento em um sobre o número da direita, ou seja, 1.2, 1.3, etc. A **Figura 3.2** mostra várias revisões, com as mais recentes à direita, de acordo com o processo de armazenamento de versões do CVS.

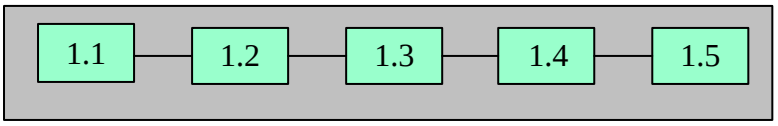


Figura 3.2 - Seqüência de revisões de um arquivo gerada pelo CVS [Cederqvist 1993]

É comum os números terminarem com mais de um dígito à direita, por exemplo, 1.3.2.2. Essas revisões representam revisões em *branches* ou ramificações geradas (mais detalhes sobre *branches* na Seção seguinte). Não existe razão para se preocupar com números de revisão, pois eles são gerados automaticamente pelo CVS, mas se o usuário desejar definir um número específico de revisão ao fazer o *commit* de um arquivo, a opção *-r* para *cvs commit* permite fazê-lo, como por exemplo: *cvs commit -r 3.0 arquivo.*

Note que o número especificado com a opção *-r* deve ser maior que qualquer número de revisão já existente do arquivo. Ou seja, se a revisão 3.0 do arquivo já existir, o usuário não pode, por exemplo, executar o comando *cvs commit -r 1.3 arquivo.* A Seção seguinte descreve o uso de *branches* e *merging* durante o desenvolvimento de um projeto por um grupo de desenvolvedores.

3.6. *Branches* e *merging*

O CVS possibilita que o usuário isole as alterações em uma linha separada de desenvolvimento, conhecida como *branch*. Quando se altera arquivos em uma *branch*, essas alterações não precisam aparecer na linha principal (*main trunk*) nem em outras *branches* pois, mais tarde, pode-se mover as alterações de uma *branch* para outra ou para a linha principal através do *merging*. A operação *merging* inclui, primeiramente, a execução do comando *cvs update -j* para atualizar as alterações dentro do diretório de trabalho. Feito isso, pode-se então fazer o *commit* para efetivamente copiar as alterações para uma outra *branch* ou para a linha principal.

Cada *branch* possui um número consistindo de um número excedente separado por inteiros decimais. O número da *branch* é criado pelo acréscimo de um inteiro ao número da revisão da qual a *branch* está sendo gerada (ou *forked*). Além disso, mais de uma *branch* pode se originar da mesma revisão. Todas as revisões em uma *branch* têm números formados pela adição de um número ordinal ao número da *branch*. A **Figura 3.3** ilustra o exemplo de uma árvore de revisões com *branches*.

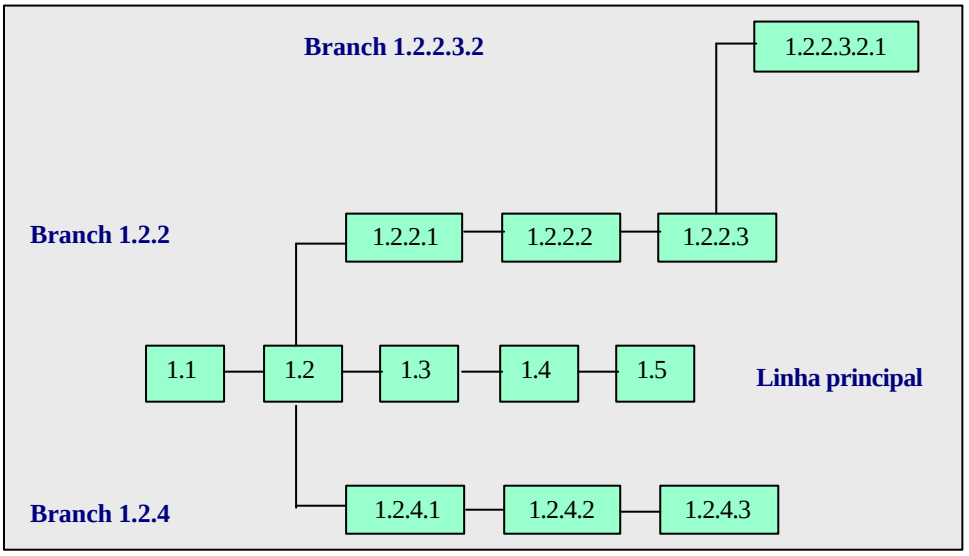
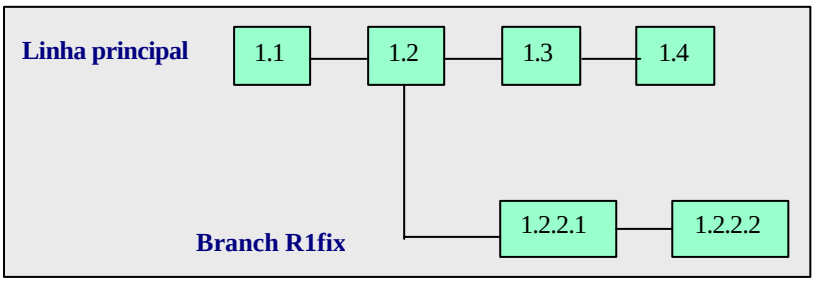


Figura 3.3 - Exemplo de várias branches geradas do mesmo arquivo [Cederqvist 1993]

O usuário pode atualizar alterações feitas em uma *branch* dentro de seu diretório de trabalho através da opção *-j* do comando *update*. Essa opção faz a união das alterações feitas do ponto onde a *branch* foi gerada até a mais nova revisão dessa *branch* (dentro do diretório de trabalho). Para melhor exemplificar a operação de *merge*, considere a seguinte árvore de revisão:



A *branch* 1.2.2 tem um nome simbólico *R1fix*. O seguinte exemplo assume que o módulo (ou diretório) *mod* contém um único arquivo, neste exemplo, *m.c*.

$ *cvs checkout mod*	# recupera a última revisão, 1.4
$ *cvs update -j R1fix m.c*	# faz a união de todas as alterações feitas na *branch*, ou seja, as alterações entre a revisão 1.2 e 1.2.2.2, dentro da cópia de trabalho do arquivo *m.c*.
$ *cvs commit -m "Included R1fix"*	# cria a revisão 1.5

Observa-se que conflitos podem resultar após uma operação *merge* e, se isso acontecer, o usuário deve resolvê-los manualmente antes de gerar uma nova revisão.

3.6.1. Quando *branches* são necessárias

Uma árvore de revisão inicial é simples. Ela consiste de apenas uma *branch*, chamada tronco ou linha principal [Tichy 1985]. Durante o desenvolvimento de um projeto, *branches* podem ser geradas. Isso acontece mais comumente nas seguintes situações:

1. Problemas temporários: um *bug* é detectado, por exemplo, na revisão 1.2, e o desenvolvimento atual está na revisão 3.1. O CVS não permite que uma revisão extra seja criada entre 1.2 e 3.1. Portanto, cria-se uma *branch* na revisão 1.2 e faz o *checkin*. O número da *branch* gerada será 1.2.1.1, 1.2.1.2, etc.
2. Desenvolvimento paralelo: algumas vezes é desejável explorar um projeto alternativo ou uma técnica de implementação diferente em paralelo à linha principal de desenvolvimento. Tal desenvolvimento resulta em uma *branch* lateral e alterações experimentais podem ser, mais tarde, movidas para a linha principal de desenvolvimento ou abandonadas.
3. Conflitos entre revisões: é comum que um desenvolvedor faça *checkout* de uma revisão e, por algum motivo demora fazer as alterações e efetuar o *commit*. Ao mesmo tempo, um outro desenvolvedor necessita fazer *checkout* da mesma revisão. Se a segunda pessoa fizer o *commit* antes da primeira, uma versão subseqüente àquela que foi feito o *checkout* será gerada. Neste caso, a primeira pessoa deverá atualizar o conteúdo da sua cópia de trabalho com a cópia do repositório (*update*) ou fazer o *commit* gerando uma *branch* lateral para depois unir as revisões (*merging*).
4. Quando versões anteriores ainda precisam ser melhoradas (alteradas).
5. Quando existe a necessidade de criar versões com propósitos alternativos.

Como o CVS permite o acesso aos arquivos através da rede, o repositório pode estar em uma máquina local ou remota. Dessa forma, a Seção seguinte descreve como proceder para trabalhar com repositórios remotos.

3.7. Repositórios remotos

A cópia de trabalho dos arquivos fontes pode estar em uma máquina diferente daquela onde reside o repositório. O uso do CVS desta maneira é conhecido como cliente/servidor. O usuário executa o CVS em uma máquina na qual ele possa "montar" seu diretório de trabalho, chamada cliente, e faz a comunicação com a máquina que contém o repositório com os arquivos, chamada servidor. Geralmente, o uso remoto do repositório é parecido com o local, exceto que o formato do nome do repositório é: *:method:user@hostname:/path/to/repository*. Os detalhes do que é exatamente necessário para fazer isso depende do método usado para se conectar com o servidor. As seções seguintes descrevem os métodos mais conhecidos.

3.7.1. Conexão com rsh

Se o usuário vai se conectar com *rsh*, o método utilizado é o *:server:* ou *:ext:*. Por exemplo, suponha que o usuário da máquina local *toe.grunge.com* seja *mozart*, e a máquina servidora seja *chainsaw.yard.com*. Em *chainsaw*, deverá ser colocada a seguinte linha dentro do arquivo *.rhosts* no diretório *bach*: *toe.grunge.com mozart*. Então deve-se testar o *rsh* fazendo: *rsh -l bach chainsaw.yard.com 'echo $PATH'*.

Feito isso, é bom ter a certeza de que *rsh* é capaz de encontrar o servidor. Para isso, o usuário deve certificar-se de que o *path* contendo o *rsh* impresso no exemplo acima inclui o diretório contendo um programa chamado cvs que é o servidor. O usuário precisa colocar esse caminho nos arquivos *.bashrc* ou *.cshrc*, etc., e não nos arquivos *.login* ou *.profile*. Alternativamente, o usuário pode definir a variável ambiente *CVS_SERVER* na máquina cliente para o nome do arquivo no servidor que ele quer usar, por exemplo, */usr/local/bin/cvs-1.6*. É necessário editar o *inetd.conf* ou inicializar (*start*) um *deamon* servidor do CVS.

Assim, se o usuário quiser acessar o módulo *foo* no repositório */usr/local/cvsroot*, na máquina *chainsaw.yard.com*, basta fazer o seguinte:

cvs -d :ext:bach@chainsaw.yard.com:/usr/local/cvsroot checkout foo.

Pode-se usar também o método *:server* em vez de *:ext*. O cliente CVS pode também se conectar ao servidor usando um protocolo de senha. Isso é muito usado principalmente se a conexão via *rsh* não for possível (por exemplo, o servidor está protegido por um *firewall*) e o *kerberos*

também não estiver disponível. Para usar esse método de conexão, é necessário fazer alguns ajustes tanto na máquina servidora como na cliente. A Seção seguinte descreve como fazê-lo.

3.7.2. Ajustando o servidor para autenticação de senha

Na máquina servidora do CVS, o arquivo */etc/inetd.conf* necessita ser editado para executar o comando *cvs pserver* quando ele receber uma conexão na porta certa, que por default, o número da porta é 2401. Se o arquivo *inetd* aceitar números de porta em *etc/inetd.conf*, então a seguinte linha é suficiente: *2401 stream tcp nowait root /usr/local/bin/cvs cvs --allow-root=/usr/cvsroot pserver*. A opção *--allow-root* especifica o diretório onde está o repositório CVS.

Devido ao cliente armazenar e transmitir senhas de forma clara, um arquivo CVS de senha separado pode ser usado, assim as pessoas não precisam comprometer suas senhas quando elas acessam o repositório. Este arquivo está em *$CVSROOT/CVSROOT/passwd*. Seu formato é semelhante ao */etc/passwd* do Unix, exceto que o primeiro possui somente dois ou três campos, *username*, *password*, e um *username* opcional para o servidor usar. Por exemplo:

cvs:ULtgRLXo7NRxs:kfogel
generic:1sOp854gDF3DY:spwang

A senha é encriptada de acordo com a função crypt() do Unix, então é possível copiar as senhas diretamente do arquivo *passwd* do Unix.

No momento da autenticação da senha, o servidor primeiro checa se o usuário existe no arquivo *passwd*. Se encontrar o usuário, ele compara a senha. Se não encontrar o usuário, ou o arquivo *passwd* não existir, então o servidor tenta usar o usuário e senha do sistema. Quando o arquivo *passwd* do CVS é usado, o servidor usa o *username* especificado no terceiro argumento, ou o primeiro se não existir o terceiro.

Assim, quando alguém tentar acessar remotamente o repositório em "chainsaw.yard.com" com o seguinte comando: *cvs -d :pserver:cvs@chainsaw.yard.com:/usr/local/cvsroot checkout foo*, estará executando o servidor sob a identidade do sistema *kfogel*, assumindo sucesso na autenticação. Entretanto, o usuário remoto não necessariamente precisa conhecer a senha de *kfogel*. Assim o *$CVSROOT/CVSROOT/passwd* pode conter uma senha diferente, usada somente

pelo usuário cvs. Dessa forma, é possível mapear múltiplos *usernames* cvs usando um único *username* do sistema.

3.7.3. Usando o cliente com autenticação de senha

Antes de se conectar com o servidor, o cliente deve fazer o *login* com o comando *cvs login*. Esse comando verifica uma senha com o servidor, e grava a senha para mais tarde efetuar transações com o servidor. O comando *cvs login* precisa do *username*, o nome do *host* do servidor e o caminho completo do repositório. Esse comando é interativo, de forma que sua execução pede uma senha:

```
cvs -d :pserver:bach@chainsaw.yard.com:/usr/local/cvsroot login
CVS password:
```

Uma vez que o usuário esteja logado no servidor, ele pode forçar o CVS se conectar diretamente com o servidor e autenticar com a senha armazenada:

```
cvs -d :pserver:bach@chainsaw.yard.com:/usr/local/cvsroot checkout foo
```

O *:pserver* é necessário porque sem ele o CVS assume que o usuário está se conectando com *rsh*. As senhas são armazenadas por default no arquivo *$HOME/.cvspass*. Seu formato é legível às pessoas, mas o usuário não deve editar esse arquivo a menos que ele saiba o que está fazendo.

A ferramenta desenvolvida neste projeto utiliza esse método de conexão (*pserver*) e faz uso do arquivo *passwd* que reside em *$CVSROOT/CVSROOT* para armazenar os autores (ou desenvolvedores das páginas) da ferramenta. O *login* de todos os autores é feito na linha de comando e suas senhas são armazenadas em *$HOME/.cvspass* como mencionado anteriormente (no nosso caso, o *$HOME* é **nobody**, pois fazemos uso de CGIs, descritos na Seção 4.2 do Capítulo 4, e estes são executados como usuário nobody). Embora todas as operações disponíveis da ferramenta sejam realizadas na mesma máquina onde reside o repositório CVS e os usuários da ferramenta não precisem do CVS em suas máquinas locais, o método *pserver* foi utilizado para autenticação dos autores para a manipulação dos arquivos através da *VersionWeb*. O uso desse método permite que o *checkout* e alterações feitas nos arquivos pelos autores sejam registrados com o seu *username* utilizado na autenticação para acesso à *VersionWeb*.

Além dos métodos descritos até aqui, é possível se conectar com o servidor CVS com os métodos GSSAPI e com *Kerberos*. A sub-seção seguinte descreve como o usuário deve proceder na utilização desses métodos.

3.7.4. Conexão direta com GSSAPI

GSSAPI é uma interface genérica para sistemas de segurança em rede tal como *kerberos* 5. Se o usuário tiver uma biblioteca de trabalho GSSAPI, ele poderá se conectar com o CVS via uma conexão TCP direta, autenticando com GSSAPI. Para fazer isso, o CVS precisa ser compilado com suporte GSSAPI; durante a configuração do CVS ele tenta detectar de qualquer forma as bibliotecas GSSAPI usando a versão 5 do *kerberos* para serem apresentadas. O usuário deverá usar o flag *--with-gssapi* para configurar.

A conexão é autenticada usando GSSAPI, mas a mensagem não é autenticada automaticamente. O usuário deve usar a opção global *-a* para requisitar a autenticação. Os dados transmitidos não são criptografados automaticamente, pois o suporte à criptografia deve ser compilado tanto no cliente como no servidor. Para isso, o usuário deve usar a opção de configuração *--enable-encrypt* para ativá-la e a opção global *-x* também deve ser usada para requisitar a criptografia.

As conexões GSSAPI são manipuladas no lado do servidor da mesma forma como foi visto na sub-seção 3.7.2 (ajustando o servidor para autenticação de senha). O usuário deverá criar um arquivo de senha "passwd" vazio e definir a variável *SystemAuth=no* no arquivo de configuração "config" do diretório *CVSROOT*. O servidor GSSAPI usa o nome principal de *cvs/hostname*, onde *hostname* é o nome da máquina *host*. Finalmente, para se conectar usando GSSAPI, o usuário deverá usar *:gserver*, como por exemplo, *cvs -d :gserver:chainsaw.yard.com:/usr/local/cvsroot checkout foo*.

3.7.5. Conexão direta com *kerberos*

A forma mais fácil de usar *kerberos* é também se usar o *rsh*, como descrito na sub-seção 3.7.1. A principal desvantagem de usar *rsh* é que todos os dados precisam passar por programas adicionais, o que pode demorar um pouco. Portanto, se o usuário tiver o *kerberos* instalado ele poderá fazer uma conexão direta via TCP, autenticando com *kerberos*. Esta Seção se refere à versão 4 do *kerberos*, pois a versão 5 é suportada via a interface genérica GSSAPI do sistema de segurança em rede, como visto na Seção anterior.

Para isso, o CVS necessita ser compilado com suporte para o *kerberos*; durante a configuração o CVS tenta detectar a versão do *kerberos*, mas o usuário pode usar o flag *--with-krb4* para configurar. Quanto à criptografia, o usuário deve seguir os mesmos passos da Seção anterior.

O usuário deve editar o arquivo *inetd.conf* na máquina servidora para executar o *kserver*. O cliente usa a porta 1999 por *default*; se o usuário quiser usar uma outra porta ele deverá especificá-la na variável de ambiente *CVS_CLIENT_PORT* na máquina cliente. Quando o usuário quiser usar o CVS, ele deverá obter um *ticket* da forma usual (normalmente *kinit*); esse *ticket* deverá permitir que o usuário faça o *login* na máquina servidora. Feito isso, o usuário poderá fazer o seguinte: *cvs -d :kserver:chainsaw.yard.com:/usr/local/cvsroot checkout foo*. Versões anteriores do CVS retrocederiam para uma conexão *rsh*, mas a versão 1.10 e sucessoras não fazem isso.

A Seção seguinte faz algumas comparações entre o CVS, o SCCS e o RCS mostrando algumas vantagens e desvantagens de cada um desses sistemas.

3.8. Considerações finais

Neste capítulo foram apresentadas as principais características do CVS e uma sessão de trabalho descrevendo o uso dos comandos (específicos do CVS) nas tarefas mais comuns que podem ser realizadas em uma ferramenta de controle de versão. Foram apresentadas também as formas de conexão com o CVS com repositórios remotos, como o CVS trabalha com números de revisões e geração de *branches*.

O CVS, diferentemente do SCCS e do RCS, possibilita o acesso concorrente sobre os arquivos entre os desenvolvedores e este é o principal motivo pelo qual as pessoas ultimamente vêm optando por usar o CVS. Além da concorrência, o CVS também permite a geração de *releases* de um projeto, a definição de módulos para um conjunto de arquivos relacionados, o acesso através da rede, a área de trabalho é distribuída e o CVS não está limitado apenas a plataformas UNIX. O acesso simultâneo aos arquivos por mais de uma pessoa e o acesso ao repositório CVS pela rede foram os dois fatores mais importantes na escolha do CVS para o desenvolvimento da ferramenta *VersionWeb*.

