ESCOLA POLITÉCNICA
DE PERNAMBUCO

Departamento de
Sistemas
Computacionais

Um Guia para Controle de Versão de Projeto de Software

Trabalho de Conclusão de Curso

Engenharia da Computação

Adriano Nântua do Nascimento Carneiro
Orientador: Prof. Sérgio Castelo Branco Soares

Recife, novembro de 2007

UNIVERSIDADE
DE PERNAMBUCO

ESCOLA POLITÉCNICA
DE PERNAMBUCO

Departamento de
Sistemas
Computacionais

Um Guia para Controle de Versão de Projeto de Software

Trabalho de Conclusão de Curso

Engenharia da Computação

Este Projeto é apresentado como requisito parcial para obtenção do diploma de Bacharel em Engenharia da Computação pela Escola Politécnica de Pernambuco – Universidade de Pernambuco.

Adriano Nântua do Nascimento Carneiro
Orientador: Prof. Sérgio Castelo Branco Soares

Recife, novembro de 2007

UNIVERSIDADE
DE PERNAMBUCO

Adriano Nântua do Nascimento Carneiro

Um Guia para Controle de Versão de Projeto de Software

Resumo

A disciplina de processo de Gerência de Configuração e Mudanças tem como uma de suas principais atividades a Gerência de Configuração, que diz respeito à estrutura do produto de software e ao monitoramento das modificações de artefatos ao longo do tempo, também conhecido como controle de versão. A Gerência de Configuração afirma que o controle de versão deve ser realizado em projetos de software. No entanto, não dá informações de como o controle de versão deve ser feito. Este trabalho, um guia para controle de versão de projeto de software, propõe uma definição de tal processo, determinando ferramentas, regras, políticas, padrões, fluxos de trabalho e melhores práticas, tendo com meta suprir a carência de orientações pragmáticas quanto ao processo de controle de versão.

Abstract

The Configuration and Change Control process discipline has, as one of its main activities, the Configuration Management. This activity deals with software product structure and change monitoring over artifacts through time, also know as version control. Configuration Management states that version control must be performed in software development projects. Nevertheless, there are not instructions about how version control should be executed. This work, a guide to software project version control, proposes a definition of such process, determining tools, rules, policies, patterns, workflows and best practices, towards removing the lack of pragmatic information on the version control process.

Sumário

Índice de Figuras

Tabela de Símbolos e Siglas

GCS	Gerência de Configuração de Software
SCM	*Software Configuration Management*, Gerência de Configuração de Software, em inglês
SVN	Subversion, um sistema de controle de versão
CVS	Concurrent Versions System, um sistema de controle de versão
MS	Microsoft
WEB	Redução de *World Wide Web* (Rede Mundial de Computadores), conhecida também como Internet
CCM	*Configuration and Change Management* (Gerência de Configuração e Mudanças)
CM	*Configuration Change Management* (Gerência de Configuração)

Agradecimentos

Gostaria de agradecer, primeiramente, a Deus pelo privilégio, acima de tudo, de estar vivo, premissa básica que as pessoas tendem a esquecer de levar em consideração.

Agradeço também a meus pais, Osmar e Teresa, por me proporcionarem uma educação de qualidade, tanto acadêmica (nas escolas em que me colocaram) quanto moral e eticamente (em casa). Sem essa base, não poderia ter chegado até aqui.

Tenho muito a agradecer a todos os Professores do DSC com que tive contato, todos grandes profissionais e, principalmente, grandes pessoas. Agradecimentos especiais a Sérgio Soares, cujas aulas não tive oportunidade de freqüentar, mas cuja orientação neste TCC tive o privilégio de receber; a Ricardo Massa, um amigo que teve paciência (eu espero) com minha indisponibilidade em semestres anteriores; e, finalmente, a Carlos Alexandre, um professor, mestre e amigo, com quem aprendi, mesmo que ele não saiba, bem mais que algoritmos e árvores binárias.

Agradeço muito aos colegas que tive ao longo do curso, que me ajudaram com seu incentivo e injeções de ânimo, em especial aos amigos Túlio Campos, Bruno Arôxa, Pedro França e César Augusto.

Por fim agradeço à minha esposa Márcia, minha companheira de vida, sem cujo apoio, incentivo e compreensão eu não teria sido capaz de concluir este trabalho, e a meu filho do coração Arthur, cujos abraços e risadas têm poderes de fortalecimento e cura para a alma. Dedico este trabalho a eles, minha família, com quem aprendo diariamente o valor do suor e do sorriso, da responsabilidade e da leveza, da felicidade, da sintonia e da alegria insubstituível e incomparável de viver nossas vidas sob tutela do amor.

Muito obrigado a todos vocês.

Capítulo 1

Introdução

Um projeto de software é composto por um conjunto de diversos tipos de arquivos, que quando devidamente processados e compilados resultam em um ou mais arquivos binários que são o produto final do projeto, o software. Como o software pode ser reconstruído a partir de seus arquivos-fonte, faz-se imprescindível o controle sobre esses artefatos.

O controle de versão é a atividade básica e primordial dentre as atividades de apoio ao desenvolvimento de software geralmente definidas na Gerência de Configuração de Software (GCS ou *Software Configuration Management* – SCM) [1]. Muitos dos problemas que ocorrem durante o processo de desenvolvimento são causados pelo controle ineficaz e/ou inexistente dos artefatos de um projeto de software.

A atividade de controle de versão dentro de uma organização é definida por políticas que determinam: os processos e procedimentos a serem seguidos, bem como a forma de executá-los, as boas práticas e situações a evitar; e a ferramenta a ser utilizada. Da falta de políticas definidas e, por conseguinte, de uma ferramenta que auxilie o controle de versão decorrem os seguintes problemas:

- Perda de versões anteriores do projeto;
- Sobrescrita e/ou exclusão acidental de item de configuração;
- Perda de trabalho decorrente de alterações feitas sobre uma versão antiga e/ou incorreta;
- Dificuldade na manutenção simultânea de diferentes versões do projeto;
- Perda de trabalho decorrente da concorrência dos membros da equipe por itens de configuração;
- Dificuldade (ou impossibilidade) na auditoria de alterações dos itens de configuração.

Dado o contexto, torna-se imprescindível que projetos de software possuam um processo de controle de versão bem definido. Toda a literatura sobre SCM disponível confirma que o controle de versão deve ser realizado. Contudo, esta mesma literatura não define como o controle de versão deve ser feito, não existindo guias previstos em SCM para este processo.

1.1 Objetivos

Existem fontes de informações, espalhadas e fragmentadas, sobre como realizar controle de versão. No entanto, não existe um guia unificado que centralize em um único documento um modelo de trabalho completo para este processo. Dada a falta de guias que definem um processo de controle de versão, este trabalho tem como objetivo fornecer subsídios concretos para a definição de um Plano de Gerência de Configuração (*Configuration Management Plan*) [1], um documento previsto nos processos de SCM. O propósito deste guia é definir, no que diz respeito ao controle de versão de artefatos de projeto, um modelo de trabalho consistente, seguro, eficaz e experimentado para projetos de desenvolvimento de software em equipe.

Este guia foi concebido tendo como base minhas experiências profissionais como gerente de projetos e de configuração, onde tive a oportunidade de planejar, executar e acompanhar o controle de versão em diversos projetos de desenvolvimento de software com equipes.

1.2 Estrutura do Trabalho

Este trabalho, um guia para controle de versão de projeto de software, foi estruturado e dividido em 5 capítulos.

No Capítulo 2 é apresentada a fundamentação teórica sobre controle de versão, introduzindo os conceitos fundamentais, noções básicas, definições e contextualizações sobre o tema. A apresentação da ferramenta utilizada no guia é feita no Capítulo 3, onde o sistema de controle de versão (Subversion), suas características, vantagens e as razões de sua escolha é apresentado. O Capítulo 4 descreve a metodologia de controle de versão, o objetivo centro do trabalho, onde os fluxos de trabalho, políticas e melhores práticas no controle de versão de projeto de software são definidos. Por fim, o Capítulo 5 apresenta as conclusões, considerações finais e trabalhos futuros.

1.3 Contribuições

Este trabalho tem como suas principais contribuições:

- A apresentação de uma sólida fundamentação teórica sobre controle de versão, abrangendo os diversos pontos indispensáveis ao entendimento desse processo;
- A definição de um processo completo de controle de versão de projeto de software:
  - As ferramentas a serem utilizadas, com os fundamentos sobre seu uso;
  - O ambiente mínimo dos projetos e sua configuração;
  - A definição de regras, políticas e padrões;
  - Os fluxos de trabalho básicos, avançados e alternativos;
  - As melhores práticas, em cada uma das etapas do processo;
- Possibilitar que um profissional da área de computação com um mínimo conhecimento em configuração de ambientes de trabalho e em projetos de desenvolvimento de software seja capaz de implantar com sucesso em uma organização uma política simples, mas eficiente, de controle de versões;
- Incentivar que empresas que não possuem controle de versão de software possam ver no guia uma oportunidade de utilizar uma metodologia pronta para esse processo.

Capítulo 2

Controle de Versão

Este capítulo tem como objetivo apresentar conceitos fundamentais sobre o controle de versão em projetos de software, expondo, ao longo das seções a seguir, noções básicas, definições e contextualizações compondo uma base teórica.

2.1 Arquitetura

Uma equipe trabalhando em um projeto de software consiste, em linhas gerais, em indivíduos que compartilham recursos, os artefatos do projeto. Estes artefatos são arquivos de diversos tipos e naturezas, tais quais códigos-fonte, *scripts*, configurações, documentação, automações de tarefas, entre outros, os quais no âmbito de SCM são também usualmente denominados **itens de configuração** (*configuration items*) [1]. O software é construído através das sucessivas modificações feitas sobre este conjunto de arquivos, pela criação, alteração e exclusão desses recursos. Desta forma, é fundamental garantir que o time tenha acesso a esse conjunto de recursos em seu estado mais atual e íntegro, com todas as modificações realizadas pelos demais membros. A maneira mais simples de atingir este cenário é criando uma área dentro da organização onde são armazenados os artefatos do projeto mais atuais, ficando disponíveis para todos os componentes da equipe. A esta área centralizada de arquivos dá-se o nome de **repositório** [2, 3, 4, 5].

Fazer com que os membros da equipe trabalhem diretamente nos itens de configuração localizados no repositório constitui uma péssima prática, apresentando os seguintes problemas:

- Em conseqüência de modificações parciais, o conteúdo do repositório estará constantemente “em alteração” e dificilmente conterá uma versão do software estável ou até mesmo compilável;
- Devido à simultaneidade de acesso, o trabalho de um componente do time tem grande probabilidade de interferir no trabalho dos demais.

Desta forma um outro modelo de trabalho deve ser adotado. Em linhas gerais, o trabalho conceitual básico da equipe com um repositório, ilustrado pela Figura 1, é extremamente simples, funcionando em um esquema de cliente/servidor. Para efetuar mudanças, corrigir problemas ou adicionar funcionalidades no software, um membro da equipe efetua uma cópia (operação representada na figura pela seta rotulada de *Leitura*) dos artefatos do projeto (representados pelo ícone de conjunto arquivos) do repositório para uma área privada em sua estação cliente. O conjunto de arquivos copiados para a estação cliente é comumente chamado de **área de trabalho**

(*workspace*) [1, 2, 5] ou **cópia de trabalho** (*working copy*) [2, 4] e sobre esta cópia é que serão feitas as modificações. Ao concluir as modificações, o componente da equipe envia para o repositório o conteúdo de sua área de trabalho (operação representada pela seta rotulada de *Escrita*), atualizando assim o repositório, que passará a dispor do conjunto de arquivos em seu estado mais atual. Assim, quando um segundo membro da equipe atualizar sua cópia de trabalho, ele terá as modificações realizadas pelo primeiro.

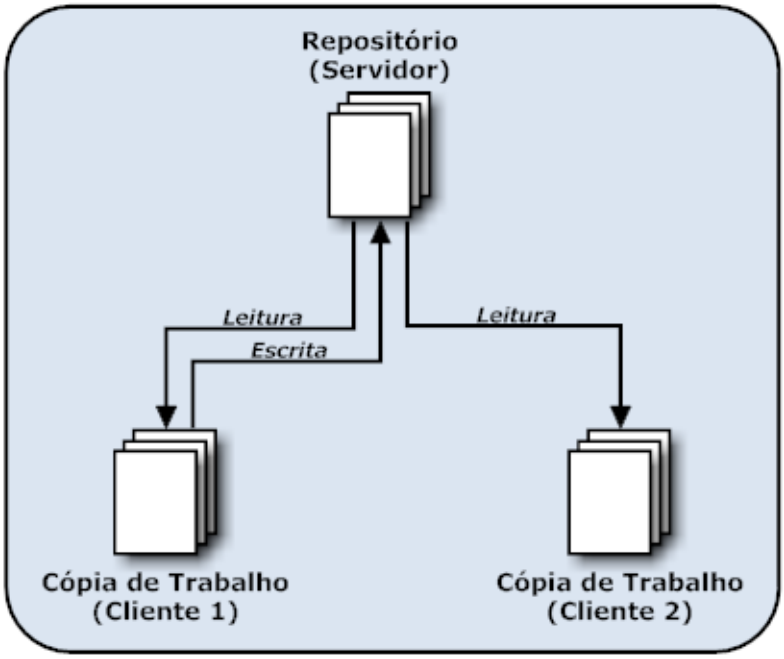

Figura 1. Funcionamento de um repositório simples (servidor de arquivos)

Conceitualmente, as noções de repositório e de área/cópia de trabalho independem de uma ferramenta ou tecnologia específica, podendo este esquema ser implementado desde através de um diretório compartilhado por um servidor de arquivos até um complexo sistema de controle de versão, como será apresentado nas próximas seções.

2.2 Modelos de Versionamento

O modelo simplificado apresentado na seção anterior apresenta brechas que podem levar a falhas que comprometem todo o trabalho da equipe. Nesta seção, é apresentado um problema clássico do tema e suas soluções, cada uma com as devidas considerações.

2.2.1 O Problema do Compartilhamento de Arquivos

Um dos problemas fundamentais que os sistemas de controle de versão devem resolver é permitir que os componentes de uma equipe compartilhem os dados do repositório de forma simultânea sem que haja interferência eles, o que pode acarretar na perda de modificações efetuadas.

O modelo de servidor de arquivos simples introduz o problema do compartilhamento de arquivos, ilustrado na Figura 2. No cenário apresentado, dois desenvolvedores (Adriano e Márcia) alteram simultaneamente o Artefato A (representado pelo ícone de documento). Ao enviar suas alterações para o repositório primeiro, Adriano corre o risco de ter sua versão sobrescrita pela versão de Márcia, devido ao formato e às condições técnicas do modelo.

