UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
PROGRAMA DE PÓS-GRADUAÇÃO EM COMPUTAÇÃO

FABRÍCIO ÁVILA DA SILVA

Um Modelo de Dados Temporal Orientado a Objetos para Gerenciar Configurações de Software

Dissertação apresentada como requisito parcial para a obtenção do grau de Mestre em Ciência da Computação

Profa. Dra. Nina Edelweiss
Orientadora

Prof. Dr. Clesio Saraiva dos Santos
Co-orientador

Porto Alegre, janeiro de 2005

CIP – CATALOGAÇÃO NA PUBLICAÇÃO

Silva, Fabrício Ávila da

Um Modelo de Dados Temporal Orientado a Objetos para Gerenciar Configurações de Software / Fabrício Ávila da Silva – Porto Alegre: Programa de Pós-Graduação em Computação, 2005.

99 f.: il.

Dissertação (mestrado) – Universidade Federal do Rio Grande do Sul. Programa de Pós-Graduação em Computação, Porto Alegre, BR-RS, 2005. Orientadora: Nina Edelweiss; Co-orientador: Clesio Saraiva dos Santos.

1. Gerência de Configuração de Software. 2. Orientação a Objeto. 3. Bancos de Dados Temporais. 4. Engenharia de Software. I. Edelweiss, Nina. II. Santos, Clesio Saraiva dos. III. Título.

" Embora ninguém possa voltar atrás e fazer um novo começo,
qualquer um pode começar agora e fazer um novo fim. "

Chico Xavier

AGRADECIMENTOS

Gostaria de começar os agradecimentos pela dupla de orientadores deste trabalho, em especial à maravilhosa professora Nina pela excelente orientação prestada nestes últimos anos. Dedico um muito obrigado pela compreensão diante dos meus desaparecimentos, ocasionados por meus compromissos profissionais.

Seguindo a linha dos orientadores, agradeço também ao professor Alfredo que orientou o meu trabalho de graduação na Universidade Federal de Pelotas. Seu apoio no momento da mudança para Porto Alegre foi fundamental para que eu atingisse meu objetivo na UFRGS com sucesso.

Agradeço a todas as pessoas que passaram pelo meu caminho dentro do II, entre funcionários, professores e amigos. Quero dedicar também um muito obrigado para meus colegas de grupo de pesquisa, especialmente a Carina, Renata, Vanessa e Eduardo, cuja contribuição científica para o meu trabalho foi praticamente nula – com exceção da Renata, com quem discuti vários conceitos de tempo e versão, os outros de fato devem ter uma vaga idéia dos princípios discutidos nestas páginas – mas seus companheirismo e amizade foram com certeza fundamentais para manter-me de alguma forma ligado à universidade e à comunidade científica, durante os meus longos períodos de ausência. Ainda entre colegas e amigos, agradeço ao meu orientando e colega profissional Robert pelo interesse na minha proposta e pelas várias discussões que tivemos durante a implementação do seu trabalho.

Finalmente agradeço também a toda minha família, começando especificamente pelo meu irmão e o tempo de convívio em Porto Alegre que aumentou bastante nossas afinidades. Assim como já fiz no meu trabalho de conclusão de graduação, dedico mais essa conquista aos meus pais – os grandes responsáveis por todos os sucessos obtidos por mim durante a minha vida. Sua dedicação e suporte tornam os meus desafios infinitamente mais fáceis de serem superados.

SUMÁRIO

LISTA DE ABREVIATURAS E SIGLAS

ASP	*Active Server Page*
BLOB	*Binary Large Object*
CI	*Configuration Item*
CoV	*Change-Oriented Versioning*
CVS	*Concurrent Versions System*
DDL	*Data Definition Language*
EPOS	*Expert System for Programming and System Development*
GCS	Gerência de Configuração de Software
HTML	*Hyper-Text Markup Language*
IC	Item de Configuração
OID	*Object Identifier*
OIDt	*Temporal Object Identifier*
OO	Orientação a objeto
OQL	*Object Query Language*
RCS	*Revision Control System*
SCM	*Software Configuration Management*
SCM_TOO	*Software Configurations Managed using a Temporal Object-Oriented data model*
SCM2	*Software Configurations Managed using a Temporal Object-Oriented data model*
SGBD	Sistema Gerenciador de Banco de Dados
SQL	*Structured Query Language*
TVM	*Temporal Versions Model*
TVQL	*Temporal Versioned Query Language*
VSS	*Visual SourceSafe*

LISTA DE FIGURAS

LISTA DE TABELAS

RESUMO

Gerência de Configuração de Software é a disciplina que define conceitos e métodos baseados nos quais engenheiros de software conseguem manter sob controle a evolução de complexos produtos de software. Todos os princípios básicos da GCS foram implementados com sucesso em diversas ferramentas comerciais, mas as abordagens mais avançadas propostas nos últimos anos ainda não são utilizadas em ambientes reais de desenvolvimento, principalmente pela alta complexidade destas propostas – impedindo a sua implementação e utilização de forma adequada.

Com o objetivo de introduzir uma solução simples e flexível para gerenciar configurações de software, este trabalho apresenta o SCM_TOO – *Software Configurations Managed using a Temporal Object-Oriented data model*, cuja principal característica é o uso da orientação a objeto para modelar o produto de software e as modificações aplicadas a ele durante o seu tempo de vida. Além de disponibilizar mecanismos específicos para manipular arquivos e usuários do sistema, o SCM_TOO aplica técnicas de bancos de dados temporais para suportar a evolução da aplicação e armazenar o histórico de modificações realizadas ao longo do tempo.

O modelo define também um mecanismo de versionamento baseado em mudanças, no qual cada alteração realizada no software é identificada como uma entidade lógica no repositório de dados e pode ser utilizada na geração de novas *baselines* do produto. Esta proposta complementa o tradicional versionamento baseado em estados e aumenta a eficiência e flexibilidade do modelo sem acrescentar uma complexidade desnecessária.

Palavras-chave: Gerência de Configuração de Software, Orientação a Objeto, Bancos de Dados Temporais, Engenharia de Software.

A Temporal Object-Oriented Data Model to Manage Software Configurations

ABSTRACT

Software Configuration Management is the discipline that defines concepts and methods based on which software engineers keep under control the evolution of complex software products. All the basic SCM principles have been successfully implemented in several commercial tools, but none of the advanced approaches proposed in the last years is currently used in real development environments, due to their high complexity – making the proper implementation and utilization almost impossible.

Aiming to introduce a simple, flexible and powerful solution to manage software configurations, this work presents SCM_TOO – Software Configurations Managed using a Temporal Object-Oriented data model, whose main characteristic is using the object-oriented paradigm to model the software product and the modifications applied to it during its lifetime. Along with specific mechanisms to handle files and system users, SCM_TOO applies temporal databases techniques to support the application evolution and store the modifications realized on it.

The model also defines a change-based versioning mechanism, in which every modification performed on the software is identified as a logical entity in the data repository and can be used to generate new product baselines. This approach complements the traditional state-based versioning and increases the model efficiency and flexibility without imposing unnecessary complexity.

Keywords: Software Configuration Management, Object-Oriented, Temporal Databases, Software Engineering.

1 INTRODUÇÃO

Um grande número de diferentes artefatos é criado e atualizado durante o desenvolvimento de um produto de software, desde documentos de requisitos e arquivos de código fonte correspondentes até casos de teste e relatórios de defeitos detectados. Uma agregação destes itens em um determinado instante de tempo, compondo um produto de software consistente, é denominada uma configuração de software. Gerência de Configuração de Software – GCS ou ainda SCM, do inglês *Software Configuration Management* – é a disciplina baseada na qual desenvolvedores podem manter sob controle a evolução de grandes e complexos sistemas de software [TIC 88], [CON 99], [DAR 91], [FRU 99], [EST 2000].

Apesar de compor uma história de vários sucessos na área da engenharia de software, a comunidade de GCS deparou-se com muitas dificuldades desde as primeiras ferramentas desenvolvidas até os mais avançados modelos propostos nos últimos anos. O produto de software propriamente dito nunca foi representado de forma a facilitar o acompanhamento e o gerenciamento das mudanças aplicadas a ele. Uma fraca abordagem para modelar o software, controlar as versões de seus itens de configuração e gerenciar suas modificações ao longo do tempo ainda são grandes problemas a serem resolvidos nos próximos anos.

SCM_TOO (ou SCM2) – *Software Configurations Managed using a Temporal Object-Oriented data model* – é um modelo de dados conceitual proposto como uma nova e flexível solução para gerenciar corretamente um produto de software e sua evolução. Fundamentado em tradicionais e consolidados conceitos previamente definidos pela engenharia de software, o modelo pode ser considerado uma eficiente ferramenta para suportar as atividades de desenvolvimento de sistemas. Além do versionamento, SCM_TOO apresenta funcionalidades temporais e a capacidade de gerenciar mudanças no software como unidades lógicas de desenvolvimento.

1.1 Motivação

O problema de acompanhar a evolução de um produto de software e construir configurações corretamente tem sido um tópico de discussão há vários anos na comunidade de engenharia de software [EST 2002]. Os primeiros sistemas gerenciadores de configurações, como RCS [TIC 85] ou seu sucessor CVS [BER 90], aplicam conceitos básicos de versionamento em arquivos simples. Ainda hoje, a maioria das ferramentas comerciais disponíveis somente captura os arquivos e diretórios que representam um produto de software, além de um pequeno número de atributos e dependências [EST 2002], [FRU 99]. Os arquivos são versionados individualmente,

tornando a construção de objetos complexos uma tarefa bastante complicada. Não é possível associar no repositório de dados os artefatos que definem os requisitos do software, como diagramas UML, com os arquivos de código que implementam fisicamente as funcionalidades do sistema, ou ainda com os defeitos detectados durante a fase de teste. Além disso, a maioria das técnicas utilizadas para a construção de configurações é inadequada, como por exemplo a seleção de determinadas versões de arquivos baseada em *tags* armazenados com eles no momento de sua criação.

Nos últimos anos foi demonstrado que um dos aspectos mais fracos das ferramentas de GCS é o modelo de dados utilizado por elas [EST 2000]. Isto acarreta diversos problemas, como por exemplo o fato de que configurações e até mesmo versões serem tratadas pelo sistema como entidades especiais, e não objetos de primeira classe. Deste modo, configurações e versões não podem participar diretamente de relacionamentos e a composição de objetos complexos torna-se uma tarefa difícil de ser alcançada.

Estes fatores incentivaram o desenvolvimento de diversos trabalhos sobre avançados modelos versionados de dados, incluindo [EST 94], [DIT 86], [BOU 88], [MUN 93], [LAM 91], [CON 97], [ZEL 97]. Alguns trabalhos chegaram a propor o versionamento de todas as entidades presentes no repositório, inclusive atributos, relacionamentos e configurações. Estas abordagens, entre outras, apresentaram soluções interessantes, mas de um ponto de vista mais prático elas se mostraram ineficientes ou complicadas demais, fornecendo mais poder do que realmente necessário [EST 2002]. A maioria destas propostas, como o modelo *Change-Oriented Versioning* (*CoV*) [MUN 93], introduziu diversas definições e funcionalidades completamente novas para engenheiros de software [CON 98]. Mesmo um especialista em gerência de configuração encontra muitas dificuldades para entender cada conceito e a forma de aplicação destas ferramentas, necessitando de um programa de treinamento específico e um período de adaptação. Além disso, não existe até o momento um SGBD comercial que suporte um destes modelos de dados avançados [EST 2002].

Trabalhos recentes concluíram que um modelo de dados não tão complexo deveria ser utilizado, suportando objetos, relacionamentos e atributos de diversos tipos, além de uma maneira uniforme de identificar os componentes e suas versões dentro do repositório [CON 99], [WEB 2001]. O paradigma de desenvolvimento de software baseado em componentes, ou CBSD (*Component-Based Software Development*), é um exemplo de modelo não suportado adequadamente pelas atuais ferramentas de GCS. Nele, os usuários enxergam cada componente como um item básico do software, que pode ser implementado por um grupo de arquivos. Gerenciamento de componentes complexos e relacionamentos entre eles é uma tarefa básica desempenhada por um sistema de GCS que venha a suportar CBSD [MEI 2001].

Outro problema reside na forma como o software é visto por todos os envolvidos no processo de desenvolvimento. Atualmente está muito claro que diferentes papéis em um projeto deveriam poder visualizar o produto de maneiras diferentes. Obviamente, um programador precisa trabalhar diretamente no código-fonte que compõe o software, mas por que o analista da aplicação deveria interagir com um punhado de arquivos distribuídos em vários diretórios quando toda a informação que ele precisa está associada a módulos e componentes? Não faz nenhuma diferença se estes módulos são implementados em cinco ou em uma centena de arquivos. Algumas ferramentas de GCS fornecem aos usuários visões específicas do produto de software – por exemplo, a área de trabalho dos desenvolvedores pode ser diferente da área de trabalho dos integrantes da equipe de teste. Mas a maioria delas ainda consiste na apresentação dos arquivos

diretamente, sem proporcionar um nível de abstração mais elevado quando o usuário assim necessitar. Esta discussão não é nova e não faz parte da disciplina de GCS diretamente, mas o desenvolvimento de um eficiente mecanismo para controlar a evolução de um software deve ser baseado numa abordagem adequada para representar o produto de software propriamente dito.

1.2 Objetivo do Trabalho

Um sistema de gerência de configuração de software desenvolvido atualmente deve disponibilizar aos seus usários um conjunto de funcionalidades avançadas para controlar a evolução de um produto de software, além dos princípios básicos de versionamento previamente desenvolvidos e implementados. O grande desafio à comunidade de GCS consiste na elaboração de uma ferramenta que atenda às atuais necessidades comercias de desenvolvimento de software, conservando as facilidades de uso encontradas nos consagrados ambientes de configuração.

Um dos grandes problemas dos ambientes atuais de GCS reside na tentativa de implementar funcionalidades avançadas utilizando o mesmo tipo de repositório de dados existente há duas décadas. A solução dos problemas descritos na seção anterior passa necessariamente pela criação de um novo modo para representar os elementos de um produto de software e a maneira como estes evoluem ao longo do tempo.

O objetivo deste trabalho é o desenvolvimento de um novo modelo de dados conceitual a ser utilizado na construção de um eficiente repositório de dados para suportar novas ferramentas de GCS. Suas principais características devem ser:

- utilização de funcionalidades temporais, facilitando o registro da evolução do sistema ao longo do tempo e a recuperação de informações históricas da aplicação;
- abordagem avançada na modelagem do produto de software que permita a criação de relacionamentos semânticos entre seus artefatos, como agregações e associações;
- representação de mudanças em um software como entidades de primeiro nível, permitindo sua participação em relacionamentos e facilitando a criação de *baselines*;
- aplicação do versionamento baseado em mudanças, permitindo que a evolução da aplicação seja vista e controlada com o foco nos requisitos implementados, e não em quais itens de configuração foram modificados individualmente;
- suporte ao versionamento baseado em estados tradicional (revisões e variantes utilizadas na evolução de elementos de software individualmente), evitando que o modelo de dados torne-se desnecessariamente complexo.

1.3 Organização do Texto

O primeiro capítulo apresentou uma rápida introdução ao tema, seguida da motivação correspondente e do objetivo específico do trabalho. O Capítulo 2 descreve os principais conceitos de gerência de configuração de software, focando nos aspectos

relacionados ao repositório de dados e nos conceitos de versionamento necessários para o correto entendimento do modelo aqui proposto. O terceiro capítulo faz uma breve descrição do TVM – *Temporal Versions Model*, cujas principais características são utilizadas neste trabalho. O quarto capítulo apresenta o SCM_TOO e sua proposta para gerenciar configurações de software, consistindo na principal contribuição deste texto. A seguir, um estudo de caso é descrito, no qual algumas modificações são realizadas em uma aplicação modelada utilizando o SCM_TOO. Finalmente, o sexto capítulo apresenta as considerações finais, identificando trabalhos futuros e as contribuições do SCM_TOO para a engenharia de software. O apêndice contém a definição de todas as classes do modelo e o seu diagrama de classes completo.

2 GERÊNCIA DE CONFIGURAÇÃO DE SOFTWARE

O termo engenharia pode ser definido como a disciplina que define maneiras para projetar, construir e trabalhar com um determinado produto. Existem hoje diversas engenharias, pois a construção de diferentes produtos exige técnicas e procedimentos específicos para cada um deles. No entanto, todas as engenharias compartilham algumas políticas e princípios comuns que podem ser aplicados a todos os produtos considerados.

A engenharia de software, porém, difere das demais engenharias em diversos e significantes aspectos, sendo o mais importante deles a possibilidade de mudança. Um software pode, e geralmente deve, sofrer alterações durante todo o seu ciclo de vida, ao contrário da maioria dos outros produtos. Um software pode ser alterado durante o processo de desenvolvimento, ou seja, antes da sua construção ser finalizada. O aspecto mais interessante, porém, é a possibilidade de modificar o software mesmo depois de ele estar pronto e ter sido entregue ao cliente. Este fato é o que torna o desenvolvimento de software uma prática tão interessante, pois podemos sempre entregar ao cliente exatamente o que ele deseja e como ele deseja.

Na prática, porém, a possibilidade de constante mudança de um produto de software acarreta em diversos problemas no processo de desenvolvimento, de forma que muitos métodos tradicionais aplicados às demais engenharias tornam-se inadequados às tarefas de construção e manutenção de um software. O desenvolvimento de um produto de software gera diversos artefatos até a entrega deste ao cliente. Estes artefatos podem ser documentos de requisitos redigidos pelo cliente, diagramas projetados por analistas, código fonte produzido por programadores, *scripts* de teste escritos por testadores de software, entre outros. Quando um requisito do produto é alterado pelo cliente, provavelmente diversos desses itens sofrerão alterações para que o produto seja modificado satisfatoriamente. Provavelmente várias pessoas estarão envolvidas nesse processo; por exemplo, um analista atualizará o modelo lógico do software e alguns programadores implementarão as modificações necessárias no código fonte do produto. A possibilidade de um único produto de software atender a diversos clientes e estes apresentarem diferentes e específicos requisitos ao mesmo tempo obrigam os desenvolvedores de software a trabalhar simultaneamente em diferentes versões do produto. Estes aspectos evidenciam a importância de gerenciar corretamente as mudanças de um software e a complexidade envolvida nesta tarefa.

O conjunto de todos os artefatos que compõem um produto de software em um instante de tempo específico determina uma configuração de software. Uma

configuração contém somente uma versão de cada artefato, de forma a compor um software completo e consistente. Com o passar dos anos foram atribuídas várias definições ao termo Gerência de Configuração de Software (GCS, ou SCM – *Software Configuration Management*). Algumas dessas definições são apresentadas aqui:

- gerência de configuração de software é a disciplina que permite aos engenheiros de software manter a evolução de produtos de software sob controle, contribuindo desta maneira para garantir a qualidade e as restrições de prazos definidas no planejamento do projeto [EST 2000];
- gerência de configuração de software é a disciplina de identificação de uma configuração de um sistema em pontos discretos no tempo, visando o controle sistemático de mudanças a essa configuração e mantendo a integridade e rastreabilidade desta configuração ao longo do ciclo de vida do software [BER 80];
- gerência de configuração de software é uma disciplina que aplica conceitos técnicos e administrativos, bem como de pesquisa, para identificar e documentar as características físicas e funcionais de um item de configuração, controlar mudanças a essas características, registrar e relatar o processamento da mudança e o status de sua implementação e verificar a conformidade desta com os requisitos [BOE 88].

Algumas definições explicitam processos de identificação e manuseio de itens de configuração, enquanto outras mantêm-se mais abstratas e tratam principalmente de aspectos conceituais. Simplificando de uma forma bastante genérica, a gerência de configuração de software é responsável pelo controle da evolução de sistemas complexos.

Segundo Estublier [EST 2000], um sistema de GCS atualmente procura fornecer serviços em três áreas distintas:

- gerenciamento de um repositório de componentes;
- controle das áreas de trabalho de cada engenheiro de software e a maneira como essas áreas são integradas ao repositório;
- suporte e controle do processo de desenvolvimento, definindo as etapas do ciclo de vida do software e impondo restrições e deveres em cada uma delas.

Como este trabalho está focado no modo de representar o software e a sua evolução ao longo do tempo, os conceitos relacionados ao repositório de componentes são os mais importantes. Alguns aspectos sobre áreas de trabalho e do processo de desenvolvimento serão também abordados por impactarem diretamente nas formas de versionamento e de evolução dos elementos componentes do software.

2.1 Conceitos de Gerência de Configuração de Software

Diversos trabalhos descrevem os conceitos fundamentais da gerência de configuração de software [EST 2000], [CON 98], [DAR 91], [LEO 2000], alguns de forma bastante abstrata, enquanto outros enfocam métodos de implementação e utilização em um ambiente de desenvolvimento profissional.

Nesta seção os principais conceitos envolvidos na disciplina de GCS serão descritos, independentemente de implementações e ferramentas comerciais disponíveis atualmente. Serão vistos os conceitos de item de configuração, a maneira como estes itens são individualmente versionados, conceitos pertinentes ao produto de software como um todo, como configurações e *baselines*, métodos de integração entre o repositório de dados e a área de trabalho de cada desenvolvedor, e a abordagem do versionamento baseado em mudanças.

2.1.1 Item de configuração

Como citado anteriormente, o processo de desenvolvimento de um software gera muitos elementos: documentos de requisitos, código fonte, casos de teste, etc. Estes elementos são chamados artefatos de software [LEO 2000].

Porém, não é necessário que todos os artefatos envolvidos na construção de um produto de software sejam controlados e gerenciados do ponto de vista das mudanças aplicadas a eles. Os artefatos cujas alterações devem ser controladas e gerenciadas durante o processo de desenvolvimento são chamados de Itens de Configuração (IC's, ou *Configuration Items – CI's*).

Um dos primeiros passos em um processo de desenvolvimento que aplica técnicas de GCS é identificar os artefatos do software considerados itens de configuração, exigindo que toda e qualquer alteração a esses artefatos ocorra de forma sistemática e controlada.

2.1.2 Versionamento de itens de configuração

Normalmente, um item de configuração passa por diversas alterações durante o desenvolvimento de um software. O histórico das modificações realizadas em um IC é armazenado mediante a representação de várias versões deste item. O item de configuração, porém, continua sendo somente uma entidade cuja evolução é descrita pelas suas versões.

Versões podem ser descritas de duas maneiras; o versionamento baseado em estados é a abordagem mais tradicional, a qual consiste em definir uma versão como um estado do item sob evolução. Em diferentes instantes de tempo, o IC assume diferentes estados e, portanto, possui várias versões. Por sua vez, o versionamento baseado em mudanças consiste em descrever cada versão em termos das modificações aplicadas a uma versão base pré-definida. O versionamento baseado em mudanças será discuto na seção 2.1.5, por tratar o versionamento em um outro nível de abstração e utilizar os conceitos de configuração e *baseline*.

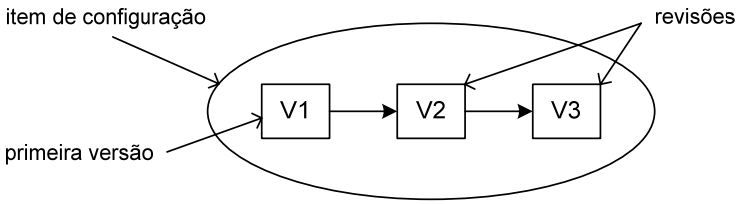


Figura 2.1: Item de configuração e suas revisões

Do ponto de vista da linha de evolução de um item de configuração, uma versão pode ser de dois tipos: revisão ou variante. Uma revisão é uma versão criada com o objetivo de substituir sua predecessora; ela apresenta funcionalidades adicionais, ou pode ter sido criada para corrigir defeitos encontrados na versão anterior. A Figura 2.1 apresenta um item de configuração que possui algumas revisões.

Uma variante, por sua vez, pode ser criada para endereçar alterações em uma das seguintes situações:

- para implementar uma solução alternativa ao mesmo problema tratado na linha principal de desenvolvimento;
- para desenvolver versões que suportem ambientes específicos de software e hardware;
- para permitir desenvolvimento paralelo, de forma que dois ou mais desenvolvedores possam trabalhar ao mesmo tempo em um mesmo item de configuração.

A Figura 2.2 apresenta um exemplo de variante em um IC. A criação de uma variante implica na derivação de uma linha alternativa de desenvolvimento a partir do tronco principal de revisões do item de configuração. O conjunto das versões desta linha de desenvolvimento chama-se ramo (ou *branch*). Um IC pode evoluir de maneira a criar diversos ramos a partir do tronco principal de versões, assim como cada ramo alternativo pode ser novamente dividido em duas ou mais linhas de desenvolvimento. A operação que unifica dois ou mais ramos chama-se de junção (ou *merge*).

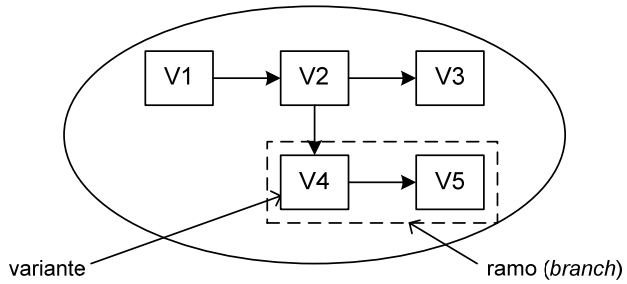


Figura 2.2: Uma variante e o ramo criado

As Figuras 2.1 e 2.2 são representações das versões de um IC em um grafo de versões, que consiste de nodos e arcos representando versões e os relacionamentos entre estas, respectivamente. Um arco partindo de um nodo V1 e chegando em um nodo V2 indica que a versão V2 é sucessora de V1 (sendo V1 a predecessora de V2), ou seja, V2 foi criada a partir da versão V1, ou simplesmente derivada de V1. Este tipo de grafo de versão representa a história da evolução de um item de configuração. Em um grafo acíclico dirigido, uma versão pode ter múltiplos predecessores, indicando que ela é o resultado da junção de duas ou mais versões (operação *merge*) e, portanto, contém características de ambas (ao menos conceitualmente). Um grafo do tipo árvore não permite que uma versão tenha mais de uma predecessora.

A Figura 2.3 contém a versão V6, criada pelo resultado da unificação do ramo alternativo ao tronco principal de desenvolvimento do IC através de uma operação *merge*. A unificação acontece quando o ramo alternativo ao tronco principal foi criado para permitir o trabalho de dois desenvolvedores que precisam modificar o mesmo item

ao mesmo tempo – um adicionando uma funcionalidade conforme previsto no planejamento do projeto e outro corrigindo um defeito apontado pela equipe de teste, por exemplo.

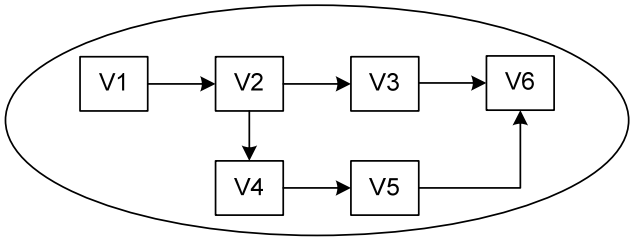


Figura 2.3: Operação *merge*

Além do grafo de versões, outro mecanismo para visualizar as versões de um IC é a construção de uma grade. Ela consiste em um espaço de várias dimensões, onde cada uma destas dimensões corresponde aos atributos variáveis do item de configuração. Revisões podem ser representadas na grade mediante a adição da dimensão tempo.

Conceitualmente, várias versões de um único item de configuração devem possuir alguma coisa em comum. Diferentes versões de um mesmo arquivo normalmente contêm várias linhas iguais. A diferença entre duas versões de um mesmo IC é denominada *delta*.

O tipo de delta a ser utilizado em um sistema de GCS é escolhido considerando requisitos nos níveis físico e lógico. Eficiência em tempo de execução e armazenamento adequado são os objetivos no nível físico, enquanto que no nível lógico deltas são usados para comparar versões e permitir que o usuário veja as diferenças entre elas em uma descrição de alto nível. Deltas podem ser dirigidos ou embutidos.

Utilizando deltas dirigidos, uma versão é construída mediante a aplicação de uma seqüência de alterações a uma versão base específica. A Figura 2.4 apresenta três versões de um componente e os deltas dirigidos correspondentes. O *delta1* contém a seqüência de alterações que, quando aplicadas à versão base, gera a *versão1* do item de software. Geralmente a versão base é definida como a que foi criada pelo usuário mais recentemente, pois esta muito provavelmente será requisitada com mais freqüência (esta prática melhora o desempenho do sistema). Deltas são aplicados, portanto, quando versões antigas são solicitadas pelo usuário.

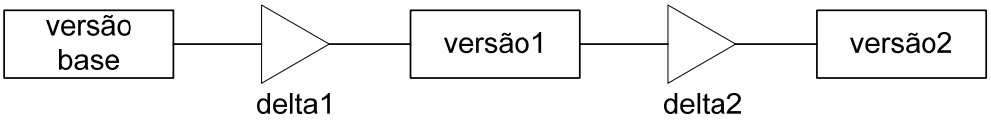


Figura 2.4: Deltas dirigidos

A utilização de deltas embutidos consiste no armazenamento de todas as versões de uma maneira sobreposta, de forma que fragmentos comuns sejam compartilhados. Cada versão aponta para seus fragmentos, ou os fragmentos são associados a expressões de controle que determinam as versões às quais eles pertencem.

Itens de configuração de software são, conceitualmente, entidades abstratas e que podem conter qualquer tipo de informação. Restrições quanto aos tipos de IC's que podem ser tratados em um projeto de GCS determinam o domínio do modelo de dados.

Um modelo é dito independente de domínio quando este suporta a criação e o gerenciamento de qualquer elemento, não importando o tipo deste. Todos os artefatos produzidos durante o ciclo de vida do software podem ser versionados e são tratados de maneira uniforme, seja uma seção de texto proveniente da documentação do projeto ou um arquivo componente do código fonte do programa. Por outro lado, um modelo é específico para um domínio se os tipos dos IC's manipulados são pré-definidos, não permitindo maior liberdade ao engenheiro modelador do sistema.

Relacionamentos entre IC's podem ser atribuídos e, geralmente, são definidos como composições ou dependências.

Composições são usadas para organizar elementos de software de acordo com sua granularidade. Os elementos são ditos complexos se compostos por outros elementos, caso contrário são denominados atômicos. Por exemplo, arquivos são elementos atômicos que podem formar uma entidade complexa, no caso um componente. Vários componentes, por sua vez, podem formar um módulo do software, de modo que a raiz desta hierarquia é o produto de software propriamente dito.

O modelo de dados pode distinguir explicitamente os itens atômicos dos compostos, ou simplesmente tratá-los de maneira uniforme e disponibilizar o relacionamento de composição para representar elementos complexos. Na representação tradicional de um produto de software através do seu código-fonte, arquivos são IC's atômicos e diretórios são utilizados para agregar estes itens, formando entidades complexas. Porém, tradicionalmente somente os arquivos são considerados itens de configuração e, portanto, versionados – os diretórios simplesmente constroem a hierarquia de representação do software e não podem ser versionados.

Dependências são conexões dirigidas entre quaisquer artefatos, sendo ortogonais às composições. A fonte e o alvo de uma dependência correspondem, respectivamente, aos elementos mestre e dependente. Semanticamente, uma dependência entre dois IC's determina que o conteúdo do dependente deve ser mantido consistente com o conteúdo do mestre. A implementação de um componente do software pode depender, por exemplo, de um conjunto de documentos e modelos que especificam os requisitos do componente.

2.1.3 O Produto de Software

Uma agregação de diversos elementos de software que, juntos, compõem um produto de software determinado, completo e consistente é denominada uma configuração de software.

Uma configuração é dita determinada pois deve ser composta por somente uma versão de cada IC, ou seja, não devem existir referências dinâmicas. Uma configuração é completa porque contém todos os itens relativos ao atual estágio de desenvolvimento do software – por exemplo, uma configuração construída no final da fase de desenvolvimento deve conter o plano de projeto escrito previamente. Finalmente, uma configuração é dita consistente pois os elementos que a compõem relacionam-se corretamente entre si – por exemplo, o plano de teste deve ser coerente com os documentos de requisitos incluídos na configuração.

A partir do conceito de configuração de software, algumas considerações quanto à granularidade do versionamento de IC's devem ser comentadas. Um sistema de gerência de configuração de software pode fornecer funcionalidades de versionamento em três

níveis diferentes: versionamento de componente, versionamento total ou versionamento de produto [CON 98].

No versionamento de componente, uma configuração é construída mediante a união de versões de vários componentes. O versionamento é aplicado a elementos de software atômicos, ou seja, itens complexos não são versionados diretamente. Configurações são representadas implicitamente, mediante o uso de um atributo específico ou de *tags* associadas a cada versão, de forma que configurações não são tratadas como entidades de primeira classe.

O versionamento total, por sua vez, generaliza o versionamento de componente de forma que todos os elementos são versionados e não somente as folhas da hierarquia de composição. Em contraste com versionamento de componente, versões de elementos complexos são representadas explicitamente como entidades; itens atômicos e compostos são tratados de maneira uniforme.