Por outro lado, o CVS apresenta algumas desvantagens em relação ao RCS tais como gerenciamento mais difícil e cuidadoso, apresenta grande número de comandos e é muito complexo, o que dificulta o seu total conhecimento por parte dos usuários.

4. Recursos para programação na WWW

4.1. Considerações iniciais

A WWW é o maior reservatório eletrônico de informações do mundo [Jamsa et al. 1997]. Em outras palavras, a *Web* se compõe de um conjunto de milhões de documentos interligados que residem em computadores de todo o mundo e, a Internet, é o veículo que permite a comunicação entre esses computadores. Adicionalmente, pode-se pensar na *Web* como um sistema de *software* e documentos interligados que se encontram "sobre" a Internet (*hardware* e *software*).

Em geral, os documentos *Web* são arquivos que contêm além de seu conteúdo principal, informações que são utilizadas pelos *browsers* para exibir seu conteúdo em uma página *Web*. O formato do arquivo descreve o tipo de informações que o arquivo contém, como áudio, vídeo ou texto. Mas, à medida em que os recursos de programação como bibliotecas multimídia e suporte de linguagens internas aos *browsers* tornam-se disponíveis aos usuários, estes passam a esperar e requerer mais de suas aplicações. Desde o surgimento da *Web*, as expectativas dos usuários não param de crescer, e inúmeras ferramentas têm sido criadas para darem suporte à incorporação de recursos mais sofisticados aos *sites*.

Como o objetivo deste trabalho de mestrado foi proporcionar que um controle de versões de páginas da *Web* fosse disponibilizado como um aplicativo na própria *Web*, neste Capítulo são brevemente descritos os recursos de programação na *Web* que foram estudados. Na Seção 4.2 é apresentado o CGI, que permite a criação de *sites Web* que incluem recursos de interação do usuário com a aplicação. Na Seção 4.3 são comentadas as principais características da linguagem Java, principalmente em relação aos recursos para o desenvolvimento de *applets* para inserção em documentos HTML. Na Seção 4.4 são apresentados os *Servlets*, um novo conjunto de aplicações escritas em Java. Na Seção 4.5 é apresentado o JSP, uma nova tecnologia de Java para criar páginas *Web* dinâmicas e na Seção 4.6 é apresentada uma breve descrição de *JavaScript*.

4.2. CGI

O CGI, ou *Common Gateway Interface*, é um padrão que especifica o formato dos dados que *browsers*, servidores e programas usam para trocar informações [Jamsa et al. 1997]. Existem alguns tipos de informações que não podem ser acessadas diretamente pelos *browsers*, como informações resultantes de acessos à base de dados e informações geradas dinamicamente. Tais informações não podem ser exibidas se não forem colocadas em um formato padrão reconhecível pelos *browsers*. Portanto, devem existir programas que façam a "transformação" daquelas informações para um formato reconhecível pelos *browsers*.

Os programas necessários para especificar ao servidor como acessar determinado tipo de informação solicitada pelo *browser* são denominados *scripts*. E os programas que possibilitam um caminho para a passagem de informação entre o *browser* e o servidor, são também chamados de *gateways*. Assim, CGI é o padrão que possibilita a comunicação entre o servidor HTTP (*HyperText Transfer Protocol*) e os *scripts*, sendo que o *script* CGI é um programa executado independentemente, escrito em qualquer linguagem aceita pelo servidor, seja ela compilada ou interpretada. As linguagens mais usadas são C/C++, Perl, Shell do Unix, Tcl/Tk, Visual Basic e Delphi [Gundavaran 1996]. Um programa CGI é executado em tempo real, ou seja, no instante em que o cliente o solicita, através da sua URL (*Uniform Resource Locator*). Dessa forma, o programa pode exibir informações dinâmicas (obtidas em tempo de execução), ao contrário de um documento puramente HTML, cujo conteúdo é estático.

4.2.1. A ação de um programa CGI

Um programa CGI não pode ser executado diretamente a partir do *browser* (cliente *Web*), é necessário localizar o *script* no servidor *Web*. De forma resumida, a ação de um programa CGI ocorre da seguinte maneira, esquematizada na **Figura 4.1**.

1. o cliente solicita uma URL (que não é um documento HTML e sim um *script*) ao servidor;
2. o servidor, ao receber o formulário, utiliza a interface CGI para contatar o *script* e passar os dados fornecidos pelo cliente;
3. o programa CGI é executado no servidor com base nos valores de entrada do cliente (se houver). Essa execução pode envolver, por exemplo, pesquisa em um banco de dados através de outros programas;

4. o resultado da execução é enviado pelo *script* ao servidor via interface CGI;

5. o servidor envia o resultado do *script* ao cliente, acrescentando as informações de cabeçalho necessárias.

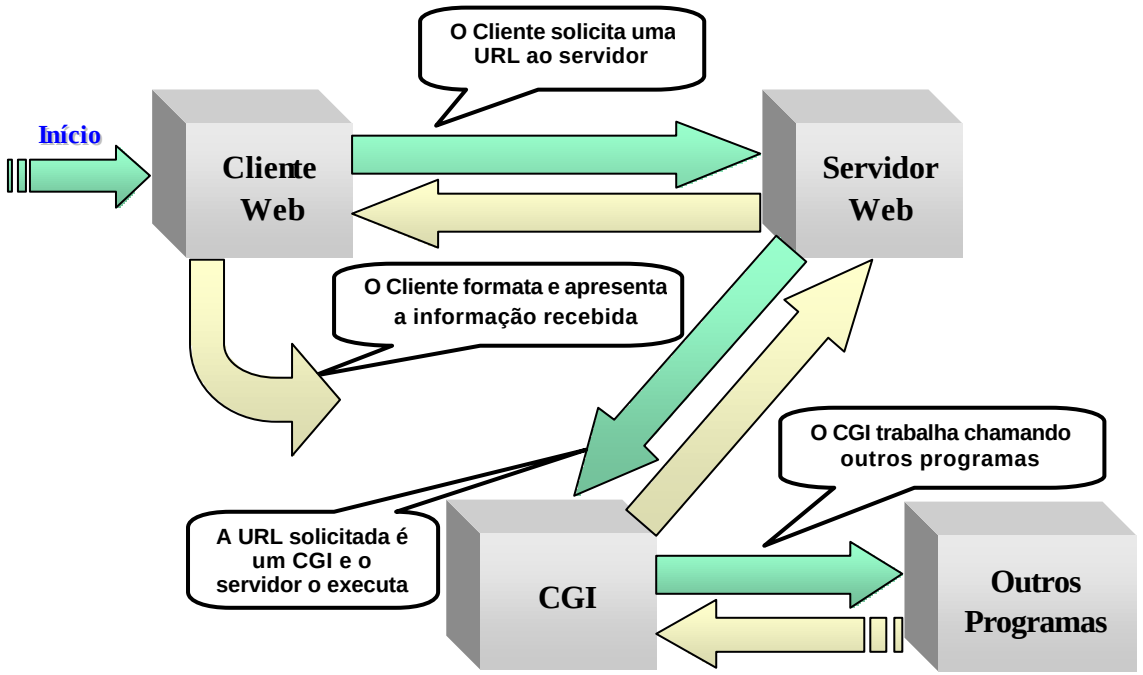


Figura 4.1 - Ação de um programa CGI

Cada vez que um CGI é executado, é necessário carregá-lo na memória e iniciar um processo que será extinto com o término da execução do aplicativo [Colla 1999]. Caso seja uma linguagem interpretada, como PERL, soma-se ainda o tempo necessário para interpretá-lo. Como esses são processos distintos, existe uma grande dificuldade no compartilhamento de recursos do sistema, por exemplo, conexão a um banco de dados e troca de informações entre os processos executados simultaneamente.

4.2.2. A interface básica do CGI

Quando um usuário da *Web* ativa uma ligação que faz o servidor carregar um *script* CGI, o servidor precisa configurar um conjunto de variáveis de ambiente utilizadas pelo *script*. Essas variáveis contêm informações sobre o *browser* que está fazendo a solicitação, sobre o servidor que está controlando a solicitação e sobre os dados (se houver) sendo passados ao *script*. A variável de ambiente *CONTENT_LENGTH*, por exemplo, determina o número exato de bytes contidos nos dados anexados [Blum 1996]. Quando o *browser* encaminha uma solicitação ao

servidor (por exemplo, usando o método "POST" HTTP), as informações que o *script* CGI recebe vêm do descritor de arquivo *stdin* (*standard input*). Então, o valor da variável *CONTENT_LENGTH* determina a quantidade de dados a serem processados a partir de *stdin*. Além disso, o servidor fornece ao *script* CGI a variável de ambiente *CONTENT_TYPE*, que permite ao *script* decidir como controlar os dados a serem recebidos.

Os dados recebidos contêm uma seqüência de pares no formato **name=value** separados pelo caractere '&'. A parte **name** contém o nome dos campos como definidos no documento HTML. A parte **value** contém os valores dos campos, ou seja, as informações inseridas pelo usuário, fornecidas de forma codificada [Gundavaran 1996]. A codificação é feita automaticamente pelo *browser*, e a decodificação deve ser feita pelo *script* CGI antes de utilizar os valores recebidos.

4.2.3. Formatando a saída do CGI

A saída de um CGI tem, basicamente, duas partes: um campo de cabeçalho e os dados de saída. O cabeçalho consiste em linhas de texto, no mesmo formato de um cabeçalho HTTP, e termina com uma linha em branco. É necessário enviar antes o cabeçalho para que o cliente conheça o tipo de informação a ser retornada pelo programa para tratá-la convenientemente. O cabeçalho não é visível ao usuário e serve para passar informações sobre a saída tanto ao servidor quanto ao cliente *Web*. Existem três tipos de cabeçalhos [Jamsa et al.1997]: *Content-type*, *Location*, *Status*.

O campo *Content-type* identifica o tipo/subtipo MIME (*Multipurpose Internet Mail Extensions*) dos dados que o *script* está enviando de volta ao *browser*. Normalmente, os *scripts* CGI enviam um documento HTML. Nesse caso, o campo ***Content-type*** conteria o seguinte:

Content-type: text/html

O campo *Location* especifica um documento. Os *scripts* usam o campo *Location* para especificar ao servidor que chamou o *script* CGI que está retornando uma referência a um documento, e não o documento em si. Para especificar um documento remoto, por exemplo, o valor de *Location* poderia conter o seguinte:

Location: http://www.jamsa.com/

O campo *Status* contém um valor de HTTP que o servidor envia do *script* para o *browser*. Os dados de saída são, realmente, a parte que será enviada ao cliente *Web*, e seu conteúdo deve ser compatível com o tipo de conteúdo (*Content-type*) informado no cabeçalho.

A principal vantagem do uso de CGI é a sua simplicidade. Para o cliente, basta possuir um *browser* para, por exemplo, ter acesso a um banco de dados. Para o desenvolvedor, há a facilidade de escrever o código em qualquer linguagem, desde que esta possa ser executada no servidor [Jamsa et al. 1997].

Além disso, os protocolos básicos do CGI são relativamente simples e a maioria dos servidores *Web* fazem uso deles. Devido à explosão da Internet, muitas pessoas têm experiência com CGI em quase todos os tipos de plataformas e servidores [Breedlove 1996].

Porém, apesar dessas vantagens, o CGI é inadequado para o desenvolvimento de sistemas complexos. Isso porque o *script* sempre é executado na máquina servidora, de forma que qualquer alteração nas aplicações requer uma intervenção do *Webmaster*, que precisa assegurar que o programa não vai causar danos ao sistema. Além disso, a execução do *script* no servidor pode representar uma grande sobrecarga para o sistema como um todo. Quanto mais trabalho puder ser distribuído nas máquinas clientes, maior será a eficiência da rede como um todo.

4.3. Java

Java é uma linguagem de programação desenvolvida pela Sun Microsystems em meados de 1990, quando o objetivo principal era o desenvolvimento de uma linguagem que permitisse a integração total de sistemas de computação com equipamentos eletrodomésticos [Lemay e Perkins 1996]. A linguagem permite a construção otimizada de aplicações distribuídas, ou seja, permite que múltiplos computadores sejam acessados através de uma rede e por múltiplos usuários simultaneamente. A sintaxe de Java é derivada da sintaxe de C++, o que permite que programadores familiarizados com C++ se adaptem facilmente à Java [Sun 1999].

4.3.1. Características da linguagem

Java é uma linguagem multiplataforma [Lemay e Perkins 1996], o que é possível devido à estrutura interpretada que a caracteriza e o processo de compilação do código-fonte (identificado pela extensão *.java*), ilustrado na **Figura 4.2**.

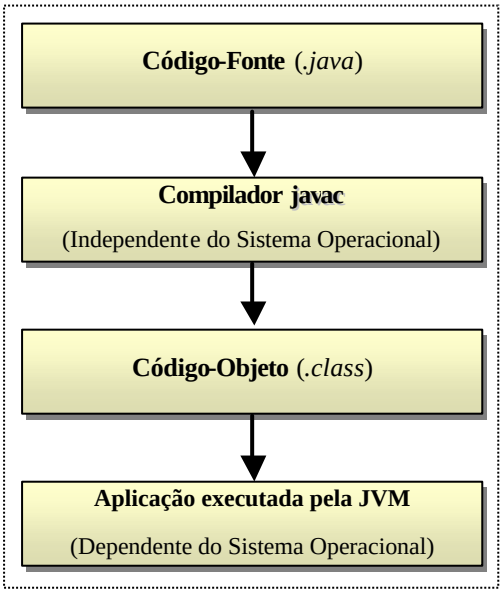


Figura 4.2 - Processo de compilação e execução em Java

O compilador Java traduz o código-fonte em um código intermediário e independente de plataforma chamado *bytecode*, que é interpretado pela JVM (*Java Virtual Machine*), também conhecida como interpretador Java ou *runtime* Java. Desse forma, é possível que um programa Java seja executado em qualquer plataforma sem a necessidade de alteração no código-fonte. Outras características da linguagem são:

- é dotada de recursos *multithreaded*, permitindo a implementação de aplicações multitarefa;
- possui um vasto conjunto de componentes para a implementação de Interfaces Gráficas com o Usuário (GUI – *Graphics User's Interface*), por meio das classes da biblioteca AWT (*Abstract Window Toolkit*);
- devido à sua natureza gráfica, inclui recursos de manipulação de elementos gráficos independentemente do dispositivo gráfico e da plataforma operacional;
- permite a manipulação de recursos de texto e características das fontes de texto;
- permite o controle dos eventos de mouse e teclado sobre os objetos "encapsulados" da interface gráfica, bem como sobre eventos definidos pelo programador;
- possui recursos para processamento e análise de imagens na maioria dos formatos gráficos padrões, como, por exemplo, arquivos GIF e JPEG;

- permite a inclusão de bibliotecas de propósito específico pela definição de interfaces, denominadas ***packages*** (como exemplo, pode-se citar bibliotecas para Computação Gráfica, Hipermídia, Programação Concorrente, Banco de Dados e outras).

Essas características fazem parte de todas as versões da linguagem Java. A partir da versão 1.2, também chamada de Java 2, foram inseridos recursos avançados para a construção de interfaces com o usuário. O *Java Swing*, que faz parte da biblioteca JFC (*Java Foundation Classes*), é uma extensão do AWT que foi integrada à linguagem [Lemay e Perkins 1996]. O *Java Swing* oferece funcionalidades bastante aprimoradas em relação ao seu predecessor, destacando-se:

- a adição de novos componentes de interface (*widgets*);
- a expansão dos recursos dos componentes existentes;
- um melhor tratamento de eventos;
- permite selecionar a aparência e o comportamento do ambiente (plataforma);
- os componentes são implementados inteiramente em Java.

Em uma rede heterogênea como a Internet, formada por diversas máquinas diferentes ligadas entre si, as características de Java a colocam em posição confortável como a linguagem ideal para este ambiente. O seu uso pode reduzir o custo de aplicativos que devem executar em várias plataformas, pois não é mais necessário transportar e compilar o código-fonte para cada plataforma diferente, e tratar os *bugs* em cada uma delas individualmente, nem gerar documentação e fornecer suporte para cada plataforma. Só há uma compilação, uma documentação e um produto comercial [Thomas et al. 1996].

4.3.2. *Applets* e aplicações *stand-alone*

Um programa Java pode tomar dois rumos com relação à sua filosofia de execução: pode ser uma *applet* ou uma aplicação *stand-alone*, ou ambos, dependendo de como o programa é escrito e dos recursos que usa [Lemay e Perkins 1996]. Uma *applet* é definida como um "*pequeno programa*" Java que necessita de uma página *Web* e um *browser* (tal como Netscape Navigator, Microsoft Internet Explorer ou outros) para ser executada. Por outro lado, uma aplicação *stand-alone* é um programa Java que pode ser executado por um comando vindo do usuário. Apenas as *applets* serão tratadas aqui, pois o uso de aplicações *stand-alone* não satisfazem o requisito de ser orientada a *Web*, contextualizado neste trabalho.

Como as *applets* são projetadas para serem descarregadas a partir de um *site* remoto e depois executadas localmente, a segurança torna-se importante. Assim, as *applets* (ao contrário das aplicações), são restritas pelos *browsers* quanto ao que podem fazer [Cornell e Horstmann 1997]. No momento, as *applets* "não assinadas" não podem:

- executar um programa que está na máquina local;
- comunicar-se com qualquer *host* além do servidor a partir do qual foram transferidas;
- ler ou gravar no sistema de arquivos do computador local (isso se aplica aos *browsers* Netscape Navigator e Internet Explorer, pois outros *browsers* podem implementar restrições de segurança menos rígidas);
- obter quaisquer informações sobre o computador local, exceto a versão de Java em uso, o nome e a versão do sistema operacional, bem como os caracteres usados para separar arquivos (por exemplo, '/' ou '\') ou caminhos (como ':' ou ';') e linhas (como '\n' ou '\r\n'). Em particular, as *applets* não podem descobrir o nome do usuário, endereço do correio eletrônico, etc.

Atualmente, a versão 1.2 de Java permite que as *applets* façam tudo o que uma aplicação Java pode fazer, utilizando assinaturas digitais, que fornecem um mecanismo para abrandar as restrições de segurança impostas às *applets.*

As *applets* são executadas na máquina do cliente, o que representa uma grande inovação na programação para a *Web*, sendo que elas podem ser embutidas em documentos HTML que, assim, tornam-se interativos. Como Java é uma linguagem de programação, é muito mais fácil tornar uma *applet* interativa do que fazer o mesmo diretamente com uma página *Web*. Sem Java, criar uma página *Web* interativa requer o envio de dados para um *script* CGI no servidor. O *script* CGI precisa processar os dados e retornar os resultados em um formato adequado ao *browser*. Com as *applets*, o processamento é passado para o sistema do usuário que, presume-se, trabalhará mais rápido do que um servidor que esteja atendendo milhares de pedidos naquele momento. Além do mais, se muitos dados tiverem que ser calculados, não é preciso se preocupar com a velocidade de transmissão da máquina servidora, pois isso será feito localmente.

4.4. *Servlets*

A linguagem Java, devido às suas características (descritas na Seção 4.3), vem sendo cada vez mais utilizada para desenvolver aplicações a serem executadas *stand-alone*, independentes da Internet. Um novo conjunto de aplicação, os *servlets*, podem popularizar ainda mais a linguagem. *Servlets* são aplicativos escritos em Java que podem ser acoplados em diversos tipos de servidores para expandir as suas funcionalidades [Colla 1999]. Estes, ao contrário das *applets*, não possuem uma interface gráfica, e podem ser escritos para trabalhar com diversos tipos de protocolo de comunicação (por exemplo, HTTP).