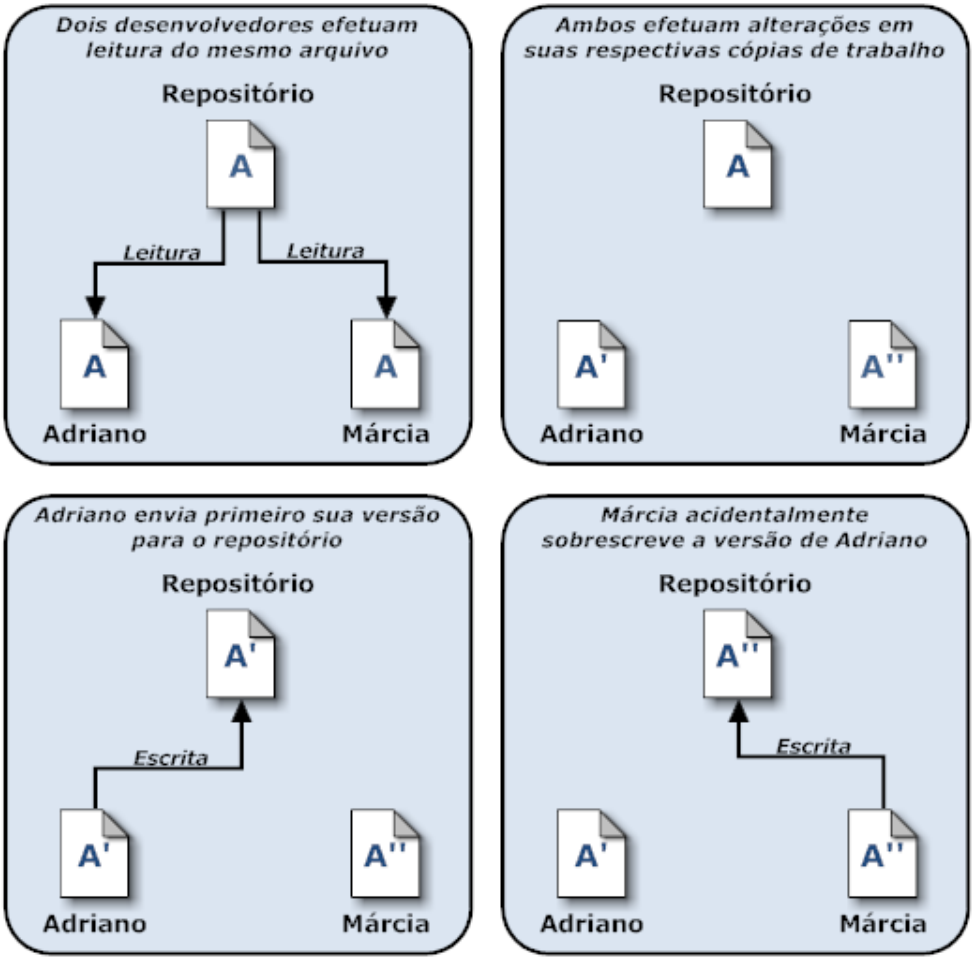

Figura 2. O problema do compartilhamento de arquivos

Obviamente, o cenário apresentado deve ser evitado, já que o trabalho de um desenvolvedor foi sumariamente desprezado e a versão do repositório não apresenta o que deveria ser a versão mais atual do software.

2.2.2 O Modelo *Lock-Modify-Unlock*

Alguns sistemas de controle de versão utilizam o modelo ***Lock-Modify-Unlock*** (Travar-Modificar-Destravar) [3,4] para solucionar o problema apresentado na seção anterior. Neste modelo, o repositório permite que somente uma pessoa por vez efetue modificações sobre um artefato, sendo esta política de exclusividade gerenciada através do uso de travamentos (*locks*).

No modelo *Lock-Modify-Unlock*, contextualizado na Figura 3, o desenvolvedor deve travar (*lock*) o arquivo antes de efetuar sobre ele modificações. Assim, um segundo desenvolvedor não conseguiria travar o mesmo artefato e, portanto, não poderia realizar modificações sobre este, podendo apenas efetuar uma leitura simples (sem possibilidade de alteração) e esperar que o primeiro desenvolvedor termine suas modificações sobre o artefato e remova a trava.

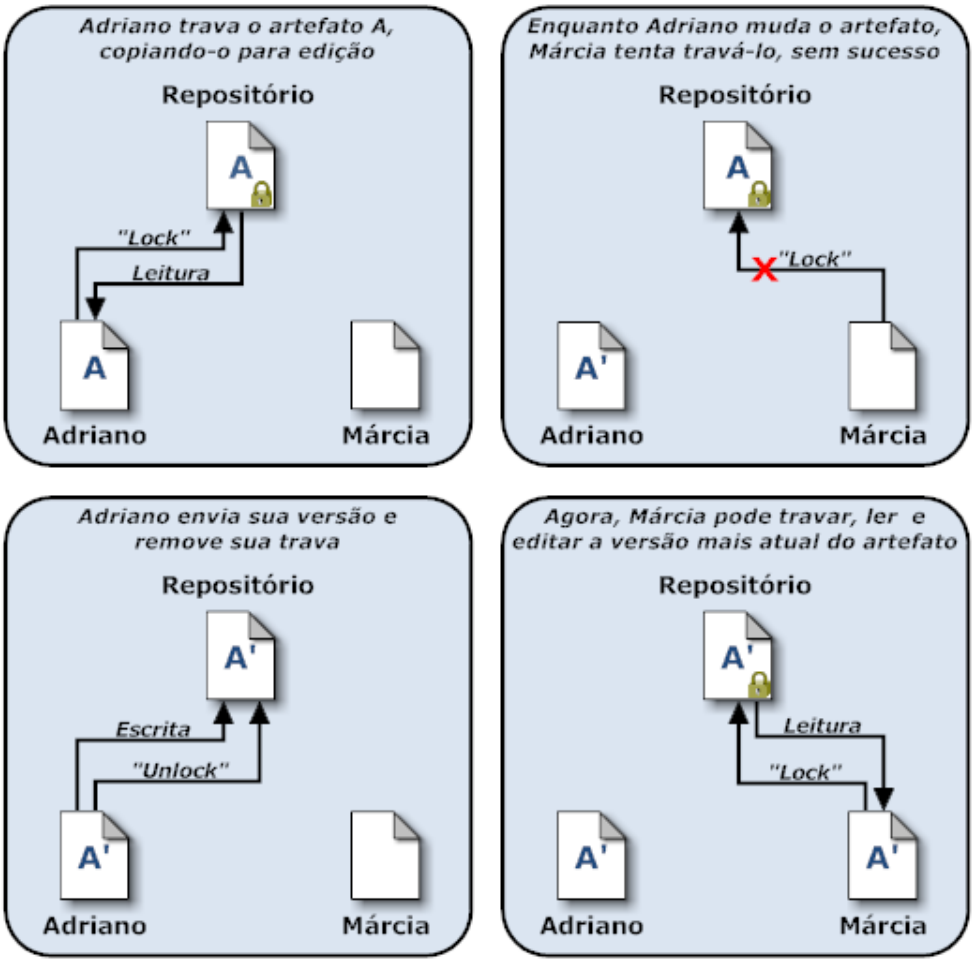

Figura 3. O modelo *Lock-Modify-Unlock*

Além de ser restritivo, o modelo *Lock-Modify-Unlock* apresenta problemas que podem tornar seu uso desaconselhado. Primeiramente, o travamento de artefatos pode causar problemas administrativos. Contextualizando um cenário onde o desenvolvedor Adriano trava um arquivo e simplesmente esquece de destravá-lo (por exemplo, focando em modificações de outros artefatos do projeto), Márcia nada pode fazer além de esperar. Piorando o cenário, Adriano pode ficar ausente (férias, por exemplo), fazendo com que seja necessária intervenção administrativa para remover a trava. A situação acaba causando desnecessários trabalho adicional e perda de tempo.

Além disso, o travamento de artefatos causa serialização de trabalho desnecessária. Em novo cenário, Adriano está modificando a porção inicial de um arquivo-fonte (um método específico de uma classe, por exemplo) e Márcia precisa modificar a porção final do mesmo arquivo (por exemplo, outro método da mesma classe, totalmente independente do primeiro). Não há razão para que o trabalho não possa ser feito simultaneamente, dado que as modificações não causam interferência mútua e presumindo que haja mecanismos eficazes de mesclar as alterações. Deste modo, a espera pelo destravamento é desnecessária e constitui perda de tempo.

Por fim, o travamento de artefatos causa uma falsa sensação de segurança. Em mais um cenário hipotético, Adriano trava e modifica o artefato A enquanto Márcia faz o mesmo com o artefato B. Se os artefatos A e B dependem mutuamente um do outro e as mudanças realizadas são semanticamente incompatíveis (uma mudança brusca de conceito, por exemplo), A e B passam a não mais funcionar juntos e o modelo nada pôde fazer para evitar o problema. Um outro cenário mostra que o modelo possibilitaria uma situação de *Deadlock* no desenvolvimento: Adriano trava e modifica A enquanto Márcia faz o mesmo com B. Se, durante o

desenvolvimento, Adriano percebe que sua modificação em A implica em um ajuste em B e Márcia, por sua vez, verifica que sua modificação em B exige uma alteração em A, surge o clássico problema do *Deadlock*, factível em contextos onde recursos são compartilhados de forma exclusiva. Concluindo, os desenvolvedores imaginam que, ao travar os arquivos, estão iniciando uma tarefa segura e isolada, o que pode não vir a se configurar.

2.2.3 O Modelo *Copy-Modify-Merge*

Os sistemas de controle de versão mais modernos e amplamente utilizados fornecem ferramentas que possibilitam a utilização do modelo ***Copy-Modify-Merge*** (Copiar-Modificar-Mesclar) [3,5] como uma alternativa ao travamento de artefatos no repositório. Este modelo permite que a equipe efetue modificações em suas cópias de trabalho privadas de forma simultânea e independente. Ao final do processo, as cópias privadas são mescladas (quando necessário) em uma nova e única versão final. Geralmente, o sistema de controle de versão em uso assiste na tarefa de mesclar dos artefatos (ou o faz por conta própria, dependendo da ferramenta), mas em última instância é responsabilidade do desenvolvedor fazer com que a mesclagem seja feita corretamente. O funcionamento do modelo *Copy-Modify-Merge* é exemplificado através do cenário ilustrado nas Figuras 4 e 5.

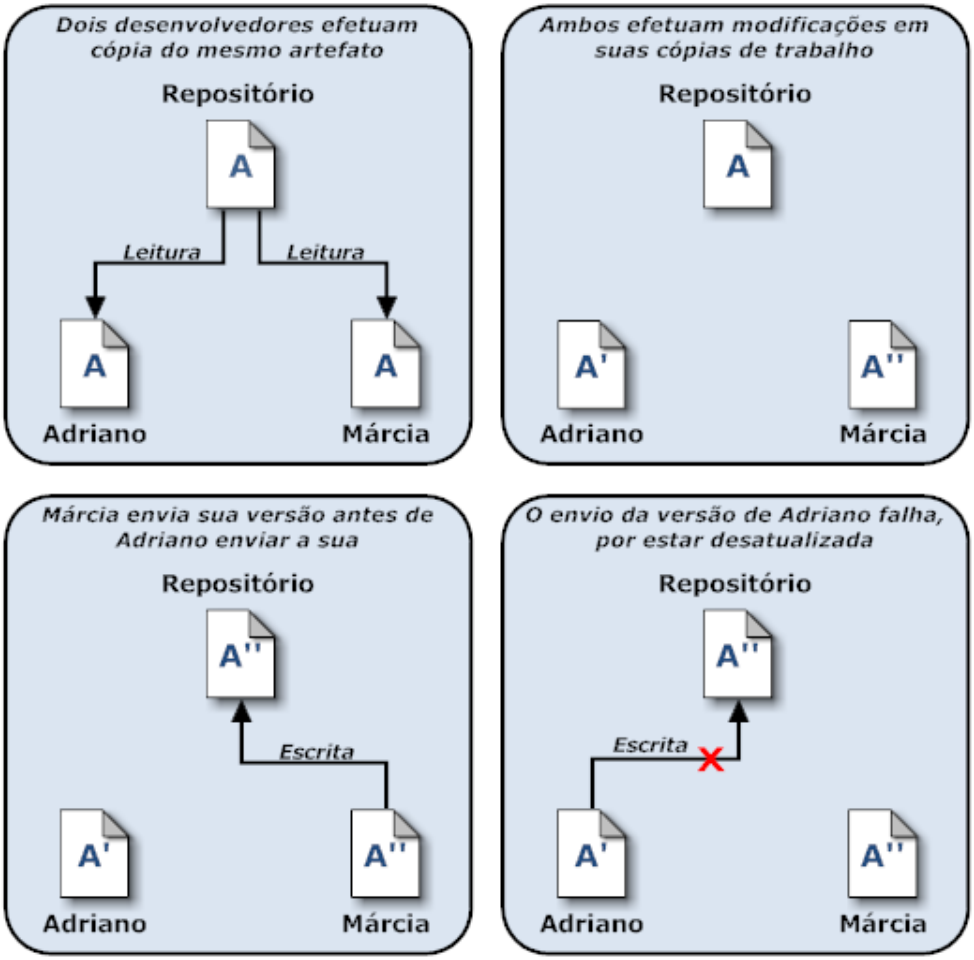

Figura 4. O modelo *Copy-Modify-Merge* (parte 1)

Nesse modelo, as ferramentas utilizadas dão condições técnicas tanto ao repositório quanto à cópia de trabalho de consistir as versões que transitam entre estas duas áreas. Desta

forma, o repositório tem a capacidade de aceitar ou não uma tentativa de escrita, comparando a versão corrente do artefato no repositório com a versão do artefato na cópia de trabalho antes da mudança.

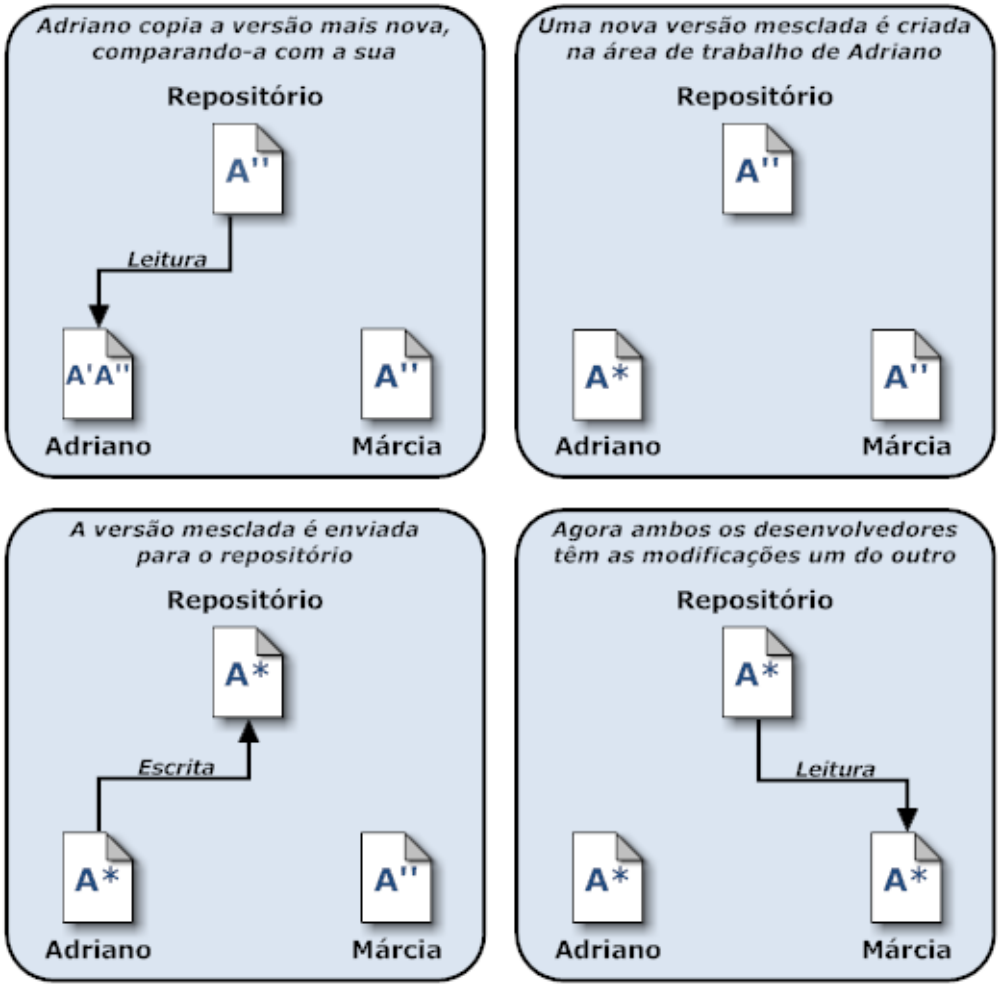

Figura 5. O modelo *Copy-Modify-Merge* (parte 2)

Ao ter rejeitada uma tentativa de escrita após verificação de versão, resta ao desenvolvedor atualizar sua cópia de trabalho, no intuito de obter as alterações feitas por outros desenvolvedores. Neste momento, é realizada a mesclagem das duas versões do artefato (a do repositório e a da cópia de trabalho do desenvolvedor), criando-se uma nova versão contendo todas as alterações feitas pelos desenvolvedores envolvidos no processo. Esta nova versão, por sua vez, terá a tentativa de escrita aceita pelo repositório.