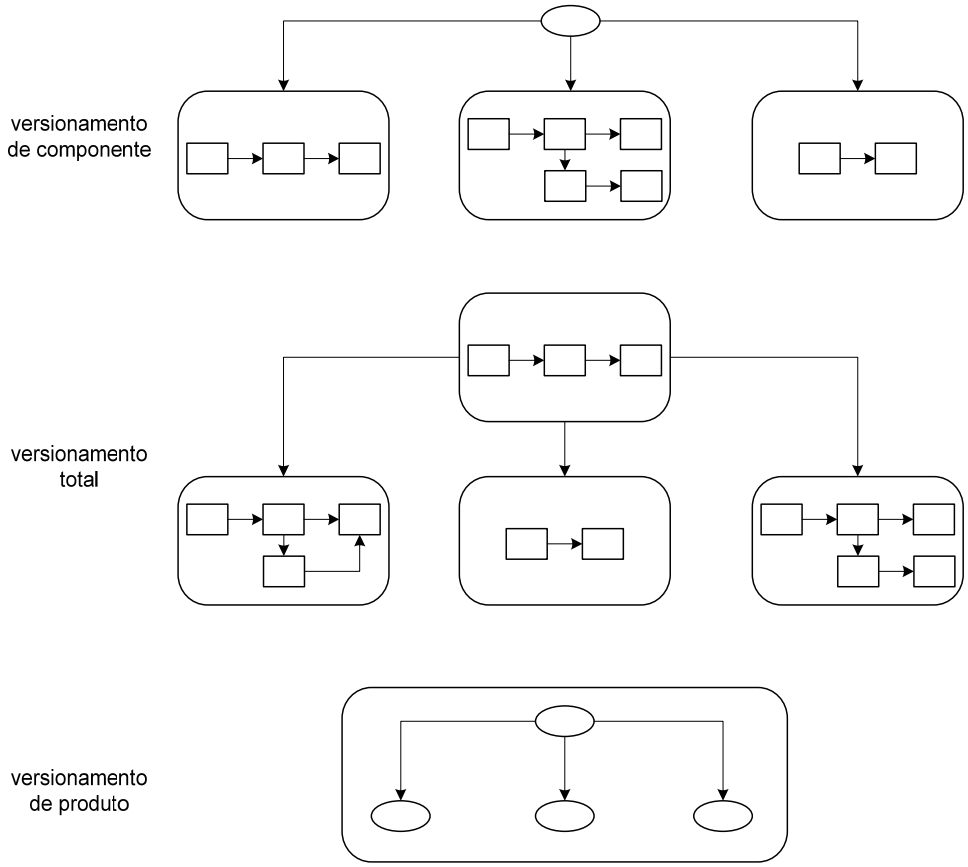


Figura 2.5: Classificação de versionamento quanto à granularidade

Versionamento de produto difere do versionamento total pelo agrupamento das versões de todos os objetos em um espaço de versão global. Esta abordagem consiste na criação de uma visão onde todos os objetos possuem somente uma versão cada, escondendo internamente versões de objetos e seus relacionamentos.

A Figura 2.5 ilustra graficamente os três tipos de versionamento classificados quanto à sua granularidade. Objetos em formas ovais são itens simples, não versionados. Retângulos com cantos arredondados são objetos versionados, cujas

versões são representadas por retângulos simples. Setas ligando versões representam derivações, enquanto que setas ligando objetos versionados e objetos não-versionados representam o relacionamento de composição.

Um estudo atualizado sobre gerência de configuração de software não pode deixar de definir também o conceito de *baseline*. Uma *baseline* é um tipo especial de configuração, a qual deve ter sido formalmente revisada e aprovada pelos responsáveis adequados no projeto de desenvolvimento (por exemplo, o gerente do projeto e o líder técnico). Uma *baseline* serve como base para as próximas atividades de desenvolvimento e somente pode ser alterada mediante procedimentos formais de controle de mudança.

Normalmente, uma *baseline* é estabelecida no final de cada fase do ciclo de desenvolvimento. Por exemplo, no fim do planejamento do projeto, uma *baseline* é criada com o objetivo de servir como base para a próxima fase, a de desenvolvimento.

Uma *baseline* também pode ser estabelecida no final do projeto, congelando os artefatos de software no momento da entrega do software ao cliente. Esta *baseline* contém exatamente quais itens de configurações (e, mais precisamente, a versão de cada item) foram entregues ao cliente. No caso de uma atividade de manutenção tornar-se necessária (e isso quase sempre ocorre), o time de desenvolvimento deve utilizar esta *baseline* como referência.

Uma vez que uma determinada versão de um item de configuração faça parte de uma *baseline*, ela não pode mais ser removida ou modificada. Quaisquer atualizações no item precisam ser formalmente aprovadas e uma nova versão do item deve ser criada para implementar estas modificações. Ou seja, uma *baseline* não pode ser apagada ou fisicamente modificada. Ela pode ser vista como um conjunto de itens congelados – novas versões derivadas destes itens podem receber alterações e, no futuro, fazer parte de uma nova *baseline*.

2.1.4 Repositório e Áreas de Trabalho

Todos os itens de configuração (e suas versões) gerenciados em um projeto de desenvolvimento de software são armazenados em um repositório. O acesso dos usuários ao repositório deve ser controlado, de forma que seja disponibilizada a versão correta de cada artefato no momento adequado. O repositório é compartilhado por todas as pessoas envolvidas no projeto, evitando que um mesmo item de configuração seja armazenado em dois locais diferentes ao mesmo tempo.

Porém, normalmente ninguém pode alterar um artefato de software diretamente no repositório. Cada engenheiro de software envolvido no projeto deve ter sua própria área de trabalho (ou *workspace*), onde os itens de configuração armazenados no repositório são duplicados para que a pessoa possa realizar tarefas comuns de desenvolvimento, como alterar documentos de requisitos ou compilar código-fonte.

O repositório armazena todas as versões de todos os itens de configurações que fazem ou já fizeram parte do produto de software sob desenvolvimento. Uma área de trabalho contém os artefatos de software necessários em determinado instante e tarefa de desenvolvimento, e somente uma versão de cada um destes artefatos. Assim, um desenvolvedor enxerga em sua área de trabalho o produto de software sem considerar os aspectos de versionamento gerenciados pelo repositório. Por exemplo, a pessoa responsável pela fase de teste do software pode manter em sua área de trabalho o plano

do projeto, documentos de requisitos, o plano de teste, *scripts* de teste e outros artefatos relacionados ao controle de qualidade, mas não precisa enxergar o código fonte do sistema (neste caso, considerando que apenas testes do tipo caixa preta serão desempenhados, onde o teste é aplicado do ponto de vista do usuário final). A Figura 2.6 ilustra um repositório e duas áreas de trabalho.

Várias formas de integrar o repositório e as áreas de trabalho foram propostas durante pesquisas e implementações de sistemas de gerência de configuração. O mecanismo mais utilizado é, sem dúvida, o modelo *check-out/check-in*. Este modelo consiste na aplicação de duas operações básicas para transferir elementos do repositório para as áreas de trabalho e vice-versa.

A operação *check-out* copia uma versão de um item de configuração do repositório para uma área de trabalho. O artefato criado na área de trabalho pode ser modificado ou não, dependendo de opções selecionadas no momento da execução do comando. A criação de versões somente-leitura pode receber outros nomes, dependendo da ferrramenta utilizada (*get_latest_versions*, por exemplo). No caso da criação de uma versão modificável, normalmente o elemento é marcado como reservado (*locked*) no repositório, prevenindo que outro desenvolvedor altere o artefato ao mesmo tempo (algumas ferramentas permitem o desenvolvimento simultâneo e disponibilizam operações de junção para unificar as modificações).

Após aplicar as modificações desejadas ao item de configuração, o desenvolvedor deve disponibilizar a nova versão do artefato aos outros engenheiros envolvidos no projeto e armazenar em um local seguro as alterações recém aplicadas. A operação *check-in* é responsável pela transferência da versão modificada na área de trabalho para o repositório. Algumas ferramentas criam uma nova versão do artefato já no momento do *check-out* (a qual é atualizada no *check-in*), outras geram a nova versão somente quando o desenvolvedor confirma o *check-in*. Independentemente da implementação, é no momento do *check-in* que as alterações realizadas pelo engenheiro serão replicadas no repositório e, portanto, disponibilizadas aos outros desenvolvedores, podendo ser incluídas em uma nova *baseline* do sistema.

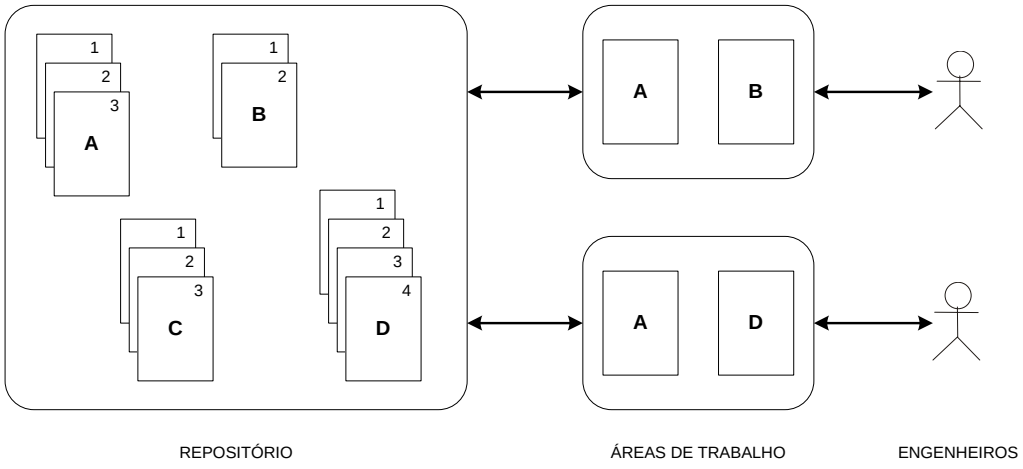


Figura 2.6: Repositório e áreas de trabalho

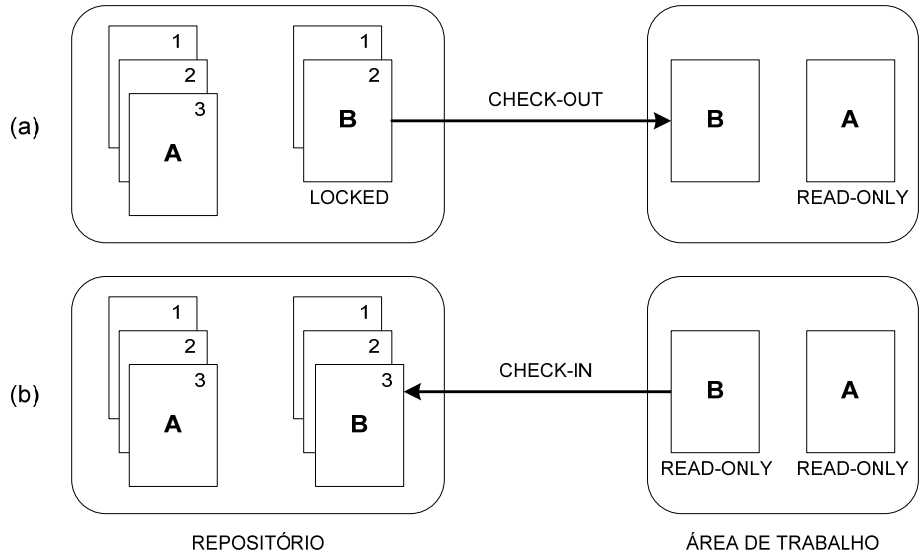


Figura 2.7: Operação *check-out* (a) e o posterior *check-in* (b)

A Figura 2.7 ilustra a utilização dos métodos *check-out/check-in* em um cenário de desenvolvimento.

2.1.5 Versionamento Baseado em Mudanças

Uma abordagem menos tradicional que o versionamento baseado em estados (descrito na seção 2.1.2) para representar diferentes versões de um IC é chamado de modelo baseado em mudanças, no qual uma versão é descrita em termos das modificações aplicadas a uma base previamente definida [MUN 93].

O versionamento baseado em mudanças utiliza o princípio de que em diversos tipos de aplicações pode ser mais natural enxergar as mudanças aplicadas em um ou mais elementos, no lugar dos elementos em si resultantes destas alterações. Uma mudança é, então, tratada como uma entidade lógica; uma versão de um grupo de elementos será representada por um conjunto de mudanças aplicadas a estes. O foco está nas alterações propriamente ditas, e os elementos resultantes destas alterações são considerados um subproduto do sistema. Cada mudança recebe um identificador (CID – Change Identifier, análogo ao OID da orientação a objeto), além de outros dados como razão, natureza e descrição da alteração.

A granularidade de um modelo de versionamento baseado em mudanças define se as alterações são aplicadas individualmente a elementos ou ao produto de software completo. No caso de versionar somente elementos atômicos, algum mecanismo de integração deve ser utilizado para associar uma mudança de um elemento a uma determinada mudança em outro elemento do software. A abordagem mais adequada considera que cada mudança registrada é aplicada ao produto de software como um todo; neste caso, uma versão completa do sistema em desenvolvimento é considerada como base (*baseline*) para que o versionamento possa ser aplicado.

As mudanças podem ser visualizadas de diversas maneiras, como em uma matriz de duas dimensões cujas linhas e colunas são versões e mudanças, respectivamente. A aplicação de uma alteração é indicada por um círculo em uma intersecção. No exemplo

da Figura 2.8, V1 é construída mediante a aplicação das mudanças M1, M2 e M3, nesta ordem.

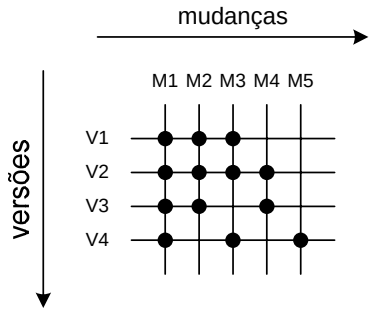


Figura 2.8: Representação em matriz

2.2 Implementações de Sistemas de Gerência de Configuração de Software

Nesta seção algumas ferramentas de GCS disponíveis serão descritas considerando o modo que estas representam o produto de software e os mecanismos de versionamento utilizados. Outros aspectos relacionados ao desenvolvimento, como suporte a trabalho cooperativo e gerenciamento de áreas de trabalho, estão fora do escopo desta análise.

Não é objetivo aqui enumerar todas as ferramentas disponíveis ou avaliar sua eficiência, mas somente ilustrar como os conceitos descritos na seção anterior foram implementados em ambientes práticos. As ferramentas foram escolhidas de forma que cada uma apresente uma característica diferente das outras, permitindo a visualização de várias possibilidades de implementações de sistemas gerenciadores de configuração de software.

2.2.1 RCS e CVS

RCS (*Revision Control System*) [TIC 85] foi um dos primeiros sistemas de GCS desenvolvidos. Na verdade, muitos autores não o consideram um gerenciador de configurações, e sim uma ferramenta para versionar arquivos, por não manusear explicitamente o produto de software e suas configurações.

De qualquer forma, o RCS foi o primeiro sistema a implementar muitos dos princípios de versionamento utilizados até hoje pelas mais modernas ferramentas. Deltas dirigidos são utilizados para armazenar diferenças entre versões. A última revisão do tronco principal de um arquivo é a única armazenada integralmente e versões anteriores são reconstruídas mediante a aplicação do conjunto de operações em cada delta. Desenvolvimento paralelo é permitido com o uso de ramos (*branches*) para cada desenvolvedor e da posterior aplicação do comando de junção (*merge*), o qual é semi-automático – no caso de modificações conflitantes, é solicitada a intervenção do usuário para decidir qual alteração deve fazer parte da versão final. O sistema implementa o modelo *check-out/check-in* para integrar o repositório com a área de trabalho de cada desenvolvedor.

O RCS, porém, não representa explicitamente configurações de software ou quaisquer relacionamentos entre arquivos. A construção de uma configuração é responsabilidade do usuário, que dispõe de um mecanismo bastante limitado para isso. Um rótulo (*label*, *tag* ou *symbolic name*) pode ser associado a cada versão no momento da sua criação. Quando acionar o comando *check-out*, o usuário seleciona as versões a serem copiadas para a sua área de trabalho com base nesses rótulos previamente armazenados. Por exemplo, pode-se considerar que toda versão cujo rótulo seja igual a "Conf_01" pertença à primeira configuração do sistema. Na prática, porém, o RCS não toma conhecimento da existência desta configuração e a sua manutenção (adição, remoção e alteração de arquivos) é um processo totalmente manual e dependente dos usuários; portanto, suscetível de erros.

O sucessor do RCS foi denominado CVS (*Concurrent Versions System*) [BER 90], que apresenta uma importante inovação em relação ao seu antecessor. O sistema permite o desenvolvimento concorrente sem a criação de ramos, baseando-se no princípio *copy-modify-merge*. O comando *check-out* não bloqueia a versão solicitada no repositório, de forma que dois ou mais desenvolvedores possam trabalhar em uma versão de um arquivo ao mesmo tempo. As versões criadas são automaticamente unificadas pela operação de junção no momento do *check-in*.

Tanto o RCS quanto o CVS apresentam os mais clássicos problemas dos modelos de dados utilizados na gerência de configuração de software: falta de suporte para a manutenção de elementos de software compostos ou total inexistência dos mesmos, fracos mecanismos de representação e manuseio de configurações e dificuldade na inserção de semântica aos itens de configuração e seus relacionamentos uns com os outros. As próximas ferramentas apresentadas tentam resolver estes problemas usando diferentes abordagens.

2.2.2 Visual SourceSafe

Microsoft Visual SourceSafe (ou simplesmente VSS) [MIC 2004] é uma ferramenta bastante parecida com os tradicionais RCS e CVS, pois apenas versiona arquivos individualmente e não existem conceitos avançados como configuração e *baseline*. A principal inovação do VSS é a definição de projetos como entidades de primeiro nível, ou seja, um repositório pode conter vários projetos e o usuário pode associar informações a cada um destes. Um projeto pode ser criado baseado em outro já existente, possibilitando a simulação do versionamento de produto de software como um todo. Porém, a ferramenta não vai muito além das funcionalidades básicas de versionamento descritas no item anterior e, portanto, apresenta os mesmos problemas na criação e na manutenção de configurações de software.

2.2.3 ClearCase

Várias funcionalidades fazem da *Rational ClearCase* [RAT 2004] uma das mais bem-sucedidas ferramentas de GCS disponíveis no mercado, algumas delas relacionadas ao desenvolvimento paralelo utilizando repositórios distribuídos. Uma importante inovação do *ClearCase* é a utilização de um sistema de arquivos virtual na representação da área de trabalho de cada usuário, gerando a ilusão de que o desenvolvedor trabalha diretamente em um ambiente não versionado.

Em relação ao modelo de dados, as principais contribuições do *ClearCase* são a aplicação de versionamento aos diretórios (além de arquivos individualmente) e do tratamento de ramos (*branches*) de desenvolvimento como entidades explícitas. O versionamento de diretórios faz do *ClearCase* uma das poucas ferramentas de GCS que permitem o versionamento de elementos de software complexos (diretórios compostos por diversos arquivos e até mesmo outros diretórios). Porém, a ferramenta não permite a inserção de semântica a estes elementos de forma adequada, gerando os mesmos problemas descritos anteriormente. A abordagem do *ClearCase* quanto aos ramos de desenvolvimento permite que um desenvolvedor (ou uma equipe, por exemplo) seja definido como proprietário de um ramo, proibindo que outros engenheiros modifiquem ou até mesmo enxerguem as revisões adicionadas a este ramo, até que estas sejam propagadas ao tronco principal do elemento mediante a operação de junção.

2.2.4 AllFusion Harvest

AllFusion Harvest (anteriormente chamado CCC Harvest), da *Computer Associates* [COM 2004], é uma ferramenta genuinamente orientada ao processo de desenvolvimento. O ciclo de vida do produto de software é definido no início do projeto, determinando todos os estados pelos quais cada item de configuração deve passar e quais as tarefas que podem ser executadas pelos engenheiros em cada um desses estados.

O produto de software é representado na maneira tradicional: arquivos são versionados e armazenados em uma estrutura de diretórios. Um aspecto interessante são as diferentes visões do software disponíveis, dependendo do estado acessado pelo usuário. Por exemplo, uma versão de um arquivo em desenvolvimento no estado *Development* não será enxergada pela equipe de teste que trabalha somente no estado denominado *Independent Test.*

A ferramenta propõe um mecanismo bastante interessante para suportar o desenvolvimento e a criação de configurações, que consiste na utilização de pacotes (*packages*). Toda versão de um arquivo, quando criada, deve ser associada a um pacote, o qual pode conter várias versões de diversos itens de configuração. Os pacotes locomovem-se pelos estados de desenvolvimento, carregando as respectivas versões e disponibilizando-as no momento correto aos engenheiros de software. O usuário pode criar configurações do produto de software selecionando uma *baseline* e os pacotes desejados, adicionando as versões corretas de cada arquivo de forma transparente.

Pacotes são utilizados, portanto, como encapsuladores de mudanças ou requisitos. Um pacote designado para implementar uma determinada mudança no produto de software percorre o ciclo de desenvolvimento agrupando todas as versões responsáveis pela implementação desta mudança. Caso surja algum defeito, o pacote pode ser deslocado novamente para uma das etapas iniciais de desenvolvimento, para que suas versões sejam corrigidas. Se o requisito for alterado ou simplesmente removido do escopo do projeto, o pacote pode ser transferido para um estado final definido como obsoleto, por exemplo – nesta situação, suas versões jamais serão utilizadas na criação de *baselines* nos estágios de desenvolvimento mais avançados.

Alguns aspectos, porém, devem ser melhorados nesta abordagem proposta pelo Harvest. Os pacotes são representados explicitamente pelo sistema, mas não existem formas de adicionar informações semânticas aos pacotes. Não é possível criar relacionamentos entre pacotes, para denotar dependências ou exclusões mútuas, por

exemplo. Não se pode utilizar um mesmo pacote para encapsular versões de código fonte do software e dos documentos de requisitos responsáveis por esta alteração, a não ser que arquivos de código e de requisitos compartilhem o mesmo ciclo de vida – o que na prática não acontece. A ferramenta disponibiliza um mecanismo muito precário para adicionar informações aos pacotes, associando a eles formulários HTML pré-definidos. Esta solução não é adequada simplesmente por não considerar as informações digitadas nos formulários no momento de promover pacotes e criar configurações.

Um aspecto positivo da ferramenta é a representação explícita de *baselines*. As versões utilizadas para criar uma *baseline* não podem ser alteradas ou removidas do projeto, conservando a propriedade de imutabilidade. O problema reside, novamente, na associação de informações a uma *baseline*. Um usuário não tem como saber quais pacotes foram utilizados para criar uma determinada *baseline* – nem mesmo a informação temporal relativa ao momento da sua criação é armazenada na *baseline*.

2.2.5 Adele

O sistema Adele [EST 94] é muito conhecido no meio acadêmico por ser uma das primeiras ferramentas a tratar o produto de software como um conjunto de objetos (atômicos ou complexos) e permitir relacionamentos entre eles. Adele usa conceitos próprios para a definição e versionamento dos artefatos de software: famílias (*families*), interfaces e realizações (*realizations*). Pode ser feita uma correspondência com os conceitos básicos de GCS descritos na seção anterior: uma família corresponde a um elemento versionado de software, o qual agrupa diversas versões; as interfaces são os ramos de implementação, ou variantes, que cada família pode ter; e finalmente, cada interface pode conter várias realizações, cujo significado é o mesmo das revisões criadas em cada ramo de desenvolvimento. Estes três conceitos são interligados por relacionamentos mantidos pela ferramenta.

Existem três tipos de versionamento no Adele: temporal, lógico e dinâmico. Versionamento temporal é utilizado para manter o histórico e o acompanhamento da evolução dos artefatos. Versionamento lógico corresponde às múltiplas variantes ou representações de uma mesmo entidade. Finalmente, o versionamento dinâmico é relacionado somente ao suporte de processos e transações aplicadas aos elementos de software.

As características mais interessantes do Adele são evidenciadas no momento que o usuário seleciona uma configuração de software para trabalhar, ou para montar uma *baseline*. Não são utilizados diretamente os identificadores de cada versão dos elementos, e sim regras de seleção construídas baseadas nos valores dos atributos armazenados em cada versão. As regras de seleção são definidas em dois grupos: a parte modelo (*model*) define a seleção de variantes, enquanto que a parte instância (*instance*) define a seleção de revisões. As regras podem ser de quatro tipos: imperativas, condicionais, exclusivas e padrão (*default*), definindo se uma regra deve ser considerada ou não em cada situação.

Adele evidencia os pontos negativos de todos os sistemas de GCS que tentam ir além do versionamento de arquivos e fornecer um modelo de dados mais completo ao usuário. A ferramenta é complexa demais e exige que os engenheiros envolvidos no projeto conheçam diversos conceitos próprios do fabricante, sendo estes inexistentes no estado da arte da disciplina ou em outras ferramentas de GCS. O custo de treinamento e preparação de uma equipe para utilizar a ferramenta satisfatoriamente é muito alto,

impossibilitando o seu aproveitamento adequado em ambientes reais de desenvolvimento de software.

2.2.6 EPOS e o modelo CoV (Change-Oriented Versioning)

O projeto EPOS ("*Expert System for Programming and System Development*") [MUN 93] tinha como principal objetivo o desenvolvimento dos principais componentes de um ambiente de engenharia de software, incluindo conceitos e mecanismos de gerência de configuração de software. Para atingir este objetivo, o grupo desenvolveu e implementou um mecanismo de versionamento próprio, chamado versionamento orientado a mudança, ou *Change-Oriented Versioning* (CoV).

Ao contrário do versionamento tradicional baseado em estados, no qual um sistema armazena explicitamente as versões de cada elemento do software, CoV registra somente as mudanças lógicas do produto de software. As versões físicas do produto são obtidas mediante a aplicação de um grupo dessas mudanças. As mudanças são aplicadas ao banco de dados como um todo, e não a itens de software individualmente.

Assim como Adele, o modelo CoV introduz diversos conceitos próprios para implementar a sua abordagem de versionamento diferenciada. Por exemplo, filtros para leitura e escrita são chamados ambição (*ambition*) e escolha (*choice*). A ambição define as versões a serem afetadas por uma mudança, enquanto que a escolha determina as versões apresentadas ao usuário.

Uma opção (*option*) é uma variável lógica representando uma mudança, determinando se esta será ou não incluída na avaliação do sistema. Outro conceito introduzido é o de visibilidade (*visibility*), que consiste em uma expressão lógica anexa a cada fragmento no banco de dados. A visibilidade indica se o fragmento será incluído ou não em uma determinada visão da base de dados.

CoV implementa mecanismos interessantes para tornar possível o versionamento baseado em mudanças, fornecendo um alto nível de abstração do produto de software e suas possíveis configurações. O ponto negativo da abordagem é a complexidade do modelo, agregando diversos conceitos e métodos não usuais para os engenheiros de software envolvidos no processo de desenvolvimento.

3 MODELO TEMPORAL DE VERSÕES

Em [GOL 95] é proposto um modelo de dados orientado a objetos estendido para suportar o versionamento de suas instâncias. Vários conceitos, mecanismos e restrições são definidos para suportar a existência de objetos, versões e configurações. Como sua principal contribuição, este modelo trata de maneira uniforme objetos simples, objetos versionados, suas versões e as configurações criadas a partir da união de diversos objetos e versões. Outro aspecto muito importante deste modelo de versões é a existência de um segundo tipo de herança, a chamada herança por extensão, além da herança por refinamento utilizada nos modelos clássicos orientados a objetos.

O TVM (*Temporal Versions Model*, ou Modelo Temporal de Versões [MOR 2001a], [MOR 2001b], [MOR 2001c]) estende o modelo de versões proposto por Golendziner de forma a apresentar características temporais, além do versionamento. Tempo é associado aos objetos, e seus atributos e relacionamentos podem ser definidos como temporais. Todas as modificações realizadas nestes atributos e relacionamentos são, então, armazenadas, mantendo o histórico da evolução da base de dados ao longo do tempo. Diversos métodos e restrições foram adicionados ao modelo de versões original para manter a consistência do modelo e de suas instâncias.

Quando utilizados os recursos de temporalidade, uma alteração realizada em uma versão pode ser armazenada sem a criação de uma nova versão. A modelagem torna-se bem mais flexível, de forma que novas versões somente são criadas quando o conjunto de alterações sofridas por uma versão é realmente significativo do ponto de vista do projeto.

Vários modelos orientados a objetos versionados foram propostos ao longo do tempo, assim como modelos temporais, por exemplo [LAM 91] e [KAK 96], respectivamente. A união de ambos aspectos fornece ao usuário um modelo de dados mais completo e flexível, o qual pode ser usado em um sistema de GCS para representar um produto de software, as pessoas envolvidas no projeto e todos os artefatos de software, como documentação e requisitos do produto.

O Modelo Temporal de Versões é um modelo de dados genérico, utilizado para representar e manipular quaisquer tipos de elementos. Como ele compõe a base do modelo proposto neste trabalho, cujo objetivo é modelar um produto de software e suportar sua evolução, o TVM será descrito nos próximos ítens de acordo com os conceitos de modelos de dados e de versionamento definidos para sistemas gerenciadores de configuração de software.

3.1 Espaço do produto de software

O espaço do produto descreve o software sob desenvolvimento com sua estrutura e componentes sem levar em consideração se estes são versionados ou não [CON 98]. TVM pode ser considerado um modelo independente de domínio, pois suas características orientadas a objeto permitem a modelagem de qualquer tipo de artefato de software. O usuário pode, por exemplo, definir uma classe “pessoa” e várias subclasses correspondentes à função que cada indivíduo desempenha no desenvolvimento do software, como “programador” e “analista”. Qualquer tipo de classe pode ser criada, de forma que o produto de software e sua interação com o mundo real pode ser representado de diversas maneiras.

O produto de software pode ser visualizado como as instâncias das classes e seus relacionamentos, os quais podem ser de dois tipos: associação e agregação. O relacionamento de agregação permite a construção de objetos complexos, onde um objeto é composto por vários outros objetos. Dependências entre módulos do software (objetos, no TVM) são representados pelo uso de associações, cujos nomes e cardinalidades podem ser definidos pelo usuário para customizar a representação do software. O modelo permite que qualquer tipo de relacionamento possa ser criado, tornando-o muito mais flexível do que a maioria dos modelos utilizados nos sistemas de GCS tradicionais.

A Figura 3.1 ilustra um produto de software representado no TVM.

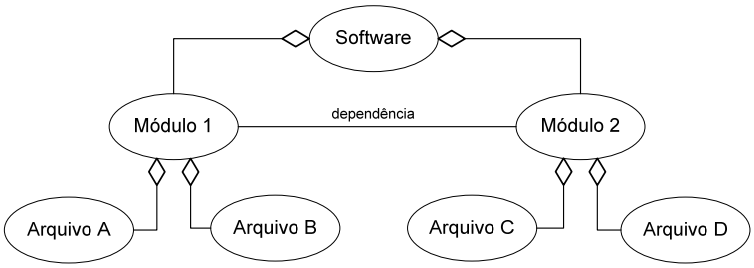


Figura 3.1: Produto de software no TVM

3.2 Espaço de versão

O modelo de versões define os objetos de software que podem ser versionados, a forma de implementação do versionamento e o modo como as versões são arranjadas.

TVM suporta versionamento baseado em estados, no qual cada versão representa um estado do objeto versionado. Por ser um modelo conceitual, ele não define como são representadas e armazenadas as diferenças entre duas versões (*deltas*).

O espaço de versão é definido por um grafo de versões, onde cada nodo é uma versão e os arcos representam o relacionamento de derivação, indicando que uma versão foi criada a partir de outra previamente armazenada. O grafo é do tipo acíclico direcionado (não podem existir ciclos e um nodo pode possuir mais de um descendente e mais de um ascendente), pois TVM suporta a operação *merge* (junção). Esta operação permite a criação de uma versão baseada em duas versões predecessoras, de forma a

reunir características de ambas versões originais. O modelo trata de maneira uniforme revisões e variantes.

Versões da mesma instância de uma classe são mantidas juntas em um objeto versionado, que mantém informações e propriedades comuns sobre as versões associadas. A criação de uma versão a partir de um objeto ainda sem versões torna este objeto a primeira versão de um objeto versionado, e a nova versão é derivada desta.

A Figura 3.2 apresenta um exemplo de instâncias possíveis em uma aplicação TVM. Há um objeto não-versionado e um versionado, cujas versões e relacionamentos de derivação também estão ilustrados.

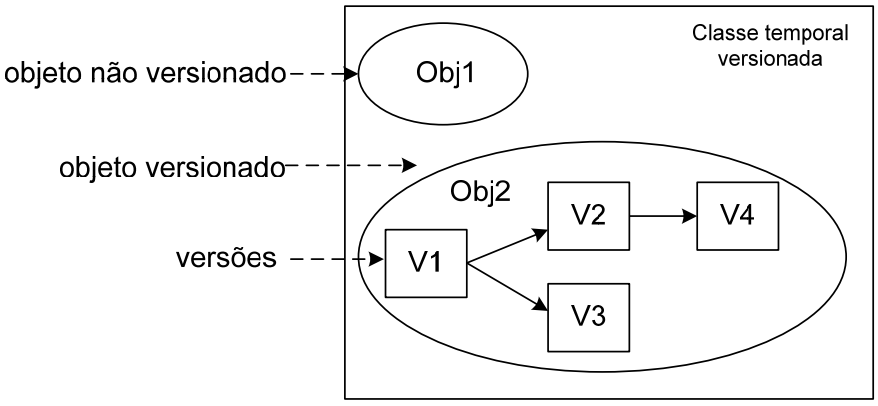


Figura 3.2: Instâncias em uma aplicação TVM

No TVM, uma versão sempre tem um determinado estado associado a ela. Operações de derivação, promoção e remoção são permitidas ou não de acordo com o estado no qual cada versão se encontra. Os estados que uma versão pode assumir durante o seu tempo de vida são: *working*, *stable*, *consolidated* e *deactivated*, (em trabalho, estável, consolidada e desativada). Transições entre eles e os respectivos eventos são apresentados em um diagrama de estados na Figura 3.3.

Alguns sistemas de GCS mantêm todas as versões imutáveis, ou seja, qualquer modificação em alguma propriedade de uma versão gera uma nova versão automaticamente. No TVM, imutabilidade é garantida por restrições associadas a cada um dos estados que uma versão pode assumir. Quando criada, uma versão assume o estado *working*, no qual ela pode ser modificada, consultada, removida, derivada e promovida. A promoção de uma versão eleva o seu estado, neste caso atribuindo-a o estado *stable*. Uma derivação cria uma nova versão no estado *working* e automaticamente promove sua predecessora para *stable*, evitando modificações que poderiam comprometer a história do objeto. No estado *stable*, a versão pode ser derivada, promovida para *consolidated*, consultada e removida, mas não pode ser modificada. No caso da operação de remoção, esta somente é permitida se não existem versões sucessoras, de modo a não prejudicar o armazenamento de toda a evolução do objeto. Uma vez no estado *consolidated*, a versão pode ser consultada e derivada, mas nunca modificada ou removida. Não existem remoções físicas previstas no TVM; a operação de remoção move a versão para o estado *deactivated* e encerra o seu tempo de vida. Neste caso, a versão somente pode ser consultada ou restaurada, retornando para o estado *working* ou *stable*.