4.4.1. Funcionalidades

Dentre algumas capacidades dos *servlets* que tornam esta tecnologia particularmente interessante, podemos citar [Colla 1999]:

- **geração dinâmica de páginas HTML** - os *servlets* podem ser instalados em servidores *Web* para processarem informações transmitidas via HTTP a partir de formulários HTML, por exemplo. As aplicações podem incluir acesso a banco de dados ou comunicação com outros *servlets*;
- **balanceamento de carga entre servidores** - para entender como utilizar *servlets* para balanceamento de carga, considere a infra-estrutura de um provedor de serviços via Internet composta de cinco servidores, dos quais quatro são capazes de executar as mesmas aplicações. O servidor restante poderia ficar responsável por monitorar as cargas dos demais e receber o acesso inicial de cada cliente às aplicações. Em seguida, poderia redirecionar os pedidos de acesso para um dos quatro servidores de aplicação, conforme a ocupação de cada um no momento em que o cliente tenta estabelecer uma conexão. Assim, o cliente passa a trocar informações somente com o servidor que foi alvo do redirecionamento;
- **modularização do código**: um *servlet* pode executar outro *servlet*, mesmo que remotamente, sendo possível executá-los em "corrente". Essa característica possibilita a modularização dos aplicativos, criando *servlets* com funções específicas. Suponha que, para acessar um conjunto de aplicativos, o cliente deva ser autenticado. Neste caso, uma configuração possível seria criar um *servlet* responsável apenas pela tarefa de autenticação. Uma vez autenticado o cliente, este *servlet* o redirecionaria para

outro *servlet*, não necessariamente instalado no mesmo servidor, e que executaria o aplicativo. A vantagem deste tipo de arquitetura é que, se por alguma razão for necessário modificar o procedimento de autenticação, por exemplo pela mudança do banco de dados de usuários, não será necessário re-escrever toda a aplicação, mas apenas o *servlet* responsável pelo processo de autenticação.

Diferentemente do CGI, o *servlet* é carregado uma única vez na memória, aonde permanece até que seja necessário modificá-lo [Colla 1999]. Mas por outro lado, é necessário reiniciar o servidor toda vez que um *servlet* sofre alguma modificação, e isso pode ser mais demorado que a recompilação de um simples *script* CGI. Entretanto, atualmente, alguns sistemas que executam *servlets* são capazes de recarregá-los sem a necessidade da reinicialização. Como os *servlets* permanecem na memória, torna-se mais fácil desenvolver aplicações que exijam persistência entre conexões sucessivas de um mesmo cliente.

Uma vez inicializado, o *servlet* estará apto a lidar com centenas de acessos simultaneamente, disparando, para cada acesso, uma nova *thread*. As *threads* de um mesmo aplicativo utilizam um espaço de endereçamento de memória comum a todas, o que permite que elas compartilhem dados e recursos do sistema. Por exemplo, todas as *threads* de um *servlet* podem usar uma única conexão estabelecida com um banco de dados no momento da inicialização do *servlet*. Esta conexão permanece aberta até que o *servlet* seja desativado, saia da memória, ou seja recarregado.

Uma vez que os *servlets* são escritos em Java, uma linguagem projetada para ser independente de plataforma, é possível escrever um *servlet* para um servidor UNIX, que poderá ser instalado em um servidor Windows NT, sem a necessidade de re-escrever o código, ou recompilá-lo. A orientação a objetos, uma das características de Java, favorece a modularização dos aplicativos, o que os torna mais escaláveis.

4.5. JSP

A tecnologia JSP, ou *JavaServer Pages*, fornece uma forma fácil de criar páginas *Web* dinâmicas. Ela simplifica a tarefa de construção de aplicações *Web* que trabalham com uma grande variedade de servidores *Web*, servidores de aplicação, *browsers* e ferramentas de desenvolvimento [McPherson 2000].

Como parte da família da tecnologia Java, JSP permite o desenvolvimento rápido de aplicações baseadas na *Web* que são independentes de plataforma. JSP separa a interface do usuário da geração do conteúdo, permitindo que os projetistas alterem todo o *layout* da página sem alterar o seu conteúdo dinâmico.

Em sua forma básica, uma página JSP é simplesmente uma página HTML que contém pedaços adicionais de código que executam a lógica de uma aplicação que gera conteúdo dinâmico. Esse código pode envolver, dentre outros, JavaBeans, objetos JDBC (*Java DataBase Connectivity*) ou objetos RMI (*Remote Method Invocation*). Por exemplo, uma página JSP pode conter um código HTML que mostra textos e gráficos estáticos, bem como um método para chamar um objeto JDBC que acessa uma base de dados e os apresenta em formato HTML; quando a página é mostrada no *browser* do usuário, ela conterá todo o conteúdo HTML com a informação dinâmica obtida da base de dados.

A separação da interface do usuário da lógica do programa em uma página JSP permite que os programadores criem códigos flexíveis que podem ser facilmente atualizados e reusados [McPherson 2000]. Em adição, devido ao fato das páginas JSP poderem ser automaticamente compiladas quando necessário, os autores *Web* podem fazer alterações no código de apresentação da página sem precisar recompilar a lógica da aplicação. Isso torna a tecnologia JSP um método mais flexível de geração dinâmica de conteúdo *Web* do que o simples uso de *servlets* Java.

As páginas JSP são automaticamente compiladas para *servlets* antes delas serem usadas a primeira vez, então essas páginas têm todos os benefícios dos *servlets*, incluindo o acesso para as APIs Java [McPherson 2000].

4.6. JavaScript

JavaScript é uma linguagem de *script* que os projetistas usam para criar páginas *Web* interativas [Jamsa et al. 1997]. Ela é uma linguagem de criação de *homepages* e suas funções podem ser embutidas dentro do código HTML. Embora os nomes sejam parecidos, Java e *JavaScript* são bem diferentes: a primeira é uma linguagem de programação, a segunda é uma linguagem de hipertexto. O *JavaScript* tem como objetivo permitir ao usuário trabalhar o dinamismo nas suas

páginas sem precisar conhecer a linguagem Java, pois está disponível num subconjunto de recursos desta linguagem.

O programa escrito em *JavaScript* é executado no cliente, ou seja, pelo *browser*, sem ser levado ao servidor. Ele está escrito na própria linguagem HTML e o *browser* deve ter capacidade de interpretar tal programa. O uso de *JavaScript* não é vantajoso por possuir funcionalidades limitadas ao subconjunto de recursos Java. Mas uma vantagem que ele possui é não exigir muita capacidade de processamento do cliente quando comparado às *applets.*

4.7. Considerações finais

Neste Capítulo foram apresentadas algumas ferramentas disponíveis para programação na *Web*, suas características, seus principais recursos de programação e as principais vantagens e desvantagens do uso de cada uma delas. Todas as ferramentas apresentadas poderiam ser utilizadas para o desenvolvimento deste projeto, mas optamos por usar o CGI devido a algumas de suas vantagens como simplicidade de programação, fácil entendimento, grande difusão no mercado de trabalho e todos os provedores de Internet suportarem o seu uso.

Além disso, os protocolos básicos do CGI são relativamente simples, a maioria dos servidores *Web* usam esses protocolos e muitas pessoas têm experiência com CGI em quase todos os tipos de plataformas e servidores.

Dessa forma, foram investigados os recursos do CGI para suportar o uso do CVS na *Web* e viabilizar o desenvolvimento da ferramenta. Como visto na Seção 4.2, o CGI pode ser escrito em várias linguagens, mas utilizamos a linguagem de programação C por já possuirmos mais conhecimento dessa linguagem e poder assim avançar mais rapidamente o projeto, por ela fornecer capacidades avançadas de programação suficientes para o desenvolvimento das funcionalidades requeridas pela ferramenta projetada e por ela ser compilada, o que lhe dá uma vantagem de melhor desempenho em relação às linguagens que são interpretadas.

Todo o processamento da aplicação desenvolvida é feito no servidor (e normalmente se constitui de comandos CVS), mas a validação da entrada de dados do usuário é feita com o uso de *JavaScript* antes que esses dados sejam enviados ao servidor. A tarefa de validação local dos dados favorece o desempenho evitando conexões desnecessárias com o CGI no servidor pois, se

informações incorretas fossem submetidas ao CGI, elas seriam verificadas somente no servidor e o CGI teria que devolver uma resposta de erro ao usuário.

O Capítulo seguinte descreve a ferramenta desenvolvida, denominada *VersionWeb*, a arquitetura adotada, seus principais módulos, as funcionalidades de cada um deles e alguns resultados obtidos através de testes informais de usabilidade realizados na *VersionWeb*.

5. A ferramenta *VersionWeb*

5.1. Considerações iniciais

Em muitas áreas de desenvolvimento, como a de *software*, e em ambientes de autoria, como a WWW, a preservação de revisões intermediárias de objetos em evolução é muito importante. Como o número de revisões cresce muito e de forma aleatória, a preservação das revisões se torna uma atividade difícil [Hicks et al. 1998].

Conforme mencionado no início desta dissertação, no ambiente WWW é muito freqüente os internautas se surpreenderem ao visitar uma página e perceberem que esta já não possui o mesmo conteúdo ou até mesmo que ela não existe mais; tudo isso é decorrente da rápida e natural evolução das informações na WWW [Sommerville et al. 1998]. E, à medida que o volume de documentos envolvidos aumenta, torna-se difícil para os desenvolvedores das páginas ou de *sites* inteiros controlarem as modificações que inevitavelmente ocorrem durante o desenvolvimento. Essas alterações, se não forem controladas, quando envolvem um grupo de pessoas trabalhando no mesmo projeto, aumentam a possibilidade de geração de um produto inconsistente e de má qualidade. Em adição, esses desenvolvedores podem estar localizados em diferentes lugares, o que dificulta ainda mais a comunicação entre eles para o desenvolvimento sincronizado do projeto.

Neste contexto, a ferramenta *VersionWeb* foi desenvolvida com o objetivo de auxiliar os desenvolvedores no desenvolvimento de páginas, especificamente para dar um suporte à evolução das mesmas, de forma coordenada e controlada evitando, assim, perdas ou sobreposições de informações. Outro objetivo da *VersionWeb* foi o de permitir que os internautas, durante a navegação, tivessem acesso às informações que alguma vez estiveram disponíveis, mas que, geralmente, como conseqüência das constantes atualizações das páginas não estão mais, naquele momento, sendo apresentadas. Por meio da recuperação de versões anteriores dessas páginas e localização de alterações, o internauta pode rever informações disponibilizadas em algum momento anterior.

Para o desenvolvimento dessa ferramenta foi utilizado o CVS, *software* para controle de versões que viabilizou o suporte à evolução das páginas *Web* em desenvolvimento, e cujas principais funcionalidades já foram descritas no Capítulo 3. A comunicação entre a *VersionWeb* e o CVS foi efetuada através de CGIs, descritos no Capítulo 4, que ficam residentes na máquina servidora juntamente com o CVS. Os programas CGIs foram implementados especialmente para viabilizarem essa comunicação.

Este Capítulo descreve a arquitetura adotada para a *VersionWeb*, seus principais módulos, suas funcionalidades, as operações CVS permitidas e alguns resultados de testes obtidos.

5.2. Arquitetura da *VersionWeb*

A ferramenta *VersionWeb* auxilia o trabalho cooperativo feito por desenvolvedores de páginas *Web* que podem estar situados em locais diferentes e remotos. Além disso, a *VersionWeb* permite que os usuários internautas, além dos desenvolvedores das páginas, encontrem alterações em uma página, recuperem versões anteriores e visualizem as diferenças entre elas. Sua principal característica é a combinação de um bom sistema de controle de versão com uma interface amplamente acessível através dos *browsers Web*.

Para apoiar sua principal característica funcional, a *VersionWeb* utiliza o CVS. A *VersionWeb* proporciona uma interface amigável baseada em formulários HTML que oferece as operações básicas de controle de versão para executar os comandos do CVS, uma vez que este é totalmente orientado a linha de comando. Para incluir os recursos do CVS em uma aplicação baseada na *Web*, uma abordagem arquitetural foi adotada e está ilustrada na **Figura 5.1**.

Para que o CVS faça o controle dos arquivos, estes devem estar armazenados em um repositório CVS. E, para tornar esse repositório disponível através da *Web*, decidimos colocá-lo na mesma máquina em que estão localizados o servidor de *Web* (servidor HTTP) e os programas CGIs por razões de segurança e de desempenho. Certamente, o repositório CVS poderia estar em uma outra máquina diferente daquela onde reside os CGIs e o servidor de *Web*. Porém, seria mais uma conexão a ser estabelecida para a execução dos pedidos do usuário (que normalmente são comandos CVS), pois o CGI (programa que faz a chamada do CVS) teria que localizá-lo nessa outra máquina, enviar os pedidos do usuário e o CVS, depois de executá-los, teria que devolver o

resultado para o CGI que o chamou na primeira máquina para que o resultado fosse enviado ao usuário.

De acordo com a abordagem arquitetural da *VersionWeb* esquematizada na **Figura 5.1**, um cenário de uso geral da ferramenta foi desenvolvido.

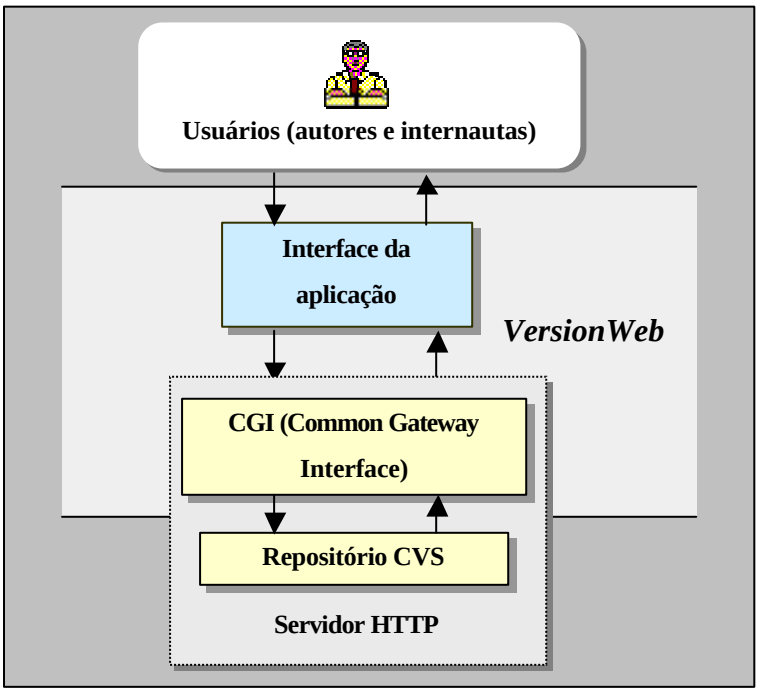


Figura 5.1 - Arquitetura básica da *VersionWeb*

Este cenário de interação do usuário com a ferramenta (representado na figura acima) juntamente com os seus dois componentes funcionais principais (interface da aplicação e os programas CGIs) realizam o seguinte procedimento para a realização das tarefas de controle de versão através da *Web*:

- os usuários, que estão em suas próprias máquinas (locais ou clientes) em qualquer lugar do mundo, interagem com o CVS através da interface de aplicação da *VersionWeb* (baseada em formulários HTML) e submetem pedidos a um programa CGI que reside fisicamente na mesma máquina que o servidor de *Web*; esses pedidos, geralmente, são comandos CVS a serem executados como *checkout*, *commit*, etc.;
- o servidor de *Web* recebe os pedidos do usuário e os passa ao CGI;
- o CGI recebe e decodifica os pedidos do usuário;

- o CGI que recebeu o pedido faz uma chamada ao CVS (também fisicamente localizado na mesma máquina) com os comandos a serem executados;
- o CVS executa os comandos e retorna o resultado ao CGI que o chamou;
- o CGI envia os resultados ao servidor de *Web*;
- servidor de *Web* devolve os resultados aos usuários que iniciaram a interação com a ferramenta.

A partir dessa arquitetura, o usuário precisa apenas de um *browser Web* em sua máquina local para utilizar a *VersionWeb*, o que favorece ainda mais a sua mobilidade em relação ao acesso à ferramenta. Os arquivos ficam em um repositório central (repositório do CVS) no servidor, mas as modificações nos arquivos podem ser remotas (no próprio servidor) ou locais (na máquina do cliente). Essas são duas características presentes na *VersionWeb* dentre outras, descritas mais detalhadamente nas seções seguintes.

5.3. Módulos da *VersionWeb*

Para que pudesse ter os recursos do CVS disponibilizados, via *Web*, a ferramenta foi construída com base em três funcionalidades principais, para as quais foram desenvolvidas interfaces básicas: **a)** gerenciamento dos usuários **b)** gerenciamento dos arquivos que estão sob o controle de versão e **c)** navegação para acesso às versões anteriores da página que estiver sendo visitada. As funcionalidades desenvolvidas a partir de cada uma dessas três interfaces serão descritas nas sub-seções seguintes.

Todas essas funcionalidades podem ser obtidas mediante verificação de que o usuário pertença a um dos grupos autorizados, ou seja, inicialmente o usuário é submetido a um procedimento de autenticação.

A **Figura 5.2** ilustra a estrutura modular da *VersionWeb*, sendo que os nós folhas representam seus três principais módulos, os quais possuem funções específicas.

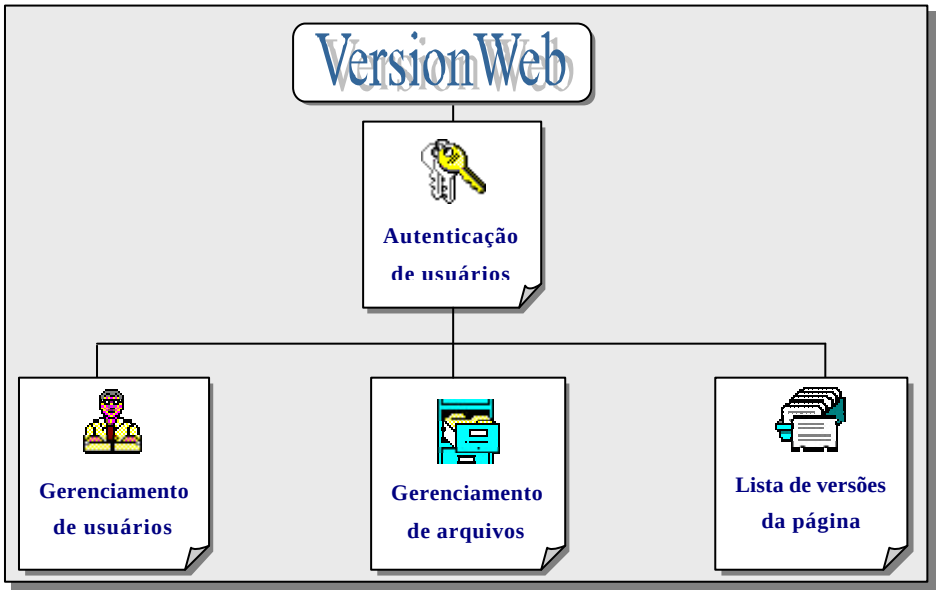


Figura 5.2 - Estrutura modular da *VersionWeb*

Por questões de projeto e de segurança, foi desenvolvido o módulo de autenticação de usuários na *VersionWeb*. Depois de autenticado, o usuário possui as permissões de acesso e, portanto, poderá ser levado, de acordo com a categoria de usuários a que ele pertence, a uma das três interfaces básicas da ferramenta: de gerenciamento de usuários, de gerenciamento de arquivos ou de acesso às diversas versões da página visitada. A sub-seção seguinte descreve o processo de autenticação de usuários.

5.3.1. Autenticação de usuários

Do ponto de vista de quais usuários se beneficiariam dos serviços da *VersionWeb*, foi possível identificar inicialmente dois tipos de usuários: os autores (ou desenvolvedores) das páginas e os internautas e/ou grupos específicos de internautas que navegam através de *browsers*.

Decidimos colocar um tipo de usuários "grupos" para os internautas porque, em algum momento, o autor de uma página pode não querer que todos os internautas tenham acesso às suas versões (se o usuário estiver navegando por essa página), mas que apenas um grupo específico de internautas, que deve estar previamente cadastrado, tenha acesso. Essa decisão é flexível, pois o autor pode decidir que as versões de algumas páginas possam ser vistas por todos os internautas e as versões de outras páginas sejam vistas apenas por um grupo específico. Para isso, no primeiro caso (onde todos o internautas podem ver as versões), o *link* para a ferramenta (que está na página que o navegador está visitando) deverá apontar para o CGI (ou o módulo

"Lista de versões da página" representado na **Figura 5.2**) que dá acesso às versões da página. E, no segundo caso (grupos específicos), o *link* deverá apontar para o CGI (ou o módulo de "Autenticação de usuários" também representado na **Figura 5.2**) que exige a autenticação de usuários.