Apesar de parecer um tanto caótico, o modelo *Copy-Modify-Merge* funciona bem na prática, eliminando os principais problemas apresentados pelo modelo *Lock-Modify-Unlock*. Quando dois ou mais desenvolvedores trabalham em paralelo em um mesmo artefato, na maioria das vezes suas modificações não se sobrepõem. Ainda assim, o tempo que se levaria para resolver este tipo de situação é muito menor que o tempo perdido pelo uso de travamento de artefatos.

2.3 Outros Conceitos Básicos do Controle de Versão

Durante as seções anteriores deste capítulo, alguns conceitos básicos do controle de versão já foram apresentados e fundamentalmente embasados, tendo em vista que se fizeram necessários

ao entendimento das noções descritas. Em resumo, foram eles: Artefatos de projeto e Itens de Configuração; Repositório; Área de Trabalho (*workspace*) ou Cópia de Trabalho (*working copy*); Travamento e Destravamento de artefatos. Esta seção tem como objetivo apresentar outros conceitos fundamentais no tema que não tiveram a oportunidade de serem citados até então.

2.3.1 *Checkout* e *Update*

Checkout [2, 3, 4, 5] consiste no ato de realizar a cópia dos artefatos do projeto do repositório para a cópia de trabalho.

Para alguns sistemas de controle de versão, como o CVS e o Subversion, o *check out* significa somente realizar a primeira leitura de artefatos do repositório, realizando-se a criação da área de trabalho do desenvolvedor; as subseqüentes leituras do repositório para obter novas modificações realizadas por outros desenvolvedores na cópia de trabalho já criada são chamadas de ***updates*** (atualizações) [2, 3, 4, 5]. Em outros sistemas de controle de versão, como o *Visual Safe Source* (VSS), não há distinção entre a primeira leitura e as demais verificações de modificações, sendo todas chamadas de *check out*.

Nas figuras apresentadas anteriormente, as operações de *check out* e *update* são indicadas pela seta rotulada de “Leitura”.

2.3.2 *Checkin*/*Commit*

Checkin [2, 3, 4, 5] consiste no ato de enviar para o repositório as alterações realizadas na cópia de trabalho do desenvolvedor. Em alguns sistemas de versão, como o CVS e o Subversion, esta operação é denominada ***commit*** [2, 3, 4, 5]. Em geral, os sistemas de controle de versão possibilitam a digitação de uma mensagem de texto associada ao *commit*, a qual o desenvolvedor deve preencher com informações sobre o que foi feito na alteração sendo enviada para o repositório. Esta mensagem é de extrema importância na rastreabilidade das modificações em um repositório.

Nas figuras apresentadas anteriormente, as operações de *checkin/commit* são indicadas pela seta rotulada de “Escrita”.

2.3.3 *Merging* e Conflito

Merging (Mesclagem ou Fusão) [2, 3, 4, 5] consiste no ato de mesclar ou fundir duas versões diferentes de um mesmo artefato, criando uma terceira e nova versão deste artefato que contém todas as modificações presentes nas duas versões anteriores. Basicamente, existem dois diferentes contextos em que a pode ocorrer *merging*: quando se realiza uma atualização de um artefato que foi modificado tanto localmente quanto no repositório (contexto que será descrito nesta seção e que também pode ser observado na Figura 5) ou quando se trabalha com *branches* (ver seção *Branches*).

Sistemas de controle de versão como CVS e Subversion, através de um mecanismo de comparação e indexação de linhas, tentam realizar mesclagem automática no momento em que é realizado um *update* em um artefato que foi alterado na cópia de trabalho e cuja versão no repositório é posterior à versão local.

Vale ressaltar que a mesclagem automática não é possível para todo e qualquer tipo de artefato: basicamente apenas arquivos que contém texto plano (como, por exemplo, arquivos de código-fonte e XML, entre tantos outros) oferecem possibilidade de mesclagem automática.

Quando as modificações nas duas versões a serem mescladas estão localizadas em diferentes trechos dentro do artefato, a mesclagem automática é bem sucedida. O caso contrário,

onde as modificações se sobrepõem, é denominado de **conflito** (*conflict*) [2, 3, 4, 5]. Conflitos devem ser resolvidos por um ser humano, contudo, mesmo sendo uma tarefa manual na essência, os sistemas de controle de versão fornecem ferramentas para auxiliar os desenvolvedores na conciliação de conflitos, apontando os pontos conflitantes no artefato e as opções de mesclagem.

2.3.4 *Revision* e *Revision Number*

Sob a tutela de um sistema de controle de versão, sempre que um artefato é modificado no repositório, é criada uma nova ***revision*** (revisão) [2, 3, 4, 5], a qual recebe um ***revision number*** (número de revisão) [2, 3, 4]. A maioria dos sistemas de controle de versão atribui os números de revisão por artefato, ou seja, cada artefato possui seu próprio número de revisão. O Subversion atribui o número de revisão à árvore inteira do projeto, conforme será mostrado mais adiante.

2.3.5 *Tags* ou *Labels*

Uma ***tag*** (Marcação) [2, 3, 4] ou ***label*** (rótulo) [5] é, de forma pragmática, uma "fotografia" do repositório em um determinado momento. Através da criação de *tags*, o sistema de controle de versão permite que seja possível dar um nome significativo (release-1.0, por exemplo) a uma versão do conjunto de artefatos do repositório correspondente a um determinado ponto no tempo, para posterior acesso.

A Figura 6 ilustra a criação de uma *tag*. A linha horizontal representa a evolução da linha principal de desenvolvimento ao longo do tempo. Cada versão dos artefatos é indicada por um círculo, contendo o número da versão ou o número da revisão dos artefatos. A *tag* é representada pelo ícone de uma etiqueta. No exemplo apresentado pela figura, foi criada uma *tag* da revisão de número 3 dos artefatos do projeto, indicando um nome significativo para esta revisão.

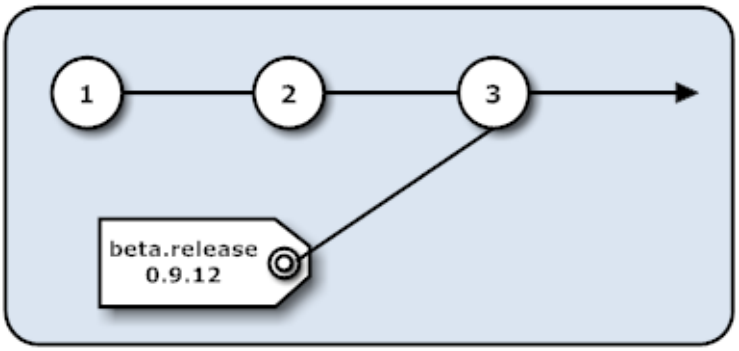

Figura 6. Definição de uma *tag*

Tags são úteis para manter versões de momentos-chave do projeto de software ao longo do tempo.

2.3.6 *Branches*

Normalmente, uma equipe trabalha na linha principal de desenvolvimento do projeto de software, onde são encontrados os artefatos em sua versão mais atual. No entanto, há situações em que é necessária a utilização de outras linhas de desenvolvimento, paralelas à linha principal. Um ***branch*** (ramificação) [2, 3, 4, 5] é uma linha de desenvolvimento independente, que funciona como um repositório paralelo do mesmo projeto, mas que mantém um vínculo histórico com a linha principal.

Tecnicamente, é possível fazer *branch* de um único artefato, de um conjunto de artefatos ou de toda a árvore do projeto, podendo esse *branch* ser criado a partir da linha principal de desenvolvimento, de uma *tag* ou mesmo de outro *branch*.

A Figura 7 ilustra uma situação em que foi utilizado o recurso de *branching*. No contexto apresentado, a linha de desenvolvimento principal de um produto evoluiu até sua versão 2.0, que foi disponibilizada para os usuários. Ao continuar a evolução do produto, através da criação de novas funcionalidades que acabarão por ser lançadas na futura versão 3.0, os usuários do produto indicam problemas e ajustes a serem realizadas na versão 2.0 lançada. Não é possível, neste momento, corrigir os problemas indicados na linha principal, pois nesta linha o produto não está numa versão estável para ser liberada para os usuários. Assim, a solução foi criar uma linha paralela de desenvolvimento, um *branch* criado a partir da versão 2.0 (indicado pela seta rotulada "*branch*"), cujos artefatos do projeto correspondem à versão do produto em posse dos usuários. Nesta nova ramificação, foram corrigidos os problemas relatados e realizados os devidos ajustes, sendo gerada a versão 2.1, que pode ser liberada para os usuários.

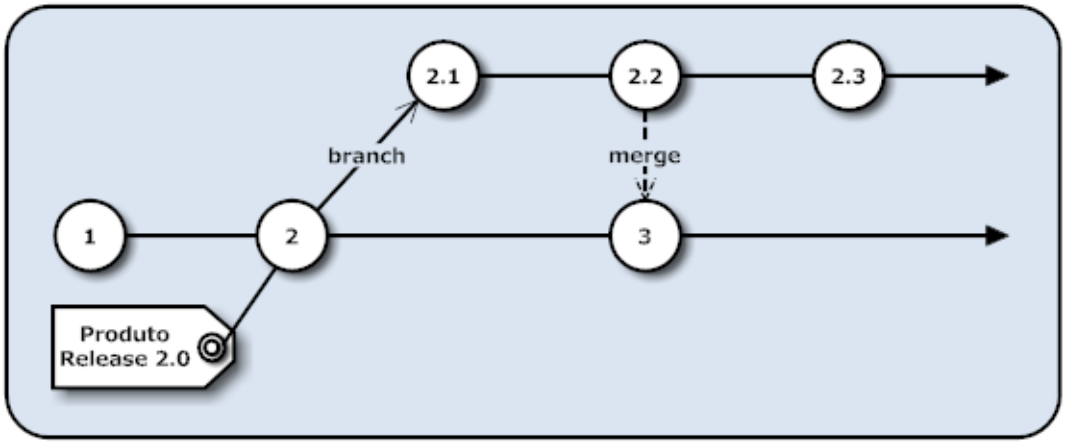

Figura 7. Uma situação de *branching* com posterior *merging*

Este mesmo *branch* tem a possibilidade de continuar a evoluir em paralelo à linha principal de desenvolvimento, através outras correções e ajustes (no exemplo, é gerada a versão 2.2 do produto). Por fim, será necessário que as correções e os ajustes realizados na linha paralela sejam incorporados na linha principal de desenvolvimento. Esta ação é realizada através de um *merge* (indicada por uma seta pontilhada, rotulada "*merge*") da versão 2.2 (a versão origem do *merge*, localizada no *branch* paralelo) para a versão 3 (a versão destino do *merge*, localizada na linha principal de desenvolvimento). Esta operação, conforme já foi descrito na seção sobre *merging*, é auxiliada pelo sistema de controle de versão, que pode desde realizar mesclagem automática a fornecer ferramentas que assistam o desenvolvedor na mesclagem manual.

Branching é um importante mecanismo no controle de versões, fornecendo ao projeto diversas vantagens no decorrer do desenvolvimento. Contudo, a ramificação de linhas de desenvolvimento é um instrumento que deve ser utilizado com inteligência, cautela e muita comunicação interna na equipe. A utilização de *branches* será explorada mais detalhadamente mais à frente neste guia.

Capítulo 3

Subversion, um Sistema de Controle de Versão

A metodologia de trabalho contida neste guia foi desenvolvida utilizando-se do Subversion, um sistema de controle de versão. O objetivo deste capítulo é apresentar esta ferramenta, incluindo um breve histórico, características, particularidades, benefícios e as razões de sua escolha como instrumento para o controle de versão deste guia.

3.1 Breve Histórico

Por muitos anos, o *Concurrent Versions System* (CVS) [2, 3, 4, 5, 6, 7] foi o mais popular sistema de controle de versão no mundo do *Open Source Software* (Software de Código Aberto), sendo considerado praticamente um padrão nesta esfera, mas também frequentemente adotado fora dela. Extensivamente utilizado ainda hoje, a popularidade desta ferramenta é facilmente explicada: o próprio CVS é um software de código aberto; seu modo de utilização não é restritivo, ou seja, não há somente uma forma de trabalhar com CVS, permitindo a adoção de diferentes políticas de acordo com o projeto; baseado no modelo de versionamento *Copy-Modify-Merge*, mas possibilitando também o uso de *locks*; amplo suporte em operações pela rede, permitindo que desenvolvedores geograficamente separados pudessem compartilhar projetos e unir esforços. Estas características tornaram o CVS uma das pedras angulares da cultura de código aberto.

Apesar de ter sua utilização altamente difundida, o CVS tem algumas deficiências, apresentando vários problemas, falhas e limitações (que serão apresentadas mais adiante) que tornavam seu uso, em alguns momentos, ineficiente, ineficaz, inconsistente e até mesmo irritante. Considerando que consertar as deficiências do CVS seria uma operação que demandaria excessivo tempo e esforço, um grupo de pessoas (entre elas desenvolvedores do CVS, usuários de CVS e autores de livros sobre CVS) apoiado por uma empresa (CollabNet) resolveu criar um novo sistema de controle de versão de código aberto. Este nova ferramenta, o Subversion (SVN) [2, 3, 4, 7], deveria fornecer funcionalidades similares às do CVS, preservar o mesmo modelo de desenvolvimento e não apresentar as falhas e limitações de seu antecessor, de forma que fosse similar o suficiente para que usuários do CVS pudessem realizar a migração com um mínimo de esforço e aprendizado. Desta forma, o Subversion foi construído, tendo sua versão 1.0 lançada no início de 2004.

Apesar de não poder ser considerado o próximo passo evolutivo das ferramentas de controle de versão, o Subversion é o sucessor melhorado do CVS e vale ressaltar que este seu antecessor já se encontra vários passos adiante de outras ferramentas do gênero. Isso coloca o Subversion muito à frente do que existe atualmente na esfera do código-aberto, sendo inclusive um competitivo sério das ferramentas pagas. Por todo o contexto apresentado, o Subversion ocupa atualmente (no lugar de seu antecessor) a posição de ferramenta padrão para os novos projetos de código aberto, sendo gradualmente adotado em projetos já existentes que utilizam seu antecessor, o CVS.

3.2 Características do Subversion

Transcorrida a devida apresentação do Subversion (e sua similaridade com o CVS), esta seção tem como objetivo descrever as características mais relevantes para este guia desse sistema de controle de versão. Serão também descritas, quando pertinente, as vantagens do Subversion sobre seu antecessor.

