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

Modelo Temporal de Versões

por

MIRELLA MOURA MORO

Dissertação submetida à avaliação,
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, junho de 2001

CIP - Catálogo de Publicação

Moro, Mirella Moura
Modelo Temporal de Versões / por Mirella Moura Moro. – Porto Alegre : PPGC da UFRGS, 2001.
120 f. : il.

Dissertação (mestrado) – Universidade Federal do Rio Grande do Sul. Programa de Pós-Graduação em Computação, Porto Alegre, BR – RS, 2001. Orientadora: Edelweiss, Nina.

1. Banco : Dados temporais. 2. Modelo de Versões. I. Edelweiss, Nina. II. Título.

Agradecimentos

Eu teria um “bocado” de gente para agradecer, porém esta página contém apenas a relação de algumas “figuras” que passaram no meu caminho.

Gostaria de começar pela pessoa maravilhosa que me acompanha de pertinho desde a graduação e que sempre consegue me colocar no melhor rumo enquanto tenho minhas “idéias geniais” (e não tão geniais também), professora Nina. Agradeço sempre a sorte que tenho de contar com uma orientadora brilhante ao meu lado durante todo esse tempo.

Como a união faz a força, e duas cabeças pensam melhor que uma, este trabalho só pode ser realizado porque eu também tive um excelente co-orientador ao meu lado, professor Clesio. Muito obrigada por formar com a professora Nina uma dupla e tanto.

Agradeço a todos os colegas do PPGC que influenciaram com sua presença e opinião em algum momento durante o curso. Embora não seja recomendado agradecimentos nominais, eu realmente preciso agradecer especialmente aos colegas que contribuíram diretamente na realização da dissertação. Agradeço aos colegas André Nácul, Anelise Jantsch, Adriana Roma, Cláudio Gomes, Lialda Rossetti, Paôla Gellati, Renata Galante, Rodrigo Moro, e Silvia Saggiorato pelas discussões a respeito das definições e implementação do Modelo e, principalmente, pelas inúmeras dúvidas que levaram a importantes aprimoramentos. À colega Carina Dorneles que além de me emprestar um livro (que diga-se de passagem resolveu algumas das minhas principais dúvidas) me “apresentou” ao DB2. Aos bolsistas Daniel Gaspary e Carlos Eduardo Peixoto pela importante participação na implementação da ferramenta de especificação de classes.

Meus agradecimentos aos funcionários e professores do Instituto de Informática. Em especial aos professores Heuser e Palazzo que tiveram a idéia e incentivaram essa “loucura” de terminar a dissertação antes.

Aos amigos que ganhei de presente no curso, aos colegas do laboratório e aos coadjuvantes que não estão citados, o meu mais sincero muito obrigada.

Encerrando os agradecimentos profissionais, agradeço ao CNPq pelo suporte financeiro.

Começando os agradecimentos pessoais, agradeço a meus pais que absolutamente sempre me incentivaram, mesmo que para isso eu tivesse que ficar tão longe e ausente. Agradeço também ao restante da família por entenderem os meus “sumiços”. Em especial, muito obrigada às minhas irmãs por agüentarem as minhas “crises de mau-humor”. E, encerrando a seção família, agradeço à minha pequena sobrinha Mariana pelos excelentes momentos de entretenimento.

Finalmente, agradeço à minha outra metade por estar comigo nas “chuvas e trovoadas” e por entender que nem sempre estou nos meus melhores dias. É verdade que o trabalho enobrece... mas só o amor engrandece. E se tudo foi possível, é porque sempre tive dedicação e um grande amor ao meu lado.

Sumário

Lista de Abreviaturas

ACID	Atomicidade, Consistência, Isolamento e Durabilidade
ASP	*Active Server Page*
BD	Banco de Dados
CAD	*Computer Aided Design*
CASE	*Computer-Aided Software Engineering*
HTML	*Hypertext Markup Language*
OID	*Object identificator*
ODMG	*Object Data Management Group*
PHP	*Personal Home Page*
SCM	*Sofware Configuration Management*
SGBD	Sistema de Gerenciamento de Banco de Dados
SQL	*Structured Query Language*
TVM	*Temporal Versions Model*
UDF	*User Defined Function*
UML	*Unified Modeling Language*
XML	*Extensible Markup Language*

Lista de Figuras

Lista de Tabelas

Resumo

O objetivo principal desse trabalho é apresentar uma alternativa para a união de um modelo de versões e dados temporais. O resultado, o Modelo Temporal de Versões – TVM (*Temporal Versions Model*), é capaz de armazenar as versões do objeto e, para cada versão, o histórico dos valores das propriedades e dos relacionamentos dinâmicos. Esse modelo difere de outros modelos de dados temporais por apresentar duas diferentes ordens de tempo, ramificado para o objeto e linear para cada versão. O usuário pode também especificar, durante a modelagem, classes normais sem tempo e versionamento, o que permite a integração deste modelo com outros modelos existentes.

A utilização de um modelo de dados temporal semanticamente rico não requer necessariamente a existência de um SGBD próprio para este modelo. A tendência é implementar o modelo sobre banco de dados convencionais, através do mapeamento das informações temporais para atributos explícitos. Como objetivo complementar, é apresentado um ambiente para o suporte do TVM e de todas suas características. Especificamente, são detalhados o mapeamento da hierarquia base do modelo para um banco de dados objeto-relacional e sua implementação em um banco de dados comercial. Desse ambiente, foi implementado um protótipo da ferramenta para o auxílio na especificação de classes da aplicação.

Palavras-chave: modelo Orientado a Objetos, modelo de dados temporal, modelo de versões

TÍTULO: Temporal Versions Model

Abstract

The main purpose of this work is to present an alternative for the union of a version model and temporal data. The result, the Temporal Versions Model - TVM, is able to store the object versions and, for each version, the history of its dynamic properties and relationships values. TVM differs from the other temporal data models by presenting two different time orders, branched time for the objects and linear time for each version. The user can also specify conventional classes (without time and version) during the modeling. This feature allows the integration with existing specifications.

The use of a temporal data model semantically rich does not require the existence of a specific database management system for this model. The tendency is to implement the model on the top of conventional databases by mapping temporal information to explicit attributes. Therefore, as a second objective, an environment to support the TVM and all its features is presented. Specifically, the mapping from the base hierarchy of the model to an object-relational database and its implementation on top a commercial database are detailed. Finally, the tool for helping the user in the system specification on the TVM environment is presented.

Keywords: Object-Oriented model, temporal data model, versions model

1 Introdução

1.1 Motivação

Muitas aplicações de banco de dados são de natureza temporal. Exemplos incluem: aplicações financeiras, como gerenciamento de ações, contas, e bancos; aplicações com registros, como médicas, gerenciamento de funcionários e inventários; aplicações de agendamento, como reservas aéreas, de trem e de hotéis, e gerenciamento de projetos; e aplicações científicas, como monitoramento de clima. Essa categoria de aplicações depende de modelagem temporal e de banco de dados temporais que armazenem dados referenciados no tempo [JEN 99].

Diversos modelos temporais foram propostos nos últimos anos. Entre as características presentes nesses modelos citam-se: associação de informações referentes ao tempo de transação e/ou validade aos dados; uso de pontos no tempo, intervalos de tempo ou elemento temporal como primitiva de tempo; e uso de objetos ou entidades temporais juntamente com não temporais no mesmo diagrama. Esses modelos de dados temporais são, geralmente, extensões de modelos de dados relacionais [CLI 87, GAD 88, LOR 88, NAV 89, SAR 90, SNO 87, TAN 86], entidade-relacionamento [ANT 97, ELM 93, LOU 91, TAU 91], ou orientados a objetos [EDE 94, KAF 92, KAK 96, ROS 91, SU 91, WUU 93, ZEL 95].

Aplicações das mais diferentes áreas necessitam de mecanismos de suporte a processos de desenvolvimento evolutivo. Nesse tipo de processo é necessário armazenar diferentes estágios de uma mesma entidade em tempos distintos ou sob diferentes pontos de vista. Esse requisito é modelado através do conceito de *versão*.

Historicamente, as primeiras pesquisas relacionadas a versões se encontram nas áreas de CAD (*Computer Aided Design*), CASE (*Computer Aided Software Design*) e SCM (*Sofware Configuration Management*) [AGR 91, BEE 88, BJÖ 89, CHO 86, CON 98, GOL 95, KIM 89, TAL 93, WUU 93]. Esse conceito foi se estendendo posteriormente para outros domínios de aplicação. Por exemplo, o uso de versões possibilita o acompanhamento da evolução de um hipertexto, a gerência de configurações alternativas, o suporte a trabalho cooperativo e a modelagem de documentos estruturados, entre outros [BIE 98, HAA 92, MOE 94, NOR 98, OST 92, SAN 99, SOA 95]. É permitido também combinar o conceito de versões com outras técnicas, como o uso de agentes para a manutenção da consistência em sistemas de hipertexto distribuídos [DAT 96].

Embora a utilização de versões armazene as alternativas de projeto, nem todo o histórico das alterações realizadas sobre os dados é registrado. Modificações importantes podem ter sido realizadas, influindo de alguma maneira no desenvolvimento geral, sem que seja possível o acesso posterior a todos os valores passados. Além disso, as informações relativas ao tempo no qual as especificações e alterações foram realizadas não é armazenado. O usuário pode estar interessado em recuperar o estado do projeto relativo a um período ou data específica, isso incluindo dados que foram descartados por qualquer motivo. Esse tipo de recuperação não é possível com o uso apenas de versionamento, pois o histórico completo somente é acessível através de um modelo de dados temporal.