No entanto, para efeito de se gerenciar esses tipos de usuários, foi definido também um terceiro tipo de usuário, denominado "administrador". Dessa forma, a ferramenta *VersionWeb* foi desenvolvida de maneira a atender esses três tipos básicos de usuários:

- administradores: pessoas responsáveis por gerenciar todos os usuários da ferramenta;
- autores: pessoas autorizadas a manipularem os arquivos que estão sob o controle de versão e que residem no repositório CVS no servidor;
- internautas e/ou grupos específicos de internautas: pessoas às quais é permitido apenas visualizar as versões de uma página, as diferenças entre elas, receber notificação de novas versões da página disponíveis e enviar comentários.

O primeiro passo para o uso da ferramenta *VersionWeb* é a autenticação do usuário (obrigatório para autores e administradores), o que dá acesso às funcionalidades principais da ferramenta. Assim, a *VersionWeb* inicia a identificação do usuário por meio da janela de autenticação ilustrada na **Figura 5.3**, para que a seguir, a interface relacionada com a devida permissão de acesso do usuário (correspondente ao grupo que ele pertence) seja disponibilizada.

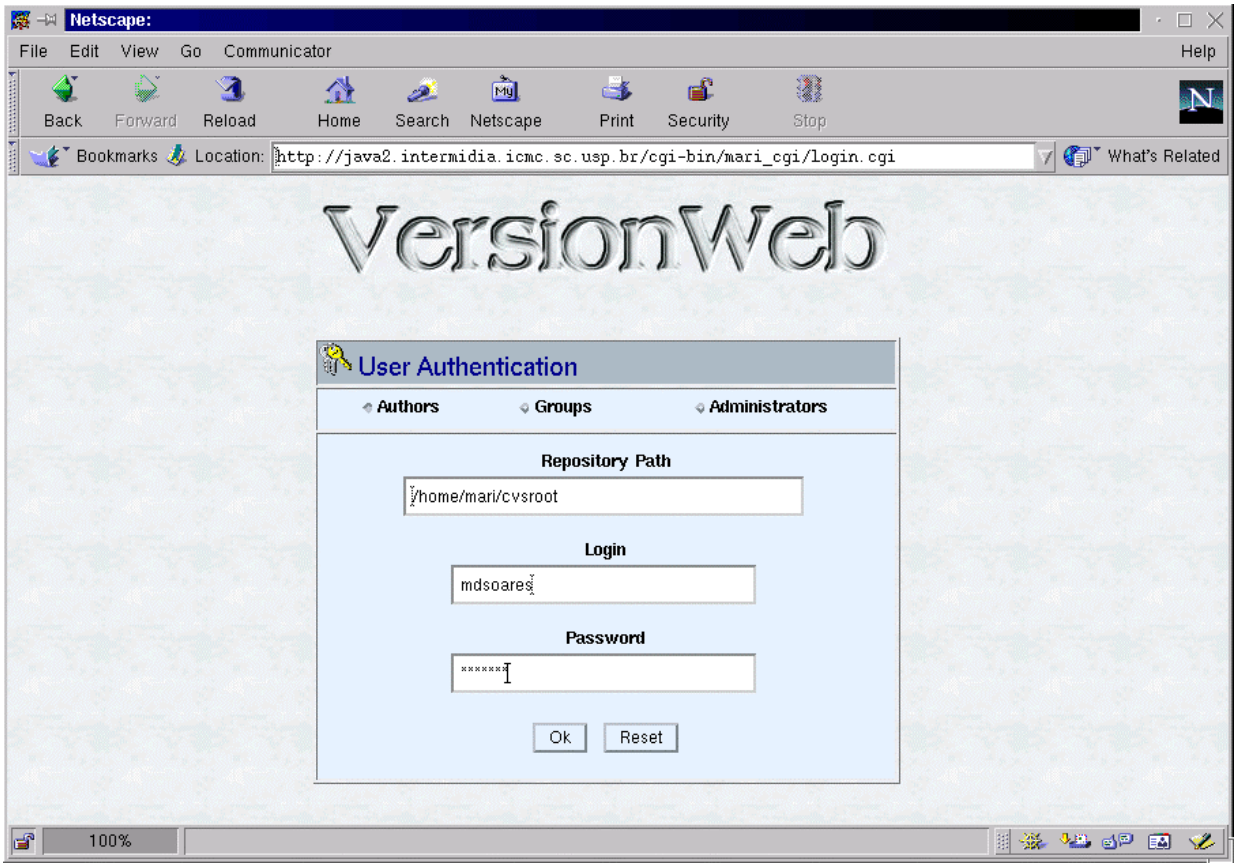


Figura 5.3 - Autenticação de usuários na *VersionWeb*

Os usuários devem, no momento da autenticação, informar o seu tipo (*Authors*, *Groups* ou *Administrators*), o caminho completo onde reside o repositório CVS (*Repository Path*), seu *username* (*login*) e sua senha (*Password*). Somente usuários do tipo "grupo" não precisam informar o caminho do repositório, pois este já é automaticamente especificado para a ferramenta quando o usuário aciona o *link* para a *VersionWeb* na página que ele está visitando.

5.3.2. Gerenciamento de usuários (por administradores)

Após o processo de autenticação, cada tipo de usuário tem acesso a um determinado conjunto permitido de serviços. Se o usuário se autenticou como um administrador, ele é responsável por gerenciar todos os usuários da ferramenta.

A partir da interface de gerenciamento de usuários (**Figura 5.4**), ao administrador são disponibilizadas operações para controle de informações dos usuários em cada um dos tipos definidos. Essas operações são:

- **Insert**: insere um usuário em uma das listas de usuários da ferramenta (*Authors*, *Groups* ou *Administrators*); para isso, deve-se selecionar a lista que o usuário irá pertencer (opções no botão de rádio da **Figura 5.4**), preencher os campos *name*, *login*, *password* e clicar sobre o botão "Insert".
- **Change**: permite alterar os dados de um usuário (*name*, *login* e *senha*); para isso, deve-se selecionar o usuário em uma das listas, preencher os campos *name*, *login* e *password* com os novos valores e clicar sobre o botão "Change".
- **Delete**: permite remover um usuário de uma das listas de usuários da ferramenta; para isso, deve-se selecionar o usuário em uma das listas e clicar sobre o botão "Delete".
- **Transfer**: permite transferir um usuário de uma lista para outra; para isso, deve-se selecionar o usuário em uma das listas, especificar a nova lista para a qual ele será transferido (marcando no botão de rádio) e clicar sobre o botão "Transfer".

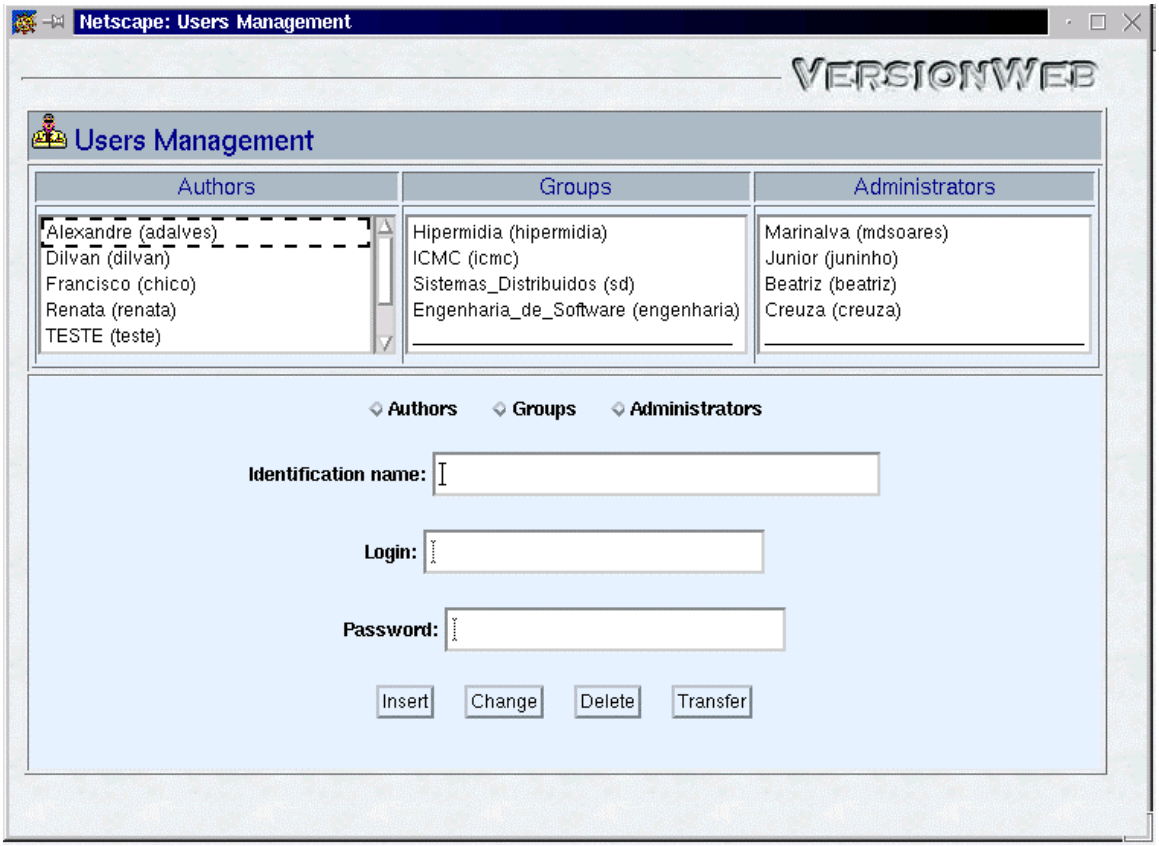


Figura 5.4. Interface de gerenciamento de usuários da *VersionWeb*

Após inserir um usuário, o administrador deve sair do ambiente da *VersionWeb*, efetuar o seu *login* no CVS através da linha de comando no servidor para que o novo usuário possa manipular

os arquivos (se o usuário for um autor ou administrador). Para os usuários do tipo "grupo" não é preciso que o administrador faça seu *login* no CVS, pois eles não têm permissão para manipular os arquivos do repositório. A Seção seguinte descreve os serviços de auxílio aos autores, relacionados com o controle de versões das páginas da *Web*, disponibilizados pela ferramenta.

5.3.3. Gerenciamento de Arquivos (por autores)

Se o usuário se identificou como um autor, ele terá acesso aos serviços relacionados com o gerenciamento de arquivos (interface ilustrada na **Figura 5.5)**. Nessa interface é exibido todo o conteúdo (arquivos e diretórios) do repositório CVS requisitado na tela de *login* pelo autor. Os arquivos gerenciados pelo CVS podem ser tanto texto como binários. A partir dessa interface (**Figura 5.5**) estão disponíveis as operações básicas de manipulação de arquivos e as operações básicas usuais do CVS, que são:

- **List Directory**: lista o conteúdo do diretório selecionado.
- **Up Level**: sobe um nível na árvore de diretório.
- **Rename**: renomea um diretório/arquivo selecionado.
- **Delete**: remove um diretório/arquivo selecionado.
- **Remote Checkout**: faz *checkout* remoto (da versão corrente) do arquivo selecionado no próprio servidor. Essa opção deve ser usada somente para alterações rápidas e pequenas em arquivos texto para que o *commit* seja feito logo em seguida.
- **Local Checkout**: faz *checkout* (da versão corrente) de um arquivo/diretório para a máquina do usuário. Essa opção deve ser utilizada para alterações mais longas e demoradas, ou seja, quando o *commit* não tiver que ser feito logo em seguida. Ao clicar nesse botão, o(s) arquivo(s) são listados em uma nova janela do *browser* para que o usuário faça o *download*.
- **Versions History**: lista as versões de um arquivo selecionado com seus respectivos autores, data, hora e mensagem de *commit*.
- **Versions List**: abre uma nova janela com uma lista de todas as versões e *branches* do arquivo selecionado e operações permitidas sobre elas.
- **Diffs**: permite o autor visualizar as diferenças de conteúdo (localizar alterações) entre as diversas versões do arquivo selecionado.
- **Create directory**: cria um diretório no repositório;

- **Add file**: adiciona um arquivo (que está local) ao repositório (no servidor) e o coloca sob o gerenciamento do CVS.
- **Commit of the local checkout**: faz o *commit* de um arquivo e gera uma versão subseqüente àquela da qual foi feito o *checkout* local. Para isso, o usuário deve procurar pelo arquivo em sua máquina (clicando em "*browser*") ou informar o caminho completo onde reside o arquivo.
- **Reload this page**: faz o *reload* da página atualizando (no *browser*) o conteúdo do repositório quando um novo arquivo é adicionado.

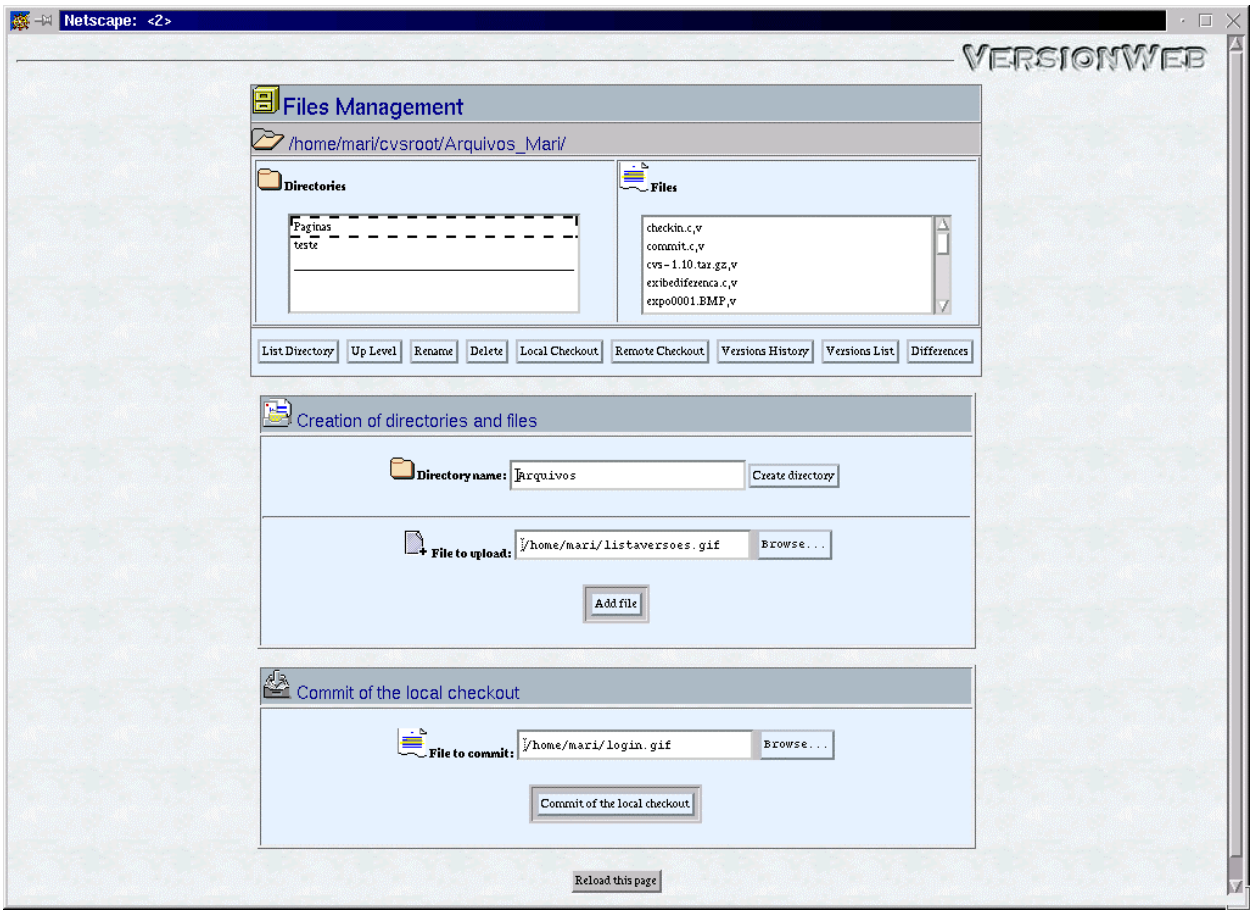


Figura 5.5 - Interface principal de gerenciamento de arquivos

De acordo com suas funções, a maioria dessas operações prossegue abrindo outras janelas de interação com o usuário, de maneira a efetuar as tarefas necessárias à sua realização. Se o usuário fizer *checkout* local de algum arquivo ou diretório (opção disponível na interface da **Figura 5.5**), ele obterá a tela da **Figura 5.6** para fazer *download* dos arquivos a serem transferidos para a máquina local.

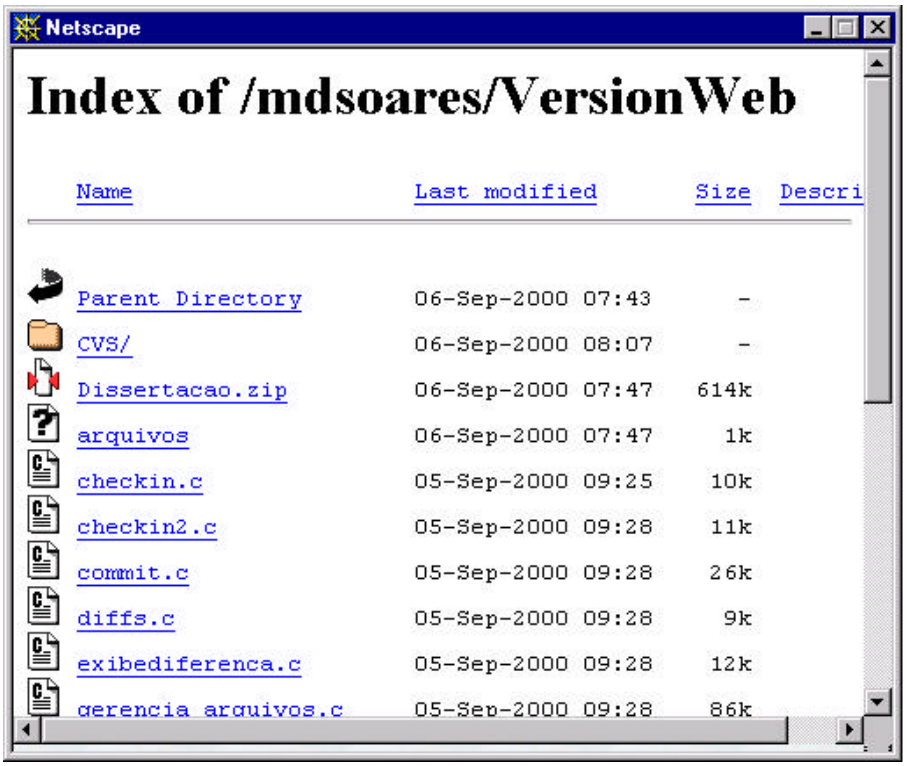


Figura 5.6 - Lista de arquivos para *download* quando se faz *checkout* local

A interface da figura acima permite que o autor faça o *download* (*checkout* local) de um arquivo ou de um diretório inteiro para alteração em sua própria máquina. A opção da interface principal de gerenciamento de arquivos que permite fazer o *commit* de um arquivo do qual foi feito *checkout local* é "Commit of the local checkout" também ilustrada na tela da **Figura 5.5**.

Se o usuário fez *checkout* local de um diretório inteiro e alterou todos os seus arquivos, o *commit* deverá ser feito para cada arquivo, um a um, e os seus nomes devem ser os mesmos, ou seja, o usuário não pode alterar os nomes dos arquivos em sua máquina e depois fazer o *commit* com nomes diferentes, senão o CVS não irá reconhecer esses arquivos e não irá gerar as versões subseqüentes àquelas das quais se efetuou o *checkout* local. Ao fazer o *commit*, a *VersionWeb* verifica se foi feito *checkout* local daquele arquivo em algum momento, e caso não tenha sido feito o *checkout* local, o *commit* retornará como resultado uma mensagem de erro dizendo que o usuário não efetuou o *checkout* daquele arquivo e que portanto o *commit* não poderá proceder.