3.2.1 Versões de Diretórios

O CVS mantém o registro histórico de modificações apenas de arquivos, individualmente. O Subversion, no entanto, implementa um sistema de arquivos capaz de registras o histórico de mudanças de árvores de diretório inteiras ao longo do tempo, ou seja, o Subversion controla não só as versões dos arquivos, como também dos diretórios.

3.2.2 Histórico de Versão Realista

Como o CVS controla somente a versão de arquivos individualmente, operações como copiar ou renomear arquivos, que podem acontecer a arquivos mas são na verdade mudanças no conteúdo de um diretório, não são suportadas. Como um efeito colateral indesejado, ao excluir um arquivo e alguns *check ins* depois incluir um outro arquivo com o mesmo nome, o novo item herdará todo o histórico do item anterior, mesmo que o novo arquivo tenha pouco ou nada a ver com o anterior. Com o Subversion, é possível adicionar, excluir, copiar e renomear tanto arquivos quanto diretórios. Cada novo item adicionado tem um histórico próprio, limpo e novo.

3.2.3 *Commits* Atômicos

Ao contrário do que acontece no CVS, no Subversion um conjunto de modificações (que pode envolver diversos arquivos e diretórios) é enviado para o repositório de forma atômica, ou seja, ou todas as modificações são registradas no repositório ou nenhuma é. Inicialmente, isso pode não parecer importante, mas em um sistema de controle de versão onde esta característica não está presente, os seguintes efeitos colaterais indesejados podem ocorrer:

- Cenário 1: Em um *commit* de três artefatos, o último na lista de envio não está atualizado na cópia de trabalho e terá seu recebimento rejeitado pelo repositório. Se o *commit* não for atômico, o repositório aceitará os dois primeiros, o que acarretará em uma inconsistência dos artefatos no repositório, já que as modificações realizadas pelo desenvolvedor só funcionariam, por exemplo, se todos os arquivos tivessem sido enviados com sucesso. No caso de *commit* atômico, todos os três itens seriam rejeitados, não permitindo inconsistências no repositório.

- Cenários 2: Um desenvolvedor realiza um *commit* com 200 artefatos, operação que tem uma latência de tempo para acontecer. Em um *commit* não atômico, enquanto esta operação está em andamento, é possível que um outro desenvolvedor efetue *update* em sua cópia de trabalho e obtenha algumas das modificações presentes no *commit* mencionado acima, mas não todas, causando uma inconsistência em sua área de trabalho. No caso de *commit* atômico, esta inconsistência não seria possível já que o segundo desenvolvedor teria em sua cópia de trabalho ou a versão do repositório antes do *commit* ou a versão depois do final do *commit*, dependendo do momento em que o *update* foi comandado.

Como conseqüência da característica de *commit* atômico, o número de revisão no Subversion tem uma regra diferente do número de revisão no CVS. No CVS, os números de revisão são atribuídos por artefato. Assim, é possível, por exemplo, um repositório conter dois artefatos: um com o número de revisão 2 (ou r2, aderindo à **notação de número de revisão** [2, 3, 4, 5]) e outro com o número de revisão r123. Neste conceito o número de revisão de um artefato significa o número de vezes que o artefato foi modificado (incluindo sua criação).

No Subversion, o conceito de número de revisão tem outro contexto: aplica-se ao repositório como um todo, ou seja, é um número de revisão global. Neste conceito, o número de revisão rN representa o estado do repositório após o enésimo *commit*. Desta forma, quando a equipe de desenvolvimento se refere, por exemplo, à "r456 do artefato `Fachada.java`", isso não significa que este artefato foi modificado 456 vezes, mais sim que a equipe se refere a "`Fachada.java` como estava na revisão 456 do repositório".

Esse conceito de número de revisão decorrente de *commit* atômico auxilia em algumas tarefas como, por exemplo, reverter a uma versão anterior do repositório (o que é bastante delicado quando cada arquivo possui um número de revisão diferente) ou obter respostas a perguntas como "o que foi modificado entre as revisões 75 e 78?".

Por fim, a introdução deste conceito parece levar à "perda da informação" da quantidade de mudanças por artefato em seu número de revisão. No entanto, o Subversion possui mecanismos que determinam facilmente a quantidade de mudanças em um artefato em um período, não havendo necessidade de esta informação estar presente no número de revisão. Em todo caso, a quantidade de vezes que um artefato foi modificado é uma informação fútil, já que uma modificação pode significar mudança em uma linha ou a criação de 20 métodos, cada um com 50 linhas, sendo mais valioso avaliar o que foi modificado em cada versão do artefato, o que também é facilmente obtido no Subversion.

3.2.4 Metadados e Versões de Metadados

O Subversion possui um mecanismo através do qual é possível atribuir informações diversas aos artefatos, sem que seja necessário armazená-los no interior do repositório. Este metadados são chamados **propriedades** (*properties*) [2, 3, 4]. Cada artefato pode possuir uma ou mais propriedades, que são em essência pares do tipo chave/valor. Mesmo havendo propriedades pré-definidas pelo Subversion, é possível criar e armazenar propriedades personalizadas, não havendo um limite para a quantidade de propriedades por artefato.

O Subversion também controla a versão das propriedades de um artefato: criação, exclusão e modificações.

3.2.5 Tagging e Branching Eficientes

No CVS, tanto o tempo para realização de *tagging* e *branching* como o espaço em disco consumido por estas operações são diretamente proporcionais ao número de artefatos envolvidos no processo. Devido à arquitetura avançada de seus repositórios, no Subversion, as operações de *tagging* e *branching* apresentam constantes (e baixas) quantidades de tempo e espaço em disco.

Devido às características do Subversion de possuir *commit* atômico e, por conseguinte, número de revisão global, tecnicamente não seria necessária a criação de *tags* neste sistema de controle de versão**:** cada revisão do repositório consiste em uma *tag*, que contém os estado global do repositório em um momento específico, identificado pelo número de revisão. A única razão da criação de *tags* no Subversion é a possibilidade de se nomear uma revisão específica com um título amigável, mais fácil de ser lembrado posteriormente.

3.3 Sobre a Escolha do Subversion

Todas as características, vantagens e benefícios apresentados na seção anterior exerceram influência na escolha do Subversion como ferramenta para este guia. Além delas, serviram como argumento para sua escolha:

- **Simples de instalar e de usar**. A instalação e manutenção básica do servidor é simples e o tempo de aprendizado da equipe para trabalhar com o Subversion é curto;
- **Funciona sobre redes TCP/IP**. Não só funciona sobre o atual padrão de redes de computador, como o faz de forma eficiente: trafegam entre cliente e servidor (em ambas as direções) somente as diferenças entre os itens alterados (artefatos e/ou diretórios) e não os itens em seu tamanho completo;
- **Leve e eficiente**. No Subversion, o custo das diversas operações é proporcional ao tamanho das mudanças e não a tamanho dos dados envolvidos;
- **Integrável ao Apache**. O Subversion tem a possibilidade de ser integrado a um servidor de rede Apache, tendo como ganho: o uso do protocolo HTTP para comunicação cliente/servidor; autenticação de usuário integrado ao domínio do servidor; compressão de dados na comunicação cliente/servidor; navegação básica de repositório em um *web browser*. A integração do Subversion com o Apache, no entanto, requer um alto grau de expertise técnica para sua implantação, configuração e manutenção e, por esta razão, a integração não será utilizada neste guia. Contudo, a integração com Apache é opcional (utilizada somente para o ganho de funcionalidades não-essenciais ao processo) e sua ausência não impede em absoluto a realização de controle de versão;
- **Multi-plataforma**. Existem versões da ferramenta para os mais populares sistemas operacionais do mercado: Linux (diversas distribuições), MS Windows, Apple Mac OS e Sun Solaris;
- **Código-aberto**. Para que o guia possa ter maior penetração e aceitação, o fato da ferramenta ser código-aberto e, por conseguinte, gratuita é um fatores de grande influência na escolha.

Capítulo 4

Controlando Versões com Subversion

Os dois capítulos anteriores apresentaram conceitos fundamentais sobre, respectivamente, a disciplina de controle de versão e o Subversion, uma base teórica necessária para o entendimento deste guia. O conteúdo deste capítulo é o objetivo central deste trabalho: a definição de um guia para controle de versão de projeto de software, utilizando a ferramenta Subversion.

4.1 Papel do Guia no Processo de Desenvolvimento

O *Rational Unified Process* (RUP) [1], uma renomada abordagem de procedimentos de desenvolvimento de software, define como uma de suas disciplinas de processo a Gerência de Configuração e Mudanças (*Configuration and Change Management* – CCM), que é composta de dois aspectos básicos:

- Gerência de Configuração (*Configuration Management - CM*), cuja responsabilidade é a estrutura do produto (projeto de software) e convenciona que os artefatos devem estar sob controle de versão;
- Gerência de Requisição de Mudança (*Change Request Management*), que tem como responsabilidade o controle das solicitações de mudança no produto.

A necessidade de um guia para controle de versão de projeto de software reside no fato de o RUP afirmar que em CM é necessário realizar controle de versão, mas não fornece informações sobre como este controle deve ser realizado. Assim, as informações contidas neste guia podem ser utilizadas para auxiliar nas seguintes atividades do processo de CCM, que estão ilustradas na Figura 8:

- Planejar Configuração de Projeto e Controle de Mudanças (*Plan Project Configuration and Chage Control*)
- Criar Ambientes de CM do Projeto (*Create Project CM Environments*);
- Monitorar e Reportar Itens de Configuração (*Monitor and Report Configuration Items*);
- Mudar e Entregar Itens de Configuração (*Change and Deliver Configuration Items*);
- Gerenciar Linhas de Desenvolvimento e *Releases* (*Manage Baselines and Releases*).

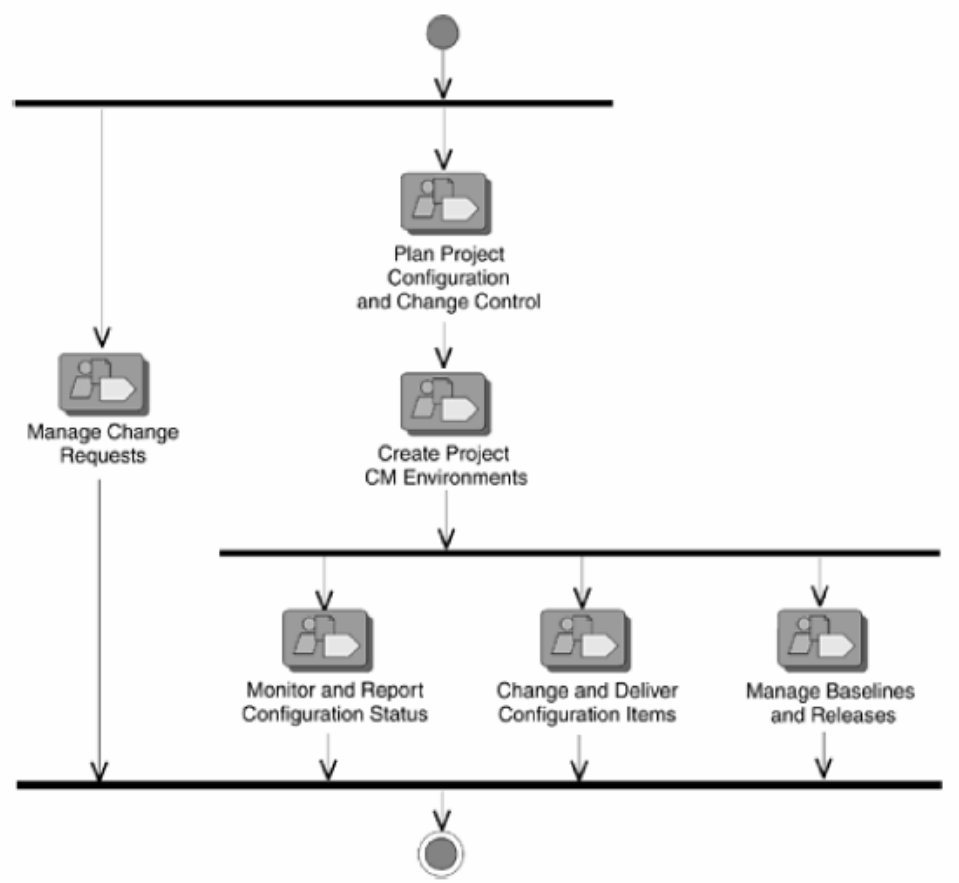

Figura 8. O Fluxo da Gestão de Configuração de Mudança, de acordo com o RUP [1]

4.2 Premissas e Pré-requisitos

Esta seção destaca os pré-requisitos para a utilização deste guia como o ambiente de desenvolvimento, ferramentas envolvidas e a natureza dos artefatos.

4.2.1 Ambiente e Ferramentas

Um dos pré-requisitos fundamentais para o controle de versão de um projeto de software é a presença de uma rede de computadores sobre o protocolo TCP/IP entre o servidor (estação central onde está situado o repositório) e os clientes (estações onde trabalham os desenvolvedores e onde estão localizadas as áreas de trabalho).

No servidor, o sistema operacional poderá ser qualquer um dos previamente indicados como compatíveis com o Subversion (Capítulo 3). A versão mínima da ferramenta para emprego neste processo é o Subversion 1.3.0, tendo como versão recomendada o Subversion 1.4.5, a mais recentemente lançada[1].

Conforme mencionado anteriormente, o processo descrito neste guia não utiliza a integração com Apache. A comunicação dos clientes com o servidor se dará através do serviço *stand-alone* chamado ***svnserve*** [2, 3, 4]. Existem tutoriais disponíveis na WEB que demonstram como fazer com que o *svnserve* funcione como serviço no Windows (ou como um *deamon* no

[1] Versão lançada em 27 de agosto de 2007.

Linux). Além disso, a autenticação de usuário deverá ser feita através dos arquivos de configuração do repositório.

Nas estações cliente, o guia utiliza como ferramenta cliente o **TortoiseSVN** [8], um aplicativo Windows que funciona sobre o Windows Explorer e cuja a interface e instrumentos internos facilitam as operações básicas do fluxo de trabalho dos desenvolvedores, promovendo um menor tempo de aprendizado e maior produtividade. Deverá ser utilizada a versão mínima compatível com a versão instalada no servidor, sendo recomendada a versão TortoiseSVN 1.4.5, a mais atual[2]. Contudo, o guia define procedimentos de utilização que independem da ferramenta cliente adotada, podendo ser utilizada a ferramenta cliente de linha de comando do Subversion, disponível para qualquer sistema operacional compatível com o SVN. Todas as operações descritas no guia estão disponíveis tanto no TortoiseSVN quanto nos clientes Subversion de linha de comando, fornecendo uma alternativa para projetos em ambientes não-Windows.

Vale ressaltar que não é objetivo deste guia ensinar a instalar e configurar as ferramentas envolvidas no processo. Este tipo de informação pode ser obtido na WEB e na literatura listada na bibliografia deste trabalho.

4.2.2 Natureza dos Artefatos

Como o objetivo deste guia é auxiliar no controle de versões de projetos de software, os tipos de artefato de projeto preferenciais devem ser aqueles considerados mescláveis, ou seja, arquivos cujo conteúdo é texto plano, como, por exemplo, arquivos de código-fonte, XML, scripts, entre diversos outros.