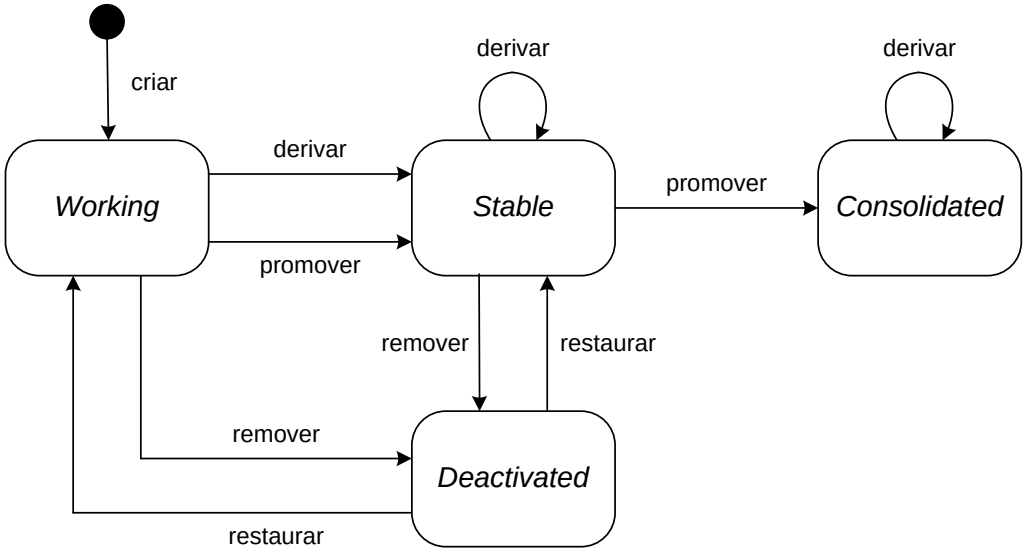


Figura 3.3: Diagrama de estados do TVM [MOR 2001a]

3.3 Interação entre espaço do produto e espaço de versão

No TVM, a integração entre o espaço do produto e o espaço de versão ocorre de maneira bastante natural, pois objetos não versionados e objetos versionados podem existir simultaneamente em um mesmo banco de dados. Para que uma classe possa conter objetos versionados e atributos e relacionamentos temporais, ela precisa ser declarada como subclasse de *TemporalVersion*. A Figura 3.4 apresenta a hierarquia de classes pré-definidas pelo TVM.

O modelo permite a definição de dois tipos de classes de aplicação:

- classe de aplicação não temporal e não versionável, definida como subclasse de *Object*. Utilizada para modelar classes nas quais os conceitos de tempo e versões não são necessários; além disso, permite a integração do TVM com outros modelos que não apresentam características de versionamento e temporalidade;
- classe de aplicação temporal versionada, definida como subclasse de *TemporalVersion*. Suas instâncias são versões (objetos com uma única versão são tratados como uma versão simples) e seus atributos e relacionamentos podem ser definidos como estáticos ou temporais.

Atributos e relacionamentos estáticos podem ser modificados livremente sem que estas modificações sejam armazenadas pelo sistema; neste caso, o versionamento do objeto determina quando seus valores podem ser alterados ou não, baseando-se no estado da versão. Atributos e relacionamentos temporais armazenam o histórico das modificações sofridas por uma versão durante o seu tempo de vida no estado *working*, antes desta ser derivada ou promovida. Os aspectos temporais serão discutidos em profundidade na seção 3.4.

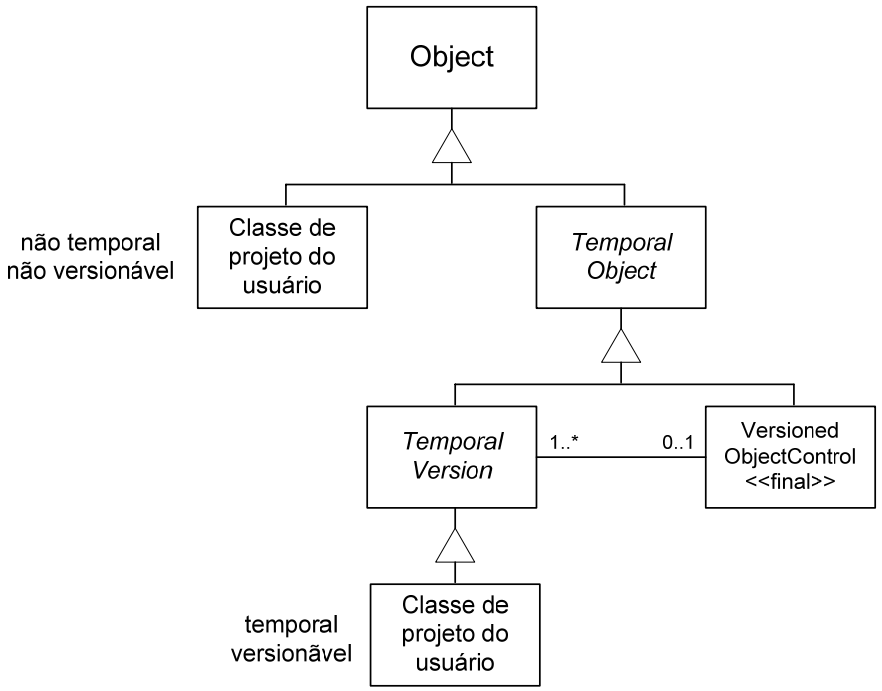


Figura 3.4: Hierarquia de classes do TVM [MOR 2001a]

O modelo identifica de maneira uniforme objetos não versionados, objetos versionados e versões propriamente ditas, de forma que qualquer um desses elementos possa ser diretamente manipulado ou consultado. O identificador de objeto, ou simplesmente OID (*object identifier*), é uma estrutura que contém a seguinte informação:

```
<Identificador de entidade, Identificador de classe, Número de versão>
```

Esta estrutura permite a existência de uma entidade composta por diversos objetos, bastando que estes tenham o mesmo identificador de entidade. O identificador de classe e o número de versão garantem a unicidade do OID do objeto.

A utilização de identificadores com a mesma estrutura tanto para objetos versionados como para não versionados permite que relacionamentos entre eles possam ser criados sem nenhuma restrição. Esta referência pode ser de dois tipos: estática ou dinâmica. Uma referência estática relaciona um objeto a uma versão diretamente, enquanto que uma referência dinâmica aponta para um objeto versionado; neste caso, a versão a ser utilizada no relacionamento é a versão mais atual, ou alguma outra especificada explicitamente pelo usuário.

TVM implementa versionamento total, permitindo que um objeto em qualquer nível da hierarquia de composição seja versionado, diferentemente dos sistemas tradicionais que utilizam versionamento de componentes, no qual somente objetos atômicos podem ser versionados.

Uma importante contribuição do TVM em relação aos modelos de dados utilizados em sistemas de GCS é a existência de dois tipos de herança: herança por refinamento e herança por extensão, apresentada em [BIL 90]. A herança por refinamento é o mecanismo tradicional encontrado nos modelos orientados a objetos e corresponde ao relacionamento *is-a*. A herança por extensão permite a descrição de uma

entidade em vários níveis da hierarquia, podendo um objeto em qualquer nível ser versionado ou não. A união dos objetos dos diversos níveis representam a entidade modelada completa.

A Figura 3.5 apresenta um exemplo que evidencia as diferenças entre os dois tipos de herança presentes no TVM. A herança por refinamento, ilustrada na Figura 3.5(a), consiste na instanciação do elemento folha da hierarquia da modelagem. Este objeto armazena as propriedades declaradas em sua própria classe, além dos atributos da(s) classe(s) ascendente(s). Na herança por extensão os valores dos atributos são armazenados no nível onde foram declarados e são compartilhados pelo(s) objeto(s) descendente(s). A Figura 3.5(b) apresenta um exemplo de herança por extensão, demonstrando que as propriedades são compartilhadas mediante o uso de um ponteiro presente no objeto folha da hierarquia.

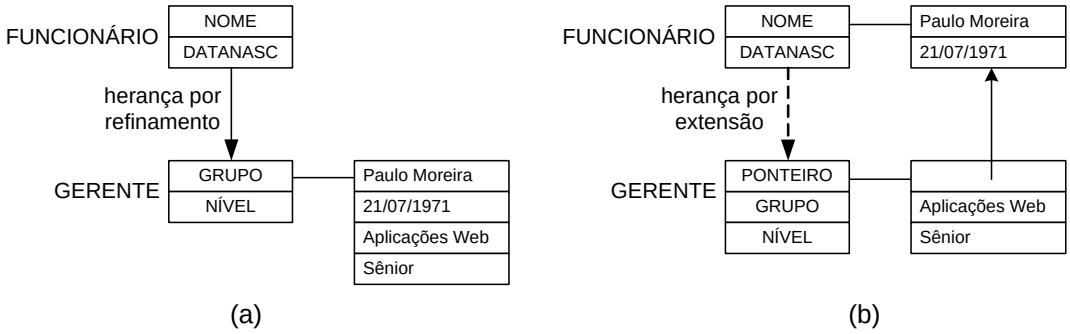


Figura 3.5: Herança por refinamento (a) e herança por extensão (b)

Uma das vantagens na utilização da herança por extensão consiste na modelagem progressiva de um objeto, acompanhando a sua evolução no mundo real. O exemplo aqui apresentado contém uma entidade “funcionário” especializada em uma classe “gerente”. Digamos que um novo empregado é contratado, neste caso de nome Paulo Moreira, e o sistema cria então um objeto a partir da classe “funcionário”. Depois de algum tempo, este empregado é promovido a gerente; no caso da utilização da herança por refinamento, deve ser criado um novo objeto da classe “gerente” e o antigo objeto do empregado deve ser descartado para evitar redundâncias. No caso da herança por extensão, basta criar um novo objeto da classe “gerente” e defini-lo como descendente do objeto empregado correspondente, pois assim os valores previamente armazenados são compartilhados com o novo objeto.

Uma configuração é um grupo de versões de diferentes objetos que juntas compõem um objeto complexo. Por exemplo, pode-se combinar uma versão específica de cada módulo de um software para formar um produto de software completo. No TVM, o usuário executa a operação *getConfiguration* em uma versão, chamada de versão base, para produzir uma versão configurada. Então, uma versão para cada ascendente na hierarquia de extensão deve ser selecionada (mediante o uso de expressões ou critérios pré-definidos), assim como uma versão para cada objeto na hierarquia de agregação. Para cada versão escolhida, uma nova é derivada, chamada de versão configurada, com seu próprio OID. Portanto, versões e configurações são tratadas de maneira uniforme no TVM, uma vez que uma configuração não é nada mais do que um grupo de versões conectadas por relacionamentos. O produto de software completo (ou componente, dependendo do nível de abstração a partir do qual a configuração foi criada) pode ser recuperado mediante a aplicação do método *getCompleteObject* na versão configurada.

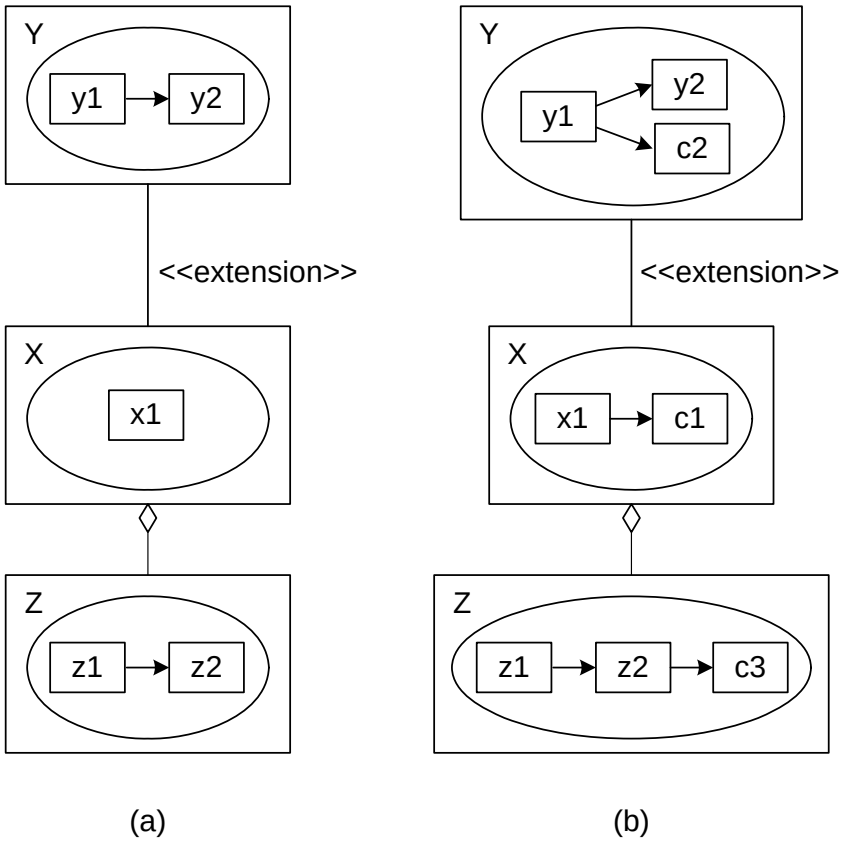


Figura 3.6: Criação de uma configuração

A Figura 3.6 apresenta um exemplo de uma configuração, antes (a) e depois (b) da sua criação. Supondo que o usuário solicite a construção de uma configuração usando a versão *x1*, uma instância da classe *X*, ele deve escolher uma versão da classe agregada, *Z*, e uma versão da classe ascendente na hierarquia de extensão, *Y*. Neste caso, *z2* e *y1* foram escolhidas e três novas versões, *c1*, *c2* e *c3*, foram derivadas das correspondentes selecionadas.

3.4 Características temporais

No TVM, o tempo pode ser associado a objetos, versões, atributos e relacionamentos, permitindo o armazenamento da evolução de todos os componentes de um sistema, sejam eles versionados ou não. Dentro de um objeto versionado, cada versão possui a sua própria linha de vida, de forma que um objeto pode ter várias versões válidas ao mesmo tempo – uma em cada ramo do objeto.

Um atributo ou relacionamento de uma classe *TemporalVersion* deve ser definido como estático ou temporal. Atributos e relacionamentos estáticos comportam-se da maneira tradicional, ou seja, qualquer alteração nos seus valores sobrepõe o valor anterior. Modificações somente são armazenadas mediante o uso de versões. A utilização de atributos e relacionamentos temporais permite que todas as alterações feitas em uma simples versão sejam armazenadas. Isto permite que todo o histórico de modificações sofridas por um objeto seja mantido, mesmo que este possua poucas ou somente uma versão. A definição dos atributos e relacionamentos como temporais ou

estáticos é feita pelo usuário durante a modelagem da base. Uma única classe pode conter atributos e relacionamentos de ambos tipos.

A variação temporal é discreta e a temporalidade é representada mediante o uso de intervalos temporais, que são grupos de instantes de tempo consecutivos e eqüidistantes. O modelo suporta duas dimensões de tempo, de forma que para cada dado temporal armazenado é associado um rótulo bitemporal que contém:

- tempo de validade: valor fornecido pelo usuário, informa quando o dado associado é verdadeiro na realidade modelada;
- tempo de transação: valor fornecido automaticamente pelo sistema, informa quando o dado associado foi inserido no banco de dados.

Regras de integridade temporal foram definidas, algumas com respeito ao tempo de vida dos objetos e suas versões e outras acerca das operações executadas sobre informações temporais (inserção e atualização).

As regras que garantem a integridade do tempo de vida dos objetos e de suas versões determinam, genericamente, que o rótulo de tempo associado a uma versão deve estar contido no tempo de vida do objeto versionado, bem como os rótulos de tempo associados aos atributos e relacionamentos temporais de uma versão devem estar contidos no tempo de vida da versão.

Quando ocorre a inserção de um dado temporal na base de dados, são associados a esta informação um tempo de transação (determinado pelo sistema) e um tempo de validade (informado pelo usuário). As regras de integridade temporal impedem que valores inválidos sejam armazenados como tempo de validade do atributo ou relacionamento, comparando os valores passados pelo usuário com os anteriormente armazenados e com o tempo de vida da versão.

O TVM não define regras para excluir fisicamente os objetos, pois o princípio de um banco de dados temporal é manter todo o histórico da aplicação. O modelo prevê somente exclusões lógicas, nas quais os tempos de validade e de transação em aberto de todos os atributos e relacionamentos temporais são fechados, recebendo o mesmo valor de tempo final definido para o objeto.

Uma discussão mais detalhada sobre aspectos temporais pode ser encontrada em [TAN 93] e [ZAN 97].

3.5 Linguagem de Consulta

Para extrair dados de uma base modelada no TVM, uma linguagem de consulta foi desenvolvida denominada TVQL – *Temporal Versioned Query Language*, ou Linguagem de Consulta Temporal Versionada [MOR 2002, ZAU 2002]. Ela é baseada na linguagem padrão de consulta SQL, de forma que elementos normais e temporais versionados sejam tratados da maneira mais homogênea possível. As características específicas de tempo e versão da linguagem serão apresentadas aqui.

Assim como na SQL, uma consulta TVQL é composta pela seguinte estrutura:

```
SELECT <lista de colunas, funções e predicados sobre colunas>
FROM <lista de tabelas>
[ WHERE <condições de seleção> ]
```

A sintaxe da TVQL, excluindo-se os elementos temporais versionados, é muito parecida com a da SQL. A cláusula DISTINCT pode ser usada para eliminar valores duplicados, operadores para comparação (<, <>, *in*, *between*, entre outros) e operadores aritméticos (+, –, *, /) são os mesmos, assim como as operações de conjunto (*union*, *intersection*, *difference*), funções de agregação (*count*, *sum*, *avg*, *min*, *max*) e outras cláusulas (*group by*, *having*, *order by*).

3.5.1 Consultando dados temporais

Uma consulta em TVQL cujo conteúdo não contenha nenhuma restrição ou cláusula referente a alguma característica temporal ou de versionamento dos dados retorna, por padrão, o estado atual das últimas versões dos objetos na base de dados, comportando-se, portanto, como uma consulta SQL tradicional.

Diversas combinações de valores históricos e atuais podem ser declaradas em uma consulta, dependendo do uso das palavras reservadas EVER e PRESENT. EVER, ou "sempre" em português, pode ser empregado após os termos SELECT e WHERE para que sejam considerados não somente os valores atuais, mas também toda a história dos atributos e relacionamentos armazenados. Utilizando-se SELECT EVER, a consulta retorna a história completa das propriedades temporais em questão. O uso de EVER na cláusula WHERE faz com que toda a história dos atributos citados seja considerada pela condição de busca. Outra possibilidade é recuperar valores históricos de acordo com uma condição atual, realizada mediante o emprego de EVER na cláusula SELECT e PRESENT após o termo WHERE.

Por exemplo, consideramos uma classe denominada *Componentes* cujo atributo *descrição* seja definido como temporal. A manutenção realizada em um componente durante o seu tempo de vida faz com que a sua descrição seja alterada diversas vezes, de modo a adequá-la às modificações sofridas. Para obter todo o histórico das alterações feitas no atributo *descrição* do componente de nome 'Grade', o usuário deve construir a seguinte consulta:

```
SELECT EVER descrição
FROM Componentes
WHERE PRESENT nome = 'Grade'
```

Nesta consulta, a palavra reservada EVER indica que todos os valores armazenados no atributo *descrição* devem ser retornados, enquanto que o termo PRESENT na cláusula WHERE determina ao sistema que somente os componentes que hoje tem nome igual a 'Grade' devem ser selecionados.

A TVQL também define rótulos temporais que podem ser utilizados tanto na condição de busca quanto nos valores a serem retornados pela consulta. Entre eles estão *tInterval* (intervalo de tempo de transação), *viInstant* (tempo de validade inicial) e *fLifetime* (tempo de vida final do objeto ou versão), por exemplo. O usuário também pode empregar a palavra reservada *Now*, que retorna o instante atual determinado pelo sistema.

3.5.2 Consultando versões

Os objetos e versões que devem ser consultados são definidos pelo usuário na cláusula FROM. Se ela contiver apenas o nome da classe, a consulta irá considerar apenas as versões correntes dos objetos.

Para obter todas as versões de um determinado objeto, a consulta deve incluir a palavra chave *versions* após o nome da classe. Por exemplo, considere a seguinte consulta:

```
SELECT comentários
FROM Arquivos.versions
WHERE nome = “system.pas”
```

Esta consulta retornará os comentários de todas as versões das instâncias da classe *Arquivos* cujo nome seja igual a *system.pas*.

A TVQL define diversas funções específicas para a recuperação das informações de versões e objetos versionados. Um grupo de funções recupera o estado da versão, como por exemplo as funções *isWorking* e *isStable* retornam um valor lógico (*true* ou *false*). Existem também funções utilizadas para navegar na hierarquia de derivação do objeto, como *isFirst, isSuccessorOf* e *isCurrent* (novamente, todas retornam um valor lógico).

Além das funções, também é disponibilizado ao usuário um conjunto de propriedades para que o controle do objeto versionado possa ser consultado de maneira transparente. Essas propriedades podem ser recuperadas na cláusula SELECT ou compor condições na WHERE. Por exemplo, a propriedade *currentVersion* retorna o identificador da versão corrente e *versionCount* o número de versões do objeto versionado em questão. As propriedades *nickname* e *entity* podem ser utilizadas para retornar o apelido do objeto e a entidade à qual ele pertence, respectivamente.

Como exemplo de uma consulta a um objeto versionado, considere o comando abaixo:

```
SELECT comentários
FROM Arquivos.versions
WHERE isWorking
```

Esta consulta considera os objetos da classe *Arquivos* e retorna os comentários de todas as versões que estiverem atualmente em trabalho, ou seja, as versões cujo estado seja igual a *working*.

Características temporais e versionadas podem ser combinadas na construção de consultas mais complexas.

4 MODELO TEMPORAL ORIENTADO A OBJETOS PARA GERENCIAR CONFIGURAÇÕES DE SOFTWARE

Como mencionado anteriormente, diversas características do Modelo Temporal de Versões seriam bastante úteis na utilização do mesmo em um ambiente de gerenciamento de configuração de software (por ambiente, entende-se um conjunto de ferramentas e funcionalidades que somadas ao modelo de dados compõem um sistema completo de desenvolvimento de software). Um estudo recente [SIL 2003a], [SIL 2003b] descreve o TVM de acordo com os conceitos de GCS e compara o modelo com diversas ferramentas gerenciadoras de configuração desenvolvidas nos últimos anos. Alguns pontos importantes deste trabalho serão apresentados na próxima seção.

4.1 Utilização do TVM em um Ambiente de GCS

Conradi desenvolveu um extenso estudo sobre modelos de versão implementados para gerenciar configurações de software [CON 98], desde o qual nenhum novo modelo foi proposto pela comunidade científica. Baseando-se neste estudo, o Modelo Temporal de Versões foi avaliado para determinar a possibilidade de sua utilização em um ambiente de GCS [SIL 2003a].

O TVM consolida todos os princípios básicos de versionamento propostos nas últimas décadas suportando a abordagem tradicional aplicada no controle e na evolução de um objeto, como o versionamento baseado em estados e o grafo de versões para representar revisões e variantes. A maioria das ferramentas de GCS implementam estes princípios para gerenciar elementos de software de tipos específicos, como arquivos, diretórios e componentes. Por sua vez, o TVM é um modelo de dados genérico, no qual é possível manipular qualquer tipo de objeto, seja um artefato de software ou um objeto do mundo real.

Todas as ferramentas de GCS conhecidas suportam o relacionamento de composição entre elementos de software [CON 98], mas muitas simplesmente desconsideram as dependências existentes entre eles. O TVM possibilita a criação de relacionamentos entre os objetos do sistema de forma explícita, permitindo a adição de informações semânticas a estas associações. Estas características fazem do TVM um modelo de dados mais completo e flexível do que a maioria dos modelos implementados pelas atuais ferramentas de GCS.

A existência de diferentes estados que uma versão pode assumir e a associação de operações e restrições a cada um destes estados representa uma importante inovação proposta pelo TVM, pois na maioria dos ambientes de GCS todos os itens são versionados e qualquer modificação no seu conteúdo implica na criação de uma nova versão no repositório de dados.

O controle da dimensão temporal em sistemas de GCS é geralmente subestimado pelos fabricantes e desenvolvedores [EST 94]. No TVM, a história completa das mudanças feitas em um atributo (ou relacionamento) pode ser armazenada se este for definido como temporal. A possibilidade da manutenção de toda a história de modificações sofridas por uma simples versão durante o seu tempo de vida é uma facilidade não oferecida por sistemas de gerência de configuração tradicionais. Além disso, o TVM automaticamente associa informações temporais a cada versão e objeto versionado criado.

O estudo desenvolvido conclui que o Modelo Temporal de Versões apresenta diversas características adequadas para modelar um produto de software, enquanto que permite também a identificação de novas funcionalidades que necessitam ser adicionados ao modelo. Os principais são:

- a utilização do TVM como repositório de dados de uma ferramenta de GCS precisa implementar mecanismos específicos para armazenar e manipular arquivos de código-fonte;
- o modelo deve dispor de classes e objetos pré-determinados para modelar as alterações realizadas no software e como estas são utilizadas para construir *baselines* da aplicação;
- métodos avançados de versionamento devem ser desenvolvidos para complementar o versionamento baseado em estados definido no TVM.

Além destas novas características, alguns aspectos do TVM precisam ser modificados para que o modelo torne-se mais adequado à modelagem de um produto de software. Esta necessidade motivou a criação de um novo modelo de dados concebido especificamente para gerenciar configurações de software, chamado SCM_TOO (ou SCM2) – *Software Configurations Managed using a Temporal Object-Oriented data model* ou, em português, configurações de software gerenciadas pelo uso de um modelo de dados temporal orientado a objeto. Modificações desenvolvidas sobre as características do TVM e novas funcionalidades propostas especificamente para a disciplina de GCS são discutidas nas próximas seções, enquanto que um estudo de caso detalhando a utilização do versionamento baseado em mudanças definido no SCM_TOO será apresentado no próximo capítulo.

4.2 Representação Gráfica

Classes são representadas graficamente utilizando as formas padrões da orientação a objeto. Não existe na literatura, porém, um formato padrão para representar objetos versionados e suas versões. A Figura 4.1 apresenta alguns objetos e versões nas formas gráficas utilizadas ao longo deste trabalho.

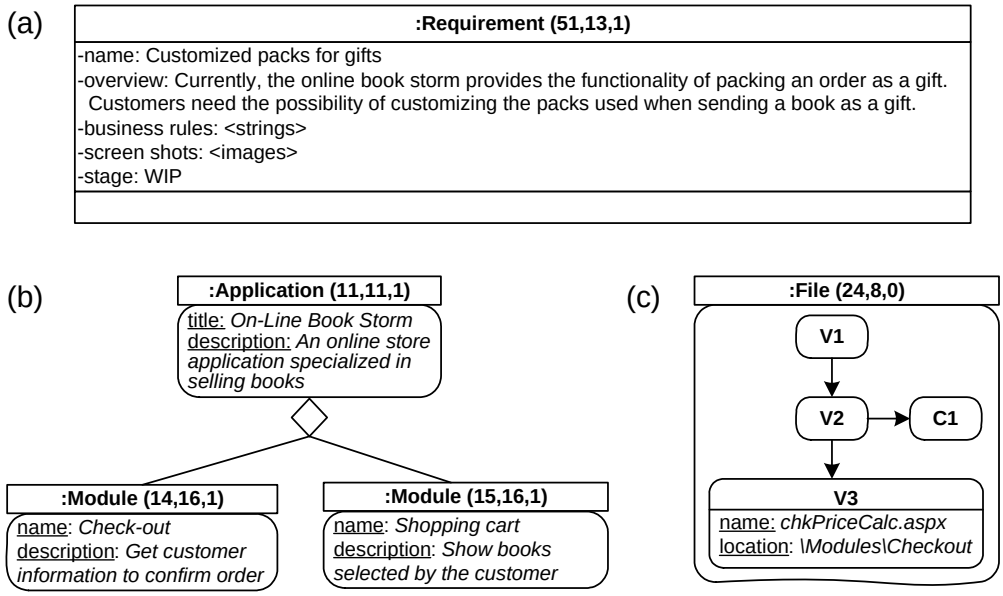


Figura 4.1: Representação gráfica de objetos, objetos versionados e versões

O objeto requisito (a), da classe *Requirement*, é um objeto não versionado e sua representação gráfica é a mesma usada na definição de classes, com duas diferenças: a barra de título apresenta o identificador do objeto (detalhado na seção 3.3 e, neste exemplo, igual a <51,13,1>) após o nome da classe, e o valor de cada atributo aparece no corpo do objeto. Esse modo de visualização também é utilizado em algumas figuras para representar objetos versionados que possuem somente uma versão. O item (b), por sua vez, apresenta alguns objetos versionados e relacionamentos entre eles, porém somente uma versão de cada objeto aparece neste tipo de representação. As informações são as mesmas disponibilizadas no item (a), mas o formato dos objetos é diferente para demonstar que estes são versionados e manter a uniformidade em relação aos outros objetos cujas versões são visualizadas, como o exemplo do item (c). Nele, todas as versões de um objeto da classe *File* são organizadas em um grafo de versões (explicado na seção 2.1.2), e somente V3 é detalhada por ser a versão mais atual do arquivo. O identificador mostrado no objeto corresponde ao objeto versionado e não a uma versão específica, por isso o número de versão contido nele é igual a zero.

4.3 Diagrama de Classes

A Figura 4.2 apresenta o diagrama de classes do SCM_TOO. As classes *Object*, *TemporalObject*, *TemporalVersion* e *VersionedObjectControl* foram previamente definidas no TVM e as suas poucas características alteradas serão descritas neste capítulo. As classes *Change*, *Baseline*, *User* e *File* foram introduzidas no SCM_TOO para modelar as funcionalidades da gerência de configuração de software e terão seus comportamentos descritos nas próximas seções. A definição completa de todas as classes é apresentada no apêndice deste trabalho.

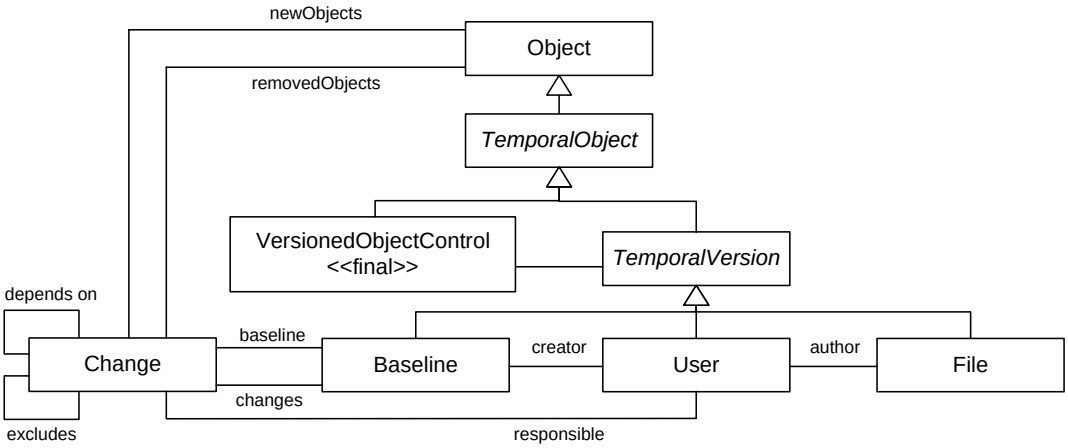


Figura 4.2: Diagrama de classes do SCM_TOO

O usuário do modelo pode refinar a classe *Object* e criar novas classes com o comportamento tradicional definido no paradigma orientado a objeto, enquanto que uma classe herdeira de *TemporalVersion* possuirá todas as características de temporalidade e versionamento definidas no modelo. As classes que implementam as características de GCS também podem ser refinadas caso o usuário queira adicionar atributos e relacionamentos específicos de seu projeto.

4.4 Temporalidade

Como explicado no capítulo anterior, o Modelo Temporal de Versões é dito bitemporal, pois para cada dado armazenado pelo modelo são associadas duas informações temporais:

- tempo de validade: valor fornecido pelo usuário, informa quando o dado associado é verdadeiro na realidade modelada;
- tempo de transação: valor fornecido automaticamente pelo sistema, informa quando o dado associado foi inserido no banco de dados.

Como exemplo, vamos considerar o registro de um funcionário e o seu salário em um sistema de folha de pagamento. A Tabela 4.1 apresenta uma atualização do salário do funcionário e os respectivos rótulos temporais associados a cada valor.

Tabela 4.1: Atualização de um atributo temporal

Funcionário	Salário	Tempo de Transação Inicial	Tempo de Transação Final	Tempo de Validade Inicial	Tempo de Validade Final
Paula Santos	R$ 2.300,00	20/12/2003	18/06/2004	01/01/2004	30/06/2004
Paula Santos	R$ 3.000,00	19/06/2004		01/07/2004	

No dia 20/12/2003 as informações sobre uma nova funcionária, Paula Santos, foram inseridas na base, definindo o tempo de transação inicial do registro. Ela começaria a trabalhar efetivamente somente no dia 01/01/2004, portanto esta data fornecida pelo usuário do sistema é armazenada como o tempo de validade inicial. No dia 19/06/2004, a informação de que Paula receberá um aumento é inserida no sistema.

O salário atual da funcionária tem seu tempo de transação encerrado, recebendo a data do dia anterior (18). O usuário do sistema registra a data de validade inicial do novo salário, neste caso 01/07/2004. Automaticamente, o valor anterior tem seu tempo de validade final registrado – 30/06/2004. Os tempos finais de transação e de validade do novo salário da funcionária permanecem indefinidos (diz-se *em aberto*) até o momento em que um novo valor seja inserido, ou que o tempo de validade seja explicitamente encerrado pelo usuário.

A bitemporalidade aplica-se perfeitamente neste exemplo pois a base de dados armazena informações sobre entidades do mundo real – uma funcionária e o seu salário. A diferenciação do momento em que um dado é armazenado no banco e o momento em que este dado é válido na realidade modelada é adequada para diversos tipos de aplicações e garante a correção da informação armazenada.