1.2 Objetivos

O objetivo deste trabalho é definir um modelo que utilize as características de versões e tempo para permitir o armazenamento das versões de um objeto e, para cada uma de suas versões, do histórico de seu tempo de vida e das alterações feitas em seus atributos e relacionamentos dinâmicos. Esse modelo é denominado Modelo Temporal de Versões (TVM – *Temporal Versions Model*).

O modelo TVM difere de outros modelos de dados temporais por permitir o uso de duas ordens de tempo diferentes (ramificado para o objeto e linear para versões) e por ser o próprio usuário quem define quais propriedades terão seus históricos armazenados. Essa segunda característica é importante por limitar o crescimento de espaço dos dados possibilitando um melhor desempenho. Além disso, o modelo apresenta dois aspectos importantes: (i) a especificação do sistema pode ser feita considerando as alternativas de projeto bem como a evolução histórica dos dados; (ii) a facilidade de integração com especificações existentes, uma vez que não é exigido que todas as classes sejam temporais versionadas.

A utilização de um modelo de dados semanticamente rico não requer necessariamente a existência de um SGBD próprio para este modelo. A tendência é implementar o modelo sobre BDs convencionais, através do mapeamento das informações temporais para atributos explícitos. Algumas experiências neste sentido foram já realizadas, mostrando sua viabilidade [ARR 94, BRA 94, CAV 95, HÜB 99, SIM 98, THE 94, TIM 99]. Como objetivo complementar, é apresentado um ambiente para o suporte do TVM e de todas suas características. Especificamente, é detalhado o mapeamento da hierarquia base do modelo para um banco de dados objeto relacional e sua implementação em um banco de dados comercial. Desse ambiente, foi implementado um protótipo da ferramenta para o auxílio na especificação de classes da aplicação.

1.3 Organização do texto

O restante do texto está organizado como segue:

- Capítulo 2, Versões e Dimensão Temporal, apresenta os principais conceitos relativos a versões e tempo, bem como um resumo do Modelo de Versões definido por Golendziner;

- Capítulo 3, Modelo Temporal de Versões, apresenta as características do Modelo Temporal de Versões referentes à representação temporal e às atualizações realizadas no Modelo de Versões;

- Capítulo 4, Ambiente Temporal de Versões, apresenta as características gerais de um ambiente para o TVM sobre um SGBD existente, detalha o mapeamento das classes base do Modelo para um sistema objeto-relacional e, finalmente, ilustra a ferramenta de apoio à especificação de classes;

- Capítulo 5, Estudo de Caso, mostra um estudo de caso que inclui a representação do mapeamento da hierarquia do TVM no DB2, a modelagem conceitual de uma aplicação e a representação gráfica do funcionamento dessa aplicação sobre o banco de dados comercial;

- Capítulo 6, Conclusões, apresenta a revisão do trabalho, conclusões, trabalhos relacionados e futuros.

Os anexos apresentam os seguintes conteúdos:

- Anexo 1 – BNF da Linguagem de Definição de Classes;
- Anexo 2 – Definição das Classes da Hierarquia;
- Anexo 3 – Sintaxe dos Principais Comandos do DB2.

2 Versões e Dimensão Temporal

Como base conceitual, este capítulo apresenta as principais definições relativas a versões e tempo, bem como um resumo do Modelo de Versões, definido por Golendziner.

2.1 Conceitos de Versões

Muitas aplicações de banco de dados que usam sistemas orientado a objetos necessitam de mais de uma *versão* do mesmo objeto. Por exemplo, uma aplicação de banco de dados para um ambiente de engenharia de software que armazena vários artefatos de software, tais como módulos de projeto, módulos de código fonte e informação de configuração para descrever quais módulos devem ser ligados e formar um programa complexo, e casos de teste para testar todo o sistema. Usualmente, atividades de manutenção são aplicadas a um sistema de software à medida em que seus requisitos evoluem. A manutenção geralmente envolve mudança de alguns módulos de projeto e implementação. Caso o sistema já esteja operando, se um ou mais módulos devem ser trocados, o projetista poderia criar uma *nova versão* de cada um desses módulos para implementar as mudanças. Similarmente, novas versões de casos de teste podem ter sido geradas para novas versões dos módulos. Entretanto, as versões existentes não podem ser descartadas até que as novas tenham sido testadas e aprovadas, para então substituírem as velhas [ELM 00].

Observa-se, portanto, que um objeto pode ter mais de uma versão. Por exemplo, considerando dois programadores que atualizam o mesmo módulo de software simultaneamente. Neste caso, duas versões, além do módulo original, são necessárias. Alguns SGBDs possuem uma facilidade que pode comparar as duas versões com o objeto e determinar se alguma mudança é incompatível, de forma a ajudar no processo de junção. Outros sistemas mantêm um *grafo de versões* que mostra os relacionamentos entre as versões. Um grafo de versões pode ajudar usuários a entender os relacionamentos entre várias versões e pode ser usado internamente pelo sistema para gerenciar a criação e a exclusão de versões.

Quando versionamento é aplicado a objetos complexos, surgem outras questões que devem ser resolvidas. Um objeto complexo, como um sistema de software, pode consistir de muitos módulos. Quando o versionamento é permitido, cada um desses módulos pode ter um número diferente de versões e um grafo de derivação. Então, uma *configuração* de um objeto complexo é uma coleção composta de uma versão de cada módulo. As versões que compõem uma configuração são compatíveis e juntas formam uma versão válida do objeto complexo.

Uma nova versão ou configuração do objeto complexo não precisa incluir novas versões para todos os módulos. Deste modo, certas versões de módulos que não tenham sido alteradas podem pertencer a mais de uma configuração do objeto complexo. Diferenciando melhor os conceitos, uma configuração é uma coleção de versões de objetos diferentes que, juntos, criam um objeto complexo, enquanto um grafo de derivação descreve versões do mesmo objeto.

Um objeto que possui versões é chamado de *objeto versionado*. Assim como as versões, também é considerado um objeto de primeira classe, pois possui um OID, mantendo suas informações e de suas versões. Um objeto versionado tem um conjunto de versões associado, no qual cada versão é representante do objeto e só pode estar associada a esse objeto versionado.

Em um objeto versionado existe sempre uma *versão corrente*, mantida pelo sistema como a mais recente à medida em que novas versões vão sendo criadas. Se o usuário especificar uma outra versão como sendo a corrente, essa permanece fixa. Sempre que o usuário envia uma mensagem a um objeto versionado sem especificar a versão, a versão corrente é utilizada.

A característica de versionamento está associada aos objetos e não às classes. Assim, um objeto não versionado pode passar a ser versionado dinamicamente. Nesses casos, o objeto não versionado passa a ser a primeira versão do objeto versionado e o antecessor da nova versão.

A criação de um objeto versionado pode ser feita:

- *implicitamente* – quando uma versão é criada a partir de um objeto não versionado através de uma derivação;
- *explicitamente* – quando um objeto é criado como sendo um objeto versionado, sem versões associadas, através da operação específica de criação de objeto versionado.

Versões de um mesmo objeto estão relacionadas por meio de um relacionamento de derivação, formando um grafo acíclico dirigido. Uma versão criada como sucessora de uma existente, é uma cópia de sua antecessora. Já uma versão criada como derivada de várias, é uma cópia de sua primeira antecessora, sendo mantidos os relacionamentos de derivação com as demais.

Toda a versão tem um estado (*status*) associado, que pode ser:

- *em trabalho* – estado que a versão recebe quando é criada. Estando nele, a versão é considerada temporária, podendo sofrer várias alterações até atingir um estado mais estável. Uma versão *em trabalho* pode ser removida;
- *estável* – uma versão nesse estado encontra-se num estágio mais avançado e consistente, podendo ser compartilhada e não mais alterada. Uma versão *estável* pode ser removida. Uma versão atinge esse estado por uma promoção explícita do usuário ou porque uma versão foi criada como sua derivada;
- *consolidada* – uma versão nesse estado encontra-se no seu estágio final, não podendo ser mais alterada, nem removida. Uma versão atinge esse estágio através de uma instrução especifica do usuário, que a promove do estágio *estável* para *consolidado*.

Objetos podem ser definidos como compostos de outros objetos. Essa composição é expressa pela inclusão de um identificador de objeto (OID) como valor de atributo de outro objeto (o composto), definindo uma referência do objeto composto para o componente. Como o objeto componente pode, ainda, ser composto de outros objetos, fica estabelecida uma hierarquia de composição de objetos. Essas referências entre os objetos podem ser representadas de duas maneiras:

- por *referência estática* quando faz referência a uma versão específica de um objeto;

- por *referência dinâmica* quando faz referência a um objeto versionado. Por exemplo, caso o usuário ainda não saiba qual versão de um componente irá usar ou queira deixar esta decisão para mais tarde, fará a referência a um objeto versionado, estabelecendo que ali, mais tarde, irá estar uma referência a uma versão específica do objeto.

Uma vez que objetos são compostos hierarquicamente, uma configuração deve incluir definições recursivamente para todos os objetos na hierarquia de agregação. Assim, sempre que o objeto referido for um objeto versionado, uma de suas versões deve ser escolhida para assumir seu lugar. A escolha da versão depende do critério utilizado, sendo a versão corrente por *default.* Os critérios podem ser pré-definidos ou uma expressão complexa, envolvendo relacionamentos entre os diversos componentes. A versão escolhida passa a ser, então, a versão configurada do objeto versionado em questão. Outra escolha a ser feita é a de um ascendente, quando a versão possuir mais de um.

Diferentes escolhas de versões para componentes e/ou ascendentes geram diferentes configurações para um mesmo objeto. Versões configuradas podem fazer referência somente a outras versões configuradas, podendo ser compartilhadas ou não. Além disso, uma versão configurada pode ser copiada para ser novamente trabalhada.