Contudo, a presença de arquivos “não mescláveis” não é proibida, já que estes tipos de arquivos podem ter seu histórico de versão mantido, o que será mostrado mais à frente neste capítulo. Dentre os artefatos “não mescláveis” comuns em projetos de software destacam-se os cronogramas de atividades, diagramas diversos (casos de uso, seqüência, classe, entre outros), planilhas de testes, imagens, entre diversos outros.

Apesar de ser não poderem ser considerados artefatos de texto plano, documentos gerados por processadores de texto (normalmente com a extensão de arquivo “doc”) têm suporte para mesclagem através da utilização da ferramenta TortoiseSVN.

4.3 Iniciando o Controle de Versão

As subseções a seguir deste tópico descrevem como iniciar o trabalho de desenvolvimento de equipe o Subversion.

4.3.1 Criando o repositório

A criação do repositório deve ser feita no servidor, através do comando `svnadmin` (consultar referências bibliográficas [2, 3, 4]). É definido um nome para repositório no momento de sua criação, bem como o diretório no servidor onde será armazenado o repositório. Vale ressaltar que ao analisar o conteúdo desse diretório, não serão encontrados os artefatos do projeto tais como em uma área de trabalho, mas sim um conjunto de arquivos que fornecem os mecanismos necessários ao controle de versão. A única forma de ter acesso aos artefatos do projeto, seja qual for a versão, é através de um cliente Subversion.

[2] Mais especificamente a versão TortoiseSVN-1.4.5.10425, lançada em 27 de agosto de 2007.

Após a criação do repositório, seu endereço na rede para acesso por clientes Subversion terá o seguinte formato:

```
protocolo://endereco_servidor/nome_repositorio
```

Existem diversos protocolos possíveis para comunicação como Subversion e neste guia utilizaremos o protocolo `svn`, por causa da previamente comentada utilização do svnserve no servidor. O endereço do servidor pode ser um endereço IP, um endereço na Internet resolvível por DNS ou o nome de um computador em uma rede interna.

4.3.2 Comandos do Subversion no Cliente

A ferramenta cliente de linha de comando do Subversion possui um conjunto de instruções (comandos) através dos quais é possível efetuar operações no repositório. Não é objetivo deste guia ensinar esses comandos e suas respectivas sintaxes. Estes conceitos podem ser observados na literatura indicada na bibliografia deste trabalho.

A aplicação cliente TortoiseSVN, a ser utilizada neste guia fornece uma interface gráfica intuitiva para os comandos do Subversion. Seu menu, integrado no Windows Explorer quando a aplicação é instalada, está ilustrado na Figura 9.

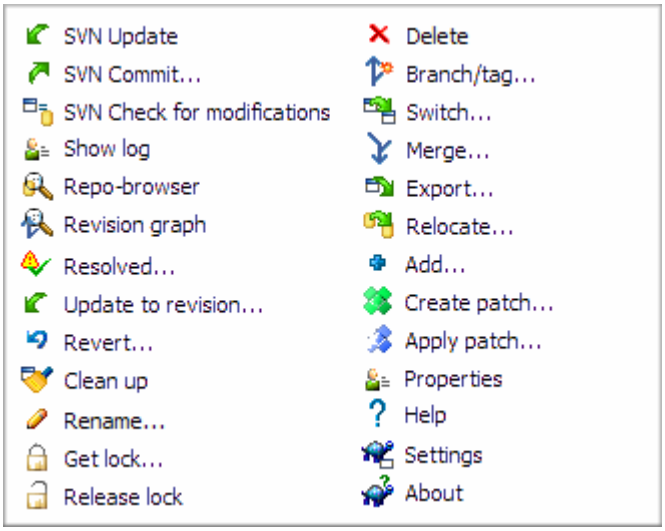

Figura 9. Menu de opções do TortoiseSVN

Além dos comandos de operação, o TortoiseSVN fornece ferramentas úteis como navegação de repositório, gráfico de revisões, log de alterações, auxílio para resolução de conflitos, entre diversas outras.

4.3.3 Layout do Repositório

Um projeto de software deve conter três diretórios considerados essenciais dentro da árvore do repositório:

- ***trunk*** – Este diretório contém os artefatos do projeto em seu estado referente à linha principal de desenvolvimento do software. A maior parte do trabalho dos desenvolvedores será feito no *trunk*;
- ***tags*** – Neste diretório são armazenadas as *tags* do projeto de software. Nele podem ser criados quantos subdiretórios quanto forem necessários, já que o nome dado a uma *tag* é

utilizado para nomear seu respectivo subdiretório. Podem ser criados outros subdiretórios para organizar internamente as *tags*, de acordo com critérios que podem ser definidos pelos gestores do projeto;

- ***branches*** – Este último diretório armazenará as linhas de desenvolvimento paralelas do projeto de software.

Tecnicamente, não há diferença no Subversion entre *tags* e *branches*, sendo ambos inclusive criados através do mesmo comando. A diferença é organizacional e processual: ao contrário dos *branches*, *tags* não devem ser modificadas, devendo permanecer estáticas ao longo do tempo.

Em situações onde dois ou mais projetos têm suas versões controladas, existem dois layouts básicos de repositório que podem ser seguidos. Quando os projetos compartilham código de forma a se tornarem mutuamente dependentes, os diretórios *trunk*, *tags* e *branches* devem estar no primeiro nível do repositório e dentro deles devem estar contidos os diretórios dos projetos. Por conta de compartilharem artefatos, engenhos ou até mesmo módulos, não faz sentido gerenciar *tags* e *branches* desses projetos separadamente, tampouco separar sua linha de desenvolvimento principal (o *trunk*).

Em contrapartida, quando os projetos não compartilham dependências, os diretórios dos projetos devem ficar no primeiro nível, cada um com seus próprios diretórios *trunk*, *tags* e *branches*. Como são projetos independentes, eles possuem seus próprios diretórios de controle de desenvolvimento (*trunk*, *tags* e *branches*). A Figura 10 ilustra as duas opções de layout de repositório, de acordo com o nível de dependência entre os projetos.

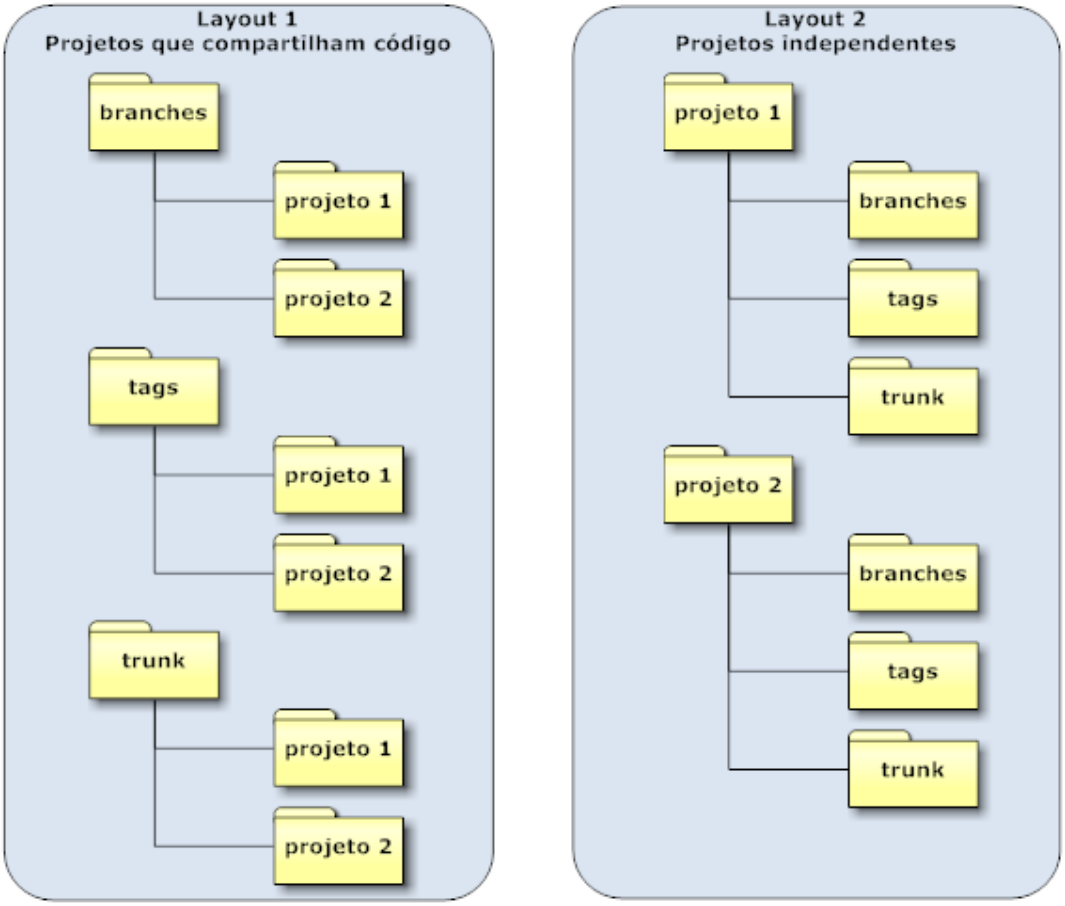

Figura 10. Opções de layout para repositórios com múltiplos projetos

4.3.4 *Checkout* e *Commit* Iniciais

No momento de sua criação, o repositório está vazio. Para que seja possível preenchê-lo, é necessário haver pelo menos uma cópia de trabalho ativa em uma estação cliente, o que é conseguido através do *checkout* inicial.

Nesse *checkout*, o desenvolvedor informa o endereço do repositório a ser acessado. Apesar de ser possível especificar a revisão do repositório a obter, no *checkout* inicial deve-se obter a versão mais recente do repositório (também chamada de ***head revision*** [2, 3, 4], sendo HEAD uma palavra-chave reconhecida pelo Subversion). Também é possível, em um *checkout*, especificar o subdiretório que está sendo obtido para trabalhar localmente. No caso do *checkout* inicial, deve ser especificado o diretório *trunk*, concatenando-o ao final do endereço, como por exemplo em `svn://servidor/repositorio/trunk`.

Além do endereço e da revisão, o desenvolvedor informa as credenciais de acesso para autenticação. O cliente Subversion (linha de comando ou TortoiseSVN) guarda localmente essas informações, para que não seja necessário autenticar-se a cada nova operação.

O *checkout* cria em um diretório indicado pelo desenvolvedor uma cópia de trabalho ativa, que consiste em uma estrutura de arquivos local capaz de operações de Subversion locais e com integração com o repositório.

Por fim, para povoar o repositório com os artefatos do projeto, é necessário copiar estes arquivos para o diretório da cópia de trabalho, adicioná-los localmente (comando `svn add` ou opção de Adicionar no menu do TortoiseSVN) e depois efetuar *commit*, operação que envia as alterações locais (no caso, a adição dos artefatos) para o repositório, deixando os artefatos recém adicionados disponíveis para os demais desenvolvedores. As operações de adição e *commit* serão mais exploradas nas seções a seguir.

4.4 Fluxo Básico de Trabalho

Esta seção tem por fim descrever o fluxo básico de trabalho, indicando o que deve ser feito, as melhores práticas e o que evitar nas operações de controle de versão.

4.4.1 Atualizando a Cópia de Trabalho

Antes de iniciar quaisquer mudanças no projeto, o desenvolvedor deverá certificar-se de que estará trabalhando com os artefatos em sua revisão mais recente da linha de desenvolvimento em que está trabalhando, a qual foi definida no *checkout* (na maioria dos casos, o *trunk*). Para isto, basta realizar um `svn update` (via linha de comando ou TortoiseSVN), ação que verificará quais os artefatos na cópia de trabalho estão defasados em relação ao repositório e obterá a *head revision* destes artefatos.

4.4.2 Modificando a Cópia de Trabalho

Basicamente, há dois tipos de modificações que podem ser feitas na cópia de trabalho: mudanças em arquivos e mudanças em diretórios. As mudanças em arquivos dizem respeito tão somente às mudanças realizadas no conteúdo de um ou mais artefatos. As mudanças em diretórios, por sua vez, podem se dar pelas operações básicas de adicionar (incluir um novo item, livre de histórico), excluir (remover um item), e copiar (criar uma cópia de um item, fazendo também uma cópia do seu histórico) arquivos ou diretórios. Existem outras operações de modificação de diretório possíveis, mas que são meramente composições da adição, exclusão e cópia: mover item

(copiar+deletar), renomear item (copiar+deletar), entre outras. Vale lembrar que todas as modificações realizadas na área de trabalho são locais à estação do desenvolvedor e só serão concretizadas no repositório quando for efetuado o *commit*.

Não é necessário, ainda, indicar previamente ao Subversion quais as alterações que serão realizadas. Para modificar artefatos, por exemplo, basta editar seu conteúdo. O Subversion automaticamente detecta que arquivos foram alterados, assim como detecta modificações nos diretórios da cópia de trabalho.

Por fim, não é imperativo estar conectado ao repositório para realizar modificações na cópia de trabalho. O Subversion fornece a vantagem de poder se trabalhar *offline*, sendo a conexão requerida somente no momento do *commit*.

4.4.3 Examinando as Modificações da Cópia de Trabalho

O Subversion dota a cópia de trabalho ativa de condições de saber, ao longo das modificações, o que foi alterado. A Figura 11 exibe o TortoiseSVN integrado ao Windows Explorer, através do qual o desenvolvedor pode verificar visualmente e de forma rápida as modificações ocorridas na cópia de trabalho.

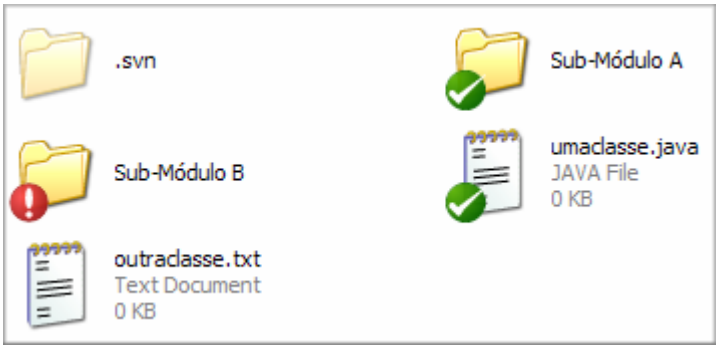

Figura 11. Windows Explorer com TortoiseSVN, indicando o *status* dos artefatos

No exemplo, os itens marcados com o ícone de *check* verde não foram alterados localmente, os marcados com o ícone de exclamação vermelha sofreram alterações e os itens sem ícone ainda não foram adicionados formalmente à área de trabalho.

Uma outra listagem, mais completa, pode ser obtida através do menu “SVN Check for Modifications” do TortoiseSVN. Nesta listagem, ilustrada na Figura 12, pode-se ver que o diretório “Sub-Módulo B” aparece como modificado na Figura 11 para alertar que o artefato “operações.java” sofreu uma alteração.

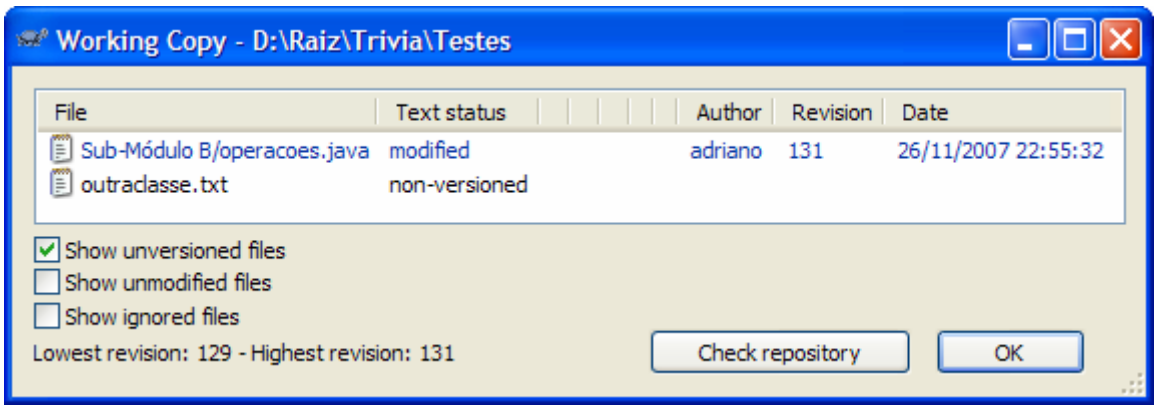

Figura 12. Listagem detalhada de modificações da cópia de trabalho.