Em um sistema de gerência de configurações de software, contudo, os elementos armazenados na base não possuem correspondentes no mundo real. Por exemplo, se um documento de requisitos for removido do banco de dados onde o software está armazenado, o mesmo simplesmente deixa de existir. Da mesma forma, a modificação de um arquivo que contém o código-fonte de um programa atualizaria simultaneamente os tempos de transação e de validade associados ao arquivo, pois não existe nenhuma diferença entre o arquivo no mundo real e no banco de dados. Na verdade, poderíamos dizer que o “mundo real” do arquivo é o próprio sistema de computação onde ele está armazenado, pois sem esse sistema ele simplesmente não existiria. Assim, tempos de transação e de validade na modelagem de um produto de software são por definição iguais. Existem, contudo, algumas situações bastante específicas onde a bitemporalidade poderia ser útil. O modelo de dados aqui proposto é flexível o suficiente para que sejam representados todos os elementos componentes de um sistema de software e quaisquer outras entidades envolvidas no processo de desenvolvimento, como pessoas e empresas. Estas entidades poderiam valer-se da bitemporalidade para registrar com fidelidade as datas de modificações de seus atributos no banco de dados e no mundo real, que poderiam ser diferentes. Porém, o esforço dispensado para manter a bitemporalidade para todos os elementos modelados e usar seus benefícios em somente uma minoria de objetos não justifica a sua implementação. Da mesma forma, a possibilidade de separar objetos em temporais e bitemporais traria uma complexidade ao modelo de dados considerada desnecessária. Assim, o SCM_TOO utiliza somente o tempo de transação (fornecido pelo sistema operacional no momento de cada operação) para que não seja necessária a interação com o usuário toda vez que uma informação temporal for criada ou alterada.

4.4.1 Atributos e Relacionamentos Temporais

Classes e tipos de dados temporais originalmente definidos como bitemporais no TVM foram, então, modificados para considerar somente uma dimensão temporal. A Figura 4.3 apresenta os tipos de dados temporais primitivos utilizados para representar e manter atributos e relacionamentos temporais no SCM_TOO.

Cada objeto da classe base *TemporalLabel* armazena um rótulo temporal com os tempos inicial e final. Esta classe contém apenas as operações básicas: construtor, métodos para leitura dos dados e modificação do tempo final – o valor inicial é atribuído no momento da construção do objeto e não pode ser alterado.

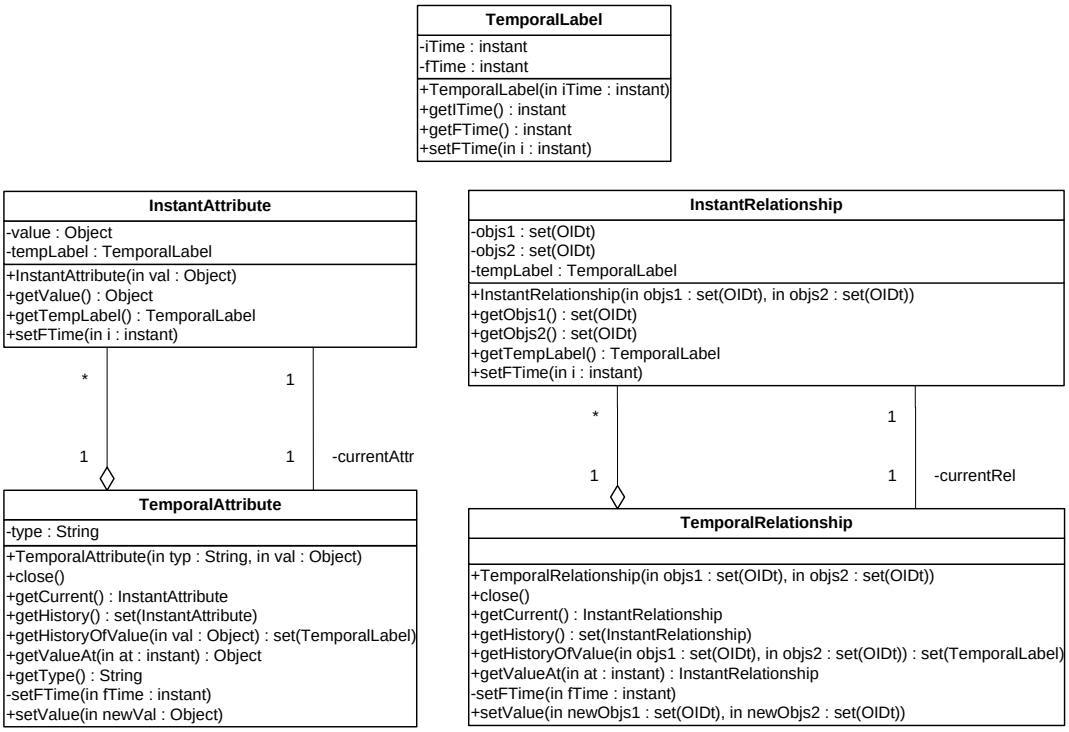


Figura 4.3: Atributos e relacionamentos temporais

Um atributo temporal (*TemporalAttribute*) é composto por uma agregação de objetos da classe *InstantAttribute*. Cada instância desta classe armazena um valor (um *Object*) associado a um rótulo temporal, contendo métodos para leitura do valor ou do tempo associado. Por sua vez, a classe *TemporalAttribute* implementa várias operações distintas, uma para a leitura do valor atual do atributo (utilizando o relacionamento *currentAttr*), outra que permite recuperar o história completa de modificações sofridas pelo atributo (*getHistory*), e ainda o valor armazenado em um instante de tempo específico (*getValueAt*). Também existe um método para recuperar os instantes durante os quais um determinado valor era válido (*getHistoryOfValue*), e outro que retorna o tipo do atributo. A operação *close* encerra o tempo de vida do atributo, definindo o tempo final do último valor armazenado igual ao instante de execução do método.

As classes *TemporalRelationship* e *InstantRelationship* comportam-se da mesma forma que as classes definidas para atributos temporais, com uma diferença: o valor de cada instante de um relacionamento temporal é definido por dois conjuntos de identificadores de objetos – ou seja, um conjunto de instâncias de uma classe relaciona-se com um conjunto de instâncias de outra classe. Dessa forma, um relacionamento temporal pode ser definido com qualquer cardinalidade (*1:1*, *1:n* ou *n:m*), o que representa uma correção em relação às classes originais do TVM, no qual relacionamentos temporais não suportavam cardinalidades múltiplas.

4.4.2 Regras de Integridade Temporal

Dois conjuntos de regras de integridade temporal são definidos originalmente no TVM para manter atributos e relacionamentos temporais consistentes. O primeiro especifica as integridades em relação aos tempos de vida dos objetos, versões e objetos

versionados. Estas regras foram adaptadas para atender somente uma dimensão temporal e são enumeradas abaixo:

1. o tempo de vida inicial de um atributo ou relacionamento temporal deve ser maior ou igual ao tempo de vida inicial da versão a qual pertence;
2. o tempo de vida final de uma versão deve ser maior que o seu tempo de vida inicial e maior ou igual ao tempo de vida final de todos os seus atributos e relacionamentos temporais;
3. o tempo de vida final de um objeto versionado deve ser maior que o seu tempo de vida inicial e maior ou igual ao tempo de vida final de todas as suas versões.

O segundo conjunto de regras de integridade temporal consiste na definição de como os valores temporais são fisicamente modificados quando operações de inserção e atualização são executadas. Considerando somente uma dimensão temporal, estas atualizações ocorrem de maneira bastante natural. SCM_TOO aplica as mesmas regras descritas no Modelo Temporal de Versões, bastanto desconsiderar o tempo de validade aplicado a cada alteração.

Assim como o TVM, SCM_TOO não define regras para exclusão física de nenhuma das instâncias criadas pelo modelo. A exclusão lógica de um objeto consiste no encerramento dos tempos de vida de todos os atributos e relacionamentos temporais em aberto, bem como o tempo de vida de todas as versões ativas e do objeto versionado. Toda versão logicamente excluída assume o estado *deactivated*, podendo mais tarde ser restaurada conforme seja necessário. Versões consolidadas (estado *consolidated*) não podem ser logicamente excluídas no SCM_TOO.

4.5 Atributos e Relacionamentos Comuns

Conceitualmente, o que faz com que um objeto seja uma versão de outro objeto? Versões de um mesmo objeto são assim definidas pois compartilham alguma característica ou propriedade. No caso de código-fonte de um software, por exemplo, algumas linhas de texto são iguais em diversas versões de um mesmo arquivo. As ferramentas de GCS tradicionalmente não implementam mecanismos específicos para aproveitar esta definição semântica de versionamento.

As implementações de sistemas de GCS desenvolveram, ao longo dos anos, diversas estratégias para reduzir o espaço em disco necessário para armazenar um item de configuração e todas as suas versões. Um delta dirigido é um conjunto de operações que, quando aplicadas a uma versão base, recupera uma versão anteriormente criada pelo usuário. Neste caso, não existe a representação explícita de quais propriedades são compartilhadas por duas versões de um mesmo elemento de software.

A utilização de deltas embutidos, por sua vez, consiste na fragmentação do conteúdo de um item de configuração e no armazenamento destes fragmentos individualmente. Uma versão é construída pela agregação de determinados fragmentos, selecionados mediante o uso de ponteiros ou de expressões de controle. Naturalmente, um fragmento será compartilhado por várias versões, garantindo que o espaço de armazenamento em disco seja bastante reduzido. Porém, o fato de que duas versões possuem um fragmento em comum é invisível para o usuário; ele enxerga versões que, na prática, podem ser completamente diferentes uma da outra.

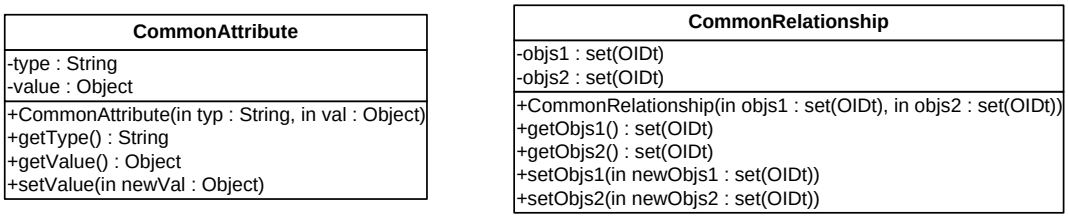


Figura 4.4: Atributos e relacionamentos comuns

O próprio Modelo Temporal de Versões implementa um objeto versionado, o qual é composto por todas as versões de um elemento modelado. Contudo, novamente o usuário não enxerga esse objeto agregador – ele existe somente com a função de gerenciar as versões e relacionamentos de um objeto versionado e é controlado automaticamente pelo sistema.

A fim de explorar o conceito de versão e fornecer ao usuário uma visão mais clara de um objeto versionado, SCM_TOO introduz atributos e relacionamentos comuns, os quais são compartilhados por todas as versões de um único objeto versionado. No momento da modelagem do produto de software, o engenheiro deve definir os atributos e relacionamentos de cada classe que serão compartilhados dentro de cada objeto. O acesso a esses atributos e relacionamentos ocorre como se eles fossem propriedades normais do objeto, ou seja, atributos tradicionais e atributos comuns dentro de uma versão são vistos pelo usuário de maneira uniforme. Quando uma atributo comum é modificado, contudo, o valor armazenado é automaticamente atualizado em todas as versões do objeto versionado.

A implementação de atributos e relacionamentos comuns no SCM_TOO é bastante simples, consistindo na utilização de duas classes: *CommonAttribute* e *CommonRelationship*, apresentadas na Figura 4.4. Assim como um relacionamento temporal, relacionamentos comuns são compostos por dois conjuntos de identificadores de objetos, permitindo cardinalidades múltiplas. Todas as versões de um mesmo objeto versionado possuem um relacionamento com cada um dos seus atributos e relacionamentos comuns, de forma que a modificação de um atributo comum em qualquer versão do objeto atualize diretamente o valor compartilhado por todas as instâncias.

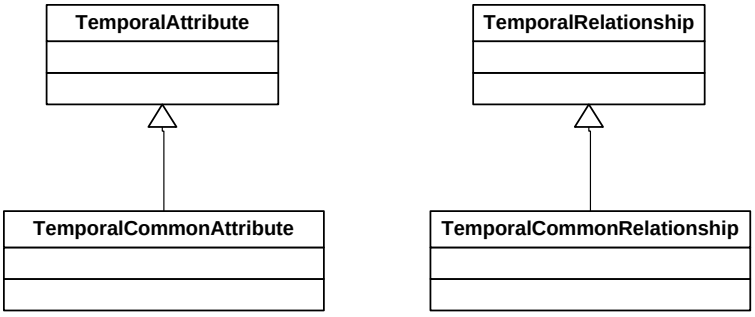


Figura 4.5: Atributos e relacionamentos comuns temporais

Naturalmente, um atributo ou relacionamento pode ser definido como temporal e comum simultaneamente. Para implementar esta funcionalidade, as classes *TemporalCommonAttribute* e *TemporalCommonRelationship* refinam as classes temporais, herdando características temporais e permitindo o compartilhamento dentro

de um objeto versionado. Estas classes são introduzidas na Figura 4.5, mas suas propriedades e métodos foram suprimidos pois são basicamente os mesmos apresentados nas figuras anteriores.

4.6 Manipulando Arquivos

Apesar da necessidade da existência de um modelo avançado para representar um produto de software e dos benefícios obtidos da orientação a objeto para esse fim, um software deve ser, irremediavelmente, implementado e composto por uma grande quantidade de arquivos. Qualquer tipo de informação pode ser representada por meio de atributos em uma classe, inclusive código-fonte. Porém, programadores de software precisam de arquivos disponíveis no sistema de arquivos de um computador para utilizar as ferramentas de desenvolvimento, como um ambiente de programação e um compilador.

Com a implementação do SCM_TOO em um banco de dados, os objetos armazenados podem ser alterados diretamente no SGBD, beneficiando-se das propriedades fornecidas pela ferramenta, como suporte a transações, concorrência e durabilidade. Os arquivos de código-fonte, por outro lado, precisam ser exportados para a máquina de cada programador e, posteriormente, importados para a base novamente.

O gerenciamento da área de trabalho (*workspace*) de cada desenvolvedor e a sua integração com o repositório de dados não está dentro do escopo deste trabalho, no qual discutimos somente o modelo de dados conceitual utilizado para representar a evolução de um produto de software. Contudo, um modelo de dados que simplesmente desconsidera como arquivos serão armazenados no repositório e que não forneça mecanismos para desenvolvedores modificarem esses arquivos seria simplesmente inútil em um ambiente prático de desenvolvimento.

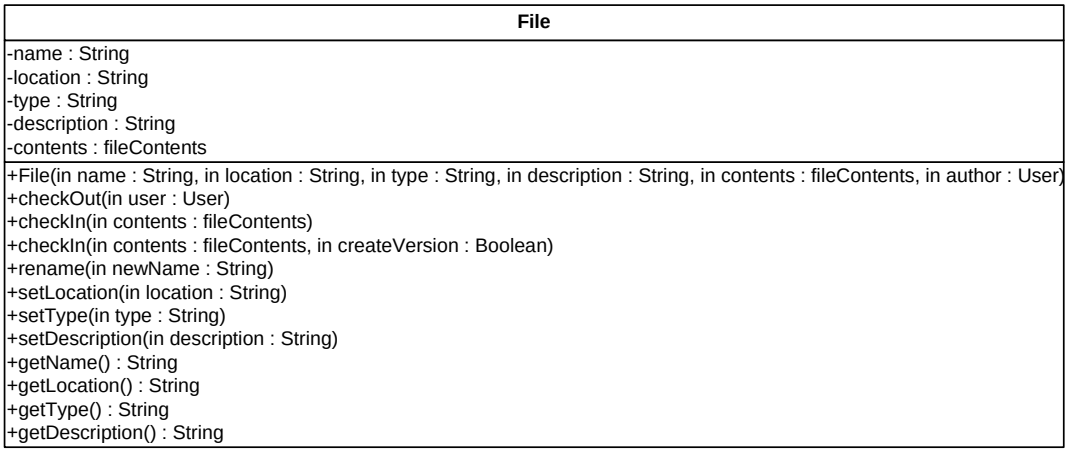


Figura 4.6: Definição da classe *File*

Para armazenar um arquivo e fornecer algumas funcionalidades específicas sobre o mesmo, SCM_TOO disponibiliza a classe *File*, apresentada na Figura 4.6. Durante a modelagem do sistema (antes de instanciar os objetos que compõem o software), o engenheiro de software pode utilizar diretamente a classe *File* para representar os arquivos do sistema ou refinar esta classe, de forma a adicionar atributos, métodos e relacionamentos adequados ao seu projeto de desenvolvimento.

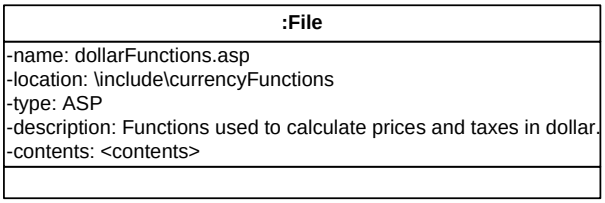


Figura 4.7: Objeto da classe *File*

A classe *File* contém cinco atributos; *name* armazena o nome do arquivo e *location* a estrutura de diretórios da aplicação onde o arquivo deve ser armazenado. O tipo do arquivo é atribuído à propriedade *type* e sua descrição aparece em *description*. O campo *contents* armazena o conteúdo do arquivo propriamente dito; o tipo de dado utilizado neste atributo depende do banco de dados no qual o modelo será implementado. Na maioria dos SGBDs, o tipo BLOB (*Binary Large Object*) seria usado, mas em alguns casos simplesmente um campo de texto longo pode ser suficiente, dependendo do tipo de aplicação. A Figura 4.7 apresenta um objeto da classe *File*, exemplificando o valor que suas propriedades podem assumir. O objeto é um arquivo do tipo ASP (*Active Server Page*) utilizado em uma aplicação web, cujo conteúdo é um conjunto de funções que calculam preços e impostos aplicados aos produtos disponíveis na loja virtual.

Um objeto da classe *File* comporta-se da mesma maneira que qualquer outro objeto modelado no SCM_TOO, com exceção das particularidades associadas ao atributo *contents*. Para controlar as alterações no conteúdo dos arquivos, SCM_TOO disponibiliza o tradicional e consagrado modelo *check-out/check-in*, descrito anteriormente na seção 2.1.4. Estes métodos interagem diretamente com a classe *User*, uma das classes pré-definidas no SCM_TOO e utilizada para representar os usuários do sistema. A definição desta classe é bastante simples e pode ser vista na Figura 4.8.

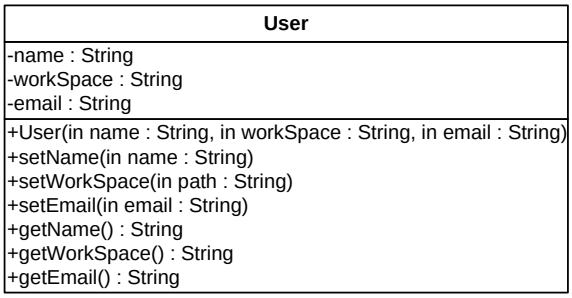


Figura 4.8: Definição da classe *User*

Um usuário possui, naturalmente, um nome e um endereço eletrônico como seus atributos básicos. A propriedade *workSpace* armazena o caminho da sua área de trabalho, por exemplo "*machine06**development**workspace*" (nome do computador na rede seguido do diretório compartilhado utilizado como área de desenvolvimento). A classe *User* disponibiliza operações de leitura e escrita de cada propriedade, além do método construtor.

Antes da modificação do conteúdo de um arquivo, o desenvolvedor aciona o método *check-out* do objeto *File* que deseja alterar. Esta operação recebe como parâmetro o usuário que vai trabalhar no arquivo, obtendo a sua área de trabalho. Neste

local, o método cria (ou sobrescreve, no caso do arquivo já exisitir no local) um arquivo utilizando o valor das propriedades *name*, *location* e *contents* do objeto em questão.

De acordo com o diagrama de estados originalmente proposto pelo TVM, um objeto pode sofrer alterações somente quando este se encontrar no estado *working* (seção 3.2). Para gerenciar melhorar as modificações realizadas em arquivos, SCM_TOO refina o diagrama de estados (Figura 4.9) alterando o nome deste estado para *available* e adicionando um novo estado, chamado *locked*. Todos os objetos na base de dados são alterados somente quando seu estado for *available*, inclusive as propriedades *name*, *location*, *type* e *description* dos objetos da classe *File*. O atributo *contents*, por sua vez, apresenta um comportamento diferenciado. Quando a operação *check-out* é executada e o arquivo criado na área de trabalho do desenvolvedor, o objeto assume o estado *locked*. Desta forma, nenhuma das suas propriedades pode ser alterada diretamente na base de dados, prevenindo que um usuário modifique o nome do arquivo enquanto um programador atualiza o seu conteúdo, por exemplo.

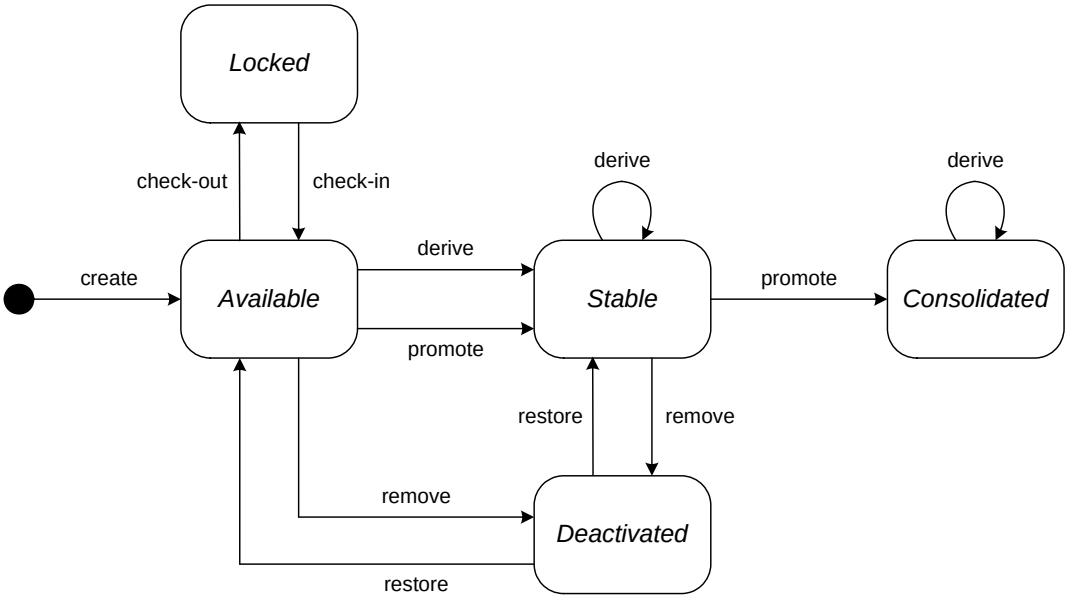


Figura 4.9: Diagrama de estados

Após a conclusão das alterações no arquivo, o desenvolvedor executa o método *check-in* e propaga as modificações realizadas na área de trabalho para a base de dados, atualizando o conteúdo do atributo *contents*. A versão do arquivo automaticamente assume o estado *available* novamente, tornando-se disponível para a todos os usuários envolvidos no desenvolvimento do software.

A operação *check-in*, por sua vez, apresenta algumas particularidades. Tradicionalmente, as ferramentas de GCS assumem que toda vez que esta operação for executada, uma nova versão do arquivo deve ser criada na base de dados para manter o histórico de alterações. SCM_TOO, contudo, apresenta um comportamento bastante flexível, permitindo que os usuários determinem quando uma nova versão de um arquivo deve ser criada ou não. O método *check-in* na classe *File* pode ser chamado recebendo como parâmetro somente o conteúdo do arquivo a ser atualizado. Neste caso, a versão do objeto somente muda de estado (retornando ao *available*), mas não é criada uma nova versão no banco de dados. Ela pode ser, mais tarde, explicitamente derivada ou promovida por um usuário do sistema, ou ainda modificada novamente. Por outro

lado, a operação *check-in* pode ser executada recebendo como parâmetro um segundo atributo, denominado *createVersion*. O valor passado é do tipo *boolean* e determina se uma nova versão deve ser criada automaticamente ou não. No caso de receber um valor *true*, o método executa a operação de derivação (*derive*), gerando uma nova versão do objeto antes de atualizar o seu conteúdo. Desta forma, a versão usada como base do trabalho do desenvolvedor é promovida para *stable* e a nova versão recém criada recebe as modificações implementadas pelo usuário na sua área de trabalho.

Além de modificar o conteúdo da versão de um arquivo, o método *check-in* também atualiza o seu relacionamento com a classe *User*, chamado *author*. Desta forma, o modelo armazena automaticamente o usuário responsável por cada versão da classe *File*.

O estudo de caso apresentado no próximo capítulo ilustrará melhor o processo de utilização dos métodos *check-out* e *check-in* para atualizar um arquivo na base de dados.

4.7 Configurações

SCM_TOO utiliza basicamente os mesmos conceitos e mecanismos aplicados a configurações definidos anteriormente no TVM. No contexto de GCS, uma configuração é a agregação de diversos elementos de software que, juntos, compõem um produto de software determinado, completo e consistente. Especificamente no SCM_TOO, uma configuração é um conjunto de versões de itens de configuração relacionados entre si, formando uma entidade de software completa.

Configurações são construídas mediante a execução do método *buildConfiguration* em um determinado objeto de software. Anteriormente denominado *getConfiguration* (no TVM), o nome da operação foi alterado para evitar confusões com os métodos de leitura de propriedades nas classes (*getDescription*, por exemplo). A operação *getConfiguration* tem uma finalidade diferente no SCM_TOO e será explicada a seguir.

A Figura 4.10 ilustra a geração de uma configuração no SCM_TOO, antes e depois da sua criação. No exemplo, são utilizados objetos das classes *Module*, *Component* e *File*. Um módulo é composto por vários arquivos (relacionamento de agregação entre as duas classes) e usa componentes na sua implementação (relacionamento de associação entre as classes *Module* e *Component*). A classe *File* é pré-definida pelo modelo (descrita na seção anterior) e as classes *Module* e *Component* são específicas deste exemplo e refinam a classe *TemporalVersion*.

O cenário inicial (a) apresenta um módulo de software chamado "*Check-out*", que implementa a parte de uma loja virtual onde o consumidor confirma a compra dos produtos previamente selecionados. O módulo é composto por três arquivos, cujas propriedades não aparecem para não sobrecarregar a figura. O componente "*Credit card verifier*" é utilizado pelo módulo "*Check-out*", de acordo com o relacionamento *uses* entre os dois objetos.

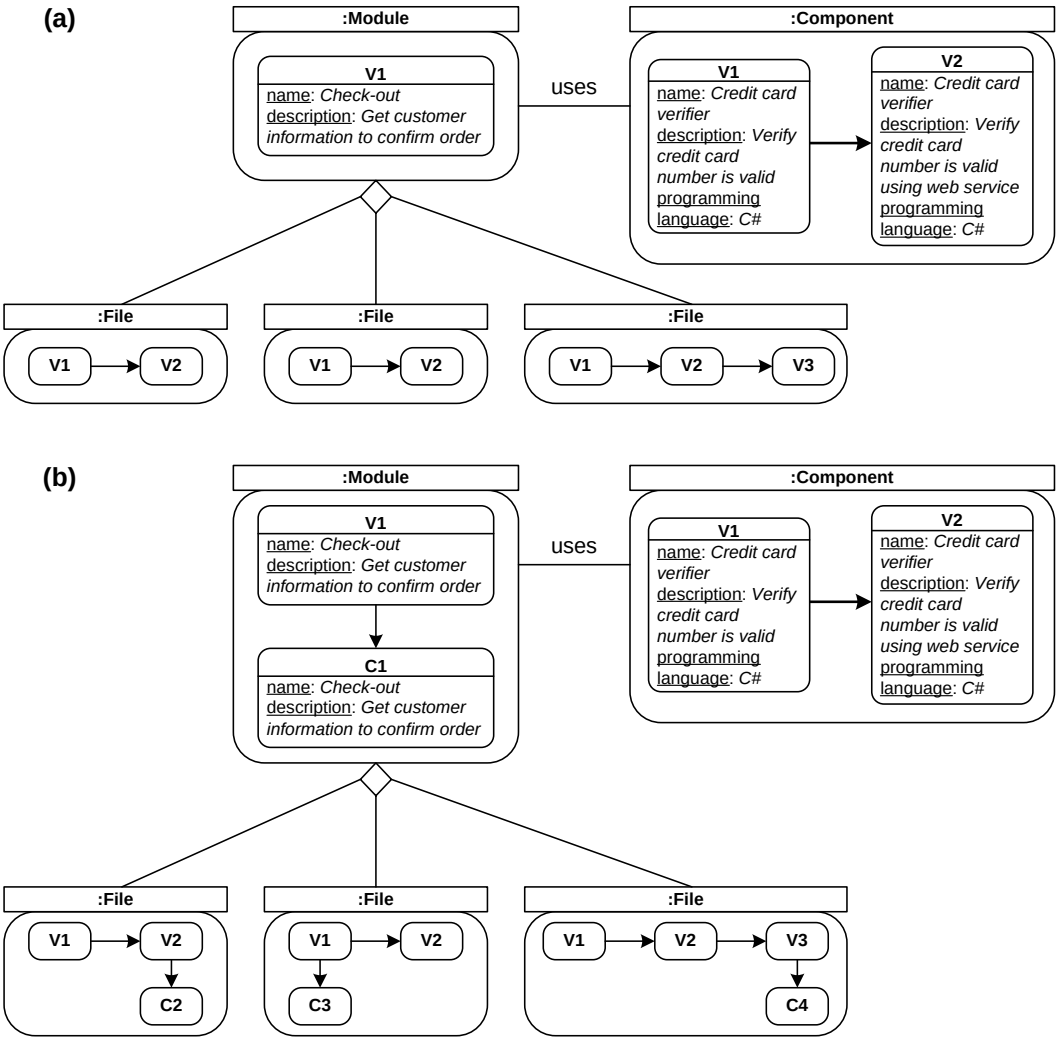


Figura 4.10: Cenário antes (a) e após (b) a construção de uma configuração

Digamos que o usuário execute o método *buildConfiguration* na primeira versão do módulo "*Check-out*", denominada versão base. Nesta situação, uma configuração do objeto módulo será construída, e não uma configuração do produto de software completo. Uma versão é derivada da versão base, assim como uma versão de cada um dos arquivos que compõem a entidade completa. A escolha de qual versão de cada objeto fará parte da configuração pode ocorrer de maneira automática, baseando-se em regras previamente definidas, ou interativamente pelo usuário do sistema. Na figura exemplo, foram criadas as versões *C1*, *C2*, *C3* e *C4*, as quais são denominadas versões configuradas por fazerem parte de uma configuração.

Versões configuradas são, estruturalmente, exatamente iguais às demais versões que compõem o produto de software. A capacidade de identificar uniformemente objetos não versionados, objetos versionados, versões e versões configuradas proporciona uma grande flexibilidade ao modelo, pois usuários podem consultar e manusear diferentes tipos de elementos de software de maneira simples e eficiente.

Uma configuração é criada como sendo uma nova entidade no produto de software, portanto todas as versões que compõem a configuração recebem um novo identificador de entidade. O identificador completo de uma versão configurada é

composto pelo identificador de classe e pelo número de versão da versão da qual ela foi derivada, mantendo os valores da versão original. O identificador de entidade, por sua vez, é igual ao valor atribuído como identificador da configuração. A Tabela 4.2 apresenta como exemplo os identificadores de algumas versões selecionadas para fazer parte de uma configuração e os identificadores das correspondentes versões configuradas criadas pelo sistema.

Tabela 4.2: Identificadores das versões e correspondentes versões configuradas

Versões Originais	Versões Configuradas
(11,07,02)	(21,07,02)
(11,09,01)	(21,09,01)
(11,09,05)	(21,09,05)

Versões configuradas diferem um pouco de versões regulares quanto ao seu comportamento em algumas situações específicas. Elas são criadas diretamente no estado *stable*, pois não podem ser modificadas pelos desenvolvedores. Além disso, uma versão configurada sempre será folha no grafo de versões de um objeto. Quando executado o método *derive* em uma versão configurada, a nova versão será derivada da predecessora da versão configurada, mantendo-a como um dos extremos no grafo.

Na Figura 4.10 existe também uma instância da classe *Component*. Este objeto não faz parte da configuração recém criada pois o seu relacionamento com a classe *Module* é uma associação simples. No SCM_TOO, somente os objetos relacionados à versão base por um relacionamento de herança por extensão ou por uma agregação devem ser selecionados e, portanto, fazem parte da configuração.

A execução do método *getConfiguration* em uma versão base retorna o conjunto de versões configuradas que compõem a configuração completa. Versões configuradas não podem ser alteradas, mas nada impede que sejam removidas, assumindo o estado *deactivated*. Versões configuradas serão automaticamente promovidas para *consolidated* quando fizerem parte de uma *baseline*, como será visto na próxima seção.

4.8 Baselines

Como explicado anteriormente no capítulo sobre GCS, uma *baseline* é um tipo especial de configuração, a qual deve ter sido formalmente revisada e aprovada pelos responsáveis adequados no projeto de desenvolvimento (por exemplo, o gerente do projeto e o líder técnico). Ela serve como base para as próximas atividades de desenvolvimento e somente pode ser alterada mediante procedimentos formais de controle de mudança.

Como um modelo genérico de versionamento, o TVM não prevê criação e gerência de *baselines* explicitamente. No processo de desenvolvimento de software, porém, as *baselines* desempenham um papel fundamental, sobretudo quando utilizada uma abordagem de versionamento baseado em mudanças como o proposto na próxima seção.

Para representar e gerenciar *baselines* de maneira eficiente, o SCM_TOO disponibiliza uma classe pré-definida com propriedades e comportamento específicos, apresentados na Figura 4.11.