Um objeto não versionado que apresenta referências dinâmicas pode ser configurado. Nesse caso, ele deixa de ser um objeto não versionado, passando a ser a primeira versão do objeto versionado recém criado e a versão base da configuração.

Uma configuração apresenta as seguintes características, que a definem como uma versão e, portanto, um objeto:

- possui um identificador (OID), construído de acordo com as mesmas regras aplicadas aos identificadores de versões;
- pode possuir ascendentes e descendentes em todos os níveis da hierarquia de herança. Seus ascendentes só podem ser versões configuradas, mas seus descendentes podem ser versões configuradas ou versões regulares;
- está associada a um objeto versionado;
- possui um *status*;
- pode ser usada como componente de outras configurações e também de outros objetos, versionados ou não;
- possui as mesmas operações definidas para versões e objetos.

Uma configuração também é uma versão especial por apresentar as seguintes peculiaridades:

- é uma definição completa de um objeto, não contendo referências dinâmicas nem múltiplos ascendentes na mesma superclasse;
- é sempre uma versão folha no grafo de derivação, possibilitando a remoção de versões configuradas. Várias versões podem ser criadas para uma mesma versão base, ficando todas como suas sucessoras. Não é gerado um histórico de configurações, mas várias configurações alternativas para uma mesma versão;
- alterações sobre uma versão configurada só podem ser mudanças nas escolhas feitas, isto é, referências a versões configuradas só podem ser substituídas por outras referências a versões configuradas.

2.2 Modelo de Versões

Apesar da literatura apresentar alguns estudos de versões sobre bases de dados relacionais [DAD 84], a grande parte da pesquisa sobre esse conceito concentra-se na sua utilização com modelos de dados orientados a objetos [AGR 91, BEE 88, BJÖ 89, CHO 86, CON 98, GOL 95, KIM 89, TAL 93, WUU 93, ZEL 95].

Dentre as alternativas de modelos com versionamento, este trabalho baseia-se particularmente no Modelo de Versões proposto por Golendziner [GOL 95]. O Modelo apresenta uma extensão aplicável a modelos de dados orientado a objetos pela incorporação de conceitos e mecanismos que suportam a definição e manipulação de objetos, versões e configurações. O Modelo de Versões utiliza herança por extensão, a qual permite o versionamento em todos os níveis da hierarquia. A grande maioria dos modelos utiliza a herança por refinamento com o versionamento dos objetos apenas nas classes mais especializadas (folhas) da hierarquia de herança.

2.2.1 Correspondência entre Objetos e Versões

Pelo fato de utilizar a herança por extensão, o modelo permite que entidades do mundo real sejam modeladas em diferentes níveis de abstração. Assim, um determinado projeto pode ser desenvolvido em um determinado nível de abstração e depois ser detalhado em níveis inferiores da hierarquia de herança, permitindo uma construção *top-down* do mesmo.

Uma versão, ao ser criada, deve possuir um ou mais ascendentes. Deste modo, criam-se correspondências entre versões de um objeto em uma classe e versões de seu objeto ascendente na superclasse. Essas correspondências estabelecem restrições de integridade (*n:m*, *1:1*, *1:n* ou *n:1*) e devem ser especificadas no momento em que o relacionamento de herança entre uma classe e sua superclasse é definido no esquema, devendo ser mantida pelo sistema.

No momento em que é preciso recuperar um objeto e todos os seus atributos herdados, a busca inicia na classe mais especializada, sendo escolhido um ascendente para cada superclasse relacionada. O ascendente pode ser explicitamente indicado, através de seu OID, ou especificado através de um dos critérios pré-definidos (mais recente, primeiro, corrente). Tanto objetos versionados como não versionados podem participar da hierarquia de herança, permitindo dessa forma que um objeto não versionado seja ascendente ou descendente de um objeto versionado.

Além das correspondências na hierarquia, há a correspondência na seqüência de derivação que pode formar três tipos de estruturas:

- *lista* – cada versão possui apenas uma predecessora e uma sucessora (cardinalidade *1:1*);
- *árvore* – inúmeras versões podem ser derivadas de uma mesma versão, mas cada versão só possui uma predecessora (cardinalidade *1:n*);
- *grafo acíclico dirigido* – cada versão pode possuir várias predecessoras e várias sucessoras (cardinalidade *n:1* e *n:m*).

2.2.2 Identificadores de Objetos

A correspondência dos objetos com seus ascendentes e descendentes é feita através do identificador do objeto (OID). O identificador possui um componente comum

a todos os objetos, que representa a *entidade* modelada. A estrutura do identificador é a seguinte:

OID = <identificador da entidade, identificador da classe, número da versão>

A entidade representa o objeto do mundo real modelado nas diversas classes do modelo conceitual. Voltando ao exemplo do software, cada classe vai representar um módulo diferente, um componente. Deste modo, a união de um objeto de cada classe vai representar o sistema de software como um todo. No momento em que se tem vários sistemas de software sendo projetados simultaneamente, cada um possui o seu próprio identificador de entidade.

O número da versão é representado com valores inteiros, gerados seqüencialmente. Esses números não podem ser reaproveitados em caso de exclusão de uma versão. Quando um objeto representar o objeto versionado, seu número de versão é nulo e, quando for um objeto sem versões seu número de versão é 1. Assim, um objeto não versionado será identificado da mesma maneira que a primeira versão de um objeto versionado.

2.2.3 Extensões Propostas pelo Modelo

Na maioria dos modelos de dados orientados a objetos, a hierarquia de herança é constituída, primeiramente, de uma classe *raiz* (normalmente denominada *Object*). A partir dela, várias subclasses são apresentadas como especializações.

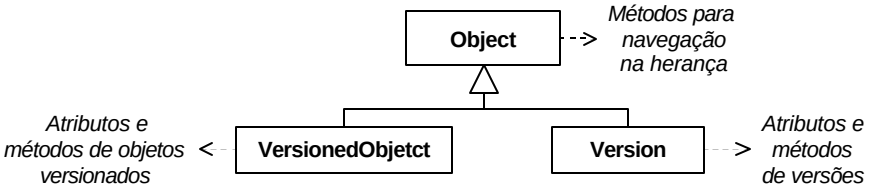


Figura 2.1 – Hierarquia definida pelo Modelo de Versões

O Modelo de Versões apresenta as seguintes extensões a um modelo orientado a objetos convencional (Figura 2.1):

- a classe raiz, chamada de *Object*, é estendida com novas operações para suportar as características de versionamento do modelo;
- duas novas classes são definidas como subclasses da classe *Object*:
  1. *VersionedObject* – permite a modelagem de objetos versionados; e
  2. *Version* – permite a modelagem de versões.
- foi proposta uma extensão na definição de classe para especificar as cardinalidades associadas às correspondências entre objetos e seus ascendentes e descendentes.

Deve-se considerar as seguintes observações importantes a respeito da nova hierarquia:

- se no momento da criação de uma subclasse não for definida sua superclasse, ela será considerada como subclasse da classe *Object*;
- se um objeto versionado for instanciado a partir de uma classe *C*, este será instanciado também na classe *VersionedObject*. Então, além do objeto pertencer à classe definida pelo usuário, também pertencerá à classe *VersionedObject*;

- se um objeto for elemento da classe *VersionedObject*, não pode ser elemento da classe *Version*, sendo a recíproca verdadeira.

As classes do modelo de versões são ilustradas na Figura 2.2 e brevemente descritas a seguir.

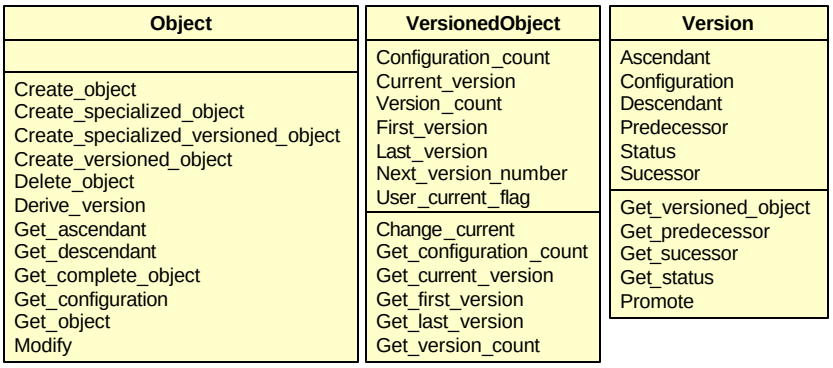


Figura 2.2 – Classes da hierarquia com atributos e operações

Classe *Object*

A classe *Object* foi estendida para que os objetos possam ser manipulados nos vários níveis da hierarquia de herança. Ela é pré-definida pelo sistema e representa a classe de mais alto nível de abstração. Possui apenas as seguintes operações:

- *Create_object* – cria um objeto não versionado pertencente à classe indicada;
- *Modify* – permite que sejam informados valores para os atributos indicados;
- *Create_specialized_object* – cria um objeto especializado na classe, devendo ser indicado o respectivo objeto ascendente;
- *Create_versioned_object* – cria um objeto versionado pertencente à classe indicada, sem nenhuma versão associada;
- *Create_specialized_versioned_object* – cria um objeto versionado como um objeto especializado;
- *Derive_version* – gera uma nova versão de um objeto, derivada de uma ou mais versões, ou de um objeto já existente;
- *Get_object* – busca os valores de atributos do objeto;
- *Get_ascendant* – permite a navegação na hierarquia de herança a partir do OID do objeto, retornando o OID do objeto ascendente da subclasse definida;
- *Get_descendant* – análoga a *Get_ascendant*, refere-se aos descendentes;
- *Get_complete_object* – retorna os ascendentes de um objeto na hierarquia de herança, um para cada superclasse;
- *Get_configuration* – produz uma versão configurada como sucessora de uma versão base, utilizada como parâmetro;
- *Delete_object* – se o OID for de uma versão, somente poderá ser excluída se estiver no estado *em trabalho* ou *estável* e for uma folha na hierarquia. Se o OID for de um objeto versionado, somente poderá ser excluído se não possuir versões associadas. Objetos referidos por outros não podem ser removidos.