Para a modificação de arquivos texto de fácil edição, que não exijam nenhuma ferramenta ou editor específico para essa alteração, a *VersionWeb* possibilita um procedimento ágil, por meio da opção "*Remote Checkout*" também ilustrada na **Figura 5.5**. Quando se faz o *checkout* remoto

de um arquivo (arquivo texto, pois a alteração de arquivos binários não pode ser feita diretamente no *browser*), obtém-se a interface da **Figura 5.7** que exibe o seu conteúdo e que permite fazer alterações nele.

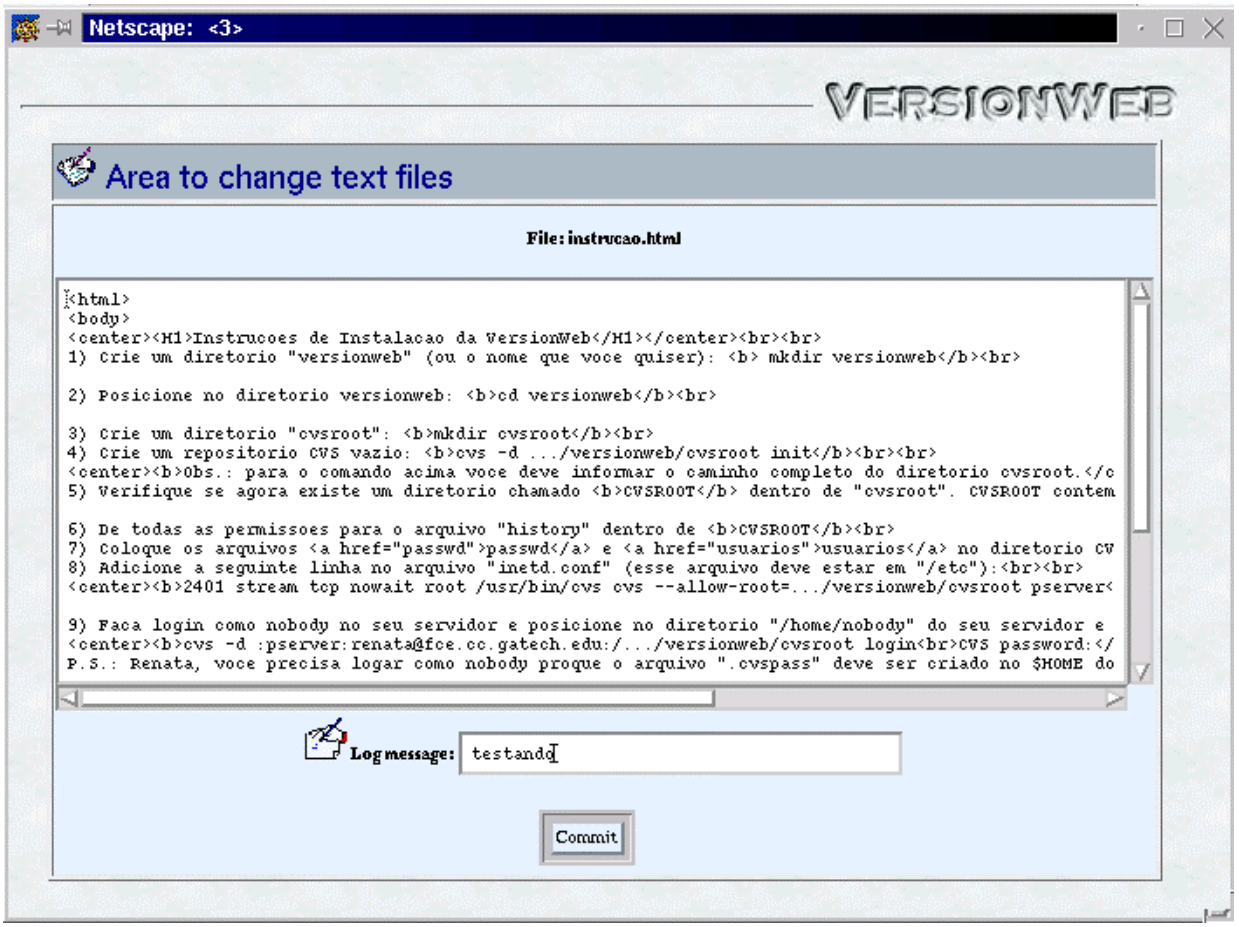


Figura 5.7 - Área de alteração do conteúdo de um arquivo texto com *checkout* remoto

A interface ilustrada acima exibe todo o conteúdo do arquivo, do qual se efetuou o *checkout* remoto, para alteração. Além disso, pode-se inserir comentários sobre as alterações que foram feitas no conteúdo do arquivo por meio de um campo de entrada (Log message) na interface. Esses comentários são registrados juntamente com o autor, a data, a hora e a nova versão gerada do arquivo após o *commit*.

O usuário poderá também, através da opção "Versions History" da interface ilustrada na **Figura 5.5,** obter mais informações sobre os arquivos do repositório, tais como: lista das versões existentes, autores, datas e comentários sobre as revisões (esses comentários são inseridos no

momento em que se faz o *commit* de um arquivo para gerar uma nova versão). Essas informações são exibidas em um tela como mostra a **Figura 5.8**.

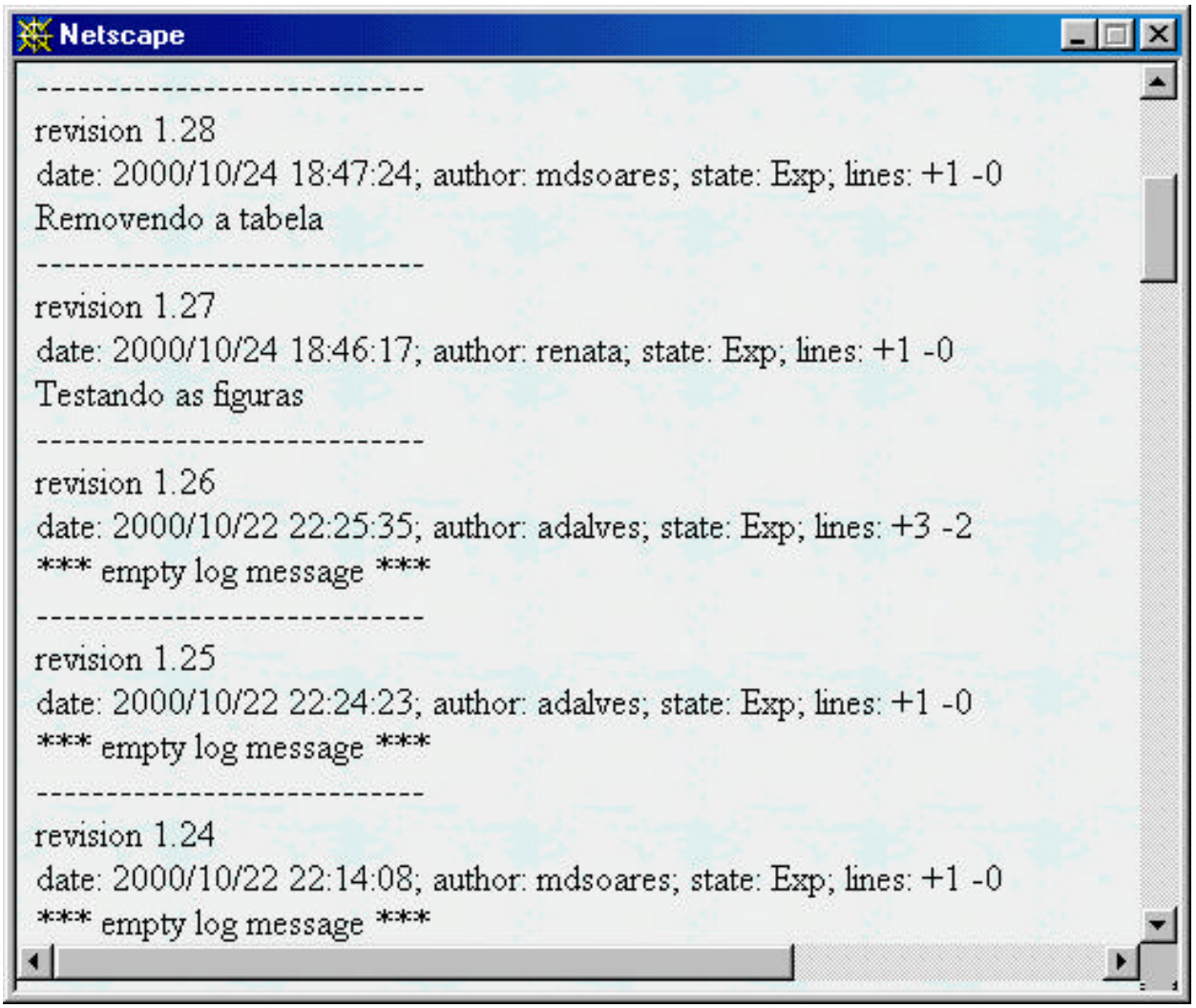


Figura 5.8 - Log de história de um arquivo

O autor poderá também, através da opção "Versions List" da interface ilustrada na **Figura 5.5**, obter a lista de todas as versões e *branches* de um arquivo com suas respectivas datas e autores (veja **Figura 5.9**), fazer *checkout* remoto e/ou local sobre qualquer versão do arquivo e fazer o *commit* das alterações efetuadas. Neste caso, para o *commit*, o autor deverá especificar uma *branch* a ser gerada (mais adiante está sendo explicado como gerar *branches*), pois o *checkout* não foi efetuado, normalmente, sobre a versão corrente do arquivo, e sim sobre uma versão anterior à atual.

Além disso, o autor poderá fazer *checkout* remoto e/ou local de uma *branch* (quando se faz *checkout* de uma *branch*, o CVS na verdade sempre faz o *checkout* da última versão daquela *branch*) e, neste caso, o *commit* gera uma versão automática para aquela *branch* e o autor não precisa especificar um número para ela.

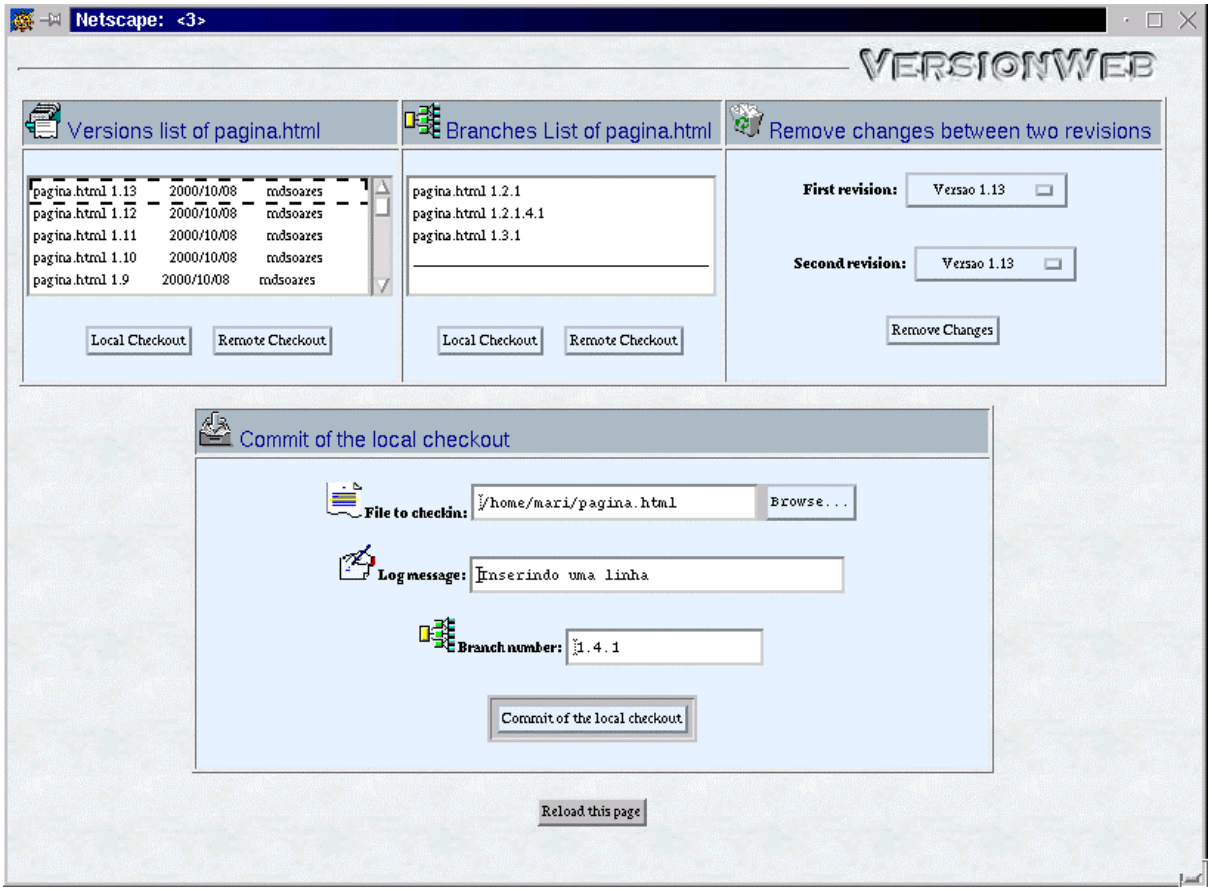


Figura 5.9 - Lista de versões de um arquivo e operações permitidas sobre suas versões

A lista de versões ilustrada na interface da **Figura 5.9** inclui todas as versões principais (linha principal de desenvolvimento - por exemplo, 1.1, 1.2, 1.3, etc) e versões das *branches* existentes (por exemplo, 1.1.1.1, 1.1.2.1, 1.1.2.1.2.1, etc), lista de *branches*, opção para remover alterações entre versões e opção para fazer o *commit* de um *checkout* local.

Se o usuário fizer *checkout* (remoto ou local), por exemplo, da revisão 1.2 e já existir a revisão 1.3, ele deverá especificar um número de versão (normalmente uma *branch*) a ser gerado no momento do *commit*, caso contrário irá surgir um conflito entre versões e o *commit* não poderá ser efetuado. A *branch* a ser gerada poderá ser 1.2.1, 1.2.2, 1.2.3, etc., e não 1.1.1, 1.1.2, etc (a menos que o *checkout* tenha sido feito da revisão 1.1).

O CVS, no momento do *commit*, gera uma versão automática para a *branch* especificada, ou seja, se o autor especificar o número 1.2.2 para a *branch* a ser gerada, o resultado será 1.2.2.1, onde "1" é a versão da *branch* (neste caso é a primeira versão da *branch*).

Se o *checkout* de uma das versões da lista de versões for remoto (opção "Remote Checkout" da **Figura 5.9**), o autor obterá a tela da **Figura 5.10**. Se for local, o *commit* deverá ser feito através da opção "Commit of the local checkout" da **Figura 5.9**.

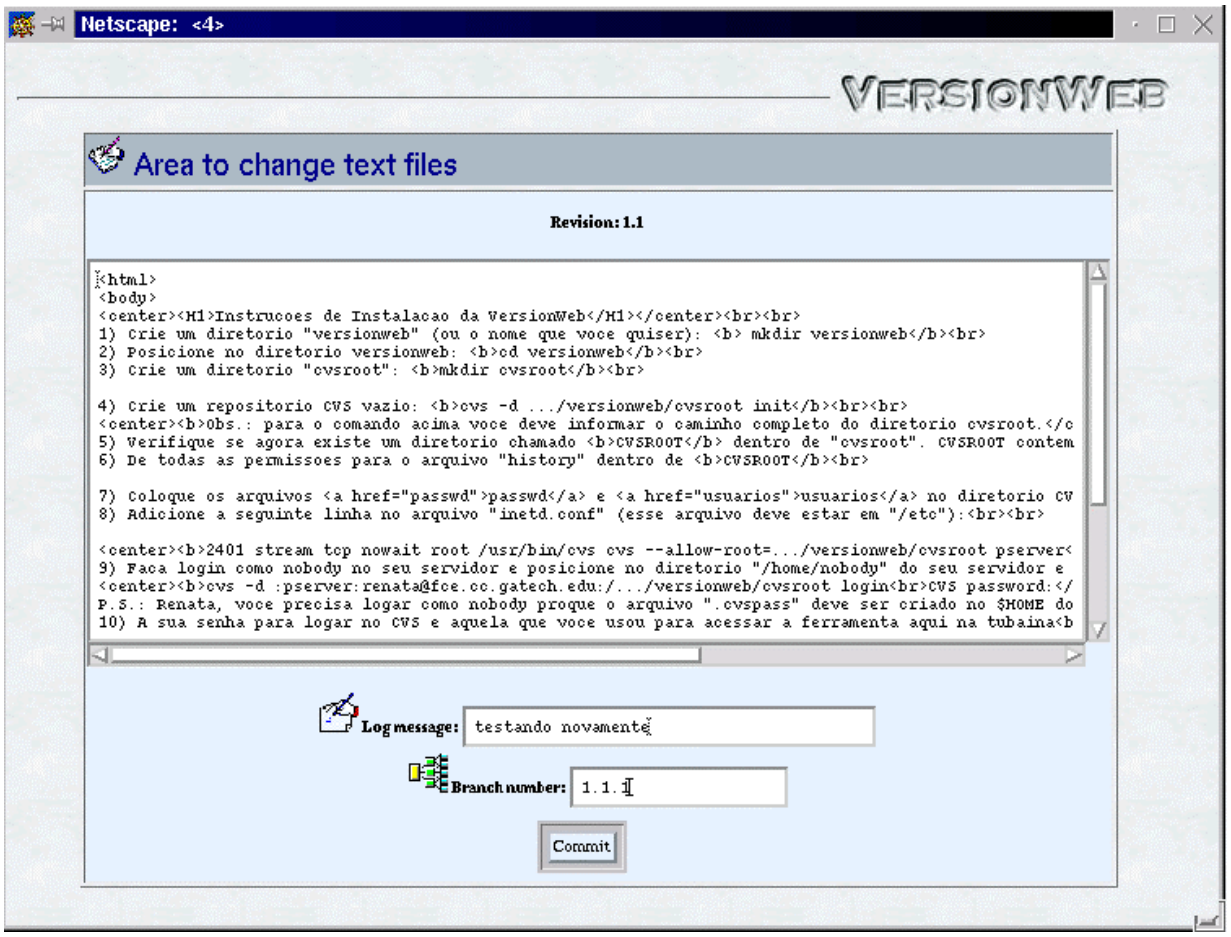


Figura 5.10 - Área de alteração do conteúdo de um arquivo texto com opção para gerar *branches*

A interface da **Figura 5.10** é semelhante àquela da **Figura 5.7**, diferenciando-se apenas pela adição de um campo para a especificação de um número para a *branch*.

O autor poderá ainda, através da interface ilustrada na **Figura 5.9**, fazer *checkout* remoto e/ou local de uma *branch* (que aparece na lista de *branches*). Neste caso, não é preciso especificar um número de versão a ser gerado, pois o *checkout* é feito sempre da última versão da *branch* e uma

versão automática será gerada para ela no momento do *commit*. Por exemplo, suponha que existam 3 versões para a *branch* 1.3.1 (1.3.1.1, 1.3.1.2, 1.3.1.3). Ao fazer o *checkout* da *branch*, o CVS pega automaticamente sua última versão (1.3.1.3) e o *commit* irá gerar a versão 4, ou seja, 1.3.1.4. Se o *checkout* for local, o *commit* deve ser feito através da opção "Commit of the local checkout" ilustrada na mesma interface da **Figura 5.9**. Se for remoto, a tela obtida é a mesma ilustrada na **Figura 5.7**, sem a opção de gerar *branches*, pois nesse caso a versão é gerada de forma automática como explicado anteriormente.