Além disso, o TortoiseSVN fornece também uma ferramenta visual a qual possibilita o desenvolvedor de verificar detalhadamente as modificações realizadas no conteúdo de um artefato, conforme pode ser visto na Figura 13.

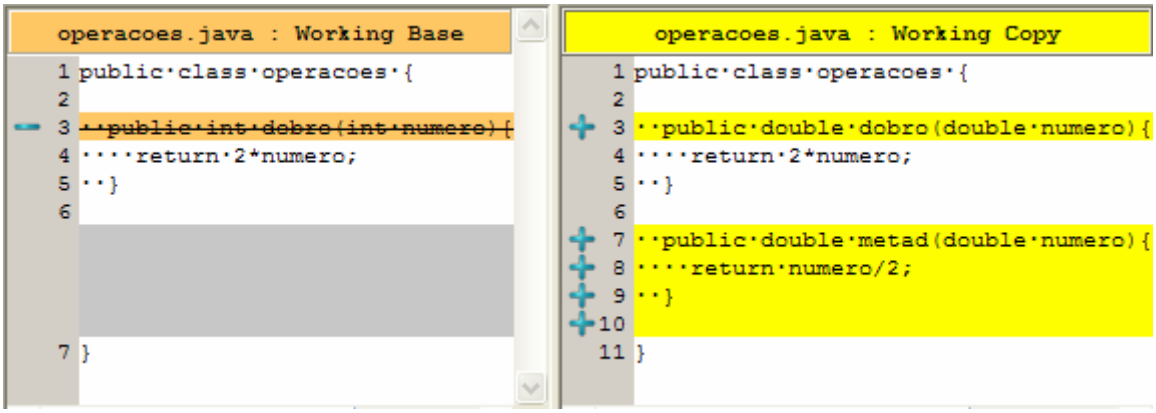

Figura 13. Modificações detalhadas em um artefato.

Neste exemplo, verificam-se as alterações realizadas no artefato "operacoes.java", sendo possível identificar que linhas do arquivo foram incluídas, modificadas ou excluídas. A janela do lado esquerdo, titulada "Working Base" mostra a versão do artefato como este se encontrava no momento em que foi obtido (por *checkout* ou *update*), ou seja, a versão local do artefato antes da modificação local. Do lado direito, é exibido o conteúdo deste artefato após a modificação, ou seja, como ele está na cópia de trabalho.

É importante destacar que os mecanismos de verificação de modificações apresentados nesta seção não necessitam que a estação esteja conectada ao repositório, podendo ser realizadas em *offline*.

4.4.4 Desfazendo Alterações na Cópia de Trabalho

O Subversion fornece a instrução *Revert* (reverter), cuja função é desfazer modificações locais na cópia de trabalho. É possível desfazer desde modificações em um único artefato a diversas modificações em vários artefatos e diretórios. Reverter um item significa perder todas as modificações realizadas neste item, revertendo-o ao seu estado original referente ao momento em que o item foi obtido pela cópia de trabalho. Esta operação é local e não exige estar conectado ao repositório, podendo ser feita em *offline*.

4.4.5 Melhores Práticas nas Modificações da Cópia de Trabalho

Ao efetuar modificações em sua cópia de trabalho, o desenvolvedor deve ter sempre e mente as seguintes práticas:

- **Sempre realizar *update* da cópia de trabalho antes de iniciar as modificações**. Apesar de já ter sido mencionada anteriormente, é uma prática que vale enfatizar. Esta ação garante que o desenvolvedor trabalhará na revisão mais recente dos artefatos e diminui a chance de conflitos;
- **Nunca efetuar travamento de artefatos "mescláveis"**. Apesar de funcionar no modelo *Copy-Modify-Merge*, o Subversion permite o travamento de itens no repositório. Este recurso, no entanto, deve ser erradicado para artefatos mescláveis;

- **Sempre efetuar travamento de artefatos "não mescláveis"**. O travamento de itens tem sua utilidade no processo de controle de versão, se usado de forma proveitosa e cuidadosa. O trabalho sobre artefatos que não podem ser mesclados deve ser serializado, por falta de alternativa. Ao modificar, por exemplo, uma planilha eletrônica, o desenvolvedor deverá travá-la, evitando assim que tenha seu trabalho sumariamente perdido. A perda de trabalho acontece quando há conflito em artefatos "não mescláveis", quando, ao final, uma das duas versões conflitantes será considerada e a outra descartada;
- **Somente reverter quando há certeza que as modificações podem ser desprezadas**. Quando se usa um sistema de controle de versão (aliado a um procedimento seguro, eficiente e eficaz de backup no servidor), virtualmente nada do que é feito se perde, com exceção das modificações locais às cópias de trabalho e reverter significa desprezar essas modificações sem deixar histórico. Portanto, antes de realizar esta operação, é necessário estar seguro que as modificações devem de fato ser desfeitas;
- **Sempre planejar suas modificações de acordo com a equipe**. As ferramentas auxiliam o desenvolvimento, contudo não substituem a comunicação entre os membros da equipe. O líder de projeto deve estar atento à distribuição das tarefas, para que dois desenvolvedores, por exemplo, não percam tempo realizando uma mesma modificação. Essa prática pode parecer óbvia, contudo é comum a diminuição da comunicação em times de desenvolvimento por conta do uso de um sistema de controle de versão.

4.4.6 Registrando Mudanças no Repositório: *Merges*, Conflitos e *Commit*

Após as modificações na cópia de trabalho, o desenvolvedor deve enviar para o repositório as alterações através de um *commit*. Antes disso, no entanto, é boa prática realizar um novo *update* no intuito de verificar se houve mudanças no repositório envolvendo os artefatos modificados localmente, caso em que um *commit* antes do *update* seria rejeitado, em todo caso. O *update* pode resultar em três situações:

- **Nenhum dos artefatos alterados na cópia de trabalho foi alterado no repositório**. Neste caso outros artefatos são atualizados (em alguns casos nenhum é), não interferindo nas modificações a serem enviadas. O *commit* pode ser efetuado sem problemas;
- **Pelo menos um dos artefatos foi modificado no repositório, mas em trechos diferentes**. Neste caso, o Subversion encarrega-se de efetuar o *merge* automático, listando os artefatos onde esta composição aconteceu. O desenvolvedor deve certificar-se de que a fusão foi realizada com sucesso, podendo fazer uso da ferramenta TortoiseMerge (mostrada anteriormente na Figura 13), que compara diferente versões e revisões de um artefato.
- **Pelo menos um dos artefatos foi modificado no repositório e em trechos que se sobrepõem**. Este caso configura uma situação de conflito.

Em uma situação de conflito, conforme exemplo ilustrado na Figura 14, o Subversion cria, para cada artefato conflitante, três arquivos que podem auxiliar a resolução.

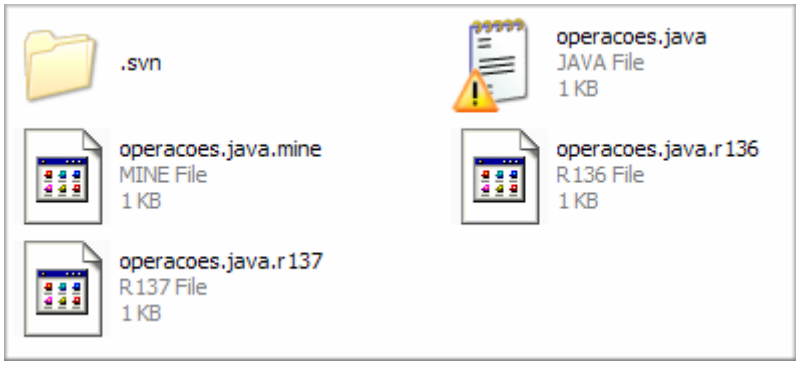

Figura 14. Artefato em situação de conflito.

Dois dos arquivos criados são cópias das duas últimas revisões mais recente do repositório (no exemplo, r136 e r137). O terceiro (com extensão “mine”) é uma cópia do artefato modificado na área de trabalho. Ao final da resolução do conflito, o resultado deve ser colocado no conteúdo do artefato, marcado no exemplo com um ícone de exclamação amarelo, indicador do conflito.

Para resolver este conflito, o desenvolvedor pode utilizar a abordagem manual (o que é desaconselhável) ou utilizar o TortoiseMerge, que pode ser evocado através do menu “Edit Conflicts” do TortoiseSVN. Esta ferramenta, conforme visto na Figura 15, presta um valioso auxílio na solução do problema.

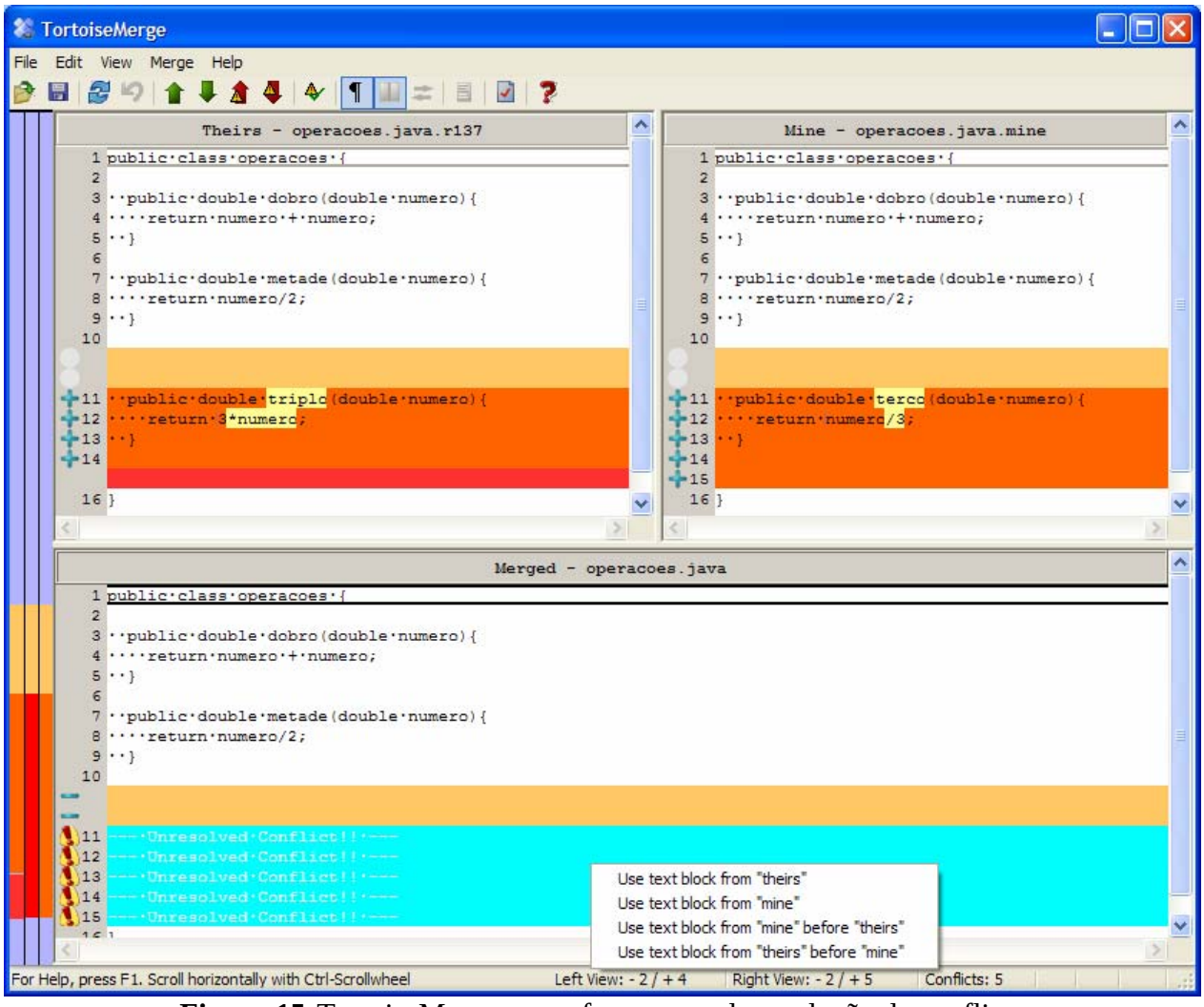

Figura 15. TortoiseMerge como ferramenta de resolução de conflitos

No exemplo, o conflito aconteceu porque dois desenvolvedores adicionaram um novo método no final do arquivo, uma área onde houve sobreposição. A ferramenta permite localizar e navegar entre pontos de conflito, bem como propor alternativas para sua resolução, através de um menu de contexto. No exemplo em questão, os dois métodos devem ser incluídos na versão final, sem haver importância na ordem deles dentro do artefato. Assim, o desenvolvedor pode utilizar uma das duas últimas opções do menu de contexto, que são "usar meu bloco de texto antes do deles" e vice-versa. O aplicativo se encarregará de distribuir o código de acordo com o especificado, o que deve ser verificado e validado pelo desenvolvedor. Ao final, o desenvolvedor marca o conflito como resolvido, o que faz com que o artefato seja mesclado e possa ser enviado ao repositório.

Somente após a realização de todas essas etapas, o desenvolvedor poderá realizar o *commit* de suas modificações. Na operação de *commit*, o desenvolvedor deverá especificar um texto descritivo das modificações sendo enviadas ao repositório, também conhecida como mensagem de log. A Figura 16 ilustra a tela de *commit* do TortoiseSVN.

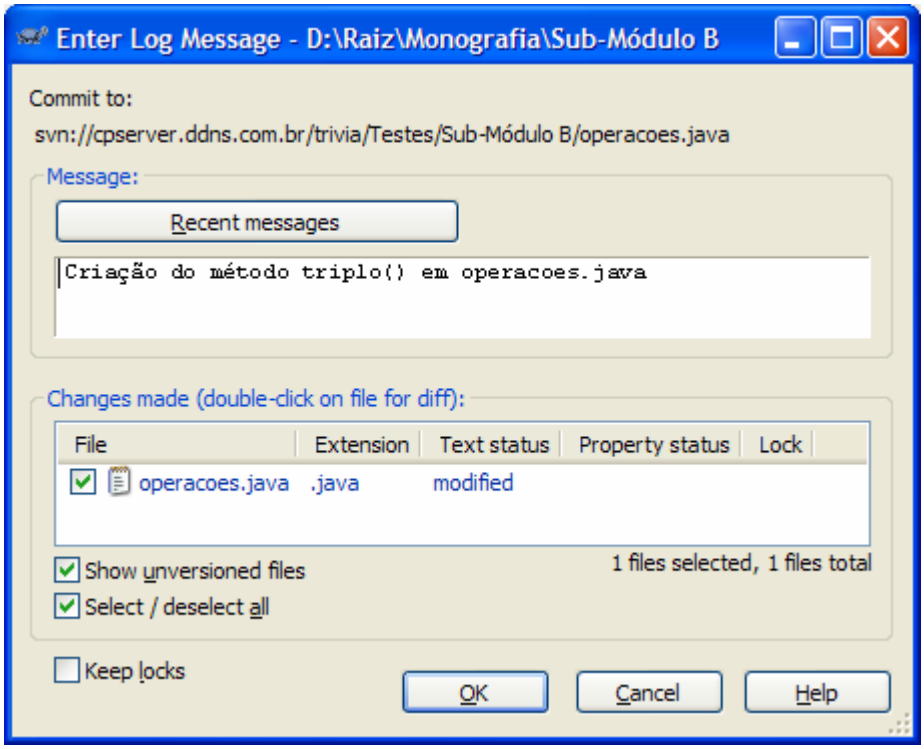

Figura 16. Tela de *commit* do TortoiseSVN.