Classe *VersionedObject*

A classe *VersionedObject* foi criada para permitir a modelagem de objetos versionados. Ela define proprie dades e operações requeridas para a manipulação de objetos versionados. As propriedades definidas são:

- *Version_count* – número total de versões do objeto versionado;
- *Configuration_count* – número de versões configuradas;
- *First_version* – primeira versão criada para o objeto;
- *Last_version* – última versão criada para o objeto;
- *Current_version* – versão corrente;
- *User_current_flag* – se a versão corrente foi escolhida pelo usuário;
- *Next_version_number* – número que será atribuído à próxima versão criada.

As operações definidas por esta classe são:

- *Get_version_count* – retorna o número total de versões do objeto versionado;
- *Get_configuration_count* – retorna o número de configurações;
- *Get_first_version* – retorna o OID da primeira versão criada;
- *Get_last_version* – retorna o OID da última versão criada;
- *Get_current_version* – retorna o OID da versão corrente;
- *Change_current* – permite ao usuário trocar a versão corrente.

Classe *Version*

A classe *Version* permite a modelagem de versões. Ela define propriedades e operações necessárias para a manipulação de versões. As propriedades definidas são:

- *Ascendant* – versão(ões) ascendente(s) na hierarquia de herança, na(s) superclasse(s) do nível imediatamente superior;
- *Descendant* – analogamente, versão(ões) descendente(s);
- *Sucessor* – versão sucessora no grafo de derivação ao qual a versão pertence;
- *Predecessor* – versão predecessora no grafo de derivação da versão;
- *Status* – estado de uma versão;
- *Configuration* – se verdadeira, indica que uma versão é uma versão configurada.

As operações definidas por esta classes são:

- *Get_versioned_object* – retorna o OID do objeto versionado correspondente, dado o OID de uma versão;
- *Get_predecessor* – retorna a(s) versão(ões) antecessora(s) imediata(s) da versão identificada;

Get_sucessor – retorna a(s) versão(ões) sucessora(s) imediata(s) da versão identificada;

Promote – promove uma versão para o próximo *status*. Se o OID for de um objeto versionado, a operação é aplicada sobre sua versão corrente;

Get_status – devolve o *status* da versão.

2.3 Conceitos de Tempo

Uma considerável variedade de aplicações de banco de dados gerencia dados que variam no tempo. Em contraste, a tecnologia existente de banco de dados provê pouco suporte para o gerenciamento de tais dados. A área de pesquisa em banco de dados temporal tem o objetivo de trocar esse estado pela caracterização de semânticas de dados temporais e pelo fornecimento de maneiras efetivas e eficientes de modelar, armazenar e consultar dados temporais [JEN 99].

A modelagem de aspectos temporais é um importante tópico da modelagem de sistemas de informação. Através desses aspectos são representadas as características dinâmicas das aplicações e a interação temporal entre diferentes processos. A possibilidade de armazenar, manipular e recuperar dados temporais deve ser considerada quando da escolha de um método de modelagem [EDE 94].

A noção de tempo surge em diferentes níveis: (i) na modelagem de dados, (ii) na linguagem de recuperação e manipulação de dados, e (iii) no nível de implementação do SGBD. A seguir são apresentadas algumas considerações sobre o primeiro nível, ou seja, a modelagem de dados.

Ao construir a especificação de um sistema de informação, não só a estrutura dos dados manipulados deve ser definida, mas também a sua dinâmica – seu comportamento com a passagem do tempo. Na coleta de requisitos dos sistemas devem ser identificados os requisitos temporais da aplicação em questão. O método utilizado na especificação do sistema de informação correspondente à aplicação deve permitir que todos os aspectos temporais sejam representados.

2.3.1 Formas de Representação Temporal

Nos bancos de dados convencionais a realidade é representada somente pelo estado presente. Quando o mundo real muda, os novos valores são incorporados no banco de dados substituindo os valores antigos. Em banco de dados temporais, as propriedades temporais são atualizadas pela inserção de um novo fato no banco.

Vários autores propõem diferentes conceitos de tempo para modelar informações temporais. Um glossário para esses termos foi estabelecido por um conjunto de pesquisadores da área em [JEN 98]. Complementando essa idéia, Snodgrass apresenta uma nova abordagem, mais sucinta e direta para conceitos temporais afirmando que tudo tem origem em uma tríade de triplas (*triad of triples*) [SNO 00]:

tipos de dados temporais;

tipos de tempo;

expressões temporais.

Esses conceitos são encontrados em todas as aplicações que variam com o tempo. Cada um deles por si só consiste de três elementos ortogonais. Os três tipos de dados temporais são:

instante – informação existe em um instante de tempo. Por exemplo, chove agora;

intervalo – informação em uma fatia de tempo. Por exemplo, chove há um mês;

período – informação com uma duração de tempo "ancorada". Por exemplo, choveu todo o primeiro trimestre, de 2 de janeiro a 30 de março.

A maioria dos SGBDs suporta somente instantes de tempo, sendo os intervalos simulados através de números inteiros e de ponto-flutuante. Períodos são sempre deixados para o desenvolvedor da aplicação simular usando tipos de dados disponíveis. Há três tipos fundamentais de tempo:

tempo definido pelo usuário – um valor de tempo sem interpretação;

tempo de validade – quando um fato é verdadeiro na realidade modelada;

tempo de transação – quando um fato foi inserido no banco de dados.

Esses tipos de tempo são ortogonais: uma classe pode ser associada com nenhum, um, dois ou mesmo os três tipos de tempo. O tempo de validade captura o histórico de uma realidade evolutiva, e o tempo de transação captura a seqüência de estados de uma classe atualizável. A associação ao tempo de transação permite reconstruir o estado da base em qualquer data passada sem necessidade de operações de recuperação de informações (*backup* e *recovery*).

Finalizando, há três tipos básicos de afirmações ou expressões orientadas a tempo, que podem ser usadas em consultas e manipulação de dados:

corrente – agora (qual informação agora);

seqüenciada – em cada instante de tempo (qual informação e quando existe);

não seqüenciada – ignorando o tempo (qual informação existe em qualquer tempo).

Essas três triplas aplicam-se igualmente a consultas, expressões de atualização, visões e regras de integridade.

2.3.2 Modelos de Dados Temporais

Diversos modelos de dados temporais têm sido apresentados na literatura, sendo a maioria constituída de modelos já existentes com extensões temporais. Entre esses, podem ser encontrados:

modelos relacionais – HRDM [CLI 87], TRDM [SNO 87], os modelos de Gadia e Yeung [GAD 88], de Lorentzos e Johnson [LOR 88], de Navathe e Ahmed [NAV 89], de Sarda [SAR 90], e de Tansel [TAN 86];

entidade relacionamentos – TempER [ANT 97], TEER [ELM 93], ERT [LOU 91], e TER [TAU 91];

orientado a objetos – TF-ORM [EDE 94] , TMAD [KAF 92], TAU [KAK 96], TOODM [ROS 91], OSAM*/T [SU 91], e OODAPLEX [WUU 93].

Uma visão concisa dos principais modelos de dados temporais existentes pode ser encontrada nas referências [ANT 97, EDE 94, HÜB 00, MOR 00, SNO 95, TAN 93].

De modo geral, todos esses modelos de dados temporais possuem basicamente os seguintes requisitos:

capacidade de modelar e consultar o BD em qualquer instante temporal;

capacidade de modelar e consultar o BD em dois instantes de tempo diferentes;

tuplas heterogêneas (diferentes períodos de existência em atributos de uma mesma tupla);

atributos multi-valorados em qualquer instante de tempo.

2.4 Considerações Finais

Esse capítulo apresentou dois conceitos distintos: versões e dados temporais. O versionamento é ideal para aplicações que envolvem projeto e para quando há necessidade de que um objeto apresente alternativas. Já os dados temporais estão presentes na maioria das aplicações. Especialmente, a associação do tempo de validade aos dados tem a vantagem de armazenar a história dos dados evolutivos. Já a utilização do tempo de transação apresenta a possibilidade de reconstruir o estado da base em qualquer data passada, sem usar operações de recuperação de informações (*backup* e *recovery*). O próximo capítulo apresenta um modelo que junta esses conceitos tratando-os homogeneamente.

3 Modelo Temporal de Versões

O Modelo Temporal de Versões (TVM) é uma extensão ao Modelo de Versões, possibilitando representar toda a história dos dados da aplicação. O conceito de versão permite que o usuário mantenha as alternativas de projeto. Adicionando a dimensão temporal, o modelo armazena o histórico e a evolução dos dados das instâncias dos objetos.

Dois níveis de gerência de versões de objetos podem ser identificados [BJÖ 89]:

nível da aplicação – suporta a representação de informações dependentes de tempo e seqüenciamento, conforme definido pelo usuário da aplicação;

nível do sistema – o histórico de versões de um objeto reflete a seqüência de modificações realizadas sobre um objeto, como percebidas pelo sistema. Neste caso o objeto nem sempre corresp onde a um objeto como visto pela aplicação, sendo denominado *unidade de versionamento*.

Este trabalho considera apenas o primeiro tipo de versão. Nesse nível, a associação da dimensão temporal proporciona o armazenamento de todo histórico e evolução dos valores dos atributos e relacionamentos das instâncias. Contudo, como isso pode sobrecarregar a capacidade de armazenamento e diminuir o de sempenho da base de dados, o usuário deve modelar para quais elementos ele deseja que o histórico seja armazenado.