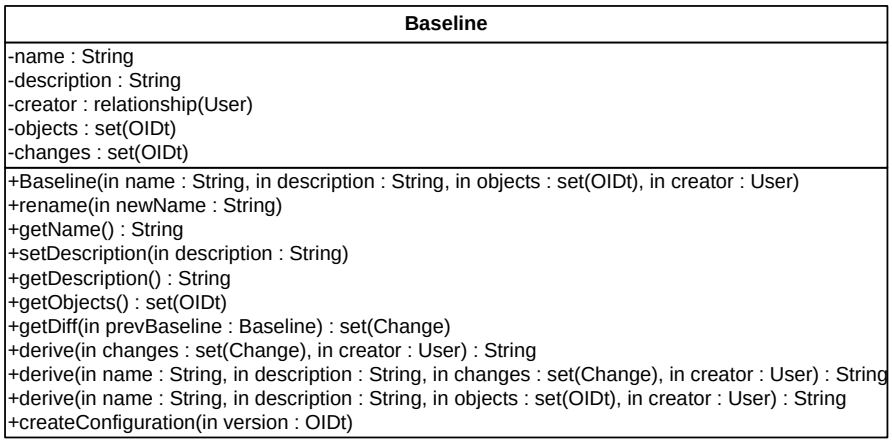


Figura 4.11: Definição da classe *Baseline*

Como atributos básicos, um objeto da classe *Baseline* possui um nome e uma descrição, cujos valores podem ser recuperados e atualizados mediante o uso de operações específicas, como *getName* e *setDescription*. A classe também contém um relacionamento com *User*, definindo sempre no momento da criação de uma *baseline* o desenvolvedor responsável por ela.

Além destas informações semânticas associadas a ela, na prática uma *baseline* não é nada mais do que um conjunto de objetos, armazenados no atributo *objects*. Estes objetos podem ser versões ou objetos não-versionados, permitindo que uma *baseline* seja composta por todos os tipos possíveis de entidades modeladas no produto de software. Algumas restrições, porém, são aplicadas a esses objetos no momento da criação de uma *baseline*.

Conceitualmente, uma *baseline* é imutável. Este é, inclusive, o principal motivo de sua existência: uma versão de um item de software adicionado a uma *baseline* é congelada e não pode ser posteriormente alterada. Novas versões do item podem ser desenvolvidas, naturalmente, mas versões definidas como parte de uma *baseline* jamais podem ser modificadas ou removidas. Dessa forma, uma *baseline* pode ser utilizada para delimitar as diferentes fases de desenvolvimento, registrando de forma segura e confiável o estado completo do produto de software entregue no término de cada etapa.

O método construtor da classe *Baseline* executa uma série de operações e validações sobre o conjunto de objetos recebidos como parâmetro, dependendo do tipo de cada um:

- objeto não-versionado – como não existe nenhum tipo de restrição definida no SCM_TOO quanto a alterações sofridas por objetos não-versionados, um novo objeto é criado cujo conteúdo seja exatamente o mesmo do objeto recebido como parâmetro, e seu OIDt é definido como parte da *baseline*. Desta forma, o objeto original pode ser posteriormente alterado e continua existindo no projeto sem restrições, enquanto que o novo objeto (que compõe a *baseline*) não pode ser modificado nem removido da base de dados, garantindo a propriedade de imutabilidade;
- versão configurada – automaticamente, as versões configuradas que compõem a configuração completa são selecionadas para fazer parte da *baseline*. Todas essas versões são, então, promovidas para o estado *consolidated*, impedindo que sejam removidas da base de dados.

- objeto versionado – no caso de receber um OIDt de um objeto versionado, o método construtor automaticamente seleciona a versão corrente do objeto para fazer parte da *baseline*;
- versão normal – o método *createConfiguration* é executado recebendo como parâmetro o OIDt da versão. Esta operação não faz nada mais do que criar uma configuração utilizando essa versão como base (ou seja, ela chama o método *buildConfiguration*).

Estas operações sobre o conjunto de objetos recebidos no momento da construção da *baseline* são executadas recursivamente, ou seja: recebendo um objeto versionado, o método seleciona a versão definida como corrente e chama a operação novamente, passando a versão como parâmetro. Recebendo a versão normal, deve ser construída uma configuração utilizando-a como base. A execução do método em uma versão configurada promove esta para *consolidated*, adiciona seu OIDt ao conjunto de objetos que compõem a *baseline* e encerra o ciclo.

A operação *getObjects* retorna a lista de OIDt's que compõem o objeto *baseline*. Os demais métodos definidos na classe *Baseline* serão explicados na próxima seção, por dependerem da utilização de objetos da classe *Change*.

4.9 Versionamento Baseado em Mudanças

O versionamento baseado em mudanças definido no SCM_TOO não impossibilita que os usuários do modelo continuem a manipular diretamente objetos versionados e suas versões, exatamente como ocorria no TVM, e que consiste no tradicional versionamento baseado em estados. Além destas funcionalidades anteriormente descritas, no SCM_TOO os desenvolvedores podem identificar uma mudança em um produto de software como uma entidade lógica, associando informações semânticas a ela e utilizando-a em operações de alto nível como criação e derivação de *baselines*. Uma mudança é aplicada ao produto de software como um todo – ela pode conter alterações em diversos artefatos de software e, portanto, encapsular versões de diferentes tipos de itens de configuração, desde documentos de requisitos até arquivos de código do sistema.

Mudanças são armazenadas no SCM_TOO mediante o uso da classe *Change*, cuja definição e principais características são apresentadas na próxima seção; seu comportamento e forma de interação com a classe *Baseline*, por sua vez, serão detalhados a seguir.

4.9.1 Classe *Change*

Uma mudança é representada pela criação de uma instância da classe *Change*, cuja definição é apresentada na Figura 4.12. A classe *Change* refina *Object* e, portanto, não pode ser versionada nem conter atributos temporais. Em um ambiente prático de desenvolvimento, é possível que uma mudança em um produto de software evolua com o tempo e seja representada por meio de várias versões, mas esta funcionalidade traria uma complexidade desnecessária para o modelo.

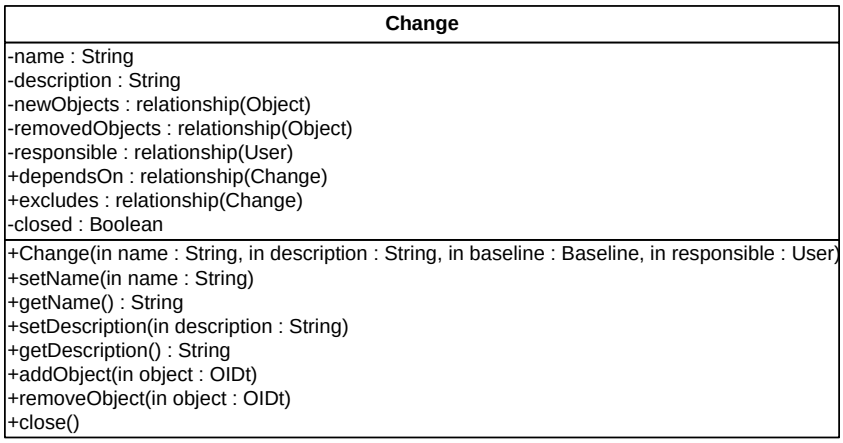


Figura 4.12: Definição da classe *Change*

Os atributos definidos diretamente pelo usuário no momento da criação de uma mudança são o seu nome e sua descrição, cujo conteúdo pode ser recuperado e alterado por meio dos métodos *get* e *set* correspondentes. Naturalmente, a classe *Change* pode ser refinada caso o desenvolvedor queira incluir outras propriedades ou relacionamentos.

O relacionamento com a classe *User* identifica o usuário do sistema responsável pela criação da mudança no software. Uma mudança sempre deve ser construída a partir de um conjunto definido de elementos de software, ou seja, todo objeto da classe *Change* deve ser associado a um objeto *Baseline*. Estes relacionamentos são passados ao construtor da classe e não podem ser posteriormente alterados pelo usuário.

Conceitualmente, sabemos que um objeto da classe *Change* identifica uma alteração realizada em um produto de software. Fisicamente, uma mudança não é nada mais do que um conjunto de objetos que devem ser adicionados ou removidos da *baseline* utilizada como ponto de partida do desenvolvimento. Esta informação é mantida pelos relacionamentos entre as classes *Change* e *Object*, denominados *newObjects* e *removedObjects*. Os métodos *addObject* e *removeObject* são utilizados para atualizar esses relacionamentos de uma maneira dinâmica e eficiente.

Uma mudança deve ser definida, não podendo conter mais do que uma versão de um mesmo elemento de software. Quando um item de configuração presente na *baseline* sofre uma modificação dentro do escopo de uma mudança, a nova versão deve ser associada ao objeto *Change*. A versão armazenada na *baseline* não precisa, porém, ser adicionada ao conjunto de objetos removidos pela mudança (relacionamento *removedObjects*). Esta exclusão será reconhecida automaticamente quando a mudança for aplicada, pois a nova versão do item presente no objeto *Change* irá sobrepor a versão identificada na *baseline*. A associação *removedObjects*, portanto, contém somente os itens de configuração que foram explicitamente removidos do software durante a implementação da mudança, como por exemplo quaisquer arquivos que se tornaram obsoletos no produto de software e não fazem mais parte do sistema. Na prática, uma mudança quase sempre é representada pela adição de um grande número de versões e até mesmo de novos objetos, enquanto que a lista de objetos removidos é normalmente muito pequena ou vazia.

Uma mudança possui, ainda, dois relacionamentos com a própria classe *Change*, denominados *dependsOn* e *excludes*. O primeiro mantém a lista de mudanças das quais a alteração no software depende. Na prática, uma mudança só pode ser aplicada a um

produto de software se todas as alterações das quais ela depende tenham sido aplicadas também. O relacionamento *excludes*, por sua vez, armazena as mudanças que são mutualmente exclusivas. Somente uma delas pode ser incluída em cada versão do produto de software.

Outros tipos de informação podem ser definidos pelo usuário caso a classe *Change* seja refinada, mas estes relacionamentos pré-definidos são especialmente importantes pois o sistema verifica o seu conteúdo no momento da construção de novas versões do produto de software.

4.9.2 Objetos *Change* e a Construção de Novas *Baselines*

Para aplicar o versionamento baseado em mudanças disponível no SCM_TOO, uma *baseline* deve ser construída e definida como ponto inicial de desenvolvimento. No caso do desenvolvimento de um produto de software novo, e não um projeto de manutenção de um sistema existente, deve ser criado um objeto *baseline* vazio.

Uma instância da classe *Change* é gerada pelo desenvolvedor antes de iniciar o seu trabalho, definindo nome e descrição da mudança, além de associar a mesma à *baseline* previamente criada. A partir deste momento, os identificadores de todas as versões e objetos criados ou modificados pelo usuário serão adicionados ao objeto mudança. Esta tarefa de associação pode ser realizada automaticamente pelo sistema de GCS, bastando que o usuário identifique na sua área de trabalho a alteração na qual ele estará trabalhando antes de começar o desenvolvimento. Esta funcionalidade, porém, é parte do controle de processo da gerência de configuração de software e está fora do escopo deste trabalho, cujo objetivo é definir um modelo de dados genérico a ser utilizado em um ambiente de GCS.

O desenvolvedor de software deve definir explicitamente os relacionamentos entre as mudanças, determinando suas dependências e exclusões mútuas. Após várias mudanças terem sido instanciadas e implementadas, o engenheiro de software deve criar uma nova *baseline* antes de mover o código para a próxima fase no ciclo de desenvolvimento (a fase de verificação e validação do sistema, por exemplo). No SCM_TOO, *baselines* são também objetos versionados, portanto deve ser criada uma nova versão da *baseline* definida como início do desenvolvimento.

Como visto na Figura 4.11, a classe *Baseline* redefine o método *derive* do TVM para implementar o seu comportamento diferenciado. No momento da derivação o usuário passa como parâmetro o conjunto de mudanças que devem ser incorporadas na nova versão da *baseline*. Assim, o conjunto de objetos que compõem a nova *baseline* é o mesmo presente na versão anterior, atualizado com a aplicação das mudanças selecionadas.

O método *derive* realiza uma avaliação das mudanças recebidas como parâmetro, verificando se estas podem ser incluídas em uma mesma *baseline*. Caso duas dessas mudanças possuírem um relacionamento de exclusão (*excludes*) entre elas, a nova versão da *baseline* não é criada e o método retorna uma mensagem para o usuário, informando as mudanças mutualmente exclusivas. A seguir, o método avalia todos os relacionamentos *dependsOn* das mudanças, verificando se todas as dependências são atendidas ou não. No caso de existir uma ou mais mudanças que dependem de um objeto *Change* não incluído na lista de objetos recebidos como parâmetro, a *baseline* não é derivada e o método retorna uma mensagem de texto com a mudança que não permitiu a criação da nova versão da *baseline*.

O usuário do sistema pode, ainda, controlar a evolução do produto de software e criar *baselines* sem instanciar objetos da classe *Change*, utilizando somente o tradicional versionamento baseado em estados implementado no TVM que obriga o usuário a selecionar explicitamente todas as versões a serem incluídas em uma *baseline*. Apesar de não recomendar esta prática, o SCM_TOO propõe-se como um modelo altamente flexível e, portanto, disponibiliza um método para derivar *baselines* cujo principal parâmetro é um conjunto de OIDt's. Assim, o usuário tem liberdade para criar novas versões de uma *baseline* incluindo quaisquer objetos e versões que desejar.

Todas as implementações do método *derive* recebem também como parâmetro o usuário (um objeto da classe *User*) responsável pela criação da nova versão da *baseline*, de forma a manter o histórico sobre qual usuário executou a criação de cada *baseline*.

A nova versão da classe *baseline* mantém em seu atributo *changes* uma lista de identificadores das mudanças utilizadas no momento da sua criação. O modelo disponibiliza um método denominado *getDiff*, que retorna o conjunto de mudanças que representam a diferença entre duas versões de uma mesma *baseline*. O método deve ser executado em uma versão de uma *baseline* e recebe como parâmetro outra versão da mesma *baseline*. A versão passada ao método deve necessariamente ser uma versão anterior a *baseline* que está executando a operação. A partir do conteúdo do atributo *changes*, o método retorna o conjunto de mudanças incluídas na criação da *baseline* que não existiam na versão anterior. Caso o grafo de versões apresente outras versões entre as duas que o usuário estiver comparando, o método é executado recursivamente para retornar a soma de todas as mudanças incluídas no processo de desenvolvimento da *baseline*. No próximo capítulo, um exemplo de execução deste método ilustrará melhor a sua utilização durante o desenvolvimento de um produto de software.

Quando incluída em uma *baseline*, uma mudança não pode mais ser modificada. Conceitualmente, a inclusão de uma mudança em uma *baseline* significa que ela está pronta e a lista de objetos que a compõe deve, portanto, ser congelada. Os objetos, por sua vez, são também congelados e não podem ser posteriormente alterados, como explicado na seção 4.6 sobre *baselines*. Todos os métodos de atualização dos atributos de um objeto *Change* avaliam o valor armazenado na propriedade lógica *closed*. A operação *close* torna o seu valor igual a *true*, impossibilitando que o usuário modifique o conteúdo da mudança e garantindo a sua imutabilidade. O método *close* é executado em todos os objetos da classe *Change* designados para fazer parte de uma *baseline* mediante o uso do método *derive*.

Naturalmente, uma mudança pode ser incluída na criação de várias *baselines* diferentes ou, ainda, de várias versões de uma mesma *baseline*. Estas versões devem fazer parte de ramos de desenvolvimento diferentes dentro do objeto versionado, caracterizando o desenvolvimento paralelo ou uma alternativa de projeto. Se o método *derive* da classe *Baseline* receber como parâmetro um objeto *Change* já incluído anteriormente na linha de desenvolvimento da *baseline*, a nova versão não é criada e a operação retorna uma mensagem de erro informando a mudança.

4.9.3 Conflitos na Derivação de uma *Baseline*

Quando várias mudanças são utilizadas no momento da derivação de uma *baseline*, podem surgir conflitos entre as versões dos objetos que compõem diferentes mudanças. O método *derive* da classe *Baseline* examina a lista completa dos objetos que

compõem todas as mudanças recebidas como parâmetro, seguindo o seguinte procedimento:

- objetos não versionados são diretamente incluídos na *baseline* pois, mesmo que apareçam em mais de uma mudança, o seu estado (valor dos atributos) será igual em todas elas;
- uma versão que seja a única de um objeto versionado presente nas mudanças é também diretamente incluída na *baseline*. No caso do objeto versionado estar presente na *baseline* original (na qual o método de derivação foi executado), o seu OIDt é simplesmente excluído da nova lista de objetos. Assim, a versão criada pela mudança torna-se a única do objeto versionado presente na nova versão da *baseline*;
- no caso de existirem duas ou mais versões de um mesmo objeto versionado nas mudanças aplicadas, torna-se necessária uma análise mais cuidadosa (além do OIDt da *baseline* original ser removido, como no item anterior). O grafo de versões do objeto versionado deve ser examinado para determinar se existe ou não um conflito. Caso as versões em questão façam parte do mesmo ramo de desenvolvimento do objeto, o sistema simplesmente seleciona a versão mais nova para fazer parte da *baseline*. Esta situação ocorre quando as mudanças são desenvolvidas uma após a outra e, naturalmente, a versão mais atual contém as modificações realizadas previamente na versão anterior. Neste caso, uma versão é uma revisão da outra. Se duas mudanças modificaram concorrentemente o mesmo objeto versionado, teremos versões variantes e localizadas em linhas alternativas de desenvolvimento do objeto. Ou seja, uma versão não inclui as modificações realizadas na outra versão e vice-versa. Neste caso, o conteúdo das versões deve ser unificado mediante a execução da operação *merge* do objeto versionado para criar uma nova versão, a qual é incluída na *baseline*.

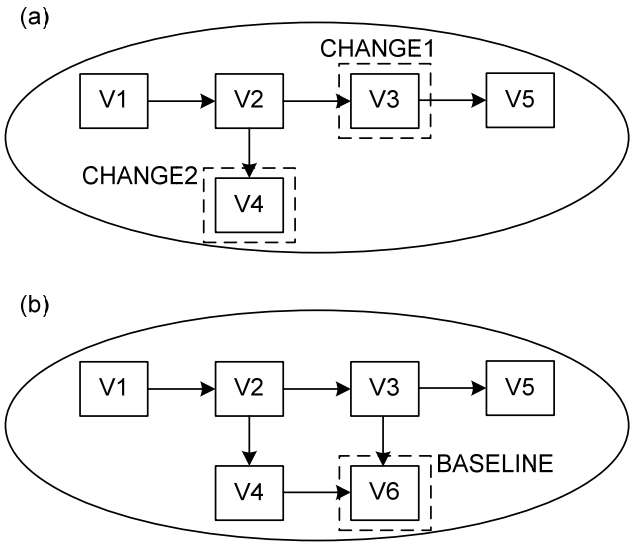


Figura 4.13: Conflito entre duas mudanças (a) e a operação *merge* (b)

A Figura 4.13 apresenta um exemplo da última situação descrita, caracterizando um conflito na criação da *baseline* resolvido com a execução do método *merge*, o qual será detalhado na próxima seção.

No primeiro cenário (a) as versões V3 e V4 fazem parte de duas mudanças diferentes, CHANGE1 e CHANGE2 respectivamente. Ambas foram selecionadas no momento da derivação de uma nova *baseline*, caracterizando um conflito por uma versão ser alternativa em relação à outra. A operação *merge* é então executada (b), gerando a versão V6 que inclui o conteúdo de ambas as mudanças selecionadas. Esta nova versão é, então, adicionada à *baseline*.

4.10 A Operação *Merge*

TemporalVersion
-ascendant : set(OIDt) = NULL -configuration : Boolean = false -descendant : set(OIDt) = NULL -predecessor : set(OIDt) = NULL -status : Char = W -successor : set(OIDt) = NULL
+TemporalVersion() +TemporalVersion(in ascendant : set(OIDt)) +TemporalVersion(in entityName : String) +TemporalVersion(in entityId : Integer, in classId : Integer, in versionId : Integer) -TemporalVersion(in predecId : set(OIDt), in ascendId : set(OIDt), in config : Boolean) -addAscendant(in ascendId : OIDt) -addDescendant(in descendId : OIDt) -addSuccessor(in succId : OIDt) +delete(in allReferences : Boolean) +deleteObjectTree(in allReferences : Boolean) +derive() +getAscendant() : set(OIDt) +getAscendant(in className : String) : set(OIDt) +getConfiguration() : OIDt +getCompleteObject() : set(OIDt) +getDescendant() : set(OIDt) +getDescendant(in className : String) : set(OIDt) +getPredecessor() : set(OIDt) +getStatus() : Char +getSuccessor(in onlyConfigured : Boolean) : set(OIDt) +getVOC() : OIDt +isConfiguration() : Boolean -isDeleteAllowed(in allReferences : Boolean) : Boolean -isDeleteTreeAllowed(in allReferences : Boolean) : Boolean +merge(in version : OIDt) : OIDt +promote(in allAscendant : Boolean, in allReferenced : Boolean) -removeAscendant(in ascendId : OIDt) -removeDescendant(in descendId : OIDt) -removeSuccessor(in succId : OIDt) +restore(in OID : OIDt) : Boolean -setAscendant(in ascendId : OIDt) -setDescendant(in descendId : OIDt) -setStatus(in newStatus : Char) -setSuccessor(in succId : set(OIDt)) -verifyAscendId(in ascendId : set(OIDt)) : Boolean

Figura 4.14: Definição da classe *TemporalVersion*

O Modelo Temporal de Versões originalmente foi preparado para suportar a operação *merge*, pois o atributo *predecessor* de uma versão é multivalorado, ou seja, pode armazenar um conjunto de OIDt's. Dessa forma, no grafo de versões do objeto versionado uma versão pode ser sucessora de duas ou mais versões ao mesmo tempo,

caracterizando a operação de junção (*merge*). O método *derive* presente na classe *TemporalVersion*, por sua vez, recebe como parâmetro uma lista de OIDt's determinando as versões a serem utilizadas na criação da nova versão. Apesar da operação *merge* ter sido prevista na definição da classe, sua execução não foi explicitamente definida no TVM.

Por desempenhar um papel fundamental no versionamento baseado em mudanças definido no SCM_TOO (como visto no item anterior), a operação *merge* será detalhada nesta seção. A classe *TemporalVersion* sofreu algumas alterações em relação ao TVM para suportar corretamente o método de junção de versões; sua nova definição pode ser vista na Figura 4.14.

O método *derive* foi redefinido para que somente crie versões sucessoras de apenas uma versão, ou seja, sem a utilização da operação *merge*. O método não recebe nenhum valor como parâmetro, pois ele criará uma nova versão derivando a versão na qual o método foi chamado.

O método *merge* é explicitamente definido na classe *TemporalVersion* para desempenhar a junção de duas versões. O algoritmo utilizado não suporta a junção de três ou mais versões ao mesmo tempo, portanto se houver a necessidade de realizar a união de mais de duas versões, a operação deve ser executada mais de uma vez, criando versões intermediárias. O método *merge* deve ser invocado em uma das versões a serem unificadas, sendo a segunda versão passada como parâmetro.

O algoritmo proposto pelo SCM_TOO para realizar o *merge* é o tradicional *three-way-merge* [MEN 2002], que consiste na comparação entre as duas versões a serem unificadas baseando-se na versão que originou ambas. No SCM_TOO, as versões devem pertencer ao mesmo objeto versionado e nenhuma delas pode ser a primeira versão do objeto. Isto garante que exista pelo menos uma versão comum na linha de desenvolvimento das duas versões a serem unificadas, mesmo que esta versão compartilhada seja a primeira versão do objeto versionado. Naturalmente, uma versão deve ser variante em relação a outra versão, ou seja, elas devem estar em ramos de desenvolvimento diferentes no grafo de versões do objeto versionado. Atendidas estas restrições, o método realiza a junção conforme o pseudo-algoritmo definido na Figura 4.15.

```
repetir o bloco para todos atributos e relacionamentos da classe
{
  se VERSÃO_1.atributo = VERSÃO_2.atributo
    então NOVA_VERSÃO.atributo = VERSÃO_1.atributo
    caso contrário
      se VERSÃO_1.atributo = VERSÃO_BASE.atributo
        então NOVA_VERSÃO.atributo = VERSÃO_2.atributo
        caso contrário
          se VERSÃO_2.atributo = VERSÃO_BASE.atributo
            então NOVA_VERSÃO.atributo = VERSÃO_1.atributo
            caso contrário
              << CONFLITO >>
}
```

Figura 4.15: Algoritmo *merge* de duas versões

A granularidade da junção é o valor de cada atributo (ou relacionamento) presente nas versões. Inicialmente, compara-se o conteúdo do atributo armazenado nas duas

versões. Encontrando valores iguais, o mesmo é copiado para a nova versão. Caso contrário, é necessária a consulta ao valor do atributo armazenado na versão base (comum nos ramos de desenvolvimento das duas versões). Caso o valor da versão base seja igual ao da primeira versão, o valor da segunda versão é copiado para a nova versão. Nesta situação, o atributo na primeira versão não foi modificado em relação à base, mas a segunda versão sofreu uma atualização. Portanto, essa modificação deve ser incorporada na nova versão. O caso dos valores armazenados nas três versões serem diferentes caracteriza um conflito, pois ambas versões sendo unificadas modificaram o mesmo atributo. Nesta situação, o sistema deve interagir com o usuário para definir qual dos valores a ser utilizado ou até mesmo para receber um novo valor para a versão final. Outra possibilidade seria a definição de uma das versões como *default* na ocorrência de conflitos, a qual poderia ser a versão na qual o método *merge* foi invocado. Por ser um modelo conceitual, SCM_TOO não define explicitamente o que fazer no momento de um conflito, deixando esta decisão a ser tomada durante a sua implementação.

O método *merge* retorna o OID da versão criada como resultado da operação de junção. Caso a nova versão não possa ser criada em virtude de alguma das exigências anteriormente descritas não terem sido atendidas, o método retorna nulo.

4.11 Considerações Finais

Este capítulo apresentou o SCM_TOO, um modelo de dados orientado a objeto desenvolvido especificamente para gerenciar configurações de software. Baseado fortemente no Modelo Temporal de Versões, o SCM_TOO mantém as importantes inovações propostas pelo seu antecessor e adiciona interessantes funcionalidades para modelar e manter sob controle a evolução de complexos sistemas de software.

Por não trazer grandes benefícios para a disciplina de GCS, a bitemporalidade do TVM foi modificada a fim de suportar somente uma dimensão temporal, tendo sido escolhido o tempo de transação. Atributos e relacionamentos comuns foram criados para adicionar valor conceitual aos objetos versionados e aumentar o poder de modelagem do desenvolvedor de software. Classes e mecanismos foram definidos a fim de permitir o armazenamento e o versionamento de arquivos que compõem o software, utilizando a consolidada técnica *check-out/check-in* para suportar de maneira adequada o trabalho dos desenvolvedores.

O versionamento baseado em mudanças, por sua vez, consiste na principal contribuição do SCM_TOO. A representação das mudanças implementadas em um produto de software como entidades de primeiro nível, complementada com os mecanismos propostos para a criação e manipulação de *baselines*, permite que a evolução da aplicação seja controlada e visualizada de várias maneiras diferentes. Apesar desta abordagem avançada, o modelo continua simples e de fácil utilização por continuar suportando o versionamento individual de objetos definido no TVM e ser fundamentado em conceitos e técnicas consagrados da engenharia de software – a orientação a objeto. O próximo capítulo apresenta um estudo de caso que ilustra a utilização do versionamento baseado em mudanças em uma aplicação real, exemplificando algumas importantes inovações disponibilizadas pelo SCM_TOO.

5 ESTUDO DE CASO

Muitos conceitos e características introduzidos pelo SCM_TOO e o modo que suas funcionalidades devem ser aproveitadas na prática podem não ter sido bem esclarecidos no capítulo anterior. Um estudo de caso onde acompanharemos um pouco da evolução de um produto de software real utilizando o modelo proposto neste trabalho será útil para exemplificar o uso de suas classes e mecanismos disponíveis para gerenciar configurações de software.

Não é objetivo deste capítulo conduzir uma experiência a partir da qual seja possível comparar formalmente o SCM_TOO com outros modelos similares descritos no capítulo sobre GCS. Apesar de existirem alguns atributos mensuráveis que poderiam ser usados como entrada para um estudo comparativo desta natureza, ainda hoje é bastante difícil determinar a eficácia de uma ou outra metodologia no processo de desenvolvimento de software. O estudo de caso descrito neste capítulo tem o propósito, portanto, de exemplificar como uma aplicação do mundo real pode ser representada no SCM_TOO e a utilização do modelo para manter a evolução do produto de software sob controle.

O comportamento de atributos e relacionamentos comuns e temporais não será descrito neste estudo de caso, apesar da aplicação modelada aqui possuir vários deles – como pode ser visto na Figura 5.2. A temporalidade foi anteriormente explicada e exemplificada em alguns trabalhos, principalmente os focados no TVM [SIL 2003b], enquanto que as características comuns introduzidas pelo SCM_TOO são bastante simples e de fácil entendimento. O estudo de caso exemplifica a utilização do versionamento baseado em mudanças, por ser este a principal inovação do modelo definido neste trabalho. A utilização das classes *Change* e *Baseline*, bem como a maneira que estas interagem com os elementos de software tradicionais e sua implicação na construção de configurações, será extensivamente demonstrada ao longo deste capítulo.

A aplicação modelada neste estudo é um software previamente desenvolvido e estável, o qual sofrerá algumas melhorias como parte do processo de manutenção do produto. Naturalmente, o SCM_TOO pode ser utilizado no desenvolvimento de sistemas completamente novos, mas optou-se pelo uso de uma aplicação legada por esta ser mais adequada na exemplificação de *baselines* e do versionamento baseado em mudanças.

5.1 Modelagem da Estrutura do Produto de Software

A etapa de modelagem da estrutura do produto de software não existe no processo tradicional de desenvolvimento simplesmente porque os tipos de elementos que compõem o software são pré-determinados pelo sistema de arquivos. Um software tradicionalmente é representado por um conjunto de arquivos distribuídos em diretórios, de forma que o engenheiro não tem a liberdade de definir outros tipos de composições ou de criar relacionamentos entre os artefatos de software. A Figura 5.1 apresenta o que seria o esquema de um produto de software – podemos fazer uma analogia com o esquema de um banco de dados antes de inserirmos qualquer tipo de informação no mesmo.

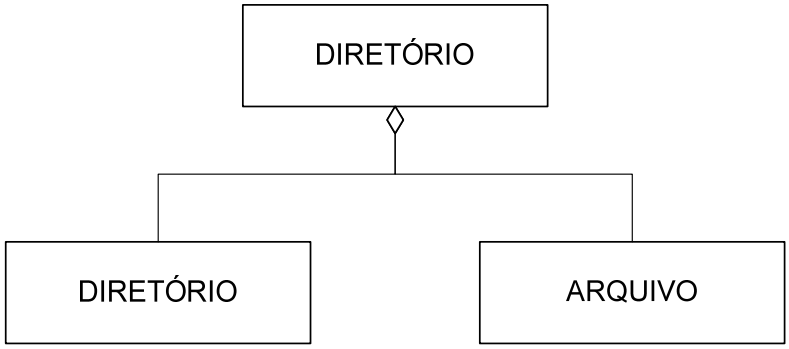


Figura 5.1: Estrutura tradicional de um produto de software

Tradicionalmente, um produto de software é constituído por um conjunto de arquivos e diretórios. Um diretório não é nada mais do que uma agregação de outros diretórios e/ou um conjunto de arquivos.

A utilização do SCM_TOO permite que a estrutura do produto de software seja completamente determinada pelo usuário, aplicando as características da orientação a objeto para definir os tipos de elementos que farão parte do sistema (classes) e a maneira que eles se relacionam entre si (associações e agregações). Dessa forma, diferentes tipos de aplicação e diferentes paradigmas de desenvolvimento podem ser representados diretamente na estrutura do software, permitindo uma modelagem mais flexível e consistente.

A aplicação utilizada como exemplo e descrita neste capítulo é uma loja virtual que vende livros na Internet, denominada "*On-Line Book Storm*". Apesar de ser um trocadilho com o termo original em inglês "*on-line book store*", ou "livraria *on-line*", a tradução mais correta seria "tempestade de livros *on-line*". A loja virtual é composta basicamente por módulos e componentes, os quais são implementados em uma grande quantidade de arquivos. Utilizando o SCM_TOO, a especificação formal da aplicação e os requisitos de software podem ser representados no esquema do produto e, portanto, relacionados com os artefatos de software que fisicamente implementarão as correspondentes funcionalidades.

A Tabela 5.1 apresenta os estereótipos utilizados nos diagramas de classe a seguir, determinando as classes que podem ser versionadas ou não e os atributos com características temporais ou compartilhados.

Tabela 5.1: Símbolos para representação gráfica

Símbolo	Significado
T_V	A classe associada é temporal e versionável (refina *TemporalVersion*)
<<t>>	O atributo associado é temporal
<<c>>	O atributo associado é comum
<<c>><<t>>	O atributo associado é comum e temporal

A Figura 5.2 apresenta as classes e relacionamentos que compõem o esquema do produto de software da aplicação – as classes determinam os tipos de elementos de software que farão parte do sistema. Vários detalhes como atributos, métodos e cardinalidades foram suprimidos da figura para não sobrecarregá-la. É importante perceber que a estrutura do software apresentada aqui pode ser utilizada para modelar outras aplicações, não sendo específica para a loja virtual utilizada como estudo de caso. Esquemas de produtos de software podem ser reutilizados no momento da criação de novos sistemas.

A classe *Application* mantém informações genéricas sobre o software, como o seu nome, descrição e versão – o conteúdo da propriedade *version* é determinado pelo fabricante, podendo ser simplesmente um número (2, 3) ou o ano do lançamento (2004), e não está diretamente relacionada com o OIDt gerado automaticamente pelo modelo. A classe *Application* herda de *TemporalVersion* para que seus objetos possam ser versionados e seus atributos temporalizados. Por exemplo, a propriedade *description* é definida como temporal para que todo valor armazenado seja mantido no histórico, registrando as modificações realizadas na descrição da aplicação durante todo o tempo de vida do software.