Ainda na interface da **Figura 5.9**, o autor poderá remover alterações feitas entre duas versões diferentes do arquivo. Para isso, ele deve selecionar as duas versões do arquivo (a primeira versão selecionada deve ser maior que a segunda) e clicar sobre o botão "Remove Changes". Por exemplo, se o autor selecionar as versões 1.5 e 1.2, o CVS remove todas as alterações que foram feitas entre a versão 1.2 e 1.5 e gera uma nova versão com a remoção efetuada, ou seja, essa nova versão irá conter, na verdade, o conteúdo da versão 1.2. Isso pode ser útil quando se deseja recuperar uma versão e trabalhar sobre ela na linha principal de desenvolvimento, e não em uma *branch* separada.

O autor poderá também visualizar as diferenças de conteúdo entre duas versões de um arquivo através da opção "Diffs" da **Figura 5.5**. A **Figura 5.11** ilustra a tela que permite escolher as versões e a forma para a visualização das diferenças quando a opção "Diffs" é acionada.

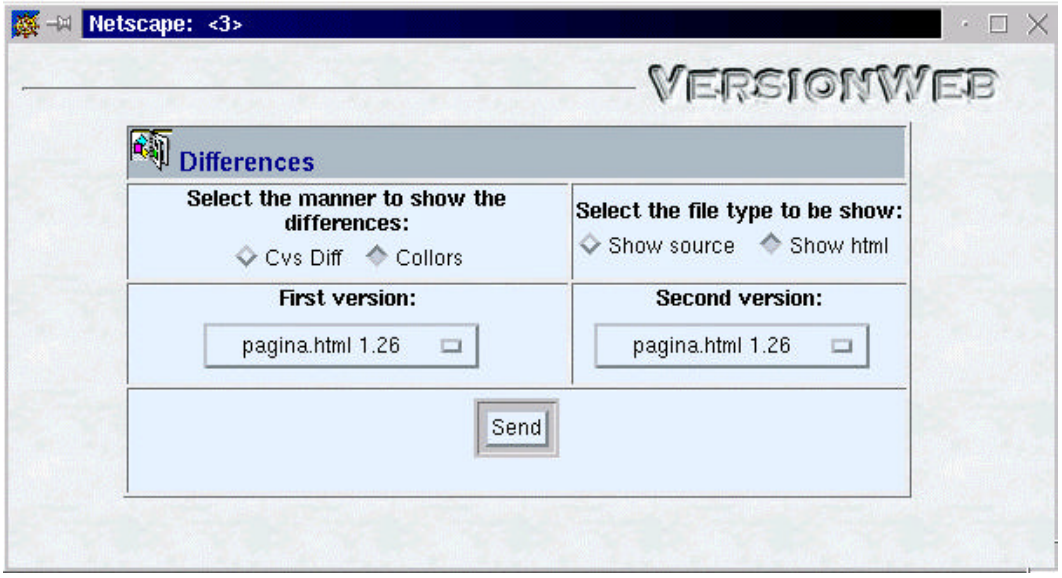


Figura 5.11 - Interface para o usuário escolher as versões para visualizar as diferenças

A interface apresentada na **Figura 5.11** exibe a lista de versões do arquivo e opções para visualizar as diferenças de conteúdo entre elas através de cores ou no formato do próprio CVS. Além disso, se o arquivo for um arquivo HTML, o usuário poderá optar por visualizar essas diferenças do fonte do arquivo ou do HTML como é interpretado pelo *browser*. Exemplos de como podem ser vistas essas diferenças estão ilustrados nas **Figura 5.14** e **5.15** e **5.16** da sub-seção a seguir.

5.3.4. Lista de versões da página (para internautas ou grupos específicos)

Um dos objetivos da *VersionWeb* foi o de permitir que os usuários, durante a navegação, tivessem acesso às informações que alguma vez estiveram disponíveis, mas que, geralmente, como conseqüência das constantes atualizações das páginas não estão mais, naquele momento, sendo apresentadas. Por meio da recuperação de versões anteriores dessas páginas e localização de alterações, o usuário pode rever informações disponibilizadas anteriormente.

Para atender esse objetivo, como mencionado na sub-seção 5.3.1, o administrador pode escolher entre permitir que todos os internautas ou um grupo específico destes tenham acesso às versões da página. Se o administrador optar por deixar que apenas grupos específicos de internautas tenham esse acesso, a *VersionWeb* requer que o usuário internauta faça parte de algum grupo já cadastrado. Para isso, a página que ele estiver visitando, se estiver sob controle de versão, conterá um *link* para a ferramenta *VersionWeb*, como a página ilustrada na **Figura 5.12**.

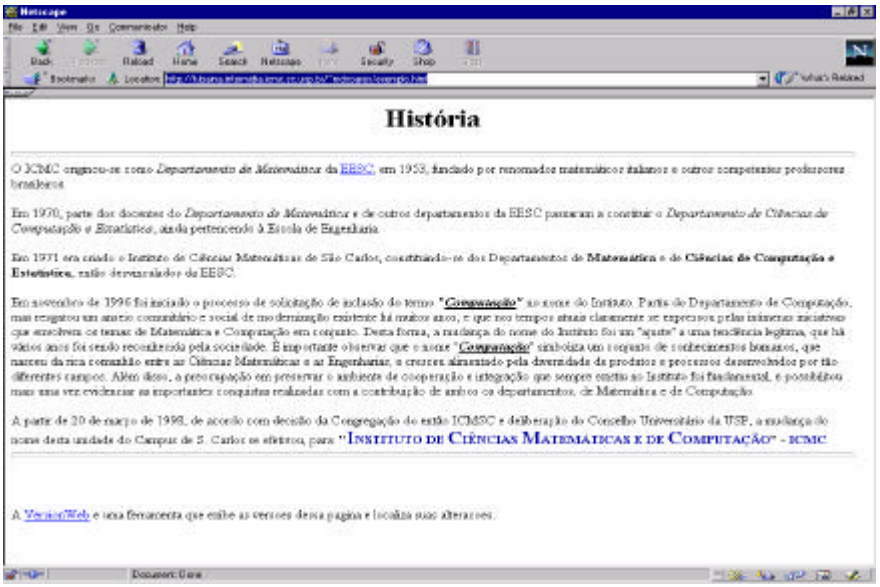


Figura 5.12 - Exemplo de uma página que contém um *link* para a *VersionWeb*

Esse *link* leva o usuário, primeiramente, à tela de autenticação de usuários ilustrada na **Figura 5.3** e, a seguir, o leva à interface de visualização das versões da página que está apresentada na **Figura 5.13**.

O *link* que dá acesso à *VersionWeb* pode estar definido em qualquer lugar da página. Nesse *link*, o autor deve incluir o caminho onde reside o repositório CVS na máquina servidora, o diretório onde está a página e o nome do arquivo HTML referente à página. Dessa forma, a ferramenta localiza a página no servidor e permite a recuperação de suas versões por parte do usuário. Após sua correta identificação (através da tela de autenticação de usuários ilustrada na **Figura 5.3)**, o usuário obterá a tela mostrada na **Figura 5.13** a seguir.

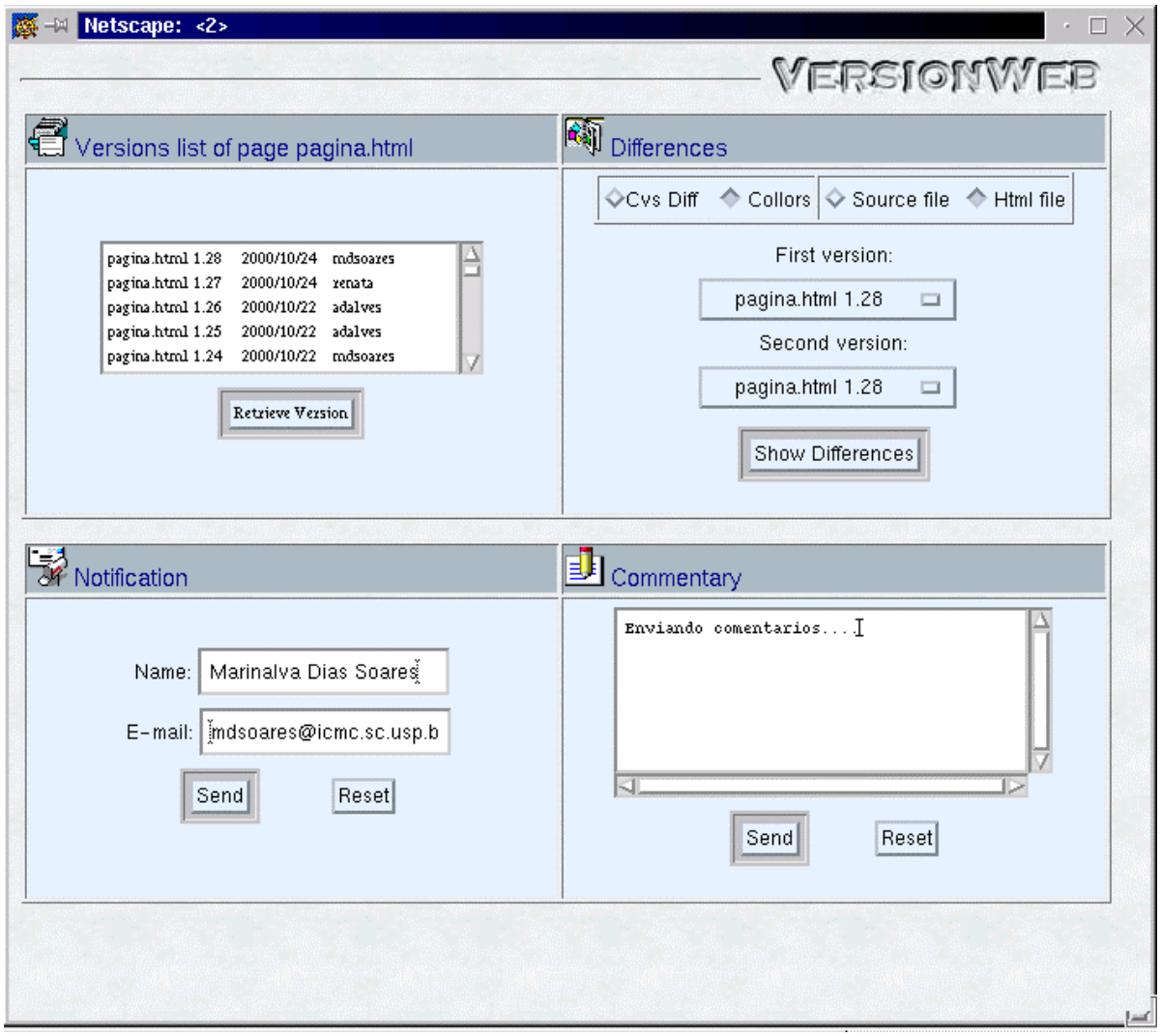


Figura 5.13 - Interface principal de recuperação de versões pelos internautas

Se o administrador permitir que todos os internautas tenham acesso às versões da página, eles terão acesso direto à interface ilustrada acima quando acionarem o *link* para a ferramenta, ou seja, eles não terão de passar pela tela de autenticação de usuários.

A partir da interface ilustrada na **Figura 5.13**, o internauta poderá realizar quatro operações:

- recuperar uma versão para visualização;
- visualizar as diferenças de conteúdo entre duas versões através de cores ou do próprio formato exibido pelo CVS;
- pedir para ser notificado de novas versões disponíveis da página;
- e enviar comentários ou críticas sobre a página e/ou ferramenta para os autores.

Como pode ser observado na **Figura 5.13**, a lista de versões para recuperação é exibida juntamente com a data e o autor da geração de cada versão. Se o usuário desejar recuperar uma versão da página para visualização, ele deverá selecionar a versão desejada e clicar sobre o botão "Retrieve Version". Feito isso, a versão da página é recuperada em uma nova janela do *browser* para visualização como uma página normal, ou seja, como se ele estivesse especificado diretamente na URL uma página para ser mostrada no *browser*.

Geralmente, quando as alterações efetuadas na página são pequenas, fica difícil para o usuário identificar o que realmente mudou naquela página apenas através da visualização de uma versão de cada vez. Dessa forma, a *VersionWeb* possibilita que o internauta visualize as diferenças entre duas versões quaisquer da página, ou seja, o que realmente mudou de uma versão para outra.

Para isso, o usuário deverá informar se quer que essas diferenças sejam exibidas através de cores ou no formato mostrado pelo CVS (o que não é muito aconselhável para quem não está familiarizado com o CVS), e se deseja ver as diferenças no fonte da página ou na sua versão em HTML. Feito isso, ele deverá então especificar as versões para as quais as diferenças deverão ser exibidas. Após essas escolhas, ele poderá clicar sobre o botão "Show Differences" e as diferenças serão mostradas em uma nova janela do *browser*.

Se o usuário optar por ver as diferenças no formato do CVS, ele obterá uma tela com o conteúdo do arquivo com caracteres de marcação das diferenças como ilustra a Figura **5.14**. Com o

formato do CVS é preciso uma análise minuciosa do conteúdo para ver e entender essas diferenças.

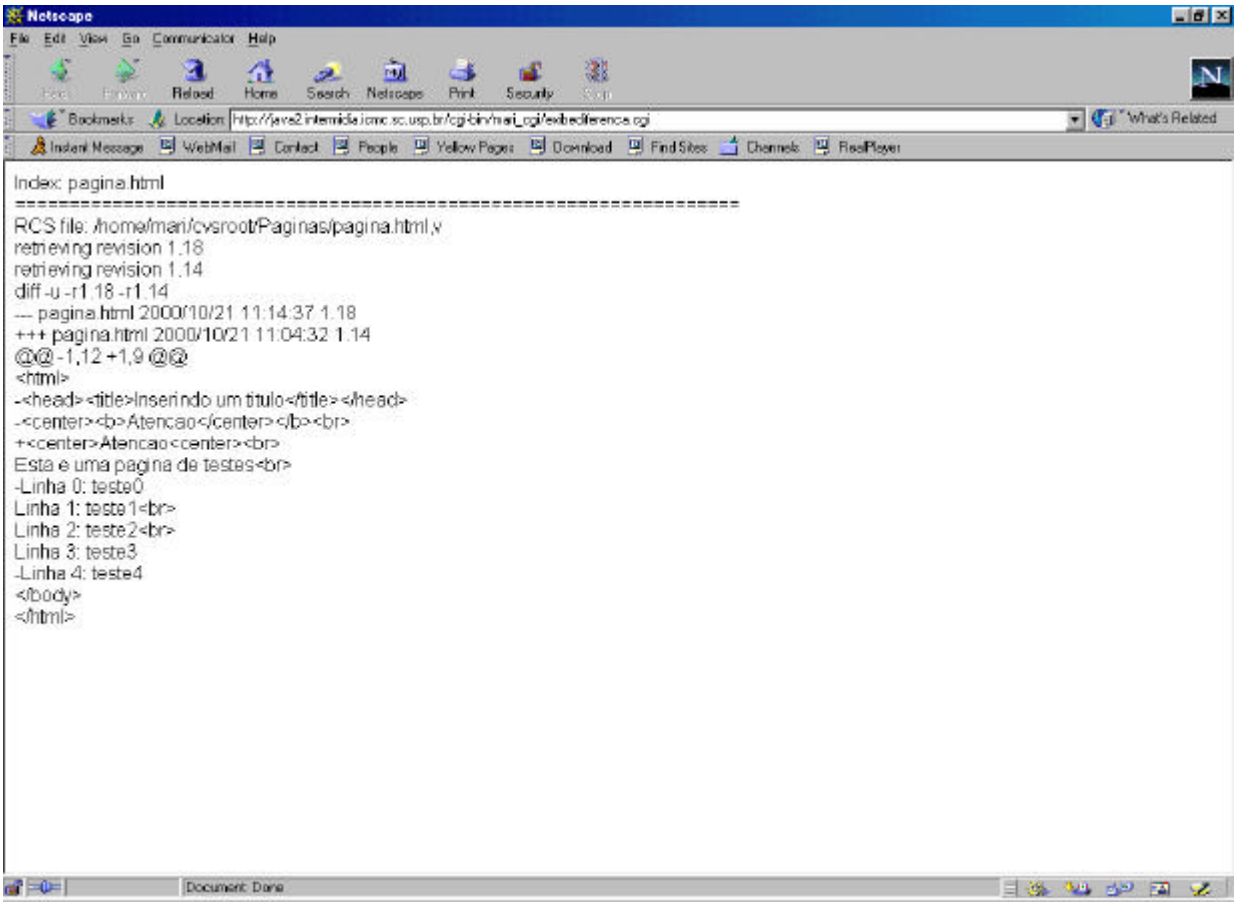


Figura 5.14 - Visualização das diferenças entre duas versões de uma página no formato do CVS

Por outro lado, se o usuário optar por visualizar as diferenças através de cores, ele obterá a tela da **Figura 5.15**, que visa facilitar a identificação das alterações. Porém, é bom lembrar que, algumas modificações como título e cabeçalho, por exemplo, não são visíveis na versão em HTML. Então, nesse caso, deve-se optar pelo código fonte. A **Figura 5.16** mostra as diferenças (do fonte) entre as mesmas versões da página correspondente àquela ilustrada na figura acima através de cores.

As **Figuras 5.14**, **5.15** e **5.16** estão ilustrando as diferenças entre as mesmas versões referentes ao mesmo arquivo.