Este capítulo apresenta as características do Modelo Temporal de Versões. Primeiramente, são descritas a representação temporal e as atualizações realizadas no Modelo de Versões. A seguir, as características de hierarquia por extensão e de configuração são melhor detalhadas devido a suas complexidades. A linguagem de definição de classes é proposta, bem como considerações sobre a linguagem de consulta. Por fim, é exposto um breve estudo comparativo com outros modelos que unem tempo e versões.

3.1 Representação Temporal no TVM

O tempo é associado a objetos, versões, atributos e relacionamentos permitindo uma modelagem mais flexível. Um objeto tem uma linha de tempo para cada versão. O fato de muitas versões do mesmo objeto poderem coexistir gera a possibilidade de duas ordens no tempo: (i) *tempo ramificado* para um objeto, devido às diferentes linhas de tempo de cada versão que são originadas da linha do objeto; e (ii) *tempo linear* para cada versão. A variação temporal é discreta, e a temporalidade é representada no modelo através do rótulo de tempo elemento temporal (conjunto de intervalos temporais), bitemporal (tempos de transação e validade) e implícito.

Cada objeto possui o atributo pré-definido *alive* associado ao seu estado de vida. O atributo *alive* recebe o valor *true* no momento da criação do objeto, armazenando também seu tempo de validade inicial. O valor do atributo é atualizado para *false*, e o tempo de validade final recebe o tempo de transação no momento da exclusão lógica.

Os atributos e relacionamentos do objeto podem ser definidos como *estáticos* (quando não têm a variação de seus valores armazenada) ou *temporalizados* (todas as alterações de seus valores são armazenadas formando seu histórico). A classificação de atributos e relacionamentos como temporalizados fica sob a responsabilidade do usuário durante a especificação, sendo permitido que uma mesma classe tenha atributos e relacionamentos de ambos os tipos (temporais ou não).

Os valores dos atributos e relacionamentos temporais estão associados com os rótulos pré-definidos *vTimei*, *vTimef*, *tTimei*, e *tTimef*, que representam respectivamente, os tempos de validade inicial e final, e os tempos de transação inicial e final.

3.1.1 Regras de Integridade Temporal

Esta seção apresenta dois conjuntos de regras de integridade temporal. O primeiro conjunto especifica as integridades em relação aos tempos de vida de objetos, versões e objetos versionados através de cinco afirmações. O segundo aborda o gerenciamento de dados em relação à inserção e à atualização de rótulos temporais. Os rótulos temporais são abreviados: TVI é o tempo de validade inicial, TVF é o tempo de validade final, TTI é o tempo de transação inicial, e TTF é o tempo de transação final.

Tempos de Vida

O rótulo de tempo associado às versões deve estar contido no tempo de vida do objeto versionado, assim como o rótulo de tempo associado às variações dos atributos ou relacionamentos temporalizados de uma versão deve estar contido no tempo de vida da versão. Sendo assim definem-se as seguintes regras de integridade temporal, estabelecidas por vida do objeto:

1. tempo de validade inicial de um atributo ou relacionamento temporal (*TVI*) deve ser maior ou igual ao tempo de vida inicial (*alive.TVI*) da versão a qual pertence;
2. tempo de transação inicial de um atributo ou relacionamento temporalizado (*TTI*) deve ser maior ou igual ao tempo de vida inicial (*alive.TVI*) da versão a qual pertence;
3. tempo de vida final (*alive.TVF*) de uma versão deve ser maior que o inicial (*alive.TVI*) e maior ou igual ao maior tempo de transação e validade final de seus atributos e relacionamento temporalizados (*TTF*, *TVF*);
4. tempo de vida inicial (*alive.TVI*) de uma versão deve ser menor que o final (*alive.TVF*) e maior ou igual ao tempo de vida inicial (*alive.TVI*) do objeto versionado;
5. tempo de vida final (*alive.TVF*) de um objeto versionado deve ser maior que o inicial (*alive.TVI*) e maior ou igual ao maior tempo de vida final de suas versões (*alive.TVF*).

Os tempos de vida inicial e final do objeto são informados pelos valores de *TVI* e *TVF* do atributo *alive* quando o mesmo é *true*. Essas regras de integridade valem dentro de cada "vida" do objeto. Por definição do Modelo, os tempos de vida de um objeto são elementos temporais. As vidas de um mesmo objeto não podem ter um (ou mais) instante em comum. No momento em que um objeto excluído é restaurado, seu novo tempo de validade inicial tem que ser obrigatoriamente maior que o tempo de validade final anterior, nem que seja por um instante temporal.

Considerando uma versão *V* com um atributo (ou relacionamento) temporalizado *a* sob uma linha de tempo *t*, as figuras 3.1a e 3.1b apresentam as possíveis combinações para os valores do rótulo temporal em relação ao atributo *alive* da versão, segundo as regras 1, 2 e 3. A Figura 3.1c apresenta os possíveis valores de validade inicial e final do *alive* para essa versão de um objeto versionado *OV*, regras 4 e 5.

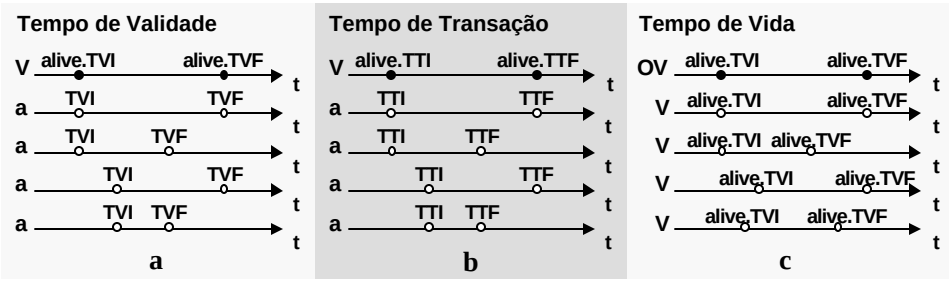


Figura 3.1 – Regras de integridade temporal

Para manter a integridade dos rótulos temporais são consideradas as regras definidas por Hübler [HÜB 00], divididas em regras para inserção e atualização de dados apresentadas a seguir. As regras de exclusão são apresentadas na seção sobre exclusão lógica e física (0).

Inserção e Atualização

A inserção é a primeira definição de um valor para um atributo. Qualquer informação que seja inserida em uma base de dados bitemporal deverá receber seus tempos de transação (fornecido pelo SGBD) e de validade (fornecido pelo usuário).

O usuário deve fornecer o tempo de validade inicial e, opcionalmente, o final da informação que está sendo inserida. O SGBD fornecerá o tempo de transação. O tempo de transação final é fornecido quando uma nova instância da informação é inserida, gerada por uma atualização da base de dados ou quando é excluída. Quando o usuário não fornecer o tempo de validade final, ele será igual a *null* (valendo até que outra informação seja definida ou o objeto seja excluído).

Em banco de dados bitemporais nenhum atributo é fisicamente modificado, exceto aqueles cujo tempo de transação final está em aberto [ELM 00]. As informações não são perdidas ou excluídas da base em atualizações, uma vez que o tempo de transação fornecido aos dados possibilita a distinção de duas ou mais informações que possuam tempo de validade iguais. Então, qualquer definição de novo valor é realizada com o tempo de transação definindo a semântica das informações. Para ilustrar como os atributos são atualizados, a Tabela 3.1 apresenta os dados (fisicamente armazenados) de funcionários de uma empresa, dos quais apenas o *nome* interessa na exemplificação. Cada tupla da tabela contém os valores do rótulo temporal associado.

A tabela informa que os dados da funcionária de nome “Maria de Lurdes Souza” foram armazenados no dia 15 de maio de 1989 e começaram a representar a realidade no dia primeiro de junho do mesmo ano, data de início de trabalho da funcionária. Supondo que a pessoa tenha casado e assumido o sobrenome do marido no dia 15 de dezembro de 1995, a base de dados foi atualizada no dia seguinte, encerrando a validade do nome de solteira e iniciando a validade do nome novo. Então, as seguintes atualizações físicas são aplicadas à tabela:

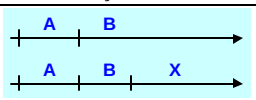
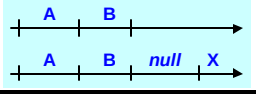
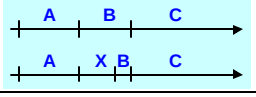
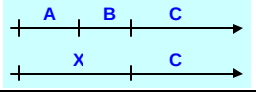
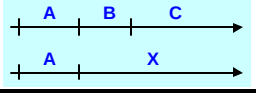
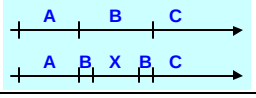
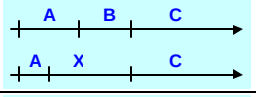
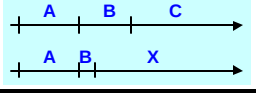
1. o valor da informação atual é copiado (*B*); *B*.TTI recebe o tempo da transação, *B*.TVF recebe TVI da informação nova menos 1, *B*.TTF recebe *null*, então *B* é uma cópia do valor corrente anterior (*A*) depois do mesmo ter sido encerrado em um tempo de validade TVI – 1;
2. novamente, copia-se o valor atual (*C*); *C*.TVI recebe o tempo de validade informado, *C*.TVF recebe *null*, *C.nome* recebe a nova informação, *C* representa a informação corrente;
3. atualiza-se *C*.TTF quando a informação corrente não representar mais a realidade.