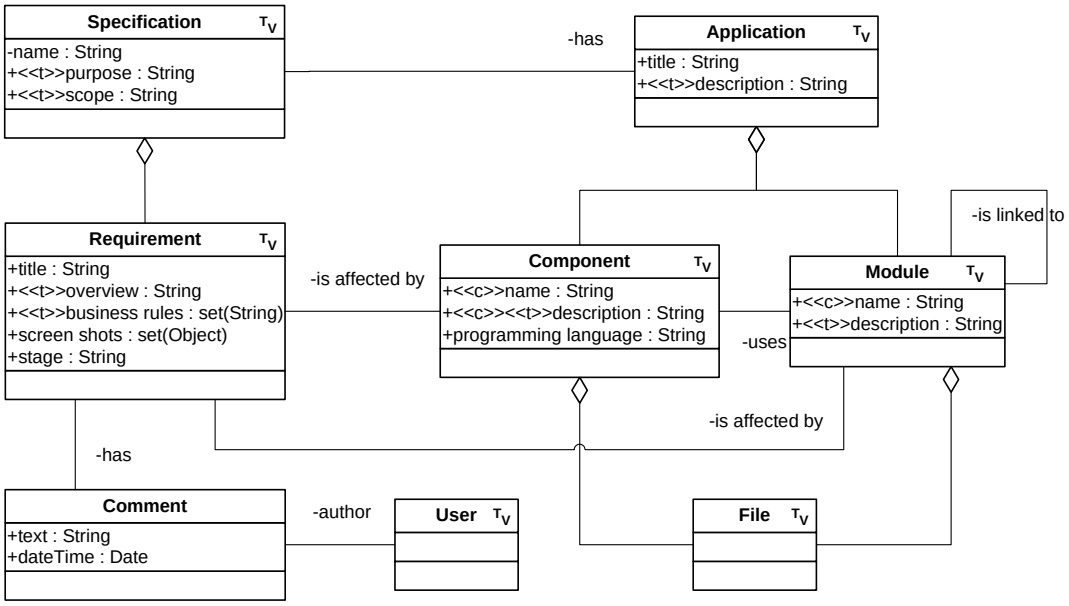


Figura 5.2: Esquema do produto de software

Um objeto *Application* será composto por uma agregação de componentes e módulos. Ambas classes (*Component* e *Module*) são definidas como *TemporalVersion* e possuem alguns atributos temporais ou comuns. Na classe *Component*, por exemplo, a

propriedade *name* é compartilhada por todas as versões de cada objeto versionado. A descrição de um componente é definida como comum e temporal, pois o histórico de atualizações no seu valor deve ser mantido automaticamente pelo sistema além de ser compartilhado entre suas versões. A classe *Module* é ligada à *Component* por um relacionamento, denomindado *uses*, determinando que um módulo pode usar um ou mais componentes do sistema. Além disso, um módulo pode ser conectado a outros módulos mediante o auto-relacionamento *is linked to*. Este relacionamento representa os caminhos possíveis de navegação entre os módulos da aplicação. O relacionamento de agregação entre as classes *Component* e *Module* e a classe *File* define explicitamente no esquema que estas classes serão compostas por um conjunto de arquivos. No diagrama as classes *File* e *User* não têm seus atributos detalhados pois elas são as mesmas apresentadas no capítulo anterior, ou seja, o usuário utilizará diretamente dessas classes pré-definidas pelo modelo para instanciar arquivos e usuários envolvidos no processo de desenvolvimento de software.

Em um sistema de GCS tradicional, vários arquivos contendo a especificação do sistema e os requisitos de desenvolvimento definidos para o produto de software seriam inseridos no repositório de componentes, mas dificilmente esses documentos seriam ligados aos elementos do sistema afetados por eles. O paradigma de orientação a objeto utilizado pelo SCM_TOO, entretanto, possibilita que o usuário represente os requisitos de desenvolvimento em classes e objetos, criando relacionamentos entre quaisquer tipos de itens de configuração armazenados no repositório.

No exemplo aqui apresentado, a classe *Specification* contém a especificação do produto de software, possuindo um relacionamento com a classe *Application*. Dessa forma, quando novas versões do objeto aplicação forem criadas, estas poderão ser associadas às versões corretas do objeto especificação. A classe *Specification* possui alguns atributos temporais, como *purpose*, cujo histórico de modificações será automaticamente mantido pelo sistema. O relacionamento de agregação entre as classes *Specification* e *Requirement* denota que uma especificação é composta por vários requisitos de software. Cada um desses requisitos pode afetar um ou mais componentes e/ou módulos do sistema – esta informação semântica é representada no esquema do software mediante os relacionamentos entre as classes *Requirement* e *Component* e/ou *Module*. Um objeto requisito posuirá, entre outros dados, um título, uma visão geral (atributo temporal *overview*) e várias *screen shots* (atributo cujo domínio é um conjunto de *Object*, onde o usuário poderá armazenar imagens). Em um projeto de desenvolvimento de software, é bastante usual que um requisito, mesmo depois de definido e descrito formalmente, ainda apresente alguns dados imprecisos que gerarão dúvidas aos desenvolvedores no momento da implementação e ao grupo de controle de qualidade durante a preparação dos testes a serem executados. Uma prática comum em um ambiente de desenvolvimento é a adição de comentários a um requisito, refinando determinadas características e associando novas informações necessárias no projeto do sistema. No esquema aqui apresentado, estes comentários são implementados na classe *Comment*, composta simplesmente por um atributo texto e um *timestamp* (data e hora). A classe possui dois relacionamentos, um com a classe *User* (para manter o usuário responsável pela criação de cada comentário) e outro com a classe *Requirement* (ligando cada requisito ao conjunto de comentários associados a este). A classe *Comment* não refina *TemporalVersion* pois não utiliza características temporais nem de versionamento. Cada comentário é adicionado somente uma vez e não deve ser posteriormente alterado, pois qualquer informação adicional deve ser inserida no projeto como um novo comentário.

5.2 Visão Estática do Produto de Software

Nesta seção a aplicação exemplo será descrita considerando somente uma versão de cada elemento de software, fornecendo uma visão estática do sistema.

Após definida a estrutura do produto de software, objetos são instanciados para compor o sistema sob desenvolvimento. O exemplo aqui apresentado é um sistema legado, portanto toda a aplicação "*On-Line Book Storm*" foi previamente instanciada e armazenada no repositório. Fisicamente, um grande número de objetos e relacionamentos são utilizados para compor a aplicação, mas por motivos de simplificação o estudo de caso utilizará somente um sub-conjunto desses elementos. A Figura 5.3 apresenta alguns dos objetos da aplicação e os relacionamentos entre eles.

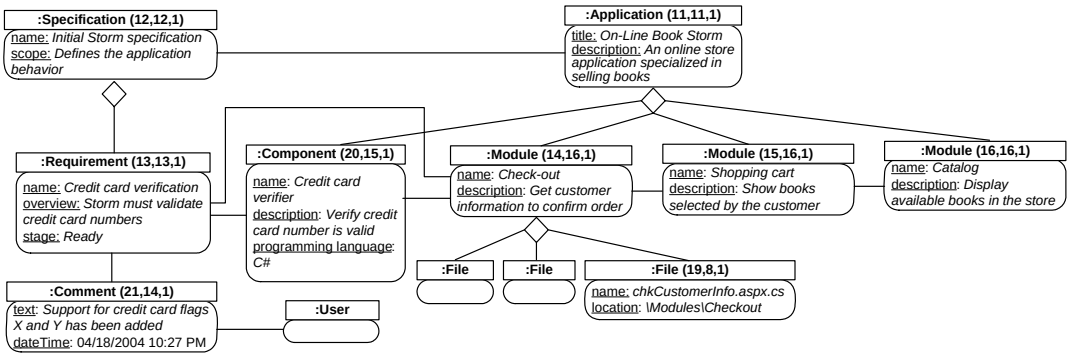


Figura 5.3: Objetos da aplicação "*On-Line Book Storm*"

Esta figura mostra apenas uma versão de cada objeto para exemplificar como um produto de software é fisicamente representado no SCM_TOO. A aplicação é composta por três módulos, denominados "*Check-out*", "*Shopping cart*" e "*Catalog*". O usuário da aplicação sempre inicia sua navegação no catálogo, onde todos os livros disponíveis na loja podem ser consultados e seus detalhes visualizados pelo consumidor. O consumidor, então, adiciona os livros escolhidos ao seu carrinho de compras, entrando no módulo *Shopping cart*. A página do carrinho mostra os livros selecionados com seus respectivos valores e adiciona as despesas de entrega. Confirmando a compra no *Shopping cart*, o consumidor entra no módulo *Check-out*, onde deve informar todos os seus dados pessoais e escolher a forma de pagamento mais adequada. Apesar de cada módulo possuir dezenas de arquivos, a figura mostra apenas três objetos da classe *File* que compõem o módulo *Check-out*, e somente alguns atributos de um dos arquivos são apresentados.

A aplicação também utiliza diversos componentes, mas a figura somente apresenta um deles, denominado "*Credit card verifier*". Este componente é utilizado pelo módulo *Check-out* (existe um relacionamento entre eles) para validar o número do cartão de crédito digitado pelo consumidor no momento da confirmação do pedido.

No exemplo apresentado na figura é também representado um objeto da classe *Requirement*, cujo nome é "*Credit card verification*". Ele mantém dois relacionamentos, um com o módulo *Check-out* e outro com o componente *Credit card verifier*, que indicam quais elementos do produto de software foram afetados pelo requisito no momento da sua implementação. O objeto *requirement* faz parte do objeto da classe *Specification*, que descreve as características básicas da aplicação, e tem um comentário associado, armazenado em uma instância da classe *Comment*.

5.3 Evolução da Aplicação

As funcionalidades de versionamento e temporalidade do SCM_TOO podem ser utilizadas individualmente nos elementos de software, praticamente da mesma forma anteriormente definida no TVM. A grande contribuição do SCM_TOO para gerenciar configurações de software, porém, aparece no uso do versionamento baseado em mudanças que consiste em algumas tarefas específicas realizadas em seqüência.

Inicialmente, uma instância da classe *Baseline* é criada contendo a configuração atual do software, cujos objetos serão utilizados pelos desenvolvedores envolvidos no projeto como ponto de partida de seu trabalho. A Figura 5.4 apresenta o objeto criado neste estudo de caso.

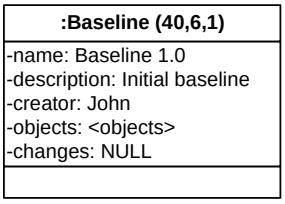


Figura 5.4: *Baseline* inicial

Apesar do SCM_TOO fazer o controle do número da versão de cada objeto automaticamente, grupos de desenvolvimento de software costumam numerar as versões de seus produtos de formas diferentes. Por exemplo, um fabricante pode utilizar o dígito antes do ponto para contar cada *release* externo do produto, enquanto que outro fabricante pode incrementar este dígito a cada *baseline* entregue ao time de controle de qualidade. Assim, o usuário tem a liberdade de inserir qualquer identificador próprio no atributo *name* do objeto *baseline*. Na figura, o nome da instância é "*Baseline* 1.0" e sua descrição indica que esta é a *baseline* inicial do projeto de desenvolvimento. A propriedade *creator* é, na verdade, um relacionamento com a classe *User*, mas para tornar o exemplo mais claro o nome do usuário (John) foi colocado na figura. O atributo *objects* contém uma lista dos OIDt's que compõem a *baseline*, entre eles todos os objetos apresentados anteriormente na Figura 5.3. Obviamente, uma grande quantidade de objetos não aparece na figura por motivos de simplificação. Como esta é a primeira versão da *baseline*, o usuário passou diretamente a lista de OIDt's e não foi utilizado nenhum objeto do tipo mudança na sua criação, portanto o atributo *changes* recebeu o valor nulo.

Criada a *baseline*, um objeto da classe *Change* deve ser instanciado para cada alteração desenvolvida no software, antes que estas modificações comecem a ser implementadas.

Inicialmente, dois requisitos são selecionados para serem desenvolvidos, um sobre disponibilizar ao consumidor a opção de receber os livros embrulhados para presente e outro que modifica a funcionalidade da busca de livros no módulo catálogo.

5.3.1 Mudança para enviar um pedido como presente

Digamos que o engenheiro John seja designado para desenvolver o primeiro requisito, denominado *packing order as a gift*. Ele instancia um objeto da classe *Change* antes de iniciar o seu trabalho, apresentado na Figura 5.5.

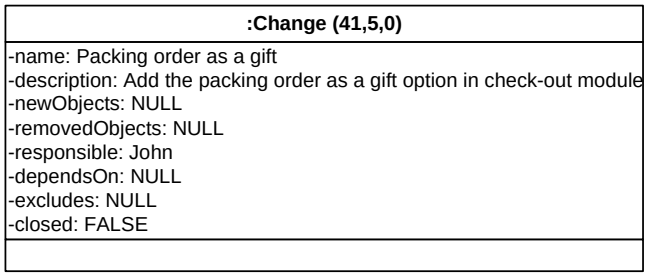


Figura 5.5: Mudança *packing order as a gift*

Como todo objeto *Change* recém criado, os relacionamentos *newObjects*, *removedObjects*, *dependsOn* e *excludes* recebem o valor nulo e serão atualizados durante a implementação da mudança. O atributo *closed* permanece como *false* até o momento em que a mudança seja incorporada em uma *baseline*, quando então ela será congelada e não mais modificada pelos desenvolvedores. O relacionamento com a classe *Baseline* desta instância aponta para o objeto "*Baseline* 1.0" previamente apresentado.

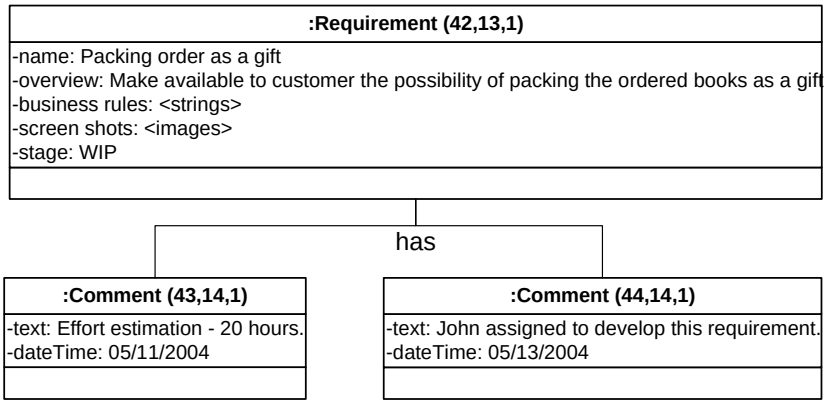


Figura 5.6: Requisito *packing order as a gift* e seus comentários

Antes de iniciar a programação da mudança, os objetos utilizados para formalizar o requisito são instanciados no repositório. A Figura 5.6 apresenta as instâncias das classes *Requirement* e *Comment* criados com esse propósito. Em uma situação real, esses objetos poderiam existir anteriormente à criação do objeto *Change*, pois os requisitos normalmente seriam inseridos na base antes da definição da sua implementação. Após a estimativa de esforço estar concluída e os requisitos priorizados de acordo com suas relevâncias, o gerente do projeto designa um ou mais desenvolvedores para cada implementação. Neste momento, é criado um objeto da classe *Change* e as instâncias do requisito apresentadas na figura são então adicionadas à mudança. Na figura, o conteúdo do atributo *business rules* foi suprimido por ser muito longo. A propriedade *stage* contém o estágio de desenvolvimento do requisito e neste

exemplo seu valor foi recém atualizado para WIP (*work in progress*). O requisito é ainda associado ao módulo *Check-out* mediante o relacionamento entre as classes *Requirement* e *Module*, denotando qual módulo da aplicação é afetado diretamente por esta mudança.

Para implementar o requisito, o desenvolvedor criou novas versões de três diferentes arquivos, como ilustra a Figura 5.7. Os identificadores dos objetos da classe *File* que aparecem na figura são associados a cada objeto versionado, por isso o número de versão de cada um é igual a zero. Os OIDt's das novas versões dos arquivos são adicionadas à lista *newObjects* do objeto *Change*, previamente composta pelos identificadores dos objetos que descrevem o requisito.

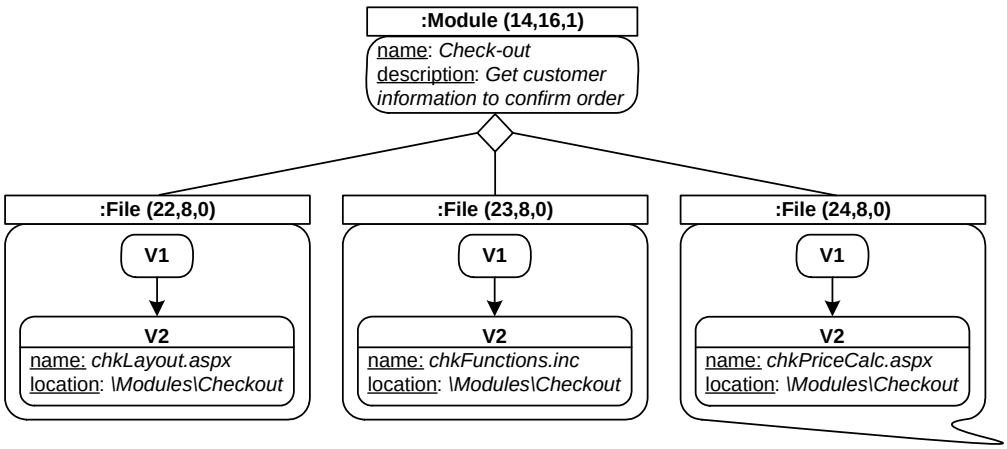


Figura 5.7: Novas versões de arquivos do módulo *Check-out*

Além de novas versões dos arquivos, uma nova classe foi criada pelo desenvolvedor para implementar adequadamente o novo requisito. A classe *Company* armazenará informações sobre outras empresas responsáveis pelo fornecimento de quaisquer materiais utilizados pela loja virtual que não sejam os próprios livros. A classe refina diretamente *Object*, pois características de versionamento ou temporalidade não serão necessárias. Neste requisito, o desenvolvedor criou um objeto de *Company* para representar no banco de dados o fabricante da embalagem utilizada nos pacotes para presente da *On-line Book Stom*. Esta situação ilustra o fato de que qualquer entidade ligada ao produto ou negócio representado no SCM_TOO, mesmo não fazendo parte diretamente do produto de software, pode ser representada e armazenada no repositório de dados, aproximando o sistema da realidade modelada. A Figura 5.8 apresenta a definição da classe *Company* e o objeto recém criado durante a implementação do requisito "*packing order as a gift*".

Naturalmente, o identificador do objeto criado para representar a companhia fornecedora das embalagens para presente é também adicionado ao relacionamento *newObjects* da mudança sendo desenvolvida.

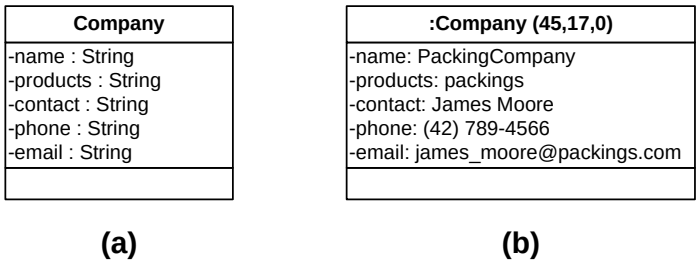


Figura 5.8: Classe *Company* (a) e sua primeira instância (b)

É bastante provável que em um ambiente real de desenvolvimento, outras versões dos arquivos envolvidos na mudança poderiam ser criadas, por exemplo no caso de a equipe de teste encontrar algum defeito inserido por esta nova funcionalidade. Casos de teste e defeitos podem ser representados também no SCM_TOO para armazenar os defeitos em um projeto e quais os artefatos afetados por cada um deles, mas esta simulação tornaria este estudo de caso desnecessariamente complexo. Assim, assumimos que somente uma versão de cada arquivo foi criada para implementar cada requisito de software.

5.3.2 Mudança da busca por preço

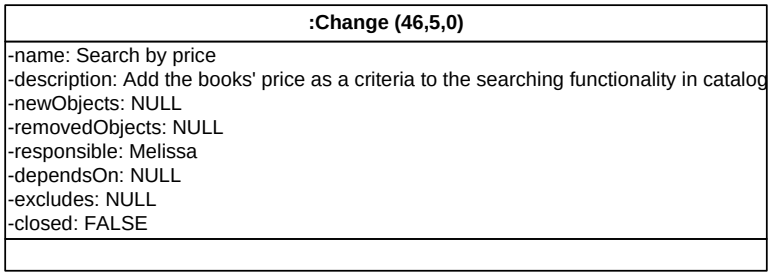


Figura 5.9: Mudança *search by price*

Conforme explicado anteriormente, o módulo catálogo da aplicação apresenta todos os livros disponíveis atualmente na loja e permite que o usuário realize buscas para encontrar produtos específicos. Esta consulta ao catálogo de livros recebe como parâmetro o nome do autor ou o título do livro procurado pelo consumidor. O requisito *search by price* consiste em adicionar uma restrição a essa busca, permitindo que o consumidor entre também a faixa de preço dos livros retornados pela consulta.

A Figura 5.9 apresenta o objeto *Change* criado para endereçar esta mudança no repositório de dados. A desenvolvedora Melissa foi designada para implementar este novo requisito.

Assim como no requisito anterior, instâncias das classes *Requirement* e *Comments* são criadas para endereçar o novo requisito e documentar adequadamente cada alteração sofrida pelo software. A Figura 5.10 apresenta estes novos objetos, cujos OIDt's são adicionados ao atributo *newObjects* do objeto *Change*.

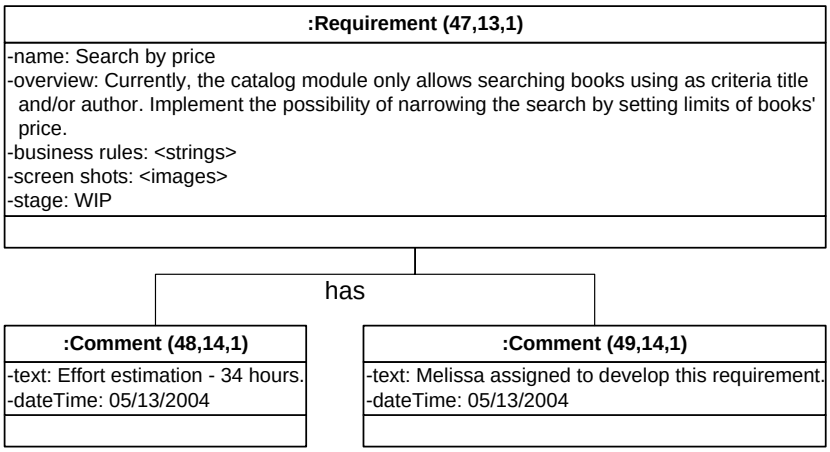


Figura 5.10: Requisito *search by price* e seus comentários

O objeto da classe *Requirement* é associado ao módulo *Catalog*, o qual será modificado para permitir a restrição da busca de acordo com o preço dos livros consultados. Quatro arquivos que compõem o módulo são modificados neste requisito, dois deles parte da camada de apresentação da aplicação. Uma página ASP é atualizada para mostrar o novo critério de restrição da busca no momento da entrada de dados, bem como uma nova versão do arquivo responsável por apresentar o resultado da busca é criado. As funções utilizadas para validar os dados entrados pelo usuário antes da consulta ser executada no banco de dados também são atualizadas. Finalmente, o arquivo que contém todas as consultas (*queries*) a serem executadas no banco de dados pela *On-Line Book Storm* também é modificado. A Figura 5.11 ilustra a implementação, apresentando as versões criadas cujos OIDt's são adicionados à propriedade *newObjects* do requisito *search by price*.

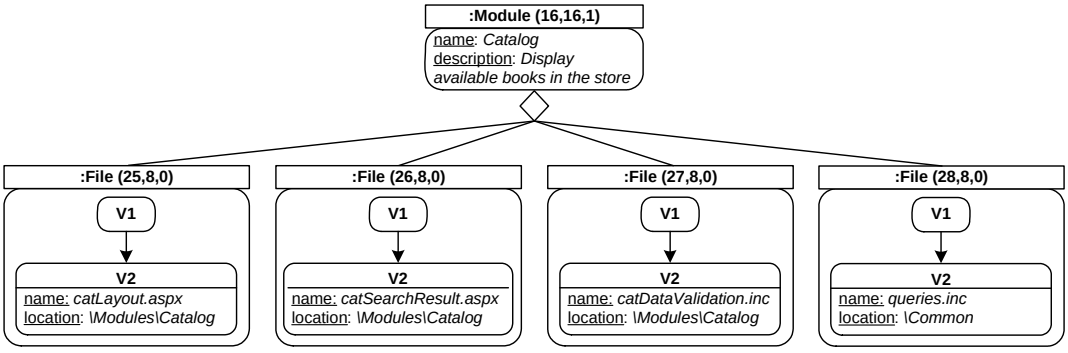


Figura 5.11: Novas versões de arquivos do módulo *Catalog*

5.3.3 Segunda *Baseline* do Produto de Software

Após o desenvolvimento dos dois requisitos solicitados estar concluído, uma nova *release* do software será disponibilizada para o cliente. Naturalmente, uma nova *baseline* do produto deve ser criada para congelar os itens de software no momento da entrega do sistema, controlando a evolução do software de maneira adequada.

No caso do SCM_TOO, uma nova versão da *baseline* anterior será instanciada mediante a execução do método de derivação definido na classe. Como explicado no capítulo anterior, a classe *Baseline* redefine este método em virtude do comportamento diferenciado de uma *baseline* em relação aos demais objetos versionados da aplicação.

O usuário do modelo pode criar uma nova versão de um objeto *baseline* de duas formas diferentes: passando como parâmetro o conjunto de mudanças (objetos *Change*) a serem incorporadas na versão atual da *baseline* ou listando explicitamente quais objetos devem ser adicionados na nova versão. Quando utilizada a segunda maneira, o usuário está assumindo totalmente a responsabilidade pelo conteúdo da nova *baseline* e, portanto, não poderá aplicar o versionamento baseado em mudanças proposto pelo SCM_TOO. A primeira abordagem, por sua vez, consiste em definir somente quais mudanças serão adicionadas à nova *baseline*, não precisando que o desenvolvedor selecione os itens de configuração individualmente no repositório de dados.

A Figura 5.12 apresenta a chamada do método de derivação no objeto *baseline* declarado anteriormente na Figura 5.4, cujo identificador é (40,6,1). O primeiro valor é o nome da nova versão da *baseline*, seguido pela descrição da mesma. O terceiro parâmetro é o conjunto de mudanças adicionadas à *baseline*, neste caso aparecem os identificadores dos objetos *Change* descritos nas seções anteriores. O último item é o usuário criador da *baseline* e na prática seria o OIDt do objeto da classe *User*, mas neste exemplo o nome Tom aparece explicitamente para simplificar o estudo de caso.

```
derive (“Baseline 2.0”,
        “Original software product plus packing order as a
         gift and search by price requirements.”,
        {(41,5,0),(46,5,0)},
        Tom)
```

Figura 5.12: Execução do método *derive* para criar a nova versão da *baseline*

O método de derivação executa algumas validações sobre os objetos *Change* recebidos como parâmetro. Por exemplo, não é permitida a inclusão de uma mudança que tenha sido desenvolvida a partir de outro objeto *baseline*, diferente do que recebeu a chamada do método. Para esta verificação, o sistema utiliza o relacionamento *baseline* do objeto *Change*. Os relacionamentos entre diferentes instâncias da classe *Change*, denominados *dependsOn* e *excludes*, também são verificados para detectar dependências entre as mudanças. No estudo de caso aqui descrito, todas essas exigências são atendidas e, portanto, o método é executado com sucesso. Caso não fosse possível a criação da nova versão da *baseline*, uma mensagem descrevendo o problema seria retornada pelo método para orientar o usuário do sistema.

A operação *derive* cria a nova versão em três etapas. Inicialmente, ela examina os objetos *Change*, coleta todos os OIDt’s criados durante a implementação das mudanças analisadas e compara com o conteúdo da *baseline* utilizada como base para a derivação. Esta etapa foi descrita anteriormente na seção 4.7.3 e é necessária para garantir que cada item de configuração tenha somente uma versão na *baseline*, detectando possíveis conflitos e a conseqüente necessidade de *merge* entre diferentes versões de um mesmo objeto.

Com a lista de OIDt’s da nova versão da *baseline* pronta, o método construtor é executado para criar uma nova instância da classe. Esta é a segunda etapa da operação de derivação e consiste em criar versões configuradas de todos os objetos presentes nas

mudanças. Posteriormente, todas as versões configuradas são promovidas para o estado *consolidated* e não podem mais ser alteradas nem removidas, conservando a propriedade de imutabilidade de uma *baseline*. Este processo foi descrito na seção 4.6.

Todos os objetos constituintes das mudanças são, portanto, transformados em versões configuradas. Como visto no capítulo anterior, a construção de uma configuração consiste na criação de novas versões, derivadas das versões originais. A lista de objetos que compõem cada uma das mudanças é, então, atualizada para conter as versões configuradas no lugar das versões originais desenvolvidas pelos usuários.

A Tabela 5.2 apresenta as duas mudanças e os identificadores das versões contidas em cada uma delas. Abaixo de cada mudança, a coluna da esquerda lista as versões originais, enquanto que a coluna da direita contém a mudança atualizada, composta somente por versões configuradas.

Tabela 5.2: Versões originais e configuradas das mudanças

***Packing order as a gift* (41,5,0)**		***Search by price* (46,5,0)**	
Versões Originais	Versões Configuradas	Versões Originais	Versões Configuradas
(42,13,1)	(90,13,1)	(47,13,1)	(91,13,1)
(43,14,1)	(90,14,1)	(48,14,1)	(91,14,1)
(44,14,1)	(90,14,1)	(49,14,1)	(91,14,1)
(22,8,2)	(90,8,2)	(25,8,2)	(91,8,2)
(23,8,2)	(90,8,2)	(26,8,2)	(91,8,2)
(24,8,2)	(90,8,2)	(27,8,2)	(91,8,2)
(45,17,0)	(90,17,0)	(28,8,2)	(91,8,2)

No exemplo de caso aqui descrito, os objetos criados para descrever os requisitos (instâncias das classes Requirements e Comments), todas as novas versões dos arquivos criados durante a implementação (nos módulos *Check-out* e *Catalog*), bem como o objeto da nova classe denominada *Company*, são adicionados à lista de objetos que compõem a nova versão da *baseline*. Todas estas versões (listadas nas duas colunas de versões configuradas na Tabela 5.2) são promovidas para o estado consolidated, impedindo que elas sejam removidas do repositório de dados.

Como as mudanças não apresentam dependências entre si e não alteram um mesmo objeto versionado, nenhum conflito é detectado e a nova versão da *baseline* é criada com sucesso.

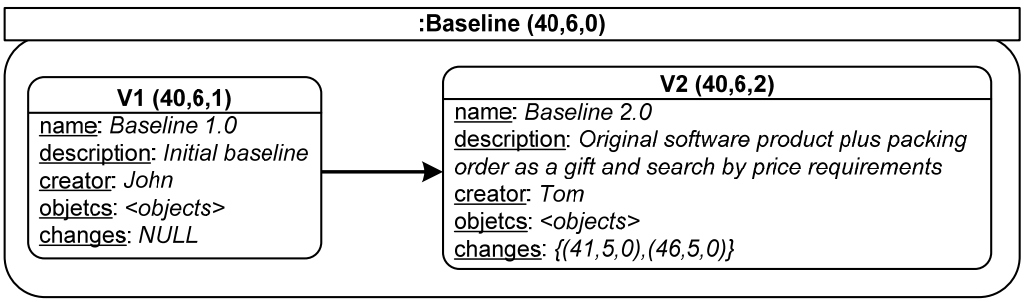


Figura 5.13: Objeto versionado *Baseline* e suas versões

A última etapa é a mais simples e consiste na finalização do processo de derivação, associando a nova versão da *baseline* à sua predecessora (propriedades

predecessor e *successor*, declaradas na classe *TemporalVersion*) e atualizando as propriedades do objeto versionado (*lastVersion*, *currentVersion*, *nextVersionNumber* e *versionCount*). As duas mudanças são fechadas para que não possam mais ser alteradas, mediante a execução do método *close*. Finalmente, o atributo *changes* da nova versão da *baseline* recebe os identificadores das mudanças, para armazenar as alterações utilizadas na sua criação e permitir a futura execução do método *getDiff*. A Figura 5.13 apresenta o grafo de versões do objeto versionado *Baseline* atualizado após a derivação.

5.3.4 Mudança para Personalizar os Pacotes dos Presentes

Após as duas mudanças desenvolvidas terem sido instaladas com sucesso no ambiente de produção, o cliente solicita o desenvolvimento de um novo requisito. Muitos clientes que usaram a nova funcionalidade de envio dos livros empacotados como presente reclamaram que a loja não disponibiliza uma forma de personalizar o pacote, seja utilizando uma cor específica ou adicionando uma mensagem de parabéns, por exemplo. Um objeto mudança denominado *customized packs for gifts* é instanciado, cuja definição aparece na Figura 5.14. Enquanto as mudanças anteriormente desenvolvidas referenciavam a *baseline* 1.0, este objeto *Change* é ligado à *baseline* 2.0 pois ela será desenvolvida a partir da nova *baseline* do sistema. Porém, o relacionamento *baseline* entre as classes *Change* e *Baseline* não aparece na figura por motivos de simplificação.

:Change (50,5,0)
-name: Customized packs for gifts -description: Provide the customer a way to customize the pack used for gifts -newObjects: NULL -removedObjects: NULL -responsible: John -dependsOn: NULL -excludes: NULL -closed: FALSE

Figura 5.14: Mudança *customized packs for gifts*

Assim como nos requisitos anteriores, diversas versões são criadas durante a implementação desta mudança. Um objeto da classe *Requirement* e dois comentários são criados, além das naturais modificações realizadas em alguns arquivos de código fonte da loja virtual. A Figura 5.15 apresenta estas novas versões criadas durante o desenvolvimento do requisito.

Assim como nos requisitos anteriores, a implementação de *customized packs for gifts* consiste, inicialmente, na criação de um objeto *Requirement* e de alguns comentários associados a ele. A codificação do requisito modifica o módulo *check-out* da aplicação, gerando novas versões de dois arquivos existentes, nomeados *chkLayout.aspx* e *chkPriceCalc.aspx*. Os dois arquivos estão na sua terceira versão, pois o requisito anteriormente desenvolvido *packing order as a gift* também alterou o conteúdo de ambos. O grafo de versões de cada objeto também contém a versão configurada de cada um, criadas durante a construção da segunda *baseline* da aplicação. As versões configuradas, porém, sempre serão folhas no grafo, portanto a derivação de uma nova versão ocorre a partir da predecessora da versão configurada. Para permitir a customização dos pacotes usados nos presentes, um novo objeto da classe *File* é

instanciado e, naturalmente, possui apenas uma versão. Como também faz parte do *check-out*, o objeto é ligado ao módulo mediante seu relacionamento de agregação. Naturalmente, os identificadores de todos os objetos apresentados na figura são adicionados à lista de novos objetos que compõem a mudança sob desenvolvimento.