Caso o *commit* seja rejeitado pelo repositório, o que aconteceria somente se uma nova modificação de um artefato envolvido fosse enviada para o repositório enquanto o desenvolvedor não enviasse a sua versão, todo processo teria que ser repetido a partir do *update*.

4.4.7 Melhores Práticas no envio de Modificações para o Repositório

Ao enviar modificações de sua cópia de trabalho para o repositório, o desenvolvedor deve ter sempre e mente as seguintes práticas:

- **Efetuar *commit* sem demora e frequentemente**. Enviar para o repositório assim que as modificações estiverem prontas evita a perda da versão local por motivos adversos como,

por exemplo, falha no hardware. Apesar de também estar suscetível a falhas de hardware, é bem mais provável o servidor contar com soluções de backup do que uma estação. Além disso, essa prática diminui os riscos de acontecer *merging* e conflitos. Quanto mais tempo se leva para submeter as modificações ao repositório, maiores são as chances de haver conflitos e mais complexo se torna resolvê-los;

- **Procurar fazer modificações atômicas e quanto menor possíveis**. Não é boa prática o desenvolvedor tentar resolver mais de um problema antes de efetuar o *commit*. Extrapolando, seria uma péssima prática desenvolver todo um módulo sem efetuar *commit*. Quanto mais atômicas as modificações, melhor o rastreamento das mudanças, mais facilmente identificável fica a introdução de problemas (*bugs*) e mais natural se torna o texto do *commit*. O desenvolvedor deve procurar corrigir no máximo um problema por vez, salvo a exceção quando ele está resolvendo uma questão de médio ou grande porte e encontra um pequeno problema que lhe custará pouquíssimo esforço para resolver, como, por exemplo, algo na ordem de até 5 linhas de código ou até 1 minuto. O planejamento das mudanças a serem realizadas é fundamental e deve ser discutido com o líder do projeto;
- **Em caso de conflito, não dê um novo *update* antes de resolvê-lo**. O TortoiseMerge só é capaz de comparar dois artefatos por vez, já que em um conflito básico são duas as versões sobrepondo código. Em caso de um novo *update*, é possível que o artefato tenha sido alterado novamente no repositório, trazendo para a cópia de trabalho mais uma versão do artefato e fazendo com que haja um conflito ternário, contexto em que a ferramenta não terá condições de auxiliar. Resolver conflitos ternários (ou de maior ordem) manualmente é uma tarefa longe de ser trivial;
- **Escrever mensagens de log significativas**. Não há ferramenta que valide uma mensagem de log, portanto isto deve ser cobrado dos desenvolvedores. Deve ser descrito, em poucas linhas, o que foi corrigido e como, em texto comum, nunca código-fonte. Caso haja algum na organização um sistema de gestão de requisição de mudanças, deve ser especificado o código/número/ID da solicitação atendida pela mudança. É preferível que as mensagens de log sejam padronizadas na organização. Quanto melhores as mensagens de log, mas rastreáveis são as modificações no repositório;
- **Destravar os artefatos previamente travados**. Após feitas as devidas modificações nos artefatos "não mesclavéis" travados anteriormente e realizado o *commit* destas alterações, não há razão pela qual manter a trava sobre os artefatos em questão. As ferramentas envolvidas no processo dão subsídios para a automação desta tarefa;
- **Evitar "quebrar a árvore do projeto"**. Apesar de parecer uma recomendação óbvia, são comuns ocasiões em que um desenvolvedor efetua *commit* de artefatos que sequer compilam. Isto é grave: outros desenvolvedores, ao atualizarem suas cópias de trabalha, obtêm esta "quebra", que pode muitas vezes impossibilitar o trabalho de toda a equipe. Deve ser vedado ao desenvolvedor tomar o *commit* por um salvamento parcial de trabalho, seja por que razão ele o faça. Uma correção não deve ser enviada pela metade, um artefato deve ser, no mínimo, compilável e o projeto, quando não for impeditivo, deve ser reconstruído (*build*) na área de trabalho do desenvolvedor. Todo procedimento viável que ajude a garantir a integridade do repositório deve ser implementado.

4.5 Examinado o Histórico

Existem várias formas de examinar o histórico de modificações de um ou mais itens de um repositório. A mais prática e versátil é através do menu "Show log" do TortoiseSVN. Na tela de

log de modificações de um item, conforme ilustrado na Figura 17, estão disponíveis as seguintes informações e ações:

- Listagem de modificações em um período, com número da revisão, data, hora, usuário e mensagem de log;
- Comparação de revisões de um artefato, através de texto e no TortoiseMerge, dando uma visão do que exatamente foi alterado em um artefato entre duas revisões quaisquer;
- Busca por mensagem de log, usuários, caminho do artefato e/ou número de revisão;
- Exibir estatísticas gerais de modificação, gráficos de *commits* por data e por usuário;
- Reverter a cópia de trabalho local para uma determinada revisão do repositório, possibilitando assim desfazer alterações já registradas no repositório.

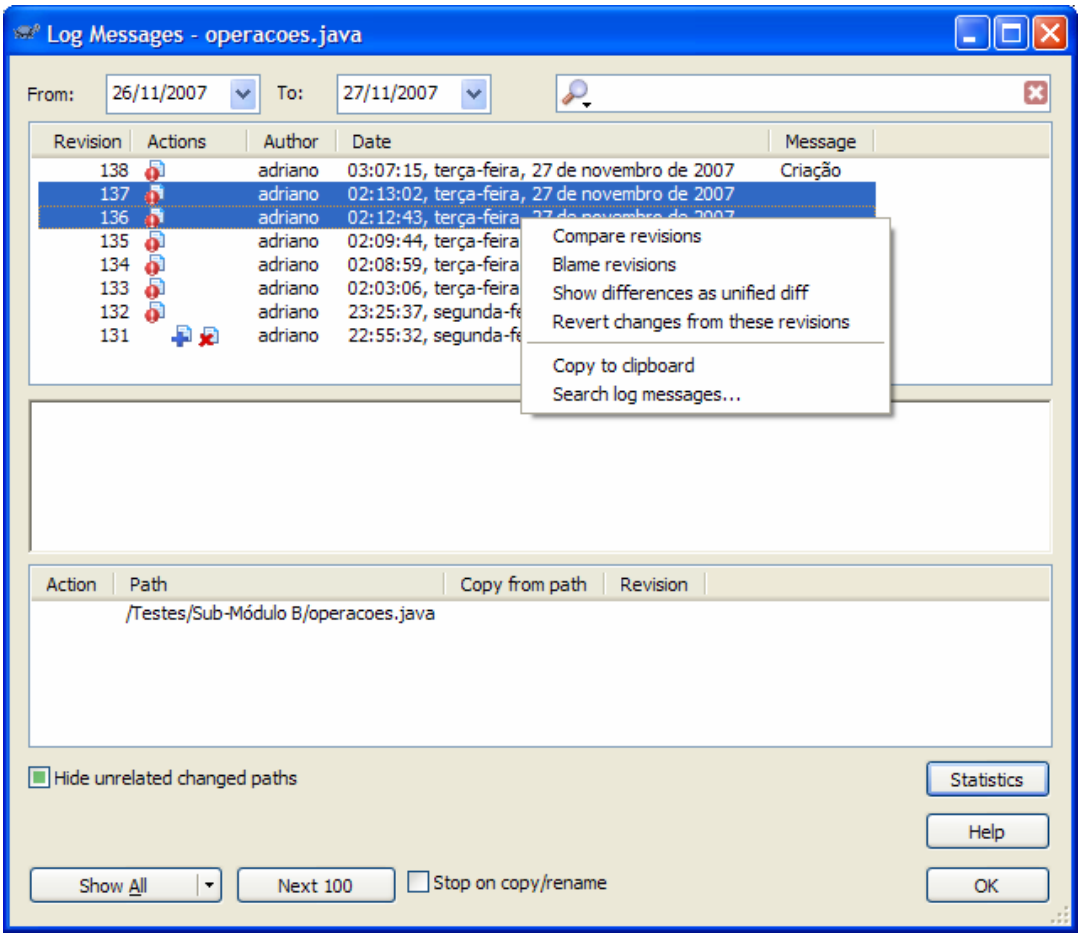

Figura 17. Tela de log de modificações do TortoiseSVN

4.6 *Tagging* e *Branching*

Esta seção tem por meta a definição das situações em que são utilizados os mecanismos de *tagging* e *branching*, bem como indicar as melhores práticas e que evitar nestes métodos não-triviais do controle de versão.

4.6.1 *Tagging*

Conforme foi mencionado anteriormente, do ponto de vista técnico, a criação de *tags* é um mecanismo redundante, já que cada revisão do repositório consiste em uma *tag*. No entanto, do ponto de vista processual e organizacional, as *tags* exercem uma função importante na identificação etapas do projeto.

Basicamente, existem duas situações em que é pertinente a criação de uma *tag*:

- ***Milestone Tag***. É uma *tag* que indica um marco interno atingido no projeto. Por exemplo, num projeto de automação comercial, cria-se uma *tag* para indicar o momento em que o módulo de estoque foi finalizado. É útil para manter um registro legível da evolução do projeto, no entanto, não pode considerada uma *tag* de criação obrigatória;
- ***Release Tag***. Esta uma *tag* de criação obrigatória, uma *release tag* deve ser criada sempre que uma versão do produto do software for liberada para seus usuários. É de extrema importância haver no repositório uma fotografia dos artefatos do projeto no momento em que houve uma liberação de versão. Este tipo de *tag* é um provável candidato a conceber, em algum momento, um *branch*.

Por fim, *tags* não devem ser alteradas. Mesmo que tecnicamente não haja diferença entre uma *tag* e um *branch*, não é boa prática a modificação de artefatos de uma *tag*: esta deve ser sempre uma "fotografia estática" do estado do repositório em um dado momento.

Para criar uma *tag*, basta utilizar a instrução "Branch/Tag" do menu de contexto do TortoiseSVN.

4.6.2 *Branching*

A ramificação do trabalho em duas ou mais linhas de desenvolvimento em um projeto de software é um mecanismo que aumenta a complexidade de todo o processo:

- O trabalho sobre a gestão do projeto aumenta, tendo em vista que é preciso mapear as ramificações e sua relação com a linha principal e controlá-las de forma ordenada, de forma a não permitir um caos organizacional no repositório;
- A atenção da equipe é dividida entre as ramificações criadas. Todos os membros da equipe devem estar cientes de quais *branches* existem, qual a finalidade de cada um deles e quais devem ser utilizados no dia-a-dia do time;
- As atividades de planejamento se intensificam, dado que a maioria dos *branches* retorna, em algum momento, à linha principal de desenvolvimento. Este retorno (*merging*) deve ser previsto e planejado, de forma que não se perca o controle sobre as ramificações em andamento.

Assim sendo, o uso de *branches* deve ser limitado somente a situações em que este mecanismo se mostre estritamente necessário. Existem algumas boas razões para se fazer uso deste recurso:

- Uma modificação grande no produto, que leve muito tempo para ser concluída e/ou que envolva duas ou mais pessoas no processo, pode ser um bom candidato a *branch*, quando a próxima versão do produto tem previsão de liberação anterior ao prazo previsto de término da grande mudança em questão. Este tipo de *branch* é mais comum em projetos de grande porte;

- Uma mudança que deixaria a linha principal do projeto inutilizável para trabalhos não necessariamente relacionados. Por exemplo, a mudança do engenho de estoque em um sistema industrial afetaria direta ou indiretamente praticamente todos os módulos do produto. A solução seria realizar esta modificação em um *branch* à parte, para não interferir no trabalho da linha principal de desenvolvimento;
- Caso a equipe tenha necessidade de uma área experimental, pode ser criado um *branch* onde os desenvolvedores possam avaliar componentes de terceiros, avaliar funcionalidades não previstas ou ensaiar mudanças de interface com o usuário. Este tipo de *branch* exige uma equipe mais experimentada no ofício de trabalhar em projetos com diversas ramificações, para evitar o caos organizacional do repositório.

É essencial que o gerente de configuração ou o gerente de projeto avalie se os benefícios trazidos pela utilização de múltiplas linhas de desenvolvimento valem o custo e o trabalho de gerenciar essas múltiplas linhas e de manter a equipe atenta a este modelo de desenvolvimento.

No entanto, o *release branch*, é um tipo de ramificação virtualmente inevitável. Ele se faz necessário sempre que é liberada uma versão do produto para os clientes e esta versão apresenta pelo menos um problema. Este tipo de *branch* nasce, em maior parte, a partir de *release tags*, sendo dificilmente criadas a partir da linha principal de desenvolvimento.

No exemplo apresentado na Figura 18, observa-se um produto que em determinado momento tem sua versão 1.0 lançada para os clientes. Logo após este *release*, a linha principal de desenvolvimento continuou sua evolução, através da criação de novas funcionalidades, mas em um dado momento, foram constatados problemas na versão 1.0. Não é viável, neste contexto corrigir os problemas na linha principal e criar um novo *release* a partir dessa linha, pois ela contém modificações não terminadas em andamento. Tampouco é viável fazer com que os clientes aguardem a versão 2.0 para que os problemas sejam corrigidos. A solução foi criar um *release branch* a partir da *release tag* “Release 1.0”, onde foram corrigidos os problemas e rapidamente lançada uma nova versão para os clientes (1.1).

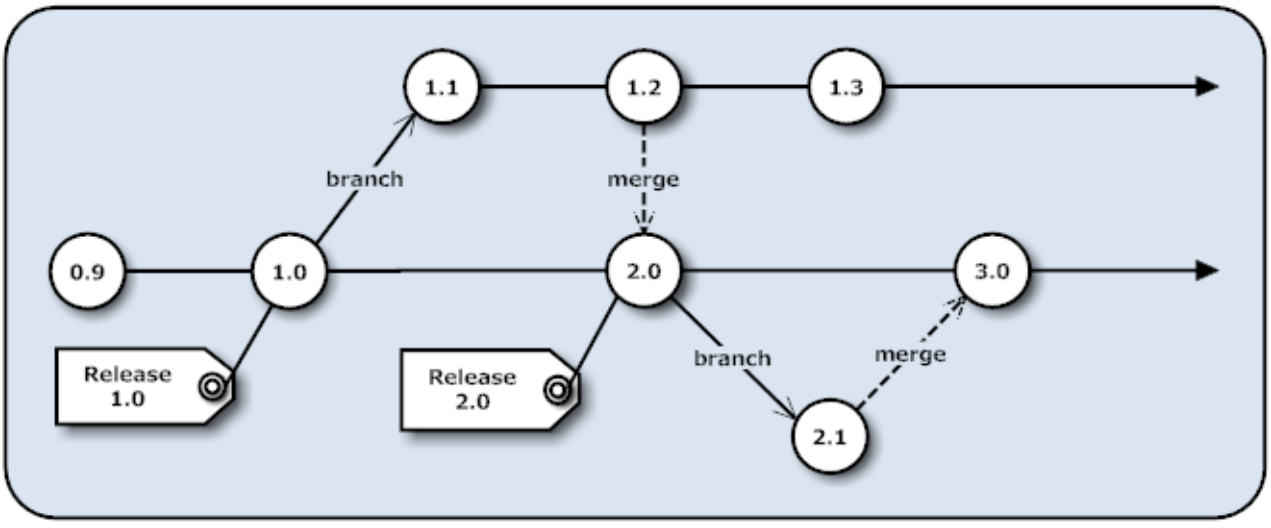

Figura 18. Produto de software com duas *release branches*

Enquanto a linha principal evolui em direção à versão 2.0, mais problemas são detectados nos clientes, que agora utilizam a versão 1.1 do produto. Por necessidade, esta será seguida pela 1.2. Pela proximidade estratégica do momento das versões 1.2 e 2.0, as correções realizadas ao longo do *release branch* superior (1.1 e 1.2) são aplicadas, por *merging*, para a versão 2.0.