Tabela 3.1 – Exemplo de atualização

	Nome	...	TVI	TVF	TTI	TTF
A	Maria de Lurdes Souza	...	01-06-1989	null	15-05-1989	**16-12-1995**
B	Maria de Lurdes Souza	...	01-06-1989	**15-12-1995**	16-12-1995	**null**
C	Maria de Lurdes Ferreira	...	16-12-1995	**null**	16-12-1995	**null**
	...	...	...	...	...	...

A Tabela 3.2 apresenta as possíveis combinações para atualização de valores e rótulos temporais, sendo adotada a seguinte notação: as informações existentes são representadas por *A*, *B* e *C*, e a nova informação por *X*.

Tabela 3.2 – Possibilidades de rótulos na atualização

Condição	Explicação e resultado	Representação gráfica
TVI_X > TVI	O TVI da nova informação (TVI_X) é maior que o TVI da informação armazenada	
- TVF atual infinito ou TVI_X < TVF	TVF da informação armazenada (B) recebe o novo TVI menos um instante	
- TVI_X maior que o último TVF	Tem um intervalo com valor *null* acrescentado cujo TVI é o TVF armazenado (B) mais um instante, e o TVF é o TVI novo menos um instante	
TVI_X = TVI	O TVI da nova informação (TVI_X) é igual a um TVI existente	
- TVF_X < TVF	O TVI da informação existente recebe o TVF da nova mais um instante.	
- TVF_X = TVF	A informação nova sobrepõe-se a uma ou mais informações armazenadas (A e B)	
- TVF_X infinito	A informação nova sobrepõe-se a uma ou mais informações armazenadas até o infinito (B e C)	
TVI_X < último TVI	O TVI da nova informação (TVI_X) é menor que o já armazenado	
- TVF_X < TVF	A nova informação fica contida dentro da já existente (B), então a já existente fica quebrada em duas partes	
- TVF_X = TVF	O TVF da informação armazenada (A) recebe o TVI da nova menos um instante	
- TVF_X infinito	O TVF da informação armazenada (B) recebe o TVI da nova menos um instante que vale a infinito	

Exemplificando melhor duas dessas situações, a Tabela 3.3 apresenta os valores das informações e dos rótulos para o caso TVI_X = TVI com TVF_X < TVF, e a Tabela 3.4 para o caso TVI_X < último TVI com TVF_X < TVF. As duas tabelas consideram que as informações *A*, *B* e *C* foram armazenados no dia primeiro de dezembro de 2000, e que a informação *X* surge no dia quinze de fevereiro de 2001. Os valores em negrito representam os dados alterados de acordo com a nova informação inserida. Para tornar as tabelas mais claras, os valores *null* estão omitidos.

Tabela 3.3 – Primeiro exemplo do uso dos rótulos

Informação	TVI	TVF	TTI	TTF
A	01-01-2001	28-02-2001	01-12-2000	
B	01-03-2001	30-04-2001	01-12-2000	**14-02-2001**
C	01-05-2001		01-12-2000	
X	**01-03-2001**	**14-03-2001**	**15-02-2001**	
B	**15-03-2001**	30-04-2001	**15-02-2001**	
...	...	...	...	...

Tabela 3.4 – Segundo exemplo do uso dos rótulos

Informação	TVI	TVF	TTI	TTF
A	01-01-2001	28-02-2001	01-12-2000	
B	01-03-2001	30-04-2001	01-12-2000	**14-02-2001**
C	01-05-2001		01-12-2000	
B	01-03-2001	**10-03-2001**	**15-02-2001**	
X	**11-03-2001**	**14-03-2001**	**15-02-2001**	
B	**15-03-2001**	30-04-2001	**15-02-2001**	
...	...	...	...	...

3.1.2 Exclusões Lógica e Física

A exclusão pode ser de duas formas: lógica e física. Quando uma versão é excluída logicamente, ela passa para o *status deactivated*, tem seu atributo *alive* atualizado para *false* e seu tempo de vida finalizado. Esse *status* não existe no Modelo de Versões e foi acrescentado para modelar o estado final de uma versão. No momento da exclusão lógica, se houver atributos ou relacionamentos temporalizados, os tempos finais de validade e transação em aberto recebem o mesmo valor de tempo final definido para o objeto.

A exclusão física é utilizada quando se deseja remover fisicamente uma informação. Essa operação é conhecida por *vacuuming* e é realizada raramente, somente quando se quer diminuir o número de dados armazenados ou por questões de segurança nas quais os dados não devem permanecer armazenados (caso comum das aplicações para Web que envolvem número de cartões de crédito, por exemplo). Entretanto, deve-se estar ciente de que essas informações serão perdidas [HÜB 00].

Outro ponto a ser considerado é que o suporte ao tempo de transação traz consigo o potencial para acessar qualquer estado passado do banco de dados. A exclusão física, por sua natureza, limita esse potencial, e ao mesmo tempo que pode ser necessária, deve-se cuidar para evitar um impacto negativo da utilidade dos dados que permanecem na base [JEN 99][1]. Em outras palavras, a exclusão física proíbe que

[1] O impacto da utilização desse tipo de exclusão na base de dados e na linguagem de consulta é descrito detalhadamente por Jensen em [JEN 99].

estados passados sejam reconstruídos [SNO 00]. O TVM não define as regras para esse tipo de exclusão, assumindo que todos os dados temporais são excluídos logicamente.

3.1.3 Hierarquia de Tipos Temporais

O termo *classe* é freqüentemente usado para referir a definição de um tipo de objeto, juntamente com as definições das operações para esse tipo. Um número de operações é declarado para cada classe, e a assinatura (*interface*) de cada operação é incluída na definição da classe. Um método (implementação) para cada operação é definido em algum lugar, usando uma linguagem de programação. Operações típicas incluem o *construtor do objeto* e o *destrutor.* Operações *modificadoras do objeto* podem também ser declaradas para alterar os atributos. Operações adicionais podem *recuperar* informação sobre o objeto [ELM 00].

Em modelos orientados a objetos, um *literal* é um valor que não tem um identificador de objeto. Entretanto o valor pode ser uma estrutura simples (atômico) ou complexa (estruturado e coleções). Esta seção apresenta o conjunto de tipos estruturados (classes) que modelam os rótulos temporais para atributos e relacionamentos. Também podem ser chamados de literais porque não armazenam o OID (Figura 3.2).

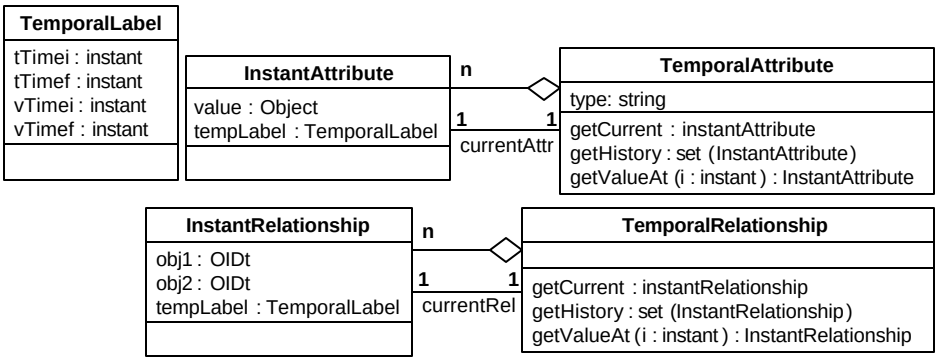


Figura 3.2 – Definição dos atributos e relacionamentos temporais

A classe base *TemporalLabel* armazena o rótulo temporal com os tempos de validade inicial e final, e transação inicial e final. Contém apenas as operações básicas: construtor, leitura dos atributos e modificação de atributos (Figura 3.3).

TemporalLabel
tTimei : instant tTimef : instant vTimei : instant vTimef : instant
TemporalLabel (iValidTime: instant, fValidTime: instant, iTransTime: instant, fTransTime: instant) getTTimei ( ) : instant getTTimef ( ) : instant getVTimei ( ) : instant getVTimef ( ) : instant setTTimef (i: instant) setVTimef (i: instant)

Figura 3.3 – Especificação da classe *TemporalLabel*

A classe *InstantAttribute* armazena o valor do atributo com seu rótulo temporal (Figura 3.4). Analogamente, *InstantRelationship* armazena os identificadores dos objetos (ou versões) que fazem parte do relacionamento e seu rótulo temporal (Figura 3.5). Ambas possuem duas opções de construtores: (i) recebe apenas o valor; (ii) recebe o valor, os tempos de validade inicial e final.

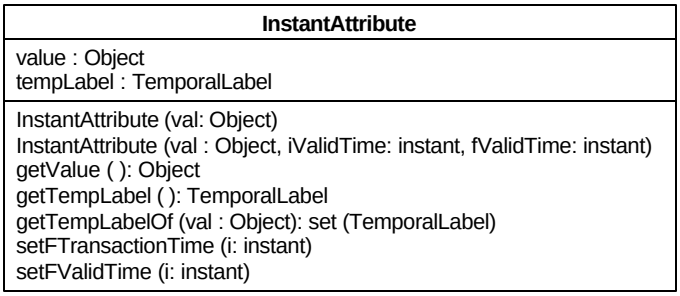


Figura 3.4 – Especificação da classe *InstantAttribute*

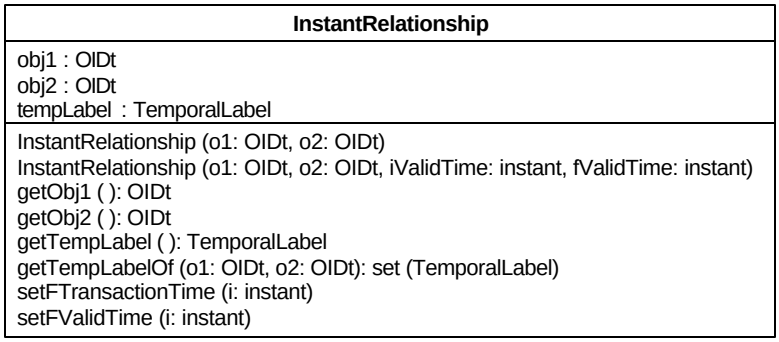


Figura 3.5 – Especificação da classe *InstantRelationship*

InstantAttribute e *InstantRelationship* possuem também as operações *getTempLabel* e *getTempLabelOf* para obter o rótulo temporal do valor corrente e o(s) rótulo(s) temporal(is) de um valor passado por parâmetro, respectivamente.