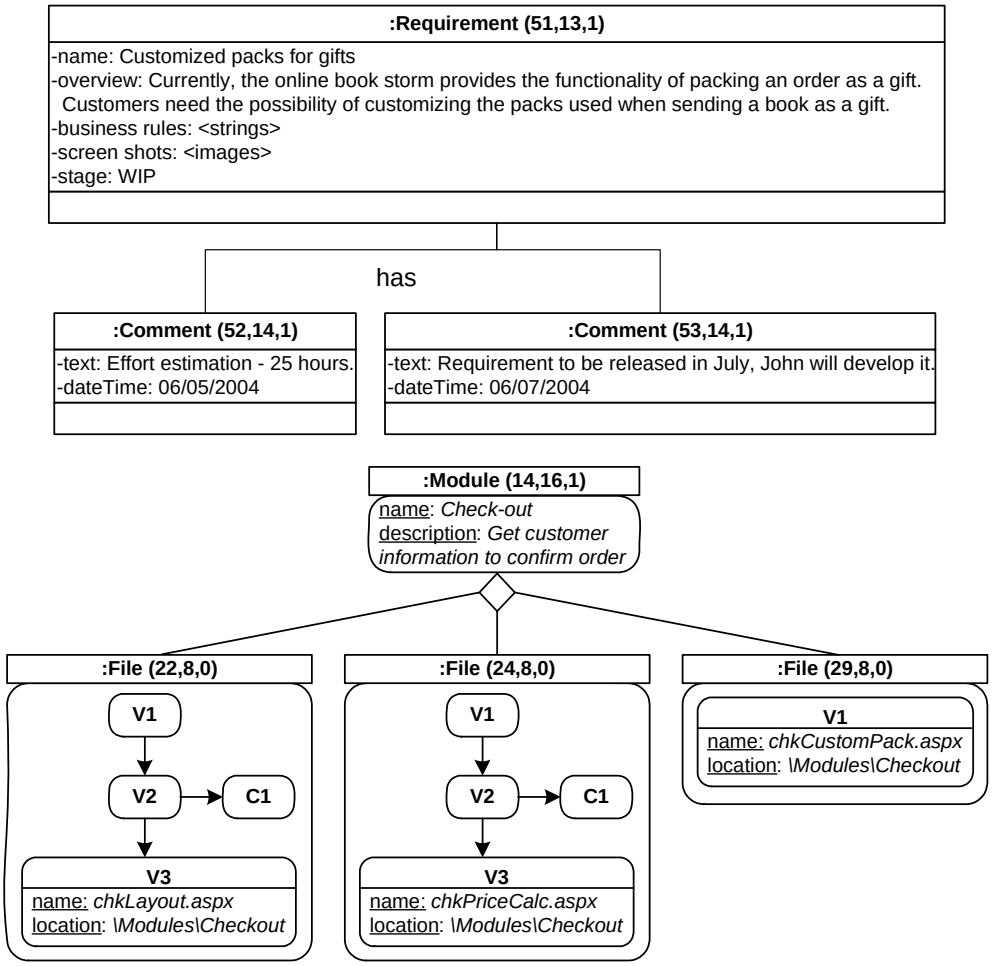


Figura 5.15: Versões que implementam o requisito *customized packs for gifts*

O requisito *customized packs for gifts* depende da mudança previamente desenvolvida denominada *packing order as a gift*, pois seria impossível disponibilizar na loja virtual a personalização dos pacotes sem que a funcionalidade de enviar os livros como presente esteja também incluída na aplicação. Para representar esta informação no repositório de dados, o desenvolvedor associa os dois objetos *Change* mediante o relacionamento *dependsOn*. Dessa forma, o sistema não permitirá que uma *baseline* seja criada incluindo o novo requisito sem que o anterior seja também adicionado à aplicação.

5.3.5 Mudança para Corrigir o Cálculo dos Preços

Durante a implementação do requisito *customized packs for gifts*, o cliente da aplicação (proprietário da loja virtual) detecta um defeito no sistema. No módulo *check-out*, uma série de cálculos são realizados para determinar o preço final do pedido de

cada usuário, basendo-se em um conjunto de dados como endereço do consumidor, cotação do dólar no caso da compra de produtos importados, entre outros. A aplicação atualmente está aplicando taxas erradas quando uma determinada combinação destes dados é considerada. Como este defeito ocasiona preços errados na loja, ele precisa ser corrigido e instalado em produção o mais rápido possível.

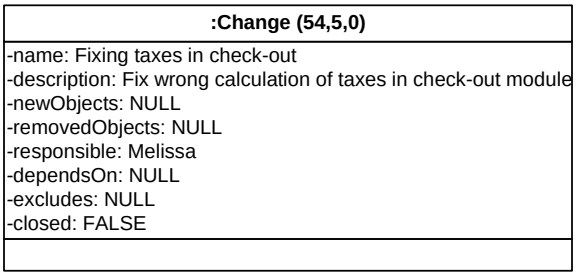


Figura 5.16: Mudança *fixing taxes in check-out*

Para implementar a correção do defeito, um novo objeto da classe *Change* é instanciado e apresentado na Figura 5.16 – *fixing taxes in check-out*. Esta mudança é ligada à *baseline* 2.0 pois ela contém as exatas versões do código da aplicação atualmente utilizadas no ambiente de produção.

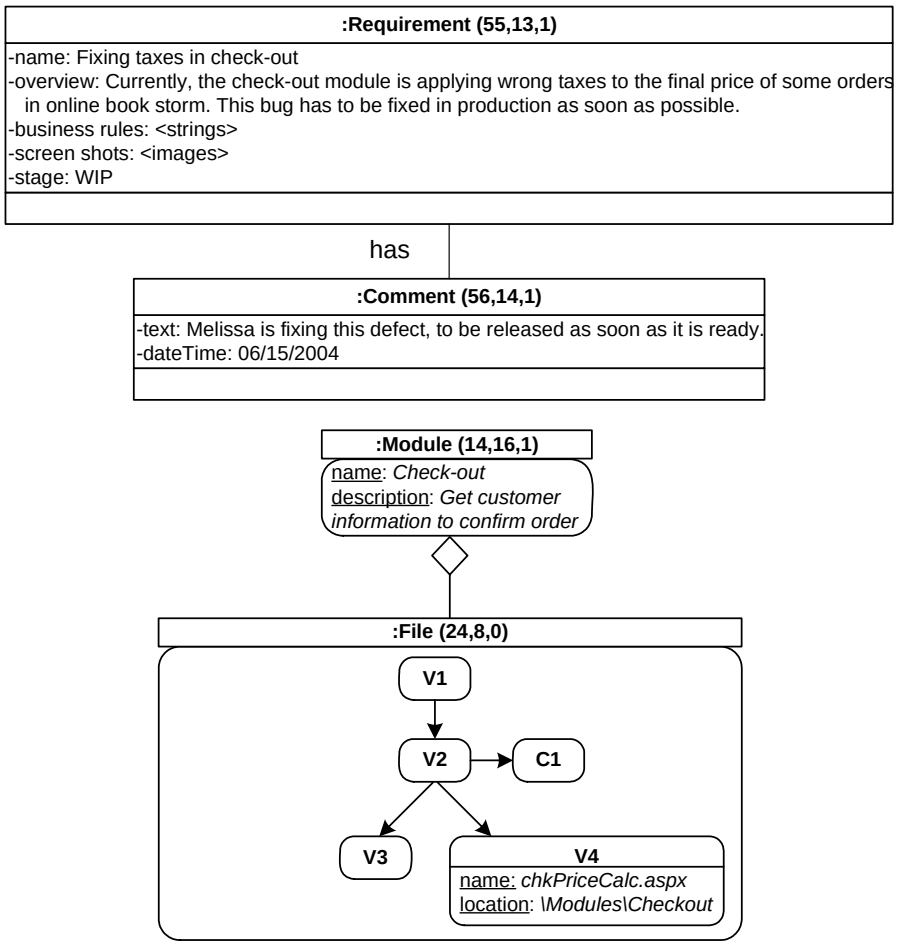


Figura 5.17: Versões que implementam o requisito *fixing taxes in check-out*

Para documentar adequadamente as modificações realizadas no software, um objeto da classe *Requirement* também é instanciado, associado a um comentário criado no momento inicial de desenvolvimento. A codificação da mudança é bastante simples e consiste na geração de uma nova versão do arquivo *chkPriceCalc.aspx*. Todos estes objetos são apresentados na Figura 5.17.

O grafo de versões do arquivo *chkPriceCalc.aspx* merece uma análise mais cuidadosa. Como a correção do defeito precisa ser desenvolvida utilizando o código atualmente em produção, a nova versão criada neste requisito deriva V2, e não V3 que é a versão mais atual do arquivo. Esta definição sobre qual versão a ser utilizada como base, porém, deve ser realizada automaticamente pela ferramenta de gerência de configuração de software. O processo de criação de uma nova versão de qualquer objeto do produto de software considera a lista de objetos que compõem a *baseline* definida como início do desenvolvimento, neste caso a *baseline* 2.0 associada ao objeto *Change*. A versão do arquivo *chkPriceCalc.aspx* que faz parte da *baseline* é exatamente a versão configurada cujo identificador é (90,8,2), conforme a Tabela 6.2. Como visto anteriormente, versões configuradas sempre são folhas no grafo de versões, portanto a nova versão criada para corrigir o defeito deriva V2, cujo conteúdo é igual ao da versão configurada. Este processo cria uma variante em relação à versão mais atual do objeto, conforme a Figura 5.17.

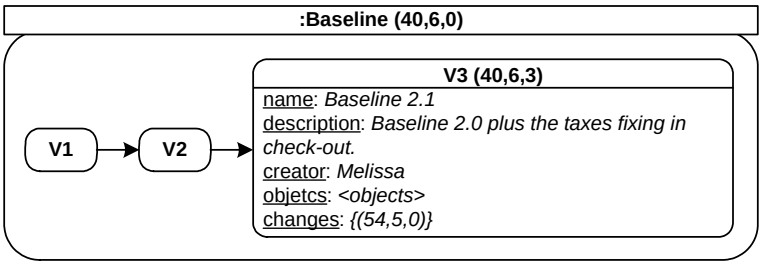


Figura 5.18: Objeto versionado *Baseline* e sua terceira versão

Uma vez finalizado o desenvolvimento da correção do defeito, o novo código deve ser instalado no ambiente de produção, gerando uma nova versão da *baseline* do produto de software. Porém, o requisito *customize packs for gifts* ainda não está pronto e não pode fazer parte desta nova *release*. Assim, a nova *baseline* inclui somente a mudança *fixing taxes in check-out* e é chamada de *baseline* ínterim, por ser somente uma versão de manutenção e não adicionar novas funcionalidades ao sistema. A Figura 5.18 apresenta o grafo de versões do objeto versionado *Baseline*, detalhando os atributos da terceira versão recém criada na correção do defeito.

5.3.6 Nova *Baseline* Unificando Dois Ramos de Desenvolvimento

Após a conclusão do desenvolvimento da mudança *cusomized packs for gifts*, uma nova *baseline* do produto de software deve ser criada para instalar o requisito no ambiente de produção. Como o código atualmente em produção está armazenado no repositótio de dados na terceira versão do objeto *Baseline*, o método mais simples seria derivar esta versão incluindo o objeto *Change* correspondente. Quando executada esta ação no SCM_TOO, o modelo não geraria a nova versão e retornaria para o usuário uma mensagem explicando que, como a mudança foi desenvolvida utilizando como base a segunda versão da *baseline*, ela não pode ser simplesmente incluída na atual versão do

objeto. Esta verificação é feita pelo sistema considerando o relacionamento do objeto *Change* com a *Baseline* definida no início do desenvolvimento do requisito.

```
derive (“Baseline 3.0”,
        “Baseline 2.0 plus customized packs for gifts
         requirement and fixing taxes in check-out.”,
        {(50,5,0),(54,5,0)},
        Tom)
```

Figura 5.19: Chamada do método *derive* em V2 para criar a nova versão da *baseline*

A forma correta de gerar a nova *baseline* do software é derivar a versão da *baseline* usada como base no desenvolvimento da mudança, no caso V2. Consultando o atributo *changes* da *baseline*, o usuário facilmente percebe que a mudança *fixing taxes in check-out* não faz parte da versão e, portanto, precisa ser adicionada nesta nova versão do objeto. A Figura 5.19 apresenta a chamada ao método *derive* de V2 passando como parâmetro as duas mudanças a serem incluídas na geração da nova *baseline*.

O nome da versão criada segue o padrão adotado pela equipe de desenvolvimento, numerada como 3.0 por ser uma *baseline* formal cujo conteúdo contém um novo requisito funcional. Os parâmetros contêm também a descrição textual da *baseline*, os identificadores dos objetos *Change* que devem ser incluídos na nova versão e o usuário do sistema de GCS responsável pela criação da *baseline*.

Como explicado anteriormente, a geração de uma nova versão da *baseline* implica em uma série de verificações quanto ao seu conteúdo. Inicialmente, as dependências individuais dos objetos *Change* incluídos são examinadas pelo SCM_TOO. A mudança *customized packs for gifts* depende do requisito *packing order as a gift*, de acordo com o relacionamento *dependsOn*. Como esta mudança foi previamente incluída na versão da *baseline* utilizada como base para o desenvolvimento do requisito *customized packs for gifts* (informação registrada no atributo *Changes* da *baseline* 2.0), o sistema permite a inclusão da mudança e passa ao próximo passo na geração da *baseline*.

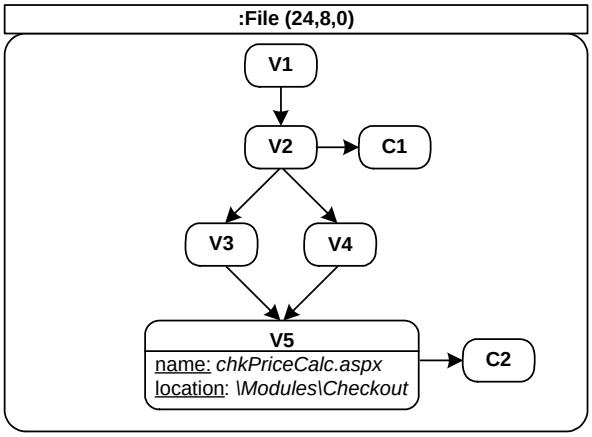


Figura 5.20: Grafo de versões do objeto *chkPriceCalc.aspx*

Examinando as dependências entre as mudanças a serem incluídas na nova versão da *baseline*, o SCM_TOO percebe que ambas modificam o mesmo arquivo, no caso *chkPriceCalc.aspx*. Conforme o grafo de versões deste objeto, apresentado anteriormente na Figura 5.17, a versão criada pela mudança *fixing taxes in check-out*

(V4) é uma variante em relação à versão gerada pelo requisito *customized packs for gifts* (V3), caracterizando um conflito. A versão deste arquivo a ser incluída na nova *baseline* precisa incorporar as modificações desenvolvidas por ambas mudanças, determinando a necessidade da operação de *merge* entre as duas versões. O algoritmo descrito no capítulo anterior e apresentado na Figura 4.15 é acionado na derivação de uma nova versão do arquivo *chkPriceCalc.aspx*.

Após unificar os dois ramos de desenvolvimento, o método *buildConfiguration* é executado para gerar uma nova configuração do arquivo, a fim de adicionar o conteúdo da versão à nova *baseline*. A versão C2 é derivada a partir de V5, atualizando o grafo de versões do objeto como ilustra a Figura 5.20.

O grafo de versões do objeto *Baseline*, por sua vez, é apresentado na Figura 5.21. A versão 3.0 (V4) é uma variante em relação à versão 2.1 (V3), mas isto não é um problema pois a única mudança incluída na *baseline* 2.1 foi também adicionada à *baseline* 3.0.

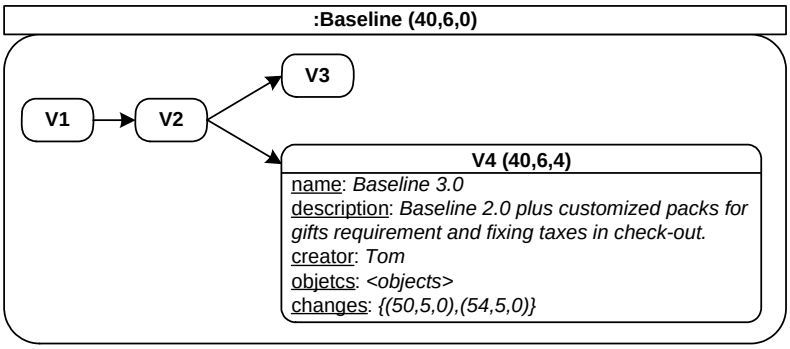


Figura 5.21: Grafo de versões do objeto *Baseline*

A derivação da versão de uma *baseline* a partir de duas versões do objeto versionado, caracterizando um *merge* entre duas *baselines*, não está previsto no SCM_TOO. Apesar de ser uma funcionalidade interessante, a construção de novas *baselines* utilizando o conteúdo de duas *baselines* existentes pode ser facilmente executada mediante a manipulação correta dos objetos *Change* previamente instanciados, como exemplificado nesta seção. A operação *merge* entre duas *baselines* resultaria na adição de uma complexidade desnecessária ao SCM_TOO, cujo princípio fundamental é implementar de maneira simples os conceitos de gerência de configuração de software no paradigma OO.

5.3.7 Considerações Finais

Este capítulo apresentou a evolução de uma aplicação sob o controle do SCM_TOO, exemplificando a utilização do versionamento baseado em mudanças proposto pelo modelo. Um recurso muito interessante é a possibilidade da utilização de diferentes níveis de visões do produto de software, dependendo do papel de cada usuário no projeto de desenvolvimento.

Por exemplo, um programador precisa trabalhar diretamente com os arquivos que compõem cada um dos módulos da aplicação e a forma como estes estão estruturados dentro do produto de software. A Figura 5.22 apresenta alguns arquivos que compõem o código fonte do módulo *check-out* e suas versões, fornecendo informações importantes

para que o desenvolvedor entenda a estrutura da aplicação e como os ítens evoluem com o tempo.

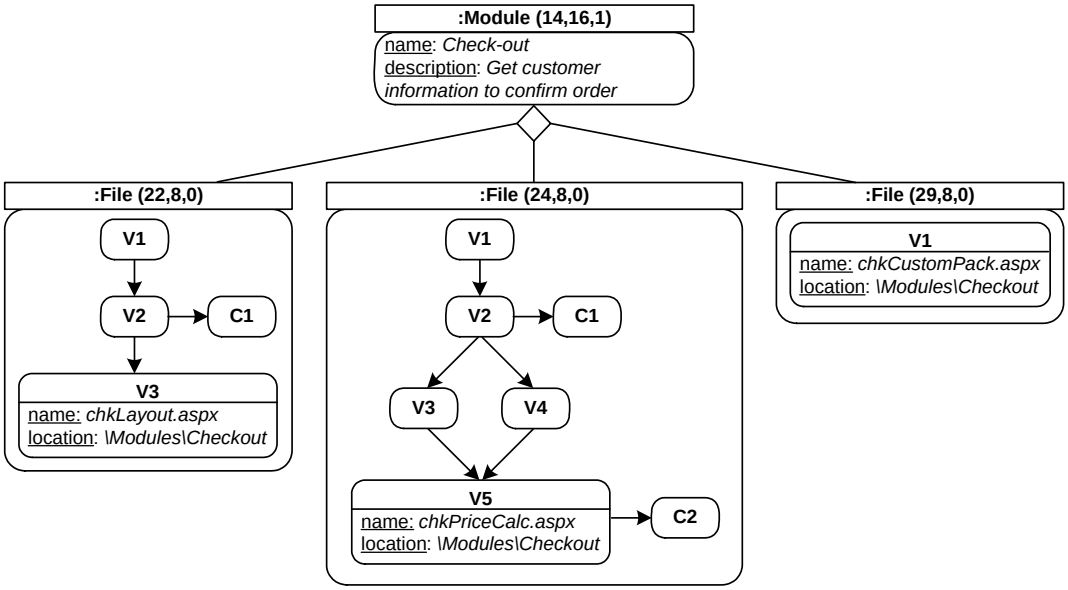


Figura 5.22: Evolução do software vista por um programador

Por sua vez, o gerente do projeto não precisa estar ciente de quantos arquivos foram modificados na implementação de cada um dos requisitos. É muito mais interessante visualizar somente as mudanças desenvolvidas ao longo do tempo e como estas foram utilizadas na criação de novas *baselines* e *releases* para o cliente. As linhas tracejadas na Figura 5.23 indicam quais mudanças foram utilizadas no momento da criação de cada versão da *baseline*. O relacionamento de dependência entre os objetos *Change* também é representado na figura. Com base nestas informações, o gerente pode compreender facilmente a evolução da aplicação, além de avaliar a produtividade de cada membro da equipe de acordo com as associações existentes entre os objetos do produto de software e os desenvolvedores responsáveis por cada alteração implementada.

Estas diferentes visões do produto de software podem ser facilmente construídas utilizando uma interface gráfica na visualização dos objetos armazenados no repositório e de uma linguagem de consulta de alto nível, como a TVQL [MOR 2002], [ZAU 2002]. O próximo capítulo apresentará as conclusões do trabalho desenvolvido, citando alguns futuros trabalhos para complementar o modelo aqui apresentado.

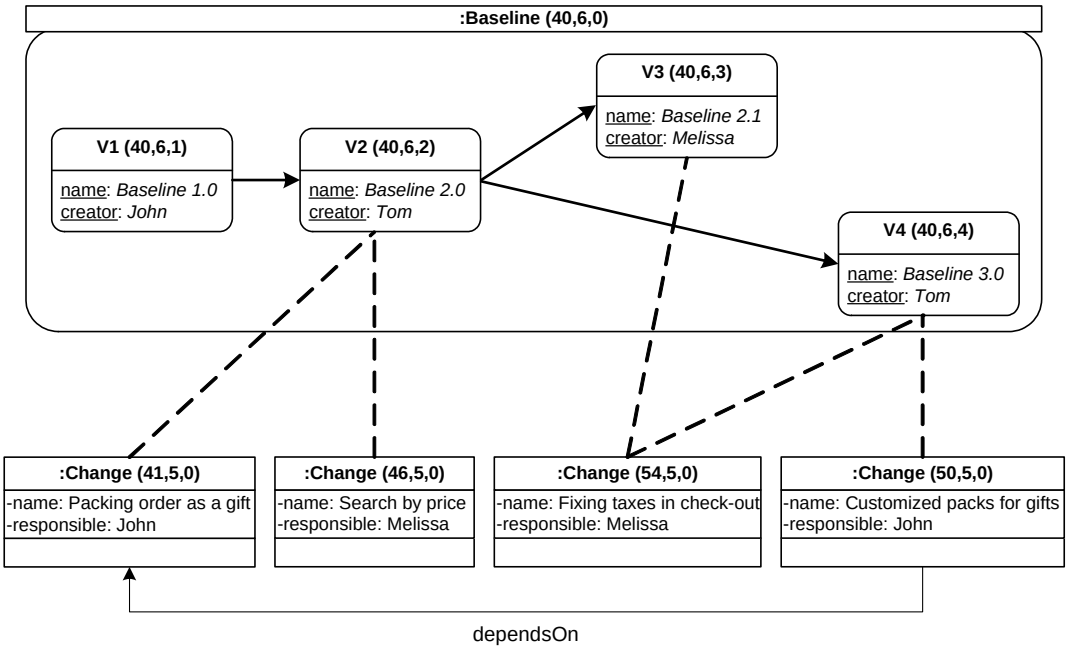


Figura 5.23: Evolução do software vista pelo gerente de projeto

6 CONCLUSÕES

O objetivo deste trabalho consiste no desenvolvimento de um modelo de dados conceitual específico para gerenciar configurações de software. Princípios e técnicas consagrados pela engenharia de software devem ser aplicados, evitando que o modelo introduza novos conceitos que venham a tornar inviável a sua implementação e utilização.

O Modelo Temporal de Versões disponibilizou todos os conceitos básicos de versionamento e temporalidade necessários na definição do novo modelo. Uma complementação destas funcionalidades básicas deveria ser desenvolvida para identificar e controlar de maneira eficiente as alterações realizadas em um produto de software ao longo de sua evolução.

O paradigma da orientação a objeto foi utilizado com o intuito de fornecer um conjunto de mecanismos eficientes que permitam ao engenheiro de software modelar a sua aplicação, versionar os itens de configuração que a compõem e aplicar práticas avançadas de desenvolvimento de software (como a construção de *baselines*) de maneira uniforme e centralizada em um único repositório de dados.

Este capítulo apresenta as considerações finais deste trabalho, iniciando pela implementação do modelo em um banco de dados comercial, destacando em seguida suas principais contribuições para a comunidade de engenharia de software e encerrando com a identificação de alguns trabalhos futuros que venham a complementar o trabalho aqui desenvolvido.

6.1 Implementação do SCM_TOO

Naturalmente, um modelo conceitual não pode ser corretamente validado e consolidado sem sua devida implementação e utilização em um ambiente real de computação. Como o desenvolvimento de um SGBD (Sistema Gerenciador de Banco de Dados) completamente novo é uma tarefa muito cara e complexa, a estratégia mais adequada consiste na implementação do modelo sobre um produto comercial disponível.

Como o principal objetivo de implementar o SCM_TOO reside na sua validação, aprimoramento e conseqüente consolidação, a implementação do modelo em um banco de dados relacional foi descartada logo no início do projeto pois certamente exigiria a

modificação de várias funcionalidades baseadas na orientação a objeto. Entre os bancos de dados OO disponíveis atualmente, o escolhido para suportar a implementação do modelo foi o *Caché* da *Intersystems* [INT 2004], destacando-se pelas ferramentas de desenvolvimento agregadas ao produto e sua extensa documentação e suporte, além da disponibilidade de licenças para a utilização do software. Além disso, o *Caché* apresenta uma arquitetura unificada de dados que disponibiliza o acesso a sua base via objetos ou instruções padrões SQL, permitindo futuras integrações com outras linguagens de programação e ferramentas relacionais. A Figura 6.1 ilustra o dicionário unificado do SGBD, sempre atualizado de forma transparente mediante o uso de classes e objetos ou instruções DDL (*Data Definition Language*), quando utilizado o paradigma relacional.

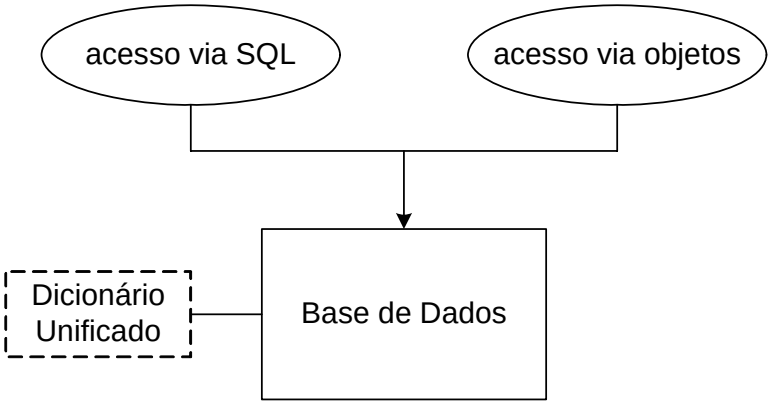


Figura 6.1: Arquitetura unificada do *Caché*

O projeto de implementação foi dividido em três etapas distintas:

- implementação da hierarquia de classes e das características de versionamento do modelo;
- implementação das características temporais nas classes anteriormente criadas;
- implementação do versionamento baseado em mudanças.

A primeira etapa foi implementada por Robert e descrita em [ROB 2004], permitindo que todas as características básicas de versionamento definidas pelo SCM_TOO sejam utilizadas em uma ferramenta comercial. Neste primeiro protótipo, os mais diversos tipos de elementos de software podem ser criados e modificados diretamente no SGBD. As classes *File* e *User* também foram corretamente construídas, disponibilizando o modelo *check-out/check-in* e integrando o repositório de dados com as áreas de trabalho dos programadores.

Esta implementação possibilita que um produto de software completo seja criado e tenha sua evolução ao longo do tempo efetivamente controlada pelo SCM_TOO. Um engenheiro de software pode modelar sua aplicação utilizando classes e relacionamentos, como o estudo de caso apresentado no capítulo anterior. Objetos versionados e não-versionados podem ser instanciados para representar os itens de configuração e relacionamentos (agregações e associações) podem ser criados para descrever de maneira adequada o produto sob desenvolvimento. Arquivos previamente existentes somente no sistema operacional podem ser instanciados no SGBD, podendo ser versionados e relacionados com os outros objetos da aplicação.

6.2 Contribuições

Diversos conceitos e idéias envolvidos na engenharia de software e na disciplina de gerência de configuração foram discutidos durante a elaboração deste trabalho. Três características do SCM_TOO devem ser destacadas como importantes contribuições geradas na construção deste modelo de dados.

O primeiro importante aspecto é a forma de representação do produto de software no SCM_TOO. A tentativa de visualizar e manipular os elementos de software em um nível de abstração mais alto do que simplesmente arquivos distribuídos dentro de diretórios é bastante antiga, mas até hoje não foi proposto nenhum modelo avançado e eficiente que pudesse ser implementado comercialmente e utilizado em ambientes práticos de desenvolvimento. A flexibilidade proporcionada pelo uso do paradigma OO permite que arquivos, módulos de programas, requisitos de software e casos de teste, por exemplo, sejam representados uniformemente, possibilitando que todos estes objetos sejam considerados itens de configuração e armazenados no mesmo repositório. Nos ambientes de GCS atuais, somente arquivos são itens de configuração, e outras ferramentas foram desenvolvidas para manter todos os outros elementos criados em um projeto, como requisitos e defeitos. A unificação de diferentes tipos de artefatos de software utilizando um modelo de dados, como o orientado a objetos, pode aumentar a eficiência e estender as funcionalidades das ferramentas de gerência de configuração de software.

A temporalidade adicionada aos tradicionais métodos de versionamento pode ser destacada como a segunda importante contribuição do modelo. Atualmente, ferramentas de GCS associam *timestamps* aos itens de configuração e suas versões, mas dificilmente aplicam quaisquer funcionalidades ou restrições quanto a esses dados de tempo. A utilização da dimensão temporal mais efetivamente, como ocorre no SCM_TOO, é um passo natural na evolução das ferramentas de gerência de configuração de software.

A terceira e principal contribuição do SCM_TOO é o versionamento baseado em mudanças proposto pelo modelo. A identificação e manipulação de cada alteração no produto de software como uma entidade lógica de primeiro nível permite o controle da evolução do sistema de maneira limpa e eficiente, associando dados semânticos a cada mudança desenvolvida. Relacionamentos e métodos descritos neste trabalho permitem a integração destas alterações com as *baselines* criadas ao longo da evolução de um produto. Algumas propostas anteriores definiram mudanças como a entidade principal de um modelo de dados para GCS, abstraindo completamente como cada alteração é implementada fisicamente, ou seja, quais artefatos de software são modificados. Esta abordagem permanece distante de um ambiente real de desenvolvimento, onde programadores e analistas envolvem-se diretamente na elaboração de novas versões de cada item do produto de software. O SCM_TOO, por sua vez, identifica cada mudança individualmente e disponibiliza técnicas para aplicar estas alterações na criação de novas *baselines*, mas ainda permite que programadores interajam diretamente com as versões dos arquivos que compõem o código do sistema e implementam as mudanças previamente definidas. Esta abordagem inovadora confere ao modelo muita flexibilidade, sem aumentar a complexidade para o usuário final da aplicação.

6.3 Trabalhos Futuros

Após a definição completa do SCM_TOO ter sido apresentada, alguns trabalhos podem ser identificados com o intuito de complementar o modelo, considerando tanto algumas contribuições conceituais quanto a implementação de ferramentas baseadas nesta proposta. As próximas seções enumeram algumas destas propostas.

6.3.1 Linguagem de Consulta

Conforme citações anteriores e uma rápida explicação na seção 3.5, a linguagem de consulta TVQL (*Temporal Versioned Query Language*) foi desenvolvida para recuperar de forma simples e eficiente versões e dados temporais de um conjunto de objetos criados utilizando o TVM. Mantendo a estrutura padrão da SQL, a TVQL adicionou diversas expressões e operadores para facilitar a busca e a filtragem de dados temporais e versionados.

Como o seu desenvolvimento foi totalmente baseado na tradicional SQL relacional, a TVQL originalmente não disponibiliza palavras chaves ou instruções específicas para manusear objetos e relacionamentos entre estes. Com o objetivo de preencher esta lacuna, Galante define em sua tese [GAL 2003] expressões de caminho que permitem recuperar versões de objetos temporais, adotando uma estrutura similar à OQL (*Object Query Language*).

O próximo passo consiste, naturalmente, na criação de um conjunto de operadores e expressões que permitam a consulta dos objetos e classes criados no SCM_TOO para endereçar as funcionalidades de gerência de configuração de software. Uma linguagem de consulta que atenda o versionamento baseado em mudanças implementado no modelo deve conter expressões específicas para retornar e filtrar objetos do tipo *Change* e *Baseline*, permitindo que diferentes visões do produto de software – como as vistas nas figuras 6.22 e 6.23 – sejam construídas de maneira rápida e eficiente. Facilidades para construir consultas sobre os arquivos armazenados no repositório também podem ser disponibilizadas, por exemplo considerando o histórico de *check-out/check-in*'s realizados ou até mesmo o conteúdo do atributo *contents* de cada objeto da classe *File*.

Alguns ajustes quanto à temporalidade devem ser feitos na TVQL, adequando esta à dimensão temporal simples do SCM_TOO. Modificações na linguagem de consulta também poderiam incluir um tratamento diferenciado aos atributos e relacionamentos comuns.

A definição de uma linguagem de consulta de alto nível para o SCM_TOO será um diferencial do modelo em relação às ferramentas de GCS disponíveis atualmente, na maioria das quais a recuperação dos elementos de software é realizada manualmente pelo usuário ou utilizando um precário sistema de busca baseado nos atributos básicos dos arquivos, como nome e data de criação.