Ainda, podem existir razões comerciais pelas quais os clientes não migrarão da versão 1.2 para a 2.0. Assim, a ramificação deverá ser mantida enquanto houver clientes ou enquanto a empresa estiver disposta (por contrato ou por mercado) a dar suporte às versões 1.X, podendo haver uma versão 1.3, criada a partir de correções da versão 1.2. Enquanto isso, a versão 2.0 apresentou problemas, justificando um *release branch* para corrigi-los. Desta vez, contudo a próxima versão da linha principal (3.0) estava agendada não muito no futuro, permitindo um *merge* das correções a partir da versão 2.1.

Para criar um *branch*, assim como uma *tag*, basta utilizar a instrução "Branch/Tag" do menu de contexto do TortoiseSVN. Para criar um *branch* a partir de uma *tag*, basta indicar a revisão a ser copiada na operação, conforme pode ser visto na Figura 19.

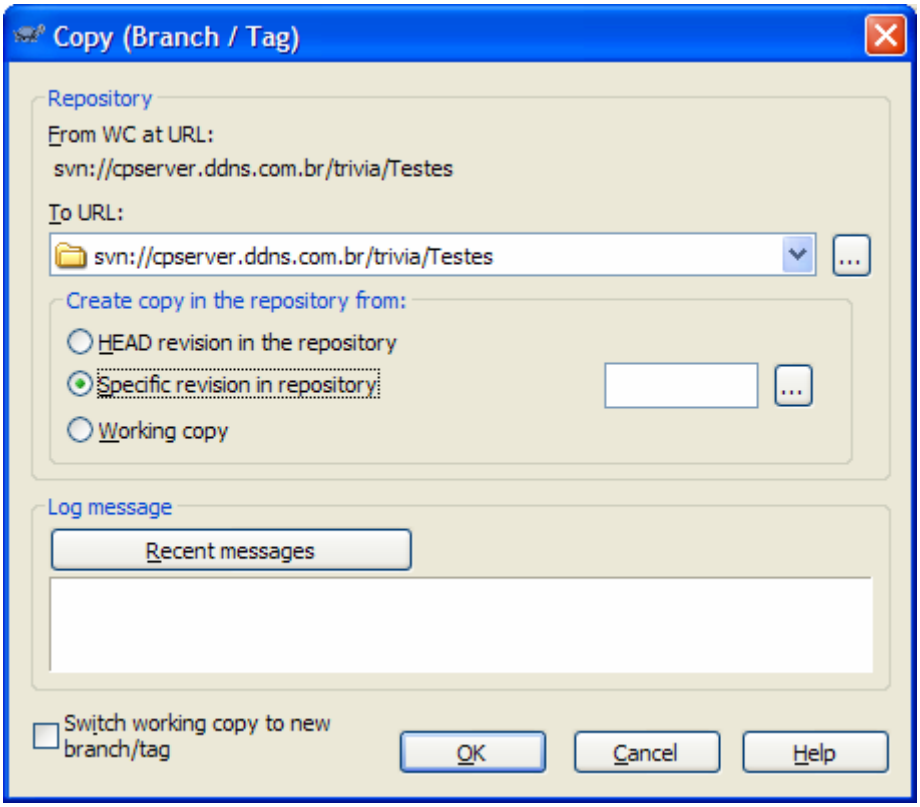

Figura 19. Tela de operação de *Branch*/*Tag*

Após a criação do *branch*, os desenvolvedores devem realizar *checkout* dessa nova ramificação para poder trabalhar nas alterações dessa linha paralela de desenvolvimento.

Para mesclar de volta à linha principal as alterações realizadas no *branch*, o TortoiseSVN fornece uma tela de fácil utilização, conforme ilustrada na Figura 20. Nesta ferramenta, é possível:

- Especificar as linhas de origem e destino do *merge*;
- Listar as diferenças entre as linhas e as modificações que serão aplicadas;
- Observar as modificações por artefato, individualmente;
- Simular o *merge* ("Dry run"), possibilitando estimar o resultado do *merge*.

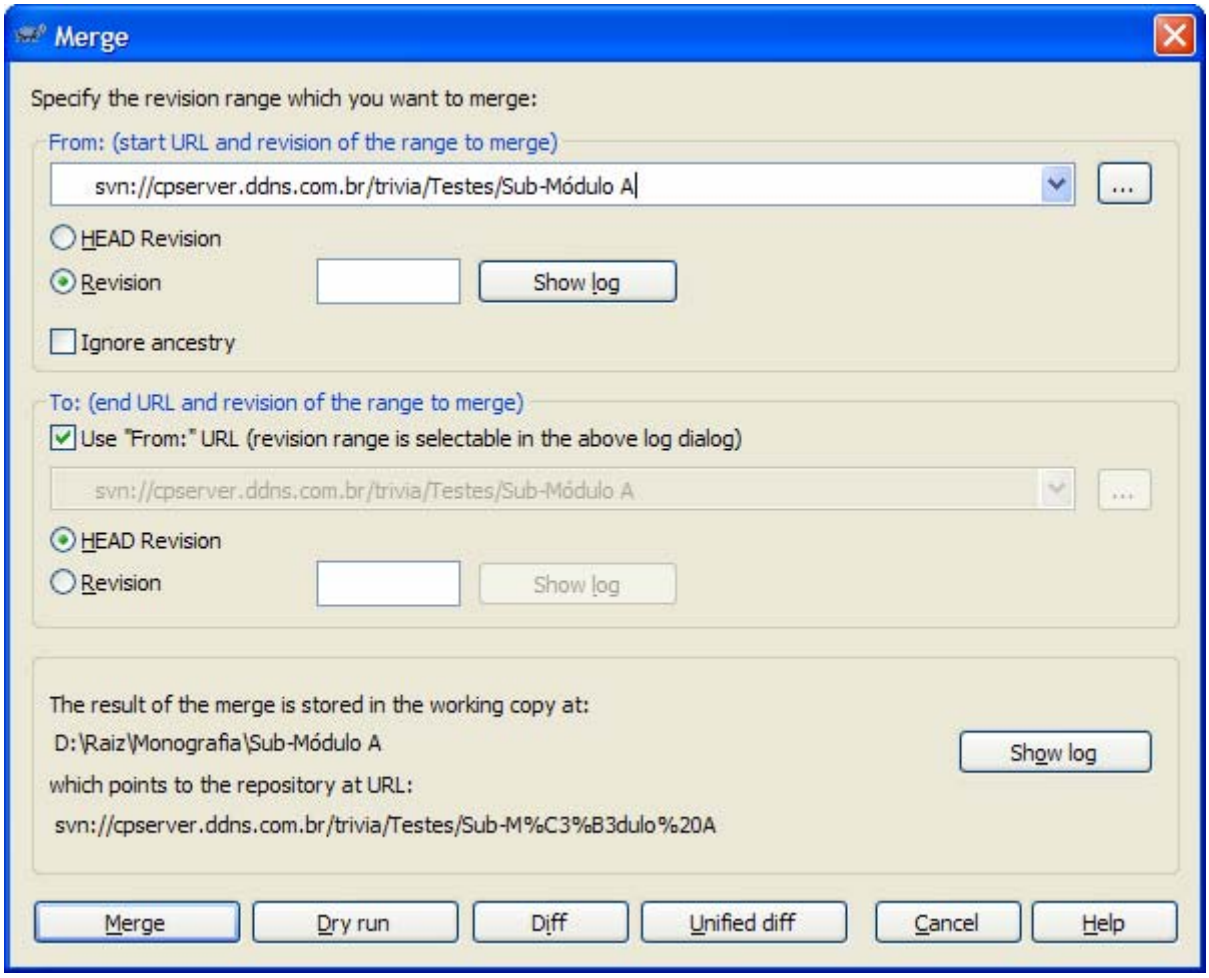

Figura 20. Tela de *Merge* entre diferentes linhas de desenvolvimento

4.6.3 Melhores práticas em *Branching*

Ao realizar operações de *branching* e *merging*, os membros da equipe devem ter sempre e mente as seguintes práticas:

- Ao trabalhar em um *branch*, o desenvolvedor deve ater-se a modificar o mínimo necessário para resolver os problemas. Colocar ou tirar linhas e espaços em branco e refazer indentação do código-fonte são modificações válidas no *trunk*. Em um *branch*, no entanto, são consideradas fúteis e prejudiciais, pois efetuam mudanças no artefato que em nada contribuem no ponto de vista do problema a resolver, já que desnecessariamente aumentam a complexidade e dificultam a operação de merge com o *trunk*;
- Apesar de vários membros da equipe poderem trabalhar no mais diversos *branches* do projeto, é recomendável que apenas pouquíssimas pessoas (uma ou duas, no máximo) seja responsável pelo *merge* das ramificações para o *trunk*. Esta operação requer experiência do profissional que irá realizá-lo, estando ele apto a resolver problemas que porventura surjam neste processo.

4.7 Um Fluxo Alternativo: Trabalhando com *Patches*

Em um projeto de código aberto, como o Subversion, por exemplo, todos os usuários têm acesso ao código-fonte do sistema, todos podem ler e até mesmo alterar os artefatos do projeto. No entanto, não são muitos os usuários que têm acesso a registrar alterações no repositório, ou seja, o acesso ao *commit* é restrito. Cada projeto código-aberto tem um grupo seleto de usuários com poderes para decidir o que entra no repositório. Isso acontece para controlar o projeto, de forma a evitar a constante "quebra da árvore" por desenvolvedores inexperientes ou mal intencionados.

Desta forma, para que possam ser aceitas contribuições de desenvolvedores comuns em um projeto código-aberto, é lançado mão do recurso de *patches*. Nesta abordagem, o desenvolvedor atualiza sua cópia de trabalho, realiza as modificações a que se propôs e ao final, ao invés de efetuar um commit, ele cria um *patch* (menu de contexto "Create Patch" do TortoiseSVN). Tecnicamente, um *patch* é um arquivo de texto plano onde são registradas as alterações realizadas em uma cópia de trabalho. Criado o *patch*, ele é enviado a um desenvolvedor privilegiado com o acesso a escrita no repositório. Este, por seu vez, atualiza sua cópia de trabalho e nela aplica o *patch* (menu de contexto "Apply Patch" do TortoiseSVN). Assim, todas as alterações realizadas pelo primeiro desenvolvedor são refeitas na cópia de trabalho do segundo, que analisa as mudanças e decide se as modificações devem ser enviadas ao repositório.

Esta abordagem é extremamente útil a projetos com muitos membros envolvidos no desenvolvimento, mostrando a flexibilidade da ferramenta de adaptar-se a novos cenários.

Capítulo 5

Conclusões e Trabalhos Futuros

A melhoria do processo de desenvolvimento de software vem sendo estimulada por um mercado cada vez mais exigente por qualidade e pela necessidade de mais produtividade no desenvolvimento. As atividades da SCM, entre elas o controle de versão, são essenciais no processo de produção de software, já que constituem um apoio para as demais atividades, mantendo o desenvolvimento controlável. No entanto, ainda existem muitos mitos sobre o custo e os benefícios destas atividades [9], o que impede que muitas empresas, principalmente as de micro e pequeno porte, façam uso de metodologias e ferramentas de SCM.

Portanto, o guia para controle de versão de projeto de software definido neste trabalho preenche uma lacuna existente em SCM, que carece de tais guias que definam seus processos internos. Conforme mencionado na lista de contribuições do Capítulo 1, este guia possibilita que um profissional da área de computação com conhecimento em ambientes e projetos de software seja capaz de implantar com sucesso uma política de controle de versões em uma organização.

Além disso, este trabalho demonstra que a implantação de uma política de controle de versão é factível e viável: pode ser implementada através de ferramentas de código aberto (Subversion); demanda um nível de gestão proporcional ao tamanho dos projetos e da organização; requer um treinamento simples de pessoal (desenvolvedores); e, por fim, um pequeno esforço de adequação a novos processos.

5.1 Trabalhos Futuros

O trabalho futuro mais natural ao atual estado deste guia é o seu aprofundamento teórico e técnico de seus principais tópicos:

- Ampliar a fundamentação teórica sobre controle de versão, incluindo padrões de SCM conhecidos, disponíveis na bibliografia [5];
- Fornecer mais detalhes técnicos quanto à instalação, configuração e manutenção de repositórios do Subversion no servidor;
- Pormenorizar mais ricamente as ferramentas cliente, tanto na ferramenta cliente Subversion de linha de comando como no TortoiseSVN;
- Dedicar mais espaço para a integração dos procedimentos do guia com os processos previstos no RUP, ou outro processo unificado de software, propondo uma instância pronta de processo para controle de versão;

- Incluir seções sobre testes, *builds* e *releases*, integrados com Subversion;
- Fornecer mais exemplos e contextualizações;
- Executar experimentos em projetos de software, através dos quais podem ser colhidas métricas para verificar o grau de eficácia dos procedimentos descrito no guia.

Além desse aprofundamento, o guia pode ser estendido de forma a adionar a outra pedra angular de SCM, a Gerência de Requisição de Mudança (*Change Request Management*), incluindo uma ferramenta que auxilie este novo controle e que se integre ao Subversion. Desta forma, o guia tem a possibilidade deixar de ser somente um guia de controle de versão para tornar-se um guia completo de SCM.

Bibliografia

[1] KRUCHTEN, P. *The Rational Unified Process: An Introduction, Third Edition*. Addison-Wesley, 2003.
[2] MASON, M. *Pragmatic Version Control Using Subversion*. Pragmatic Bookshelf, 2006.
[3] COLLINS-SUSSMAN, B.; FITZPATRICK, B.W.; PILATO, C. M. *Version Control with Subversion*. O'Reilly Media, 2004.
[4] BERLIN, D. e ROONEY, G. *Practical Subversion, Second Edition*. Apress, 2006.
[5] BERCZUK, S e APPLETON, B. *Software Configuration Management Patterns: Effective Teamwork, Practical Integration*. Addison-Wesley, 2002.
[6] *CVS, Open Source Version Control*, http://www.nongnu.org/cvs/, ultimo acesso em 26 de novembro de 2007.
[7] *Subversion Home Page*, http://subversion.tigris.org/, último acesso em 24 de novembro de 2007.
[8] *TortoiseSVN Home Page*, http://tortoisesvn.tigris.org/, ultimo acesso em 25 de novembro de 2007.
[9] LEON, A. *Software Configuration Management Handbook, Second Edition*. Artech House Publishers, 2004.