As classes *TemporalAttribute* e *TemporalRelationship* são agregados dos conjuntos de valores que os atributos e relacionamentos temporais assumem durante as vidas dos objetos. Além disso, essas classes possuem um relacionamento com o instante corrente (*currentAttr* e *currentRel*, respectivamente). As operações *getHistory* retornam todos os valores com os rótulos de tempo, *getHistoryOfValue* retornam o histórico do valor passado por parâmetro, *getValueAt* retornam os valores em um instante de tempo específico passado como parâmetro (Figura 3.6 e Figura 3.7). O hífen antes do nome da operação indica que sua visibilidade é particular.

Essas classes mantêm as duas opções de construtores das classes *InstantAttribute* e *InstantRelationship*, e seguem as regras de integridade descritas na seção anterior (3.1.1). As operações de modificação dos tempos finais são definidas como particulares porque apenas os métodos construtores, de atualização (*updateValue*) e de encerramento (*close*) podem modificar os valores dos tempos de transação e validade finais.

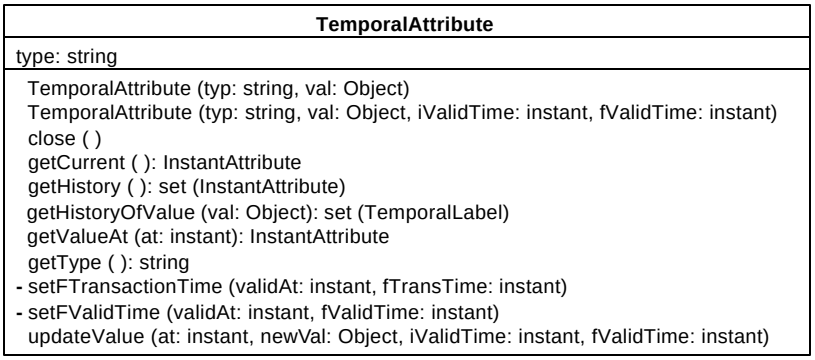


Figura 3.6 – Especificação da classe *TemporalAttribute*

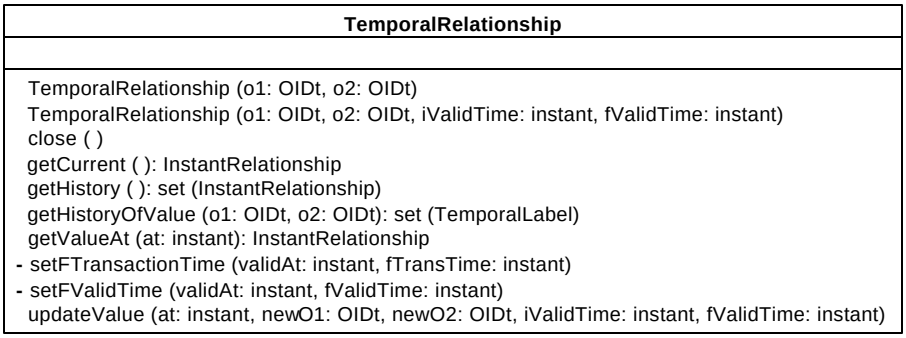


Figura 3.7 – Especificação da classe *TemporalRelationship*

3.2 Modificações do Modelo de Versões

A alteração mais simples realizada no Modelo de Versões foi a forma de representação gráfica. A notação apresentada na Tabela 3.5 foi estabelecida para representar graficamente as características temporais e de versões em um diagrama de classes.

Tabela 3.5 – Representação gráfica para os elementos do modelo

Símbolo ou Notação	Significado
T_V	Classe temporal e versionável
<<final>>	Classe final, não pode ser especializada
Itálico	No nome da classe, indica classe abstrata
<<temporal >>	Relacionamento temporal
<<T>>	Atributo temporal
t	
<<extension>>	Relacionamento de herança por extensão
negrito	Operação final (*final*), não pode ser redefinida
$	Atributo ou operação de escopo de classe (*static*)
+	Atributo ou operação pública (*public*)
-	Atributo ou operação particular (*private*)
#	Atributo ou operação protegida (*protected*)

A Figura 3.8 apresenta um breve exemplo do uso de alguns desses elementos gráficos. São ilustradas quatro classes:

- *ClasseNormal* – classe normal, ou seja, sem tempo e versões. Apresenta cinco tipos de atributos (normal, de classe, público, particular e protegido), os seis tipos de operações (normal, final, de classe, público, particular e protegido), um relacionamento normal (não temporal) com a classe *ClasseTVfinal*, e um relacionamento de herança por extensão com a classe *ClasseAbstrata*;

- *ClasseAbstrata* – classe normal abstrata;

- *ClasseTVfinal* – classe temporal versionada final. Apresenta um atributo normal e um temporal, os seis tipos de operação, e um relacionamento temporal com a classe *ClasseTVnormal*;

- *ClasseTVnormal* – classe temporal versionada.

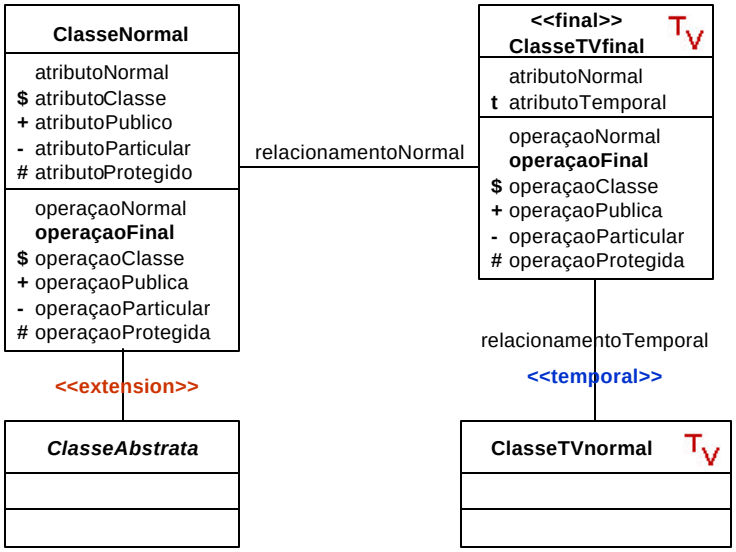


Figura 3.8 – Exemplo do uso da notação

3.2.1 Estados de uma Versão

Durante seu tempo de vida, uma versão pode passar por alguns estados. As transições entre eles e os eventos que ocasionam tais transições estão representados no diagrama de estados na Figura 3.9.

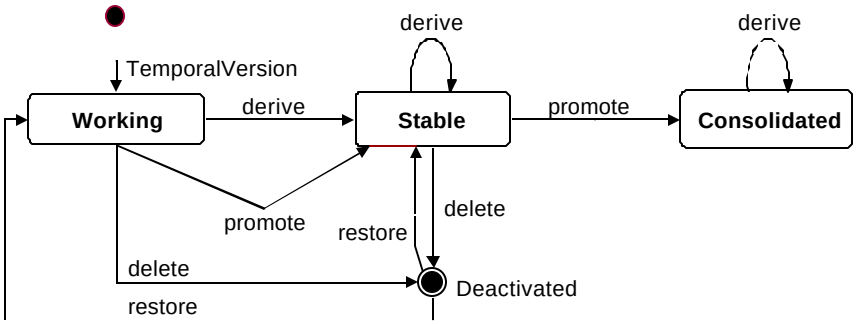


Figura 3.9 – Diagrama de estados de uma versão

Cada estado define o conjunto de operações que podem ser aplicadas sobre uma versão, conforme a Tabela 3.6. Desse modo, quando uma versão está no estado:

- *Working* – pode servir como base de uma derivação, ser promovida para *Stable*, ser alterada, consultada ou ainda excluída (logicamente);
- *Stable* – pode servir como base de uma derivação, ser promovida para *Consolidated*, consultada, compartilhada por outros usuários, excluída (logicamente) desde que não possua nenhuma versão sucessora, e não pode ser alterada;
- *Consolidated* – pode servir como base para derivação, ser consultada, compartilhada por outros usuários, e não pode ser alterada nem excluída;
- *Deactivated* – pode ser apenas consultada e restaurada.

Tabela 3.6 – Estados das versões e respectivas operações

Estados / Operações	*Working*	*Stable*	*Consolidated*	*Deactivated*
Derivar	**S**	**S**	**S**	–
Promover	**S**	**S**	–	–
Alterar	**S**	**N**	**N**	**N**
Excluir	**S**	**S** (se não possui sucessora)	**N**	**N**
Consultar	**S**	**S**	**S**	**S**
Compartilhar	**N**	**S**	**S**	**N**
Restaurar	–	–	–	**S**

Legenda: **S** *pode ser realizada* **N** *não pode ser realizada* – *não definida*

3.2.2 Hierarquia de Classes

A hierarquia de classes proposta por Golendziner foi alterada para atender às características temporais acrescentadas ao modelo. A definição das classes da hierarquia pode ser encontrada no Anexo 2. A Figura 3.10 apresenta a nova hierarquia.