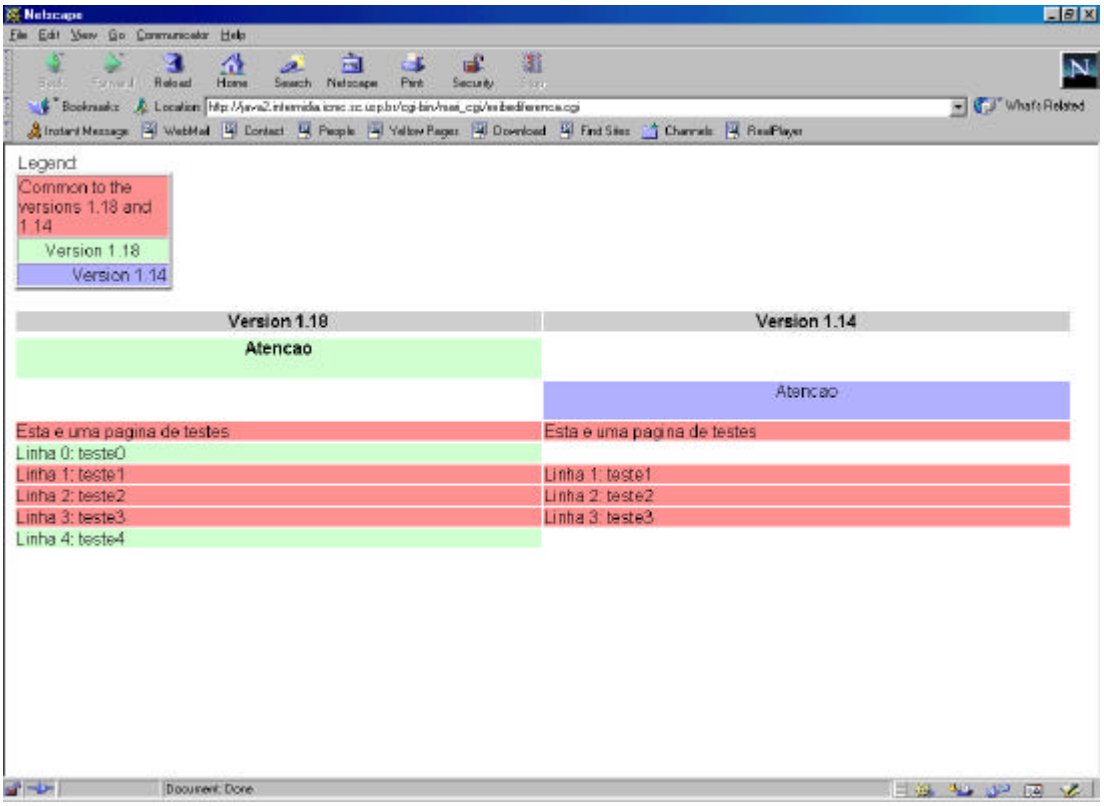


Figura 5.15 - Visualização das diferenças (do HTML) entre duas versões de uma página através de cores

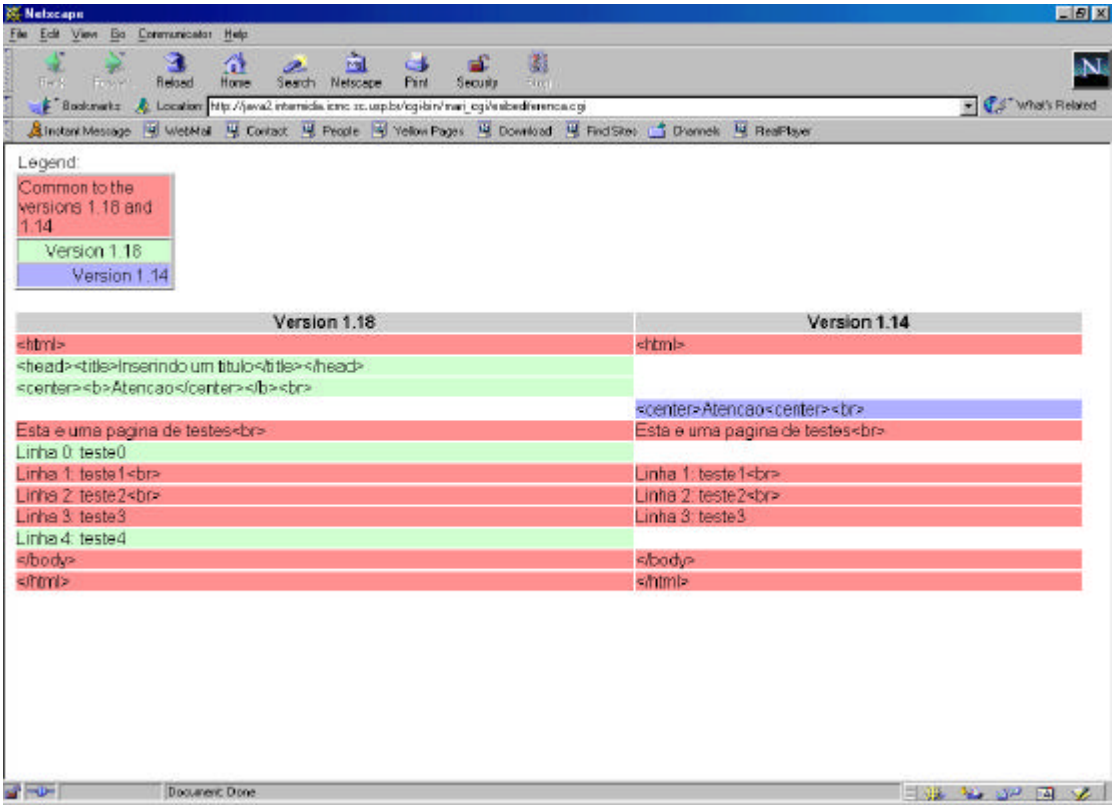


Figura 5.16 - Visualização das diferenças (do fonte) entre duas versões de uma página através de cores

Ainda através da interface ilustrada na **Figura 5.13**, o usuário poderá pedir para ser notificado de novas versões disponíveis da página. Essa notificação é feita automaticamente pelo CVS quando uma nova versão daquela página é gerada. Porém, o autor pode não disponibilizar essa nova versão, o que vai causar uma certa decepção no usuário, pois ele vai receber uma notificação de que uma nova versão da página foi gerada, mas que pode não estar disponível. Além disso, o usuário pode enviar sugestões ou críticas sobre as alterações efetuadas na página. Todos os autores da *VersionWeb* terão acesso aos comentários enviados pelo usuário.

5.4. Testes de usabilidade da *VersionWeb*

A *VersionWeb* foi desenvolvida como um sistema protótipo, apresentada nas seções anteriores, projetada para viabilizar o controle de versões de páginas *Web* na própria *Web*. Ela visa atender desde usuários de *Web* mais especialistas (desenvolvedores de páginas ou *sites* por exemplo) e que têm intensa interação com a *Web*, assim como usuários da *Web* menos especializados em autoria de páginas, cujo objetivo de navegação pelas páginas é buscar informações, mas que freqüentemente não as encontram da forma como haviam obtido alguma vez anterior.

Com o propósito de se avaliar a usabilidade da proposta viabilizada na *VersionWeb*, nesta Seção apresentamos dados experimentais e uma análise de respostas fornecidas por um conjunto de usuários após terem utilizado a *VersionWeb* na realização de um conjunto pré-definido de tarefas.

Para essa avaliação, foi elaborado um formulário contendo um conjunto de 14 tarefas básicas a serem executadas utilizando-se a *VersionWeb*, um questionário com 7 perguntas simples sobre como o usuário realizou as tarefas requeridas, além de 8 solicitações de dados sobre o usuário e de opiniões de respostas livres que ele forneceria conforme sua conveniência. O referido formulário pode ser visto no Apêndice 1.

O teste de usabilidade, em linhas gerais, consistiu da entrega do formulário contendo uma descrição inicial sobre *VersionWeb*, o conjunto de tarefas a serem executadas e o questionário para coleta de *feedback* sobre a usabilidade da ferramenta. Posteriormente, as respostas foram devolvidas. A partir de um levantamento das respostas obtidas, foi possível se analisar aspectos favoráveis e desfavoráveis sobre o uso da *VersionWeb*.

O objeto em estudo para o levantamento dos dados experimentais foi o uso da *VersionWeb* por um número de usuários da WWW. Especificamente neste experimento, a *VersionWeb* foi avaliada por 20 usuários "entrevistados" via formulário. A opção de utilizarmos formulário foi para que os usuários entrevistados tivessem maior flexibilidade de horário, se sentissem bem à vontade e que as respostas tivessem caráter confidencial (pessoal) preservado.

O objetivo desse experimento foi a coleta de impressões sobre a interação com a ferramenta, dado um conjunto pré-definido de tarefas a serem realizadas. Por conveniência, para a execução dessas tarefas e respectiva obtenção dos dados em questão, foi solicitado a realização dos testes a usuários do próprio Departamento de Computação do ICMC-USP. Esses usuários, em geral, possuíam nível de conhecimento computacional alto, mas com diferentes níveis de conhecimento sobre controle de versões. Esses usuários são estudantes de Computação, sendo 70% em nível de Mestrado, 15% em nível de Graduação e 15% em nível de Doutorado.

Esse teste de usabilidade, embora tenha sido aplicado a um número relativamente pequeno de usuários, foi muito importante, pois apresentou uma perspectiva de necessidades que não tinham sido consideradas até esta etapa de desenvolvimento da ferramenta. Por outro lado, mostrou também que as funcionalidades estão atendendo às tarefas propostas de forma satisfatória, uma vez que as respostas às solicitações de opinião sobre a ferramenta, em sua maioria, foram relativas à aparência e apresentação das telas. Um aspecto também importante observado foi de que, nas solicitações de opiniões (que o usuário forneceria conforme sua conveniência), 100% dos entrevistados forneceram respostas livres, variadas e indicando aspectos de melhoria da interface (o que de certa forma, representou que a ferramenta tenha despertado interesse).

A seguir, estão transcritas algumas sugestões, exemplificando os tipos de comentários fornecidos à solicitação ***"O que poderia ser melhorado na ferramenta?"***

.... Diminuir interface para que caiba no monitor 800x600;

.... O download dos arquivos que estão remotos;

.... Deixar as janelas sempre maximizadas, na interface que exibe lista de versões informar qual é a última versão, colocar help em todas as janelas;

.... Colocar mensagens explicativas durante os processos e durante click nos botões;

.... Não utilizar várias janelas (a menos que o usuário queira);

.... ter um *feedback* maior nas tarefas 6 e 10;

.... Acréscimo de botões de fechar nas janelas, não utilizar barras de *scroll*;

.... Clicar no diretório (tarefa 2) apenas e mostrar o conteúdo sem ter de clicar em "List Directoy";

.... Colocar opção de help para cada uma das funcionalidades;

Um aspecto interessante observado foi de que, embora os usuários possuíssem conhecimentos de Computação e muitos deles fossem usuários freqüentes da *Web* (65% responderam que sim à pergunta ***"Você utiliza freqüentemente a Web?"***), a maioria desconhecia o CVS (pois 23% deram resposta afirmativa à questão (a) ***"Você já conhecia o CVS utilizado em software?"***), e metade dos estudantes se enquadraram com níveis de conhecimento "pouco" ou "nenhum" sobre os termos relacionados com controle de versões (questão c) ***"Qual o nível de conhecimento dos termos relacionados com controle de versões que você possuía?"***), mesmo com 46% dos entrevistados realizando pesquisas em Engenharia de Software. Embora a amostragem de usuários não seja tão representativa, essa pouca incidência de conhecedores de CVS e/ou controle de versões pode indicar que controle de versões é um tema que talvez seja pouco estudado ou que esse tema se apresenta de "difícil" aplicação.

A maioria dos usuários, 77% dos entrevistados, manifestou que a *VersionWeb* facilita de alguma forma o entendimento da funcionalidade do CVS, embora nenhuma resposta negativa tenha sido emitida (os 77% responderam que sim à questão b) ***"Você acha que a ferramenta facilita de alguma forma o entendimento da funcionalidade do CVS?"***).

Entre as desvantagens apontadas pelos usuários, destacamos duas, as quais consideramos também como potenciais temas de pesquisas futuras:

1. segurança
2. alterações concomitantes por vários autores

A primeira desvantagem requer uma atenção especial pois esse é o grande perigo que corre qualquer trabalho profissional na *Web*. Muitas companhias e instituições têm investido em alternativas para contornar esse problema, por exemplo, através de Intranets com mecanismos mais sofisticados de autenticação. A segunda desvantagem se enquadra como o desafio dos trabalhos distribuídos. Consideramos que a inclusão de mecanismos para notificação inteligentes seria uma alternativa para sincronizar os esforços. Como exemplo, implementar um alerta automático do tipo "agente" que avise, no momento adequado aos desenvolvedores, de que outros colegas estão também tendo acesso aos seus artefatos e quem são essas pessoas.

Os valores das respostas fornecidas revelaram ainda que a maioria dos usuários entrevistados apreciou a ferramenta e que gostaria de utilizá-la novamente (85% deram resposta afirmativa à questão g) "***Você gostaria de utilizar novamente a VersionWeb?***").

5.5. Considerações finais

Neste Capítulo foram descritas todas as funcionalidades da ferramenta *VersionWeb*, sua arquitetura e seus principais módulos. Observa-se que os módulos que atendem especificamente aos objetivos deste trabalho são "Gerenciamento de arquivos" e "Lista de versões da página" (ilustrados na **Figura 5.2**), onde o primeiro permite o gerenciamento de todo o conteúdo do repositório, especificado na tela de *login* pelo autor, mantendo as diversas versões geradas dos arquivos e possibilitando o trabalho sobre eles por vários autores de forma remota ou local através da *Web*. Já o módulo "Lista de versões da página" permite ao navegador visualizar as diferentes versões da página que estiver visitando, bem como as diferenças de conteúdo entre elas. Além disso, esse módulo permite também que o usuário informe aos autores que querem ser notificados quando houver uma nova versão da página e também fazer seus comentários e críticas.

Entretanto, para controlar melhor o acesso à *VersionWeb* e principalmente às funcionalidades de gerenciamento de arquivos, tornou-se necessário implementar um módulo de autenticação de

usuários, onde estes podem ser autores, administradores ou grupos específicos de internautas. Consequentemente, foi preciso também implementar o módulo de gerenciamento de usuários para que a manutenção destes pudesse também ser feito através da *Web*.

Com a finalidade de obter impressões sobre a interação dos usuários com a ferramenta, foram feitos alguns testes de usabilidade da *VersionWeb* com 20 usuários através de um conjunto de tarefas pré-definidas. Os testes mostraram resultados satisfatórios ao uso da *VersionWeb* e a importância de controle de versões tanto no ambiente WWW como em ambientes de desenvolvimento de *software*. Esses dois últimos comentários relativos à importância do controle de versões foram respostas sugeridas pelos próprios usuários que fizeram os testes, pois eles observaram que a *VersionWeb* não se adapta somente a autoria de páginas através da *Web*, mas ao desenvolvimento de qualquer tipo de *software* que envolva equipes de desenvolvedores. Logicamente, para o uso da *VersionWeb* em outros ambientes de trabalho, se faz necessária uma investigação sobre adaptação àquele ambiente, e até mesmo a inclusão de alguma funcionalidade extra que a *VersionWeb* não oferece, pois no geral, ela oferece as funcionalidades básicas requeridas para o gerenciamento de versões de arquivos em qualquer ambiente.

Embora os usuários que testaram a *VersionWeb* tenham feito algumas considerações com relação à interface, mais especificamente com relação ao seu tamanho, os testes mostraram que os objetivos deste trabalho, descritos inicialmente nesta dissertação, foram atingidos.

O Capítulo seguinte descreve algumas conclusões obtidas deste trabalho, as vantagens e desvantagens da ferramenta e algumas sugestões para trabalhos futuros.

6. Conclusões

6.1. Contribuições

O processo de mudança é uma realidade que ocorre com extrema freqüência na *World Wide Web*. No ambiente WWW, os leitores freqüentemente se queixam quando ao visitar uma página, percebem que esta já não possui o mesmo conteúdo ou até mesmo que ela não existe mais, decorrente da rápida e natural evolução das informações na WWW [Sommerville et al. 1998].

Os desenvolvedores de páginas *Web*, por sua vez, encontram dificuldades quando muitas pessoas estão envolvidas na construção em paralelo de uma mesma página ou de um conjunto de páginas relacionadas. Isso se deve do fato de que os autores trabalham independentemente em suas próprias cópias, tendo como principal problema a integração dessas cópias em um hiperdocumento final [Sommerville et al. 1998]. Além disso, em geral, o volume de documentos envolvidos é significativamente grande e foge a um controle simples da evolução de suas cópias, pois pouco ou nenhum gerenciamento de informação é fornecido.

As várias formas de atuação nesses dois cenários típicos para o controle de versões das páginas *Web* mostram que um suporte ao controle de versão dos arquivos para os autores, os quais trabalham em um desenvolvimento colaborativo, e um suporte à navegação por versões anteriores das páginas, por parte dos internautas, são alvos de investigação de muito interesse. Neste contexto foi desenvolvida a ferramenta *VersionWeb*.

Para tornar o controle de versão de páginas disponível na *Web*, de forma a facilitar o trabalho dos autores e de manter disponíveis aos internautas as informações de versões anteriores das páginas, foram pesquisados alguns assuntos relacionados ao controle de versão em geral, controle de versão na *Web*, exemplos de ferramentas automatizadas para o controle de versão e mecanismos para programação na *Web*.

Dessa forma, foram estudadas as principais funcionalidades de SCM durante todo o ciclo de vida de um *software*, alguns modelos de versões de *software* para SCM descritos na literatura e

também as formas de representação do produto de *software* (seus componentes e relacionamentos entre eles) e do espaço da versão (a forma como são representadas e organizadas as diversas versões geradas de um item de *software*). Com esse estudo, obteve-se o conhecimento sobre a importância do controle de versões para arquivos e/ou produtos de *software*, bem como o entendimento de diversos termos relacionados como versões, revisões, variantes, *branches*, etc. Todos esses conceitos foram descritos no Capítulo 2 dessa dissertação.

Após conhecidos esses conceitos, procuramos por ferramentas automatizadas de controle de versões que aplicassem alguns desses conceitos. Dentre as diversas ferramentas existentes, foram pesquisadas o SCCS, o RCS e o CVS. O CVS mostrou ser a ferramenta mais adequada para a realização deste trabalho devido a algumas de suas características adicionais em relação às duas primeiras como, por exemplo, o acesso remoto aos arquivos (através da rede) e o acesso simultâneo sobre arquivos pelos autores. O SCCS e o RCS foram descritos com mais detalhes na Seção 2.7 do Capítulo 2 e o CVS no Capítulo 3.

Foi realizado também um estudo da maioria das funcionalidades de ferramentas de apoio ao controle de versões em ambientes de autoria da *Web* atuais. As ferramentas encontradas foram WebRC, V-Web e AIDE, descritas no Capítulo de Introdução. Diferentemente de SCCS, RCS e CVS, elas não estavam disponíveis para utilização. No entanto, possibilitou uma visão geral das características relacionadas às tarefas de controle de versão na *Web* que se apresentam na literatura.

Com base nos estudos realizados sobre os requisitos principais para o versionamento de páginas *Web* (criação de páginas de forma cooperativa e gerenciamento das informações das páginas), foi possível definir uma arquitetura para a ferramenta (descrita na Seção 5.2 do Capítulo 5) de forma que o repositório, contendo os arquivos, ficasse armazenado no servidor. O acesso a esse repositório seria feito através da rede por meio de uma interface de aplicação que exibisse todo o seu conteúdo juntamente com as operações de controle de versão mais usuais.

Com base nessa arquitetura, foram pesquisados alguns mecanismos de programação para a *Web* existentes como CGI, Applets Java, Servlets e JSP (descritos no Capítulo 4) para a viabilização do desenvolvimento da ferramenta e disponibilização da mesma na *Web*, de forma que a criação e controle da maioria das informações fossem feitos através da *Web*. Após o estudo desses mecanismos, procuramos por um que fosse simples de se programar e bem difundido no

mercado de trabalho para que obtivéssemos rápido desenvolvimento. Assim, optamos pelo uso dos *scripts* CGI (cujas definições, características e formas de ação já foram descritas na Seção 4.2 do Capítulo 4) para estabelecer a comunicação e a interação entre os usuários (através de uma interface baseada em formulários HTML) e o CVS.

De forma geral, a disponibilização de uma ferramenta para o gerenciamento de versões de arquivos na *Web* com a arquitetura adotada na *VersionWeb* e, além disso, usando o CVS para permitir o acesso simultâneo sobre os arquivos (esses arquivos podem ser páginas *Web*, fontes de programas, documentação de projetos, textos, e outros) visa estimular a colaboração entre os autores que podem estar localizados em diferentes lugares, pois eles não precisam se preocupar tanto em comunicar uns aos outros que vão trabalhar em um determinado arquivo e não necessariamente precisam ter os arquivos locais em suas máquinas.

A abordagem adotada pela *VersionWeb*, de possuir o repositório centralizado no servidor, facilita ainda mais esse processo, pois as informações se tornam mais prontamente disponíveis aos autores à medida que cada um vai modificando os arquivos e fazendo o *commit*. O fato do CVS não bloquear os arquivos para edição por mais de um autor ao mesmo tempo gera uma produtividade maior entre os integrantes da equipe, pois a espera de liberação de um arquivo para escrita não é necessária e isso com certeza retardaria a obtenção do produto final.

Os recursos de localização das alterações feitas pelo autor, juntamente com data e comentários da versão gerada, dão um certo conforto e agilidade ao processo de desenvolvimento no sentido de que os autores não precisam estar se comunicando e analisando cada linha de código para ver o que foi alterado e quem alterou, pois isso é feito automaticamente. Além disso, os desenvolvedores dos *sites* não precisam colocá-los "indisponíveis" (fora do ar) enquanto estiverem fazendo modificações ou adaptações. Isso certamente causa menos aborrecimento aos internautas que estão visitando essas páginas, pois têm sempre uma versão disponível. Além disso, os desenvolvedores (empresários, por exemplo) não correm o risco de perder clientes pelo fato do *site* estar sofrendo alterações e estar fora do ar.

A princípio pode-se observar três contribuições iniciais deste trabalho: **a)** a disponibilização de uma ferramenta de controle de versões na *Web*, através de um ambiente não orientado a linhas de comando (como são a maioria das ferramentas de controle de versões atualmente), indicando claramente que o controle de versões de arquivos através do CVS pode ser feito via *Web*; **b)** o

uso de scripts CGI, tecnologia amplamente difundida, conhecida e fácil de usar mostrou ser eficiente e prático no estabelecimento da comunicação e interação dos usuários com o CVS; **c)** o fato dos autores não precisarem "memorizar" os comandos do CVS para manipulação dos arquivos é, com certeza, um outro aspecto positivo para aceitação da ferramenta para o trabalho colaborativo. Em geral, os desenvolvedores de páginas *Web* alegam que seu objetivo principal não é "aprender os comandos CVS" e que o conhecimento dos recursos de desenvolvimento na *Web*, por apresentarem freqüentes inovações, já lhes demandam um alto nível de aprendizado.