6.3.2 Implementação Completa e Validação do SCM_TOO

A seção 6.1 apresentou resumidamente o projeto de implementação do SCM_TOO em um banco de dados comercial e as suas características já desenvolvidas em um trabalho paralelo a este [ROB 2004]. As etapas seguintes da implementação são:

- implementação das características temporais nas classes anteriormente criadas;
- implementação do versionamento baseado em mudanças.

Após as etapas acima terem sido concluídas, será possível utilizar o SCM_TOO em um ambiente real de desenvolvimento de software e coletar uma série de dados a respeito desta utilização, de forma a medir a produtividade do modelo e o nível de satisfação dos desenvolvedores envolvidos no projeto. Neste momento será possível identificar efetivamente quaisquer pontos negativos do modelo e as oportunidades de melhoria deste trabalho, bem como saber a real extensão da sua contribuição à disciplina de gerência de configuração de software.

6.3.3 Ambiente Completo de GCS

Segundo Estublier [EST 2000], um sistema gerenciador de configurações de software precisa fornecer serviços nas seguintes áreas:

- gerenciamento do repositório de componentes;
- controle da área de trabalho dos desenvolvedores;
- suporte e controle de processo.

O escopo do SCM_TOO é exatamente o primeiro item, responsável por organizar e manter eficientemente todos os elementos que compõem o produto de software e suas diversas configurações. Alguns aspectos da interação do repositório com a área de trabalho – o segundo item – foram também endereçados durante a modelagem da classe *File* e a aplicação do modelo *check-out/check-in*, cuja discussão encontra-se na seção 4.6. Porém, nada sobre o controle do processo de desenvolvimento de software é abordado pelo SCM_TOO, cujo escopo restringe-se ao modelo de dados do produto de software.

Uma ferramenta de gerência de configuração de software que venha a utilizar o SCM_TOO precisa disponibilizar uma série de funcionalidades específicas a fim de aproveitar todas as características definidas neste trabalho. Após a implementação do SCM_TOO em um SGBD comercial, é necessário que um ambiente completo de GCS seja desenvolvido sobre o modelo, criando métodos de controle de processo e interfaces adequadas entre os usuários do sistema e o banco de dados em questão. A Figura 6.2 ilustra esta situação.

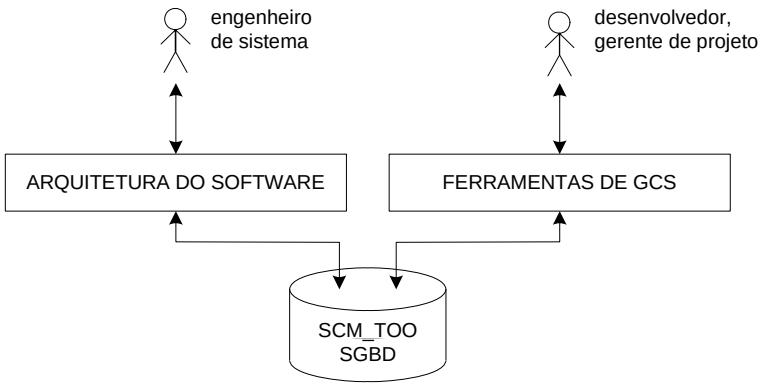


Figura 6.2: Ambiente de GCS sobre o SGBD

O SGBD utilizado na implementação do modelo aparece na parte inferior da figura, onde todas as classes definidas neste trabalho devem estar corretamente codificadas para permitir o armazenamento e manipulação de objetos versionados, características temporais, comuns, arquivos e o versionamento baseado em mudanças. Sobre o banco de dados, temos dois ambientes de desenvolvimento:

- Arquitetura do software: onde todas as classes e seus relacionamentos são criados – podemos dizer que os metadados do produto de software são mantidos por esta ferramenta. Ela geralmente é utilizada somente antes do projeto começar, como exemplificado na Figura 5.2 do estudo de caso. Algumas vezes pode ser necessário alterar o esquema do software durante o desenvolvimento, como apresentado na seção 5.3.1, quando uma nova classe foi criada para endereçar a implementação de um requisito de projeto.
- Ferramentas de gerência de configuração de software: um ambiente de GCS tradicional, disponibilizando aos desenvolvedores um conjunto de funcionalidades para visualizar e modificar os itens de configuração armazenados no repositório. Além de consultar todos os elementos do produto de software, nesta ferramenta os usuários também têm acesso aos métodos *check-out/check-in* e à construção de configurações e *baselines* do sistema. Uma interface amigável deve ser disponibilizada aos desenvolvedores, de forma a facilitar o uso do versionamento baseado em mudanças presente no SGBD, podendo inclusive automatizar a criação de objetos *Change* e a adição de novas versões a estas mudanças. Além disso, esta ferramenta deve oferecer uma série de funcionalidades relativas ao suporte de processo, por exemplo definindo diferentes papéis de usuários, o ciclo de vida da aplicação, restrições de acesso em cada fase do projeto de desenvolvimento, etc.

REFERÊNCIAS

[BER 90] BERLINER, B. CVS II: Parallelizing Software Development. In: USENIX TECHNICAL CONFERENCE, 1990, Washington, DC. **Proceedings…** Washington, DC: [s.n.], 1990.

[BIL 90] BILIRIS, A. Modeling Design Object Relationships in PEGASUS. In: INTERNATIONAL CONFERENCE ON DATA ENGINEERING, 6., 1990, Los Angeles. **Proceedings…** Los Alamitos: IEEE Computer Society, 1990. p.228-236

[BOE 88] BOEHM, B. W.; PAPACCIO, P. N. Understanding and Controlling Software Costs. **IEEE Trans. Software Engineering**, New York, v.14, n.10, p.1462-1477, 1988.

[BOU 88] BOUDIER, G. et al. An Overview of PCTE and PCTE+. In: ACM SIGSOFT/SIGPLAN SOFTWARE ENGINEERING SYMPOSIUM ON PRACTICAL SOFTWARE DEVELOPMENT ENVIRONMENTS, 3., 1988, Boston. **Proceedings…** [S.l. : s.n.], 1988. p.248-257.

[COM 2004] COMPUTER ASSOCIATES. **AllFusion Harvest Change Manager**. Disponível em <http://www3.ca.com/Solutions/Overview.asp?ID=255&TYPE=S>. Acesso em: out. 2004.

[CON 97] CONRADI, R.; WESTFECHTEL, B. Towards a Uniform Version Model for Software Configuration Management. In: INTERNATIONAL CONFERENCE ON SOFTWARE ENGINEERING, SCM, 7., 1997, Boston. **Software Configuration Management**: proceedings. Berlin: Springer-Verlag, 1997. p.1-17. (Lecture Notes in Computer Science, v.1235).

[CON 98] CONRADI, R.; WESTFECHTEL, B. Version Models for Software Configuration Management. **ACM Computing Surveys – CSUR**, New York, v.30, p.232-282, 1998.

[CON 99] CONRADI, R.; WESTFECHTEL, B. SCM: Status and Future Challenges. In: INTERNATIONAL SYMPOSIUM ON SYSTEM CONFIGURATION MANAGEMENT, SCM, 9., 1999, Toulouse, France. **Software Configuration Management**: proceedings. Toulouse: Springer-Verlag, 1999.

[DAR 91] DART, S. Concepts in Configuration Management Systems. In: INTERNATIONAL WORKSHOP ON SOFTWARE CONFIGURATION MANAGEMENT, SCM, 3., 1991, Trondheim, Norway. **Software Configuration Management**: proceedings. New York: ACM Press, 1991. p.1-18.

[DIT 86] DITTRICH, K.; GOTTHARD, W.; LOCKEMANN, P. DAMOKLES, a Database System for Software Engineering Environments. In: INTERNATIONAL WORKSHOP ON PROGRAMMING ENVIRONMENTS, 1986, Trondheim, NO. **Proceedings…** Berlin: Springer-Verlag, 1986. p.353-371. (Lecture Notes in Computer Science, 244).

[EST 94] ESTUBLIER, J.; CASALLAS, R. The Adele Configuration Manager. In: TICHY, W. F. (Ed.): **Configuration Management, Trends in Software**. New York: Wiley, 1994. v.2, p.99-134.

[EST 2000] ESTUBLIER, J. Software Configuration Management: A Road Map. In: INTERNATIONAL CONFERENCE ON SOFTWARE ENGINEERING, 22., 2000, Limerick, Ireland. **The Future of Software Engineering**. New York: ACM Press, 2000. p.279-289.

[EST 2002] ESTUBLIER, J. Impact of the Research Community On the Field of Software Configuration Management. **Software Engineering Notes**, New York, v.27, n.5, p.31-39, 2002.

[FRU 99] FRÜHAUF, K.; ZELLER, A. Software Configuration Management: State of the Art, State of the Practice. **Software Engineering Notes**, New York, v. 14, n. 7, Nov. 1999. Trabalho apresentado no International Symposium on System Configuration Management, SCM, 9., 1999, Toulouse, France.

[GAL 2003] GALANTE, R.M. **Modelo Temporal de Versionamento com Suporte à Evolução de Esquemas**. 2003. 150f. Tese (Doutorado em Ciência da Computação) – Instituto de Informática, UFRGS, Porto Alegre.

[GOL 95] GOLENDZINER, L. **Um Modelo de Versões para Banco de Dados Orientados a Objetos**. 1995. Tese (Doutorado em Ciência da Computação) – Instituto de Informática, UFRGS, Porto Alegre.

[INT 2004] INTERSYSTEMS. **Caché – Post-relational database**. Disponível em: <http://www.intersystems.com/cache/index.html>. Acesso em: out. 2004.

[KAK 96] KAKOUDAKIS, I.; THEODOULIDIS, B. **The Tau Temporal Object Model**. UK: Department of Computation, UMIST, 1996.

[LAM 91] LAMB, C.; LANDIS, G.; ORENSTEIN, J.; WEINREB, D. The ObjectStore Database System. **Communications of the ACM**, New York, v.34, n.10, p.50-63, 1991.

[LEB 94] LEBLANG, D. The CM Challenge: Configuration Management That Works. In: TICHY, W. F. (Ed.). **Configuration Management, Trends in Software**. New York: Wiley, 1994. v.2, p.1-38.

[LEO 2000] LEON, A. **A Guide to Software Configuration Management**. Norwood, EUA: Artech House Computer Library, 2000.

[MEI 01] MEI, H.; ZHANG, L.; YANG, F. A Software Configuration Management Model for Supporting Component-Based Software Development. **Software Engineering Notes**, New York, v.26, n.2, p.53-58, 2001.

[MEN 2002] MENS, T. A State-of-the-Art Survey on Software Merging. **IEEE Trans. Software Engineering**, Piscataway, NJ, EUA, v.28, n.5, 2002.

[MIC 2004] MICROSOFT. **Microsoft Visual Studio Developer Center – Visual SourceSafe**. Disponível em <http://msdn.microsoft.com/vstudio/previous/ssafe/>. Acesso em: out. 2004.

[MOR 2001a] MORO, M.M. **Modelo Temporal de Versões**. 2001. Dissertação (Mestrado em Ciência da Computação) – Instituto de Informática, UFRGS, Porto Alegre.

[MOR 2001b] MORO, M.M.; SAGGIORATO, S.M.; EDELWEISS, N.; SANTOS, C.S. A Temporal Versions Model for Time-Evolving Systems Specification. In: INTERNATIONAL CONFERENCE ON SOFTWARE ENGINEERING & KNOWLEDGE ENGINEERING, 13., 2001, Buenos Aires. **Proceedings**… Skokie: Knowledge Systems Institute, 2001. p.252-259.

[MOR 2001c] MORO, M.M.; SAGGIORATO, S.M.; EDELWEISS, N.; SANTOS, C.S. Adding Time to an Object-Oriented Versions Model. In: INTERNATIONAL CONFERENCE ON DATABASE AND EXPERT SYSTEMS APPLICATIONS, 12., 2001, Munich, Germany. **Proceedings…** Berlin: Springer-Verlag, 2001. p.805-814. (Lecture Notes in Computer Science, v.2113).

[MOR 2002] MORO, M. M.; EDELWEISS, N.; ZAUPA, A. P.; SANTOS, C. S. TVQL – Temporal Versioned Query Language. In: INTERNATIONAL CONFERENCE ON DATABASE AND EXPERT SYSTEMS APPLICATIONS, DEXA, 13., 2002, Aix-en-Provence, France. **Proceedings. . .** Berlin: Springer-Verlag, 2002. p.618–627. (Lecture Notes in Computer Science, v.2453).

[MUN 93] MUNCH, B.P. **Versioning in a Software Engineering Database – The Change Oriented Way**. 1993. Tese (Ph.D.) – NTNU, Trondheim, Norway.

[RAT 2004] RATIONAL. **Rational ClearCase**. Disponível em: <http://www-306.ibm.com/software/awdtools/clearcase/>. Acesso em: out. 2004.

[ROB 2004] ROBERT, R. **Implementação do SCM_TOO em um Banco de Dados Orientado a Objetos**. 2004. 54f. Projeto de Diplomação (Curso de Graduação em Ciência da Computação) – Instituto de Informática, UFRGS, Porto Alegre.

[SIL 2003a] SILVA, F.A.; COSTA, R.V.C.; EDELWEISS, N.; SANTOS, C.S. Using the Temporal Versions Model in a Software Configuration Management Environment. In: SIMPÓSIO BRASILEIRO DE ENGENHARIA DE SOFTWARE, SBES, 17., 2003, Manaus, Amazonas. **Anais...** Rio de Janeiro: Programa de Engenharia de Sistemas e Computação da COPPE/UFRJ, 2003. p.302-317.

[SIL 2003b] SILVA, F.A. **Um Estudo Sobre a Utilização do Modelo Temporal de Versões em um Ambiente de *Software Configuration Management***. 2003. 47f. Trabalho Individual (Mestrado em Ciência da Computação) – Instituto de Informática, UFRGS, Porto Alegre.

[SOF 90] SOFTWARE MAINTENANCE AND DEVELOPMENT SYSTEMS. **Aide-de-Camp Product Overview**. Concord, MA, 1990.

[TAN 93] TANSEL, C.G. **Temporal Databases – Theory, Design and Implementation**. Redwood City: Benjamin/Cummings, 1993.

[TIC 85] TICHY, W. F. RCS – A System for Version Control. **Software – Practice and Experience**, London, v.15, p.637-654, 1985.

[TIC 88] TICHY, W. F. Tools for software configuration management. In: INTERNATIONAL WORKSHOP ON SOFTWARE VERSION AND CONFIGURATION CONTROL, 1988, Grassau. **Proceedings…** [S.l.:s.n.], 1988.

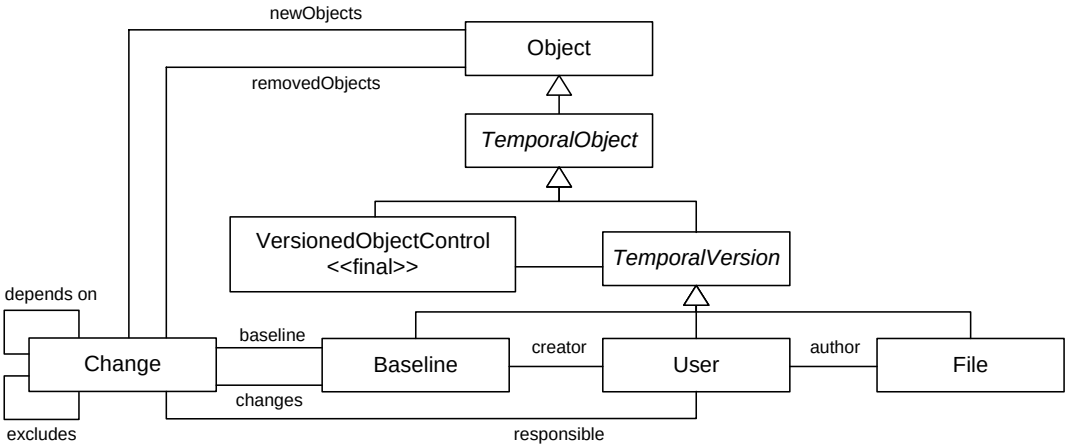
[WEB 2001] WEBER, D.W. Requirements for an SCM Architecture to Enable Component-Based Development. In: INTERNATIONAL WORKSHOP ON SOFTWARE CONFIGURATION MANAGEMENT, 10., 2001. Toronto, Canada. **Proceedings…** [S.l.:s.n.], 2001.

[ZAN 97] ZANIOLO, C. **Advanced Database Systems**. San Francisco: Morgan Kaufmann, 1997.

[ZAU 2002] ZAUPA, A.P. **Suporte a Consultas no Ambiente Temporal de Versões**. 2002. Dissertação (Mestrado em Ciência da Computação) – Instituto de Informática, UFRGS, Porto Alegre.

[ZEL 97] ZELLER, A.; SNELTING, G. Unified Versioning through Feature Logic. **ACM Transactions on Software Engineering and Methodology**, New York, v.6, n.4, p.397-440, 1997.

APÊNDICE A CLASSES DO SCM_TOO

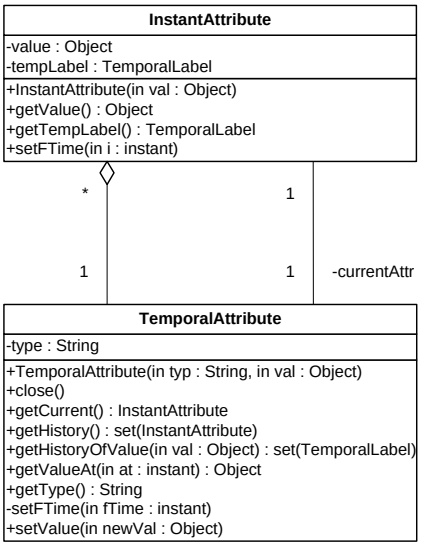


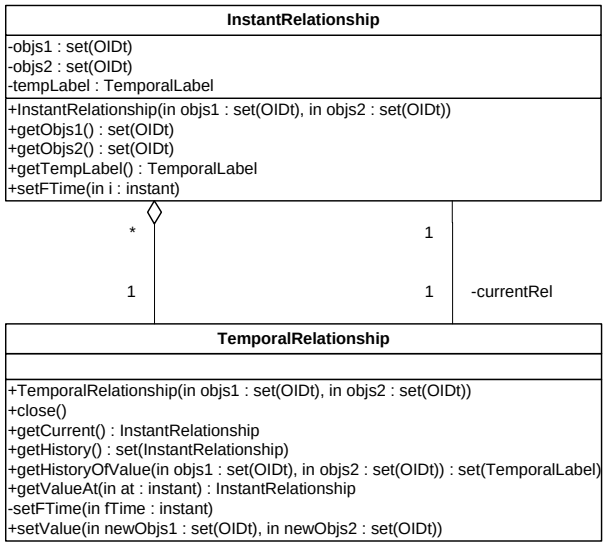
TemporalLabel
-iTime : instant -fTime : instant
+TemporalLabel(in iTime : instant) +getITime() : instant +getFTime() : instant +setFTime(in i : instant)

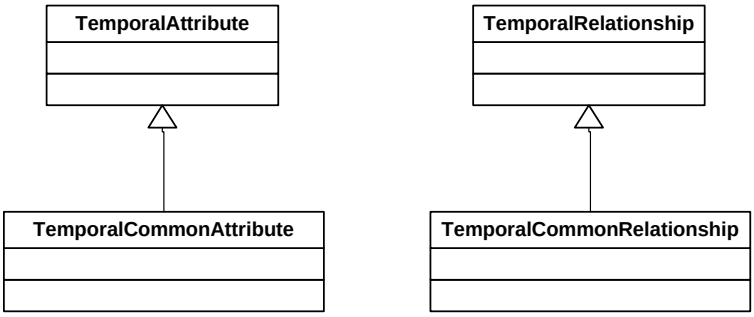
OIDt
-value : String
+OIDt(in E : Integer, in C : Integer, in V : Integer) +getClassNr() : Integer +getEntityNr() : Integer +getOID() : String +getVersionNr() : Integer

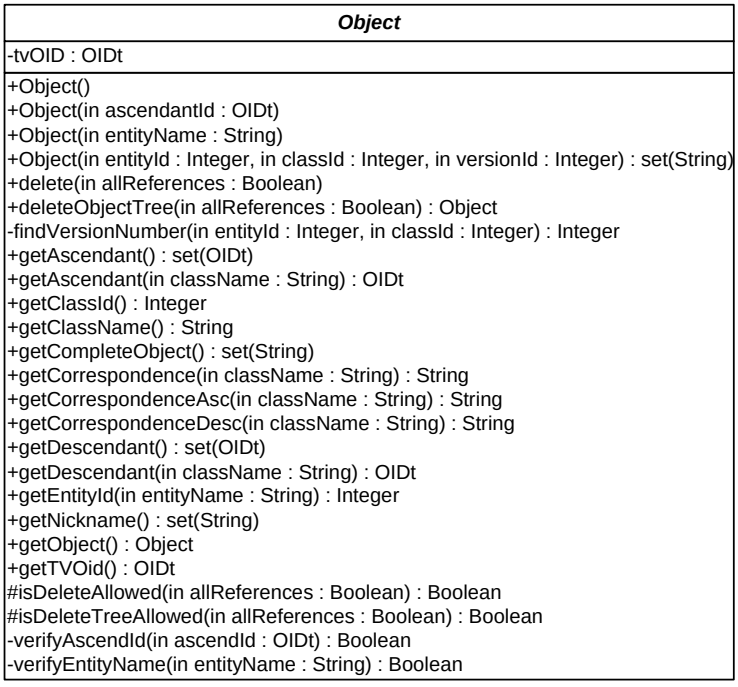
CommonAttribute
-type : String -value : Object
+CommonAttribute(in typ : String, in val : Object) +getType() : String +getValue() : Object +setValue(in newVal : Object)

CommonRelationship
-objs1 : set(OIDt) -objs2 : set(OIDt)
+CommonRelationship(in objs1 : set(OIDt), in objs2 : set(OIDt)) +getObjs1() : set(OIDt) +getObjs2() : set(OIDt) +setObjs1(in newObjs1 : set(OIDt)) +setObjs2(in newObjs2 : set(OIDt))


InstantAttribute
-value : Object
-tempLabel : TemporalLabel
+InstantAttribute(in val : Object)
+getValue() : Object
+getTempLabel() : TemporalLabel
+setFTime(in i : instant)
*
1
1
1
-currentAttr
TemporalAttribute
-type : String
+TemporalAttribute(in typ : String, in val : Object)
+close()
+getCurrent() : InstantAttribute
+getHistory() : set(InstantAttribute)
+getHistoryOfValue(in val : Object) : set(TemporalLabel)
+getValueAt(in at : instant) : Object
+getType() : String
-setFTime(in fTime : instant)
+setValue(in newVal : Object)

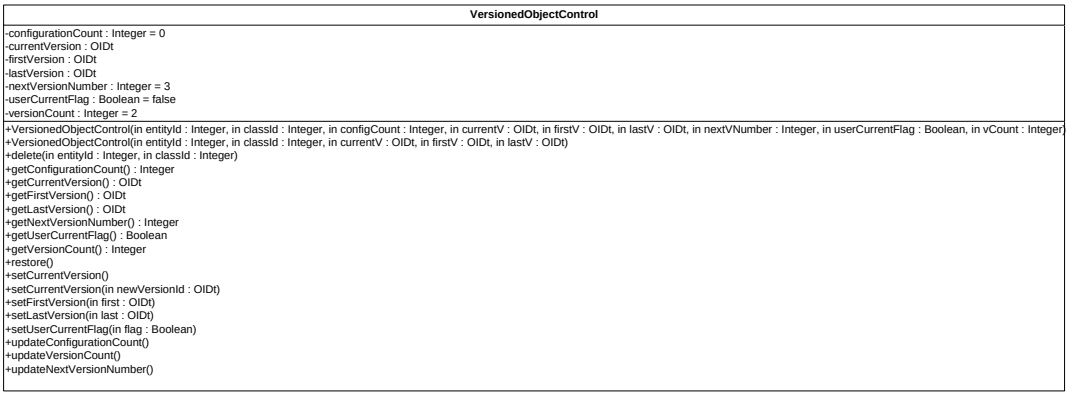

InstantRelationship
-objs1 : set(OIDt)
-objs2 : set(OIDt)
-tempLabel : TemporalLabel
+InstantRelationship(in objs1 : set(OIDt), in objs2 : set(OIDt))
+getObjs1() : set(OIDt)
+getObjs2() : set(OIDt)
+getTempLabel() : TemporalLabel
+setFTime(in i : instant)
*
1
1
1
-currentRel
TemporalRelationship
+TemporalRelationship(in objs1 : set(OIDt), in objs2 : set(OIDt))
+close()
+getCurrent() : InstantRelationship
+getHistory() : set(InstantRelationship)
+getHistoryOfValue(in objs1 : set(OIDt), in objs2 : set(OIDt)) : set(TemporalLabel)
+getValueAt(in at : instant) : InstantRelationship
-setFTime(in fTime : instant)
+setValue(in newObjs1 : set(OIDt), in newObjs2 : set(OIDt))

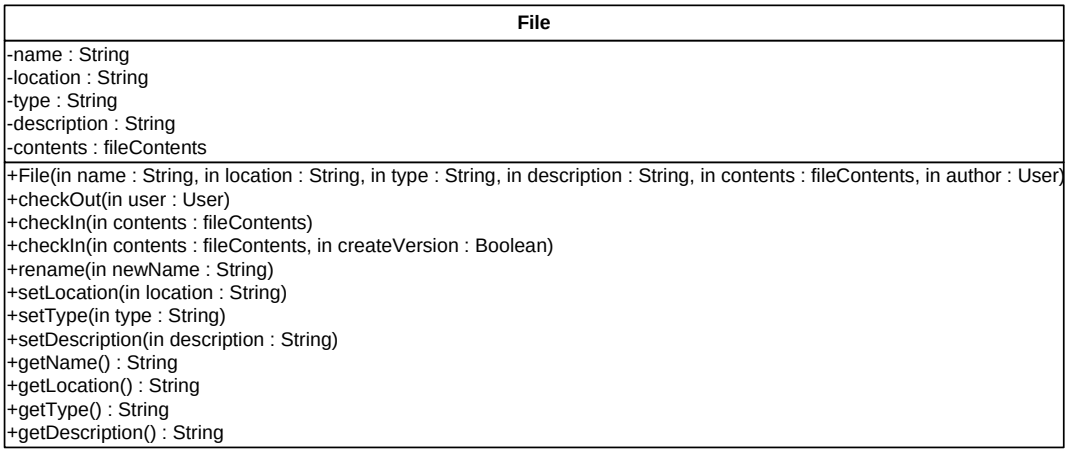

TemporalAttribute
TemporalCommonAttribute
TemporalRelationship
TemporalCommonRelationship

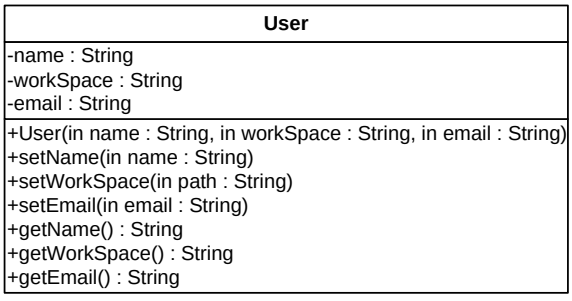

Object
-tvOID : OIDt
+Object()
+Object(in ascendantId : OIDt)
+Object(in entityName : String)
+Object(in entityId : Integer, in classId : Integer, in versionId : Integer) : set(String)
+delete(in allReferences : Boolean)
+deleteObjectTree(in allReferences : Boolean) : Object
-findVersionNumber(in entityId : Integer, in classId : Integer) : Integer
+getAscendant() : set(OIDt)
+getAscendant(in className : String) : OIDt
+getClassId() : Integer
+getClassName() : String
+getCompleteObject() : set(String)
+getCorrespondence(in className : String) : String
+getCorrespondenceAsc(in className : String) : String
+getCorrespondenceDesc(in className : String) : String
+getDescendant() : set(OIDt)
+getDescendant(in className : String) : OIDt
+getEntityId(in entityName : String) : Integer
+getNickname() : set(String)
+getObject() : Object
+getTVOid() : OIDt
#isDeleteAllowed(in allReferences : Boolean) : Boolean
#isDeleteTreeAllowed(in allReferences : Boolean) : Boolean
-verifyAscendId(in ascendId : OIDt) : Boolean
-verifyEntityName(in entityName : String) : Boolean

TemporalObject
-alive : Boolean = true
+TemporalObject() +TemporalObject(in ascendId : OIDt) +TemporalObject(in entityName : String) +TemporalObject(in entityId : Integer, in classId : Integer, in versionId : Integer) +closeTemporalLabels() +delete() +getAlive() : Boolean +getAttributeHistory(in attribName : String) : set(InstantAttribute) +getAttributeValueAt(in attribName : String, in i : instant) : InstantAttribute +getILifetime() : instant +getFLifetime() : instant +getObjectHistory() : set(InstantAttribute) +getRelationshipHistory(in relatedObjId : OIDt, in relatName : String) : set(InstantRelationship) +getRelationshipHistoryAt(in relatedObjId : OIDt, in relatName : String, in i : instant) : InstantRelationship +setTemporalAttribute(in attribName : String, in newValue : Object) +setTemporalRelationship(in relatName : String, in newO1 : OIDt, in newO2 : OIDt)

TemporalVersion
-ascendant : set(OIDt) = NULL -configuration : Boolean = false -descendant : set(OIDt) = NULL -predecessor : set(OIDt) = NULL -status : Char = W -successor : set(OIDt) = NULL
+TemporalVersion() +TemporalVersion(in ascendant : set(OIDt)) +TemporalVersion(in entityName : String) +TemporalVersion(in entityId : Integer, in classId : Integer, in versionId : Integer) -TemporalVersion(in predecId : set(OIDt), in ascendId : set(OIDt), in config : Boolean) -addAscendant(in ascendId : OIDt) -addDescendant(in descendId : OIDt) -addSuccessor(in succId : OIDt) +delete(in allReferences : Boolean) +deleteObjectTree(in allReferences : Boolean) +derive() +getAscendant() : set(OIDt) +getAscendant(in className : String) : set(OIDt) +getConfiguration() : OIDt +getCompleteObject() : set(OIDt) +getDescendant() : set(OIDt) +getDescendant(in className : String) : set(OIDt) +getPredecessor() : set(OIDt) +getStatus() : Char +getSuccessor(in onlyConfigured : Boolean) : set(OIDt) +getVOC() : OIDt +isConfiguration() : Boolean -isDeleteAllowed(in allReferences : Boolean) : Boolean -isDeleteTreeAllowed(in allReferences : Boolean) : Boolean +merge(in version : OIDt) : OIDt +promote(in allAscendant : Boolean, in allReferenced : Boolean) -removeAscendant(in ascendId : OIDt) -removeDescendant(in descendId : OIDt) -removeSuccessor(in succId : OIDt) +restore(in OID : OIDt) : Boolean -setAscendant(in ascendId : OIDt) -setDescendant(in descendId : OIDt) -setStatus(in newStatus : Char) -setSuccessor(in succId : set(OIDt)) -verifyAscendId(in ascendId : set(OIDt)) : Boolean


VersionedObjectControl
-configurationCount : Integer = 0
-currentVersion : OIDt
-firstVersion : OIDt
-lastVersion : OIDt
-nextVersionNumber : Integer = 3
-userCurrentFlag : Boolean = false
-versionCount : Integer = 2
+VersionedObjectControl(in entityId : Integer, in classId : Integer, in configCount : Integer, in currentV : OIDt, in firstV : OIDt, in lastV : OIDt, in nextVNumber : Integer, in userCurrentFlag : Boolean, in vCount : Integer)
+VersionedObjectControl(in entityId : Integer, in classId : Integer, in currentV : OIDt, in firstV : OIDt, in lastV : OIDt)
+delete(in entityId : Integer, in classId : Integer)
+getConfigurationCount() : Integer
+getCurrentVersion() : OIDt
+getFirstVersion() : OIDt
+getLastVersion() : OIDt
+getNextVersionNumber() : Integer
+getUserCurrentFlag() : Boolean
+getVersionCount() : Integer
+restore()
+setCurrentVersion()
+setCurrentVersion(in newVersionId : OIDt)
+setFirstVersion(in first : OIDt)
+setLastVersion(in last : OIDt)
+setUserCurrentFlag(in flag : Boolean)
+updateConfigurationCount()
+updateVersionCount()
+updateNextVersionNumber()


File
-name : String
-location : String
-type : String
-description : String
-contents : fileContents
+File(in name : String, in location : String, in type : String, in description : String, in contents : fileContents, in author : User)
+checkOut(in user : User)
+checkIn(in contents : fileContents)
+checkIn(in contents : fileContents, in createVersion : Boolean)
+rename(in newName : String)
+setLocation(in location : String)
+setType(in type : String)
+setDescription(in description : String)
+getName() : String
+getLocation() : String
+getType() : String
+getDescription() : String


User
-name : String
-workSpace : String
-email : String
+User(in name : String, in workSpace : String, in email : String)
+setName(in name : String)
+setWorkSpace(in path : String)
+setEmail(in email : String)
+getName() : String
+getWorkSpace() : String
+getEmail() : String

Baseline
-name : String -description : String -creator : relationship(User) -objects : set(OIDt) -changes : set(OIDt)
+Baseline(in name : String, in description : String, in objects : set(OIDt), in creator : User) +rename(in newName : String) +getName() : String +setDescription(in description : String) +getDescription() : String +getObjects() : set(OIDt) +getDiff(in prevBaseline : Baseline) : set(Change) +derive(in changes : set(Change), in creator : User) : String +derive(in name : String, in description : String, in changes : set(Change), in creator : User) : String +derive(in name : String, in description : String, in objects : set(OIDt), in creator : User) : String +createConfiguration(in version : OIDt)

Change
-name : String -description : String -newObjects : relationship(Object) -removedObjects : relationship(Object) -responsible : relationship(User) +dependsOn : relationship(Change) +excludes : relationship(Change) -closed : Boolean
+Change(in name : String, in description : String, in baseline : Baseline, in responsible : User) +setName(in name : String) +getName() : String +setDescription(in description : String) +getDescription() : String +addObject(in object : OIDt) +removeObject(in object : OIDt) +close()