Ainda neste contexto, pode-se concluir que, entre as vantagens do controle de versão encontradas em ambientes de engenharia de *software* e ambientes de autoria, uma das mais significativas está relacionada com a reconstrução de versões anteriores de seu trabalho. Outras vantagens como localização das alterações, seus autores, data de alteração, etc., são também muito relevantes.

Como resultado deste trabalho, obteve-se uma publicação nacional no IV Workshop de Teses em Engenharia de Software (ocorrido juntamente com o XIII SBES - Simpósio Brasileiro de Engenharia de Software) em outubro de 1999 [Soares 1999], e foi submetido outro artigo ao IMSA - *International Conference on Internet Multimedia Systems Applications* em julho/2000 [Soares et al. 2000]. Recebemos a notificação de aceitação do artigo em agosto/2000 e o mesmo será apresentado pela Prof. Drª. Renata Pontin de Mattos Fortes entre os dias 19 e 23 de novembro/2000 em Las Vegas, EUA.

6.2. *VersionWeb*: vantagens e limitações

A *VersionWeb* oferece recursos de controle de versões de arquivos embutidos em formulários HTML que ativam o CVS. Uma vantagem óbvia é que o repositório e os arquivos de outros usuários ficam disponíveis diretamente a todos os autores. Para isso, basta que eles tenham um *browser Web* para permitir o acesso ao repositório no servidor.

Além disso, a *VersionWeb* pode ser facilmente estendida para incorporar outros recursos do CVS, bastando para isso acrescentar as opções na interface da ferramenta (formulários HTML) e construir o CGI para a tarefa correspondente. Uma outra vantagem da ferramenta é que ela pode ser usada tanto por desenvolvedores de páginas *Web*, como desenvolvedores de *software*, projetos, redações e textos para jornais pelos repórteres etc., pois, uma vez que a ferramenta gerencia páginas HTML, e estas não incluem somente arquivos texto, sua utilização em outras

áreas de trabalho e desenvolvimento não seria problemática. Por exemplo, o uso da *VersionWeb* em ambientes jornalísticos não somente permite a manutenção e o acesso das diversas versões das matérias e informações geradas, como também o trabalho cooperativo entre os jornalistas e/ou repórteres no sentido de revisões das matérias. Além disso, essas pessoas não precisam estar na mesma sala ou na mesma cidade para efetuarem essas tarefas através da *VersionWeb*.

A interação da *VersionWeb* se dá através de formulários HTML e scripts CGI, o que favorece ainda mais sua difusão e o seu uso, pois quase todos os servidores executam CGIs e os *browsers* trabalham com formulários HTML de forma eficiente. Além disso, o usuário não precisa de nenhum tipo de *plug-in* ou aplicação auxiliar para usar a *VersionWeb*. Embora o uso de CGI faça com que todas as operações sejam executadas no servidor, todas as entradas do usuário são pré-validadas com o uso de *JavaScript* na própria máquina cliente, pois assim, esse tipo de validação não precisa ser feita pelo CGI, o que seria mais demorado. As informações já são passadas no formato correto e válido para o CGI para que ele possa efetuar a operação com as chamadas ao CVS.

Por outro lado, a *VersionWeb* conforme implementação atual, possui algumas limitações. Uma delas é executar apenas em servidores Unix e/ou Linux, mas a instalação da ferramenta em outras máquinas (Unix ou Linux) requer apenas a recompilação dos programas fontes devido a algumas informações específicas da máquina que são utilizadas pelos CGIs. Não é preciso modificar nada no código fonte.

No caso de se instalar a ferramenta em um servidor Windows 95, 98, 200 ou NT, é preciso localizar a linha de código no CGI que faz uso do sistema de arquivos e de comandos do Linux e trocá-los pelos seus correspondentes do Windows e compilar os CGIs novamente. Essas seriam as maiores dificuldades do uso da *VersionWeb* em outras plataformas. Além disso, com o uso de CGIs, o acesso à ferramenta por um número muito grande de usuários ao mesmo tempo pode sobrecarregar o servidor, pois cada vez que um CGI é executado ele é inicializado na memória.

Em relação ao sistema V-Web descrito na Seção 1.3 de Trabalhos Relacionados no Capítulo de Introdução desta dissertação, a *VersionWeb* apresenta algumas vantagens como: exibe as diferenças entre versões de uma página, permite a edição simultânea dos arquivos por autores ao mesmo tempo e fornece vários recursos relacionados ao controle de versão.

Com relação ao sistema AIDE, a *VersionWeb* tem a vantagem de usar o CVS, o qual possui vantagens e características adicionais ao RCS e dá suporte ao desenvolvimento paralelo entre autores. O AIDE não oferece esse recurso. Além disso, no AIDE, o usuário deve especificar a URL a ser localizada para exibir as diferenças e as versões. Na *VersionWeb*, porém, isso é feito no momento em que o usuário está navegando pela página e ele pode ver as diferenças entre as versões e recuperar uma delas acionando o *link* que dá acesso à ferramenta. Esse *link* pode estar em qualquer lugar da página. Porém, essa página deve estar sob o gerenciamento da *VersionWeb*, ao contrário do que é feito no AIDE.

6.3. Sugestões para trabalhos futuros

A *VersionWeb* pode ser estendida de forma a possibilitar a geração de configurações de páginas *Web* ou de *sites* inteiros, pois um *site* é composto de páginas e estas podem ser compostas por outros tipos de informações (arquivos) como imagens, *link* para outras páginas, etc. Para isso, os documentos e arquivos que compõem as páginas devem também estar no repositório sob o controle de versão. Os conceitos de espaço do produto, espaço da versão e grafos AND/OR descritos no Capítulo 2 auxiliam nesse processo.

Outra funcionalidade que pode ser acrescentada à ferramenta é o gerenciamento de permissões de arquivos/diretórios do repositório por parte dos autores, pois, na *VersionWeb* todos os autores têm as mesmas permissões sobre todo o conteúdo do repositório. Além disso, pode-se acrescentar uma opção de notificação automática aos autores no momento da renomeação, de um arquivo ou de um diretório, feita por um deles, pois assim ninguém vai ficar pensando que o arquivo ou diretório foi removido, quando na verdade foi apenas renomeado.

Como o CVS não suporta a exibição de diferenças entre duas versões de arquivos binários, pode-se também estudar a possibilidade de construir uma ferramenta que exibe as diferenças entre versões para esses tipos de arquivos e incorporá-la à *VersionWeb*.

Um trabalho de validação do uso da ferramenta em outros ambientes de trabalho como salas de aula, empresas de *software* e redação de jornais pode também ser realizado de forma a avaliar o desempenho e usabilidade da *VersionWeb* em diferentes ambientes, bem como proporcionar outras necessidades de usuários mais específicas a seus domínios.

Referências Bibliográficas

[Berliner 1990] Berliner, B. CVS II: parallelizing software development. Proceedings of the Winter 1990 USENIX Conference. Washington, DC, January, 1990.

[Bieliková 1999] Bieliková, M. Space-efficient version storage. Disponível *on-line* em: wysiwyg://63/http://www.dcs.elf.stuba.sk/~bielik/scm/delta.htm. Visitado em agosto de 1999.

[Blum 1996] Blum, A. – **Building Business Web Sites**, MIS:Press, 1996.

[Bolinger e Bronson 1995] Bolinger, D.; Bronson, T. **Applyng RCS and SCCS - From Source Control to Project Control**. Disponível *on-line* em: http://www.oreilly.com/catalog/rcs/chapter/index.html. Visitado em agosto de 2000.

[Breedlove 1996] Breedlove, R. F. **Web Programming Unleashed**. Disponível *on-line* em: http://ebooks.cs.biu.ac.il/1575211173/ch12.html. Visitado em agosto de 2000.

[Cederqvist 1993] Cederqvist, P. Version Management with CVS. Disponível *on-line* em: ftp://java.icmc.sc.usp.br/library/books/cvs.pdf. Visitado em agosto de 1999.

[Colla 1999] Colla, E. C. Servlet, java do lado servidor. Disponível *on-line* em: http://www.insite.com.br/docs/develop-servlet95.html. Visitado em maio de 1999.

[Conradi e Westfechtel 1998] Conradi, R.; Westfechtel, B. Version Models for Software Configuration Management. **ACM Computing Surveys**, vol. 30, Nº. 2, 1998.

[Cornell e Horstmann 1997] Cornell, G.; Horstmann, C. S. – **Core Java**, Makron Books, 1997.

[CVS 1999] Concurrent Versions Systems. Disponível *on-line* em: http://www.cyclic.com, http://www.loria.fr/~molli/cvs-index.html. Visitado em agosto de 1999.

[Douglis et al. 1998] Douglis, F.; Ball, T.; Chen, Y.-F.; Koutsofios E. The AT&T Internet Difference Engine: Tracking and viewing changes on the web. **World Wide Web**, Volume 1, Nº. 1, 1998.

[Ehrig et al. 1989] Ehrig, H.; Fey, W.; Hansen, H.; Lowe, M.; Jacobs, D. Algebraic software development concepts for module and configuration families. **Lecture Notes in Computer Science**, 405, Springer, 181-192, 1989.

[Estublier e Casallas 1994] Estublier, J.; Casallas, R. *The Adele configuration manager*. **In Configuration Management**, W. F. Tichy, Ed., Vol. 2 of Trends in Software, Wiley, New York, 99-134, 1994.

[Fröhlich e Nejdl 1997] Fröhlich, P.; Nejdl, W. WebRC - Configuration Management for a Cooperation Tool. **Lecture Notes in Computer Science**, 1235, Springer, 175-185, 1997.

[Gundavaran 1996] Gundavaran, S. – **CGI Programming on the World Wide Web**, O'Reilly & Associates, Inc., 1996.

[Hicks et al. 1998] Hicks, D. L.; Leggett, J. J.; Nürnberg, P. J.; Schnase, J. L. A Hipermedia Version Control Framework. **ACM Transactions on Informations Systems**, vol. 16, Nº. 2, Pages 127-160, 1998.

[Hungs e Kunz 1992] Hungs, T.; Kunz, P. UNIX Code Management and Distribution. **Conference on Computing in High Energy Physics**, Annecy, France, September 21-25, 1992.

[Jamsa et al. 1997] Jamsa, K.; Lalani, S.; Weakley, S. – **Web Programming**, Jamsa Press, 1997.

[Kilpi 1997] Kilpi, T. Product Management Requeriments for SCM Discipline. **Lecture Notes in Computer Science,** 1235, Springer, 186-200, 1997.

[Leblang 1994] Leblang, D. The CM Challenge: Configuration management that works. **In Configuration Management**, W. F. Tichy, Ed., Vol. 2 of Trends in Software, Wiley, New York, 1-38, 1994.

[Lemay e Perkins 1996] Lemay, L.; Perkins, C. L. – **Teach Yourself Java in 21 days**, Sams Net Publishing, 1996.

[Lie et al. 1989] Lie, A., Conradi, R.; Didriksen, T.; Karlsson, E.; Hallsteinsen, S. O.; Holager, P. Change oriented versioning. **Lecture Notes in Computer Science**, 387, Springer, 101-117, 1989.

[McPherson 2000] McPherson, S. JavaServer Pages: A Developer's Pesperctive. Disponível *on-line* em: http://developer.java.sun.com/servlet/. Visitado em agosto de 2000.

[Munch 1995] Munch, B. Versioning. Disponível *on-line* em: http://www.idt.unit.no/~bjounmu/thesis/node45.html. Visitado em agosto de 1999.

[Nagl 1996] Nagl, M. Building Tightly-Integrated Software Development Environments: The IPSEN Approach. **Lecture Notes in Computer Science**, 1170, Springer, 1996.

[Pressman 1995] Pressman, R. S. **Software Engineering** – MacGraw Hill, 3ª edition, 1995.

[Pressman 1997] Pressman, R. S. **Software Engineering** – MacGraw Hill, 4ª edition, 1997.

[Rochkind 1975] Rochkind, M. J. The source code control system. **IEEE Transactions Software Engineering.** Vol.1, Nº 4, 364-370, 1975.

[Soares 1999] Soares, M. D. Gerenciamento de Controle de Versões de Páginas Web. XIII Simpósio Brasileiro de Engenharia de Software, IV Workshop de Teses em Engenharia de Software, Florianópolis, Santa Catarina, Brasil, 33-37, outubro de 1999.

[Soares et al. 2000] Soares, M. D.; Fortes, R. P. M.; Moreira, D. A. *VersionWeb*: A Tool for Helping Web Pages Version Control. In: International Conference on Internet Multimedia Systems and Applications (IMSA 2000), Las Vegas, EUA, pp. 275-280.

[Sun 1999] The Java Tutorial. Disponível *on-line* em: http://java.sun.com. Visitado em dezembro de 1999.

[Sommerville 1995] Sommerville, I. **Software Engineering,** 5ª edição, Addison-Wesley, 1995.

[Sommerville et al. 1998] Sommerville, I., Rodden, T., Rayson, P., Kirby, A., Dix, A. Supporting information evolution on the WWW. **World Wide Web**, Vol. 1, Nº 1, 45-54, 1998.

[Thomas et al. 1996] Thomas, M. D.; Patel, P. R.; Hudson, A. D.; Ball Jr., D. A. – **Java Programming for the Internet: A Guide to Creating Dynamic, Interactive Internet Applications**, Ventana Comunications Group, 1996.

[Tichy 1985] Tichy, W. F. RCS – A system for version control. **Software Practice and Experience**, Vol. 15, Nº 7, 637-654, 1985.

[Vitali e Durand 1999] Vitali, F.; Durand, D. G. Using versioning to support collaboration on the WWW. Disponível *on-line* em: http://cs-pub.bu.edu/students/grads/dgd/version.html. Visitado em janeiro de 1999.

[Zeller e Snelting 1995] Zeller, A.; Snelting, G. Handling version sets through feature logic. **Lecture Notes in Computer Science**, 989, Springer, 191-204, 1995.

Apêndice

Neste apêndice, é apresentado o formulário distribuído aos usuários para a realização dos testes de usabilidade da *VersionWeb*. O formulário elaborado está transcrito a seguir.

Tarefas para teste de uso da *VersionWeb*

A *VersionWeb* é uma ferramenta de Gerenciamento de Versões de Páginas Web que auxilia os autores no desenvolvimento paralelo de uma página ou de um *site*. Essa ferramenta utiliza o CVS (Concurrent Versions System) para fazer o controle das versões geradas dos arquivos. Em um sistema de controle de versões de arquivos nenhuma informação é perdida, ou seja, todas as alterações feitas são gravadas (gerando novas versões) e uma versão específica pode ser recuperada a qualquer momento.

- A realização das tarefas abaixo constitui uma avaliação inicial de uso da ferramenta *VersionWeb*. O tempo necessário para a realização desse teste não será superior a 20 minutos. Para iniciar as tarefas que estão descritas abaixo, utilize a ferramenta *VersionWeb* em **http://java2.intermidia.icmc.sc.usp.br/cgi-bin/mari_cgi/login.cgi**
- Para obter auxílio na realização da lista de tarefas, consulte o help da *VersionWeb* em **http://java2.intermidia.icmc.sc.usp.br/~mari/Help/help.htm**. Para maior flexibilidade, abra essa página de help em uma nova janela do *browser*.

Dados importantes:	Repository Path: **/home/mari/cvsroot**
	login: **teste**
	Password: **secret**
Obs.: as informações acima são todas com letras minúsculas	

Obs.: Trabalhe somente sobre o diretório especificado

Definição de termos mais utilizados:

- ***checkout* remoto** - obtém uma cópia do arquivo selecionado para edição remota, ou seja, no próprio servidor.
- ***checkout* local** - obtém uma cópia do arquivo selecionado para edição local (através do *download* do arquivo).
- ***commit*** - registra todas as alterações feitas sobre o arquivo gerando uma nova versão.

Tarefas:

1. Faça login como autor (para isso, selecione a opção “Authors” do botão de rádio na tela de *login* e utilize os dados fornecidos acima para os campos Repository Path, Login e Password). Se sua autenticação estiver ok, você terá acesso à interface principal de gerenciamento de arquivos do repositório CVS especificado.
2. Liste o conteúdo do diretório **Teste** (opção List Directory) e verifique se dentro dele existe um arquivo chamado **teste.html**. Verifique quantas versões existem deste arquivo (opção Versions List). A seguir, feche essa janela.
3. Faça um *checkout* remoto do arquivo **teste.html** (opção Remote Checkout). O conteúdo do arquivo será exibido em uma nova janela.
4. Depois do <body> insira a seguinte linha: "**Inserindo uma linha: seu_nome
**"
5. Edite uma mensagem de *log* (algum comentário) sobre a alteração que você fez no arquivo.
6. Faça *commit* das alterações feitas (opção Commit). A seguir, feche a janela de área de alteração do arquivo.
7. Verifique quantas versões existem agora do arquivo **teste.html**.
8. Faça um *checkout* local do arquivo da versão 1.2.
9. Abra-o em sua máquina e insira a seguinte linha antes do </body>: "**Alterando arquivo com checkout local:_seu nome
**".
10. Faça o *commit* dessa alteração gerando uma *branch* com o número 1.2.1 e se quiser digite uma mensagem de log. Para essa tarefa, você deve utilizar a opção "Commit of the local checkout".
11. Faça um *reload* dessa janela (janela que exibe a lista de versões) através do botão “Reload this page” no final da janela e verifique se a *branch* 1.2.1 foi gerada.
12. Faça um *checkout* remoto de cada versão que você gerou e verifique se suas alterações realmente foram gravadas. Não precisa fazer *commit*, é apenas para verificar.
13. Remova as alterações feitas entre a primeira e a última versão do arquivo (opção Remove Changes). Essa tarefa deve gerar uma nova versão com a remoção das alterações das duas versões. A seguir, feche essa janela.
14. Veja as diferenças entre as versões do arquivo, e a seguir, feche todas as janelas.

Faça uma rápida avaliação do uso da ferramenta, assinalando somente uma das alternativas:

a) Você já conhecia o CVS (Sistema de Controle de Versões Concorrentes), utilizado em software?	sim ☐ não ☐
b) Você acha que a ferramenta facilita de alguma forma o entendimento da funcionalidade de CVS?	sim ☐ não ☐
c) Qual o nível de conhecimento dos termos relacionados com controle de versões que você possuía?	alto ☐ médio ☐ pouco ☐ nenhum ☐
d) Como foram suas dificuldades para a realização das tarefas ?	☐ **grandes**, difíceis de explicar ☐ **médias**, conseguiria repetí-las ☐ **pequenas**, com pouco auxílio você conseguiria superá-las
e) Quais tarefas foram muito fáceis?	(dê o(s) nro(s) entre 1 e 14):
f) Quais tarefas foram muito difíceis ou mesmo impossíveis de se executar?	(dê o(s) nro(s) entre 1 e 14):
g) Você gostaria de utilizar novamente a *VersionWeb?*	sim ☐ não ☐

Dê sua opinião, se quiser:

a. O que poderia ser melhorado na ferramenta?

_____

_____

b. Quais as vantagens/desvantagens em usar controle de versão na Web?

_____

_____

c. Em quais ambientes de trabalho (além dos ambientes de autoria de páginas Web) essa ferramenta seria útil?

_____

_____

- Você é aluno de: ( ) Graduação ( ) Mestrado ( ) Doutorado

- Se você é aluno de graduação, você faz iniciação científica? ( ) Sim ( ) Não
- Você tem conhecimento de Engenharia de Software ou faz pesquisa em Engenharia de Software? ( ) Sim ( ) Não
- Você utiliza freqüentemente a *Web*? ( ) Sim ( ) Não
- Você faz autoria de páginas *Web*? ( ) Sim ( ) Não
- Se possível, deixe seu endereço (e-mail) para contato: _____

Meu e-mail: mdsoares@icmc.sc.usp.br

Muito obrigada!