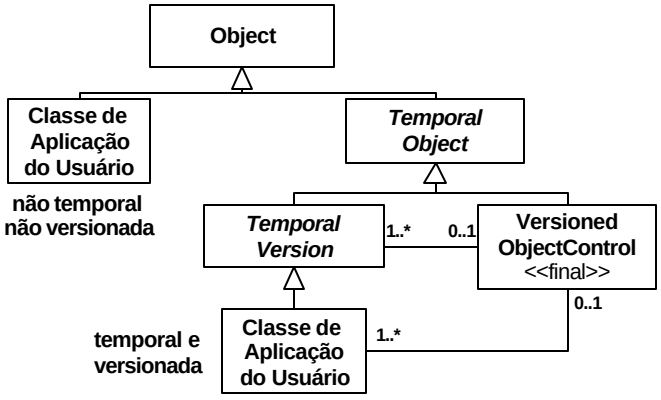


Figura 3.10 – Hierarquia de classes do TVM com as classes de aplicação

A hierarquia do TVM permite especificar dois tipos de classes da aplicação:

- *classe de aplicação não temporal e não versionável* – definida como subclasse de *Object*. Sua modelagem pode ser usada para contemplar classes que representam tabelas de uma base já existente ou classes auxiliares nas quais os conceitos de tempo e versões não são necessários. As informações temporais dessas classes não são armazenadas, ou seja, só são armazenados os valores atuais dos dados;

- *classe de aplicação temporal versionada* – definida como subclasse de *TemporalVersion*. Possui um relacionamento de associação com a classe *VersionedObjectControl*. Seus atributos e relacionamentos podem ser definidos como estáticos ou temporalizados, e suas instâncias são versões com o rótulo de tempo associado (atributo *alive* herdado de *TemporalObject*). As instâncias desse tipo de classe de aplicação podem ser objetos não versionados, versionados e as próprias versões.

Olhando essa hierarquia, a possibilidade de criar classes temporais diretamente de *TemporalObject* poderia ser cogitada. Essa hipótese foi descartada porque o objetivo principal é modelar ambas as características de tempo e versões nos objetos. O Modelo permite somente os dois tipos de classe descritos.

As classes *TemporalObject* e *VersionedObjectControl* são controladas pelo sistema gerenciador e ficam transparentes ao usuário. As classes *TemporalObject* e *TemporalVersion* são *classes abstratas*, não podem ser instanciadas diretamente. A classe *VersionedObjectControl* só pode ser instanciada pelo sistema gerenciador com o intuito de administrar os objetos que possuem versões. Além disso, essa classe é denominada *classe final* não sendo permitido especializá-la em subclasses.

Duas características do Modelo de Versões são diretamente atingidas por essa nova hierarquia:

- herança múltipla – o objeto versionado não é mais uma instância da classe de aplicação e *VersionedObject*. O controle sobre as versões de um objeto versionado, em vez de ser exercido pelas características herdadas de *VersionedObject*, agora faz parte da classe *VersionedObjectControl*, com a qual cada versão e cada objeto versionado tem um relacionamento estabelecido;

- dinamicidade de não versionado para versionado – na nova hierarquia, não é permitido que uma instância mude de não versionada para versionada dinamicamente, como no Modelo de Versões. O usuário deve estipular em tempo de projeto quais classes podem ter objetos versionados e defini-las como temporal versionadas (subclasses de *TemporalVersion*). Completando, é totalmente garantido o direito de um objeto de uma classe temporal versionada não ter versão associada, ou seja, ser um *objeto sem versões*.

Resumindo, a Figura 3.11 apresenta graficamente os objetos instanciados da classe de aplicação temporal versionada *CL* (subclasse de *TemporalVersion*). São apresentados um objeto sem versões (*Obj1*) e um objeto versionado (*Obj2*) com suas respectivas versões (*V1*, *V2*, *V3*, *V4*).

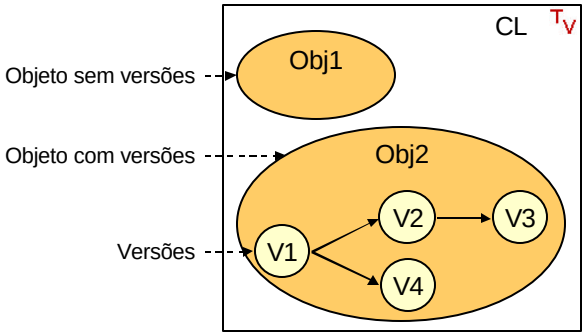


Figura 3.11 – Objetos instanciados de uma subclasse de *TemporalVersion*

3.2.3 Atributos e Operações das Classes

Algumas operações das classes do Modelo envolvem o acesso a metadados, os quais são utilizados para armazenar informações a respeito de todo o esquema (definições das classes). Além das informações padrão (tvOID, nome, cardinalidade, ...) as classes de metadados devem armazenar todos os novos conceitos definidos pelo TVM. Por exemplo, em algum lugar deve ficar registrado se a classe é temporal versionada, se o relacionamento é herança por extensão e o nome de sua classe ascendente ou se é classe raiz, quais atributos e relacionamentos são temporais, e assim por diante. Partindo do pressuposto que as classes de metadados já estão definidas no banco de dados gerenciando todas as informações, esta seção apresenta em detalhes os atributos e operações das classes base do TVM.

A inclusão das novas classes gera a necessidade de uma realocação dos atributos e operações da hierarquia base do Modelo. Além disso, sendo o TVM um modelo orientado a objetos, optou-se por fazer uma releitura do Modelo de Versões deixando-o mais próximo de alguns conceitos dos padrões adotados em modelagem (UML e ODMG) e linguagens orientadas a objetos (C++ e Java). Por exemplo, o TVM atualiza a operação *create_object* (responsável pela instanciação de um objeto) do Modelo de Versões para construtor da classe no TVM (*Object ( )*).

A Figura 3.12 mostra o resultado da realocação das operações do Modelo de Versões. As operações mais comuns de obter atributos, atribuir valores e criar objetos estão implícitas em todas as classes nesta figura. O conjunto completo das operações pode ser conferido no detalhamento das classes. Os nomes dos atributos e operações foram atualizados mantendo o seguinte padrão:

- nome de atributo e operação começa com caracter minúsculo (com exceção dos construtores);
- nome de atributo e operação formado por mais de uma palavra tem as palavras seguintes à primeira iniciadas com caracter maiúsculo;
- nome de operação começa por verbo, sendo que operações de leitura de atributos começam por *get* e de modificação por *set*. Por definição, atributos que não têm um método de modificação próprio recebem seu valor na instanciação do objeto pelo construtor da classe ou pela execução de outro método.

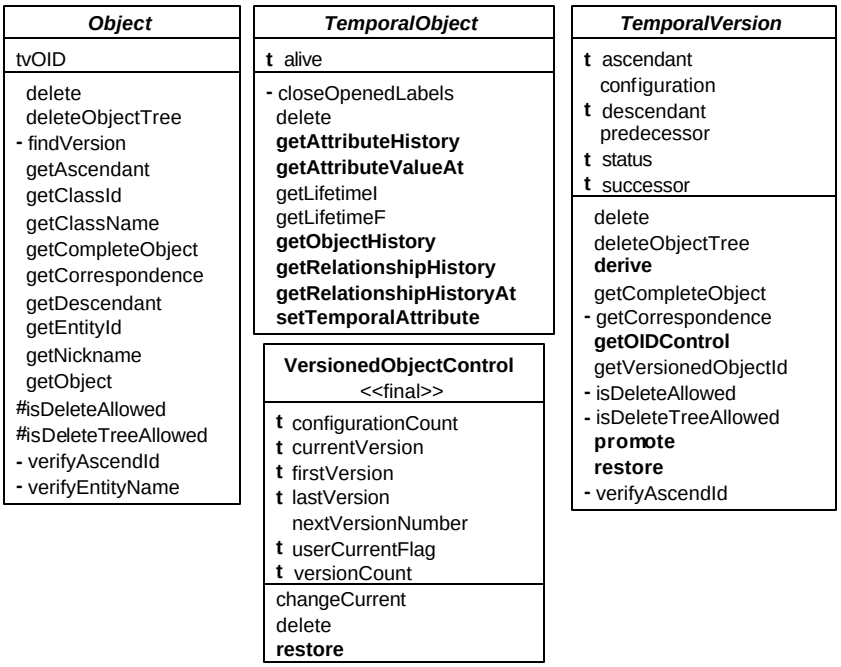


Figura 3.12 – Classes da hierarquia com principais atributos e operações

Classe *Object*

A classe *Object* tem os atributos e operações comuns a todos os objetos (Figura 3.13). O OID é renomeado para tvOID para não confundir com o OID gerado automaticamente pelo SGBD orientado a objetos. O tvOID mantém a estrutura definida anteriormente (identificador da entidade, da classe, e número da versão) e é modelado

através da classe *OIDt*. Os objetos das classes comuns (sem tempo e versão) têm a mesma estrutura de tvOID, com a diferença que o número da versão é sempre *null*. Quando o construtor recebe um tvOID como parâmetro, o método particular *verifyAscendId* verifica se o tvOID é válido para a classe ascendente. Quando o construtor recebe um identificador de entidade como parâmetro, deve conferir nos metadados se é um valor válido. Quando recebe entidade, classe e versão por parâmetro deve verificar se os valores são válidos e se o objeto já não existe.

As operações *create_object* e *create_specialized_object* são substituídas pelos construtores da classe *Object*. A operação *modify* é substituída pelas operações que atualizam os atributos. As operações *derive_version*, *get_configuration*, *create_versioned_object*, *create_specialized_versioned_object* são realocadas para a classe *TemporalVersion*. As operações relativas aos ascendentes e descendentes são explicadas na seção sobre herança por extensão (seção 3.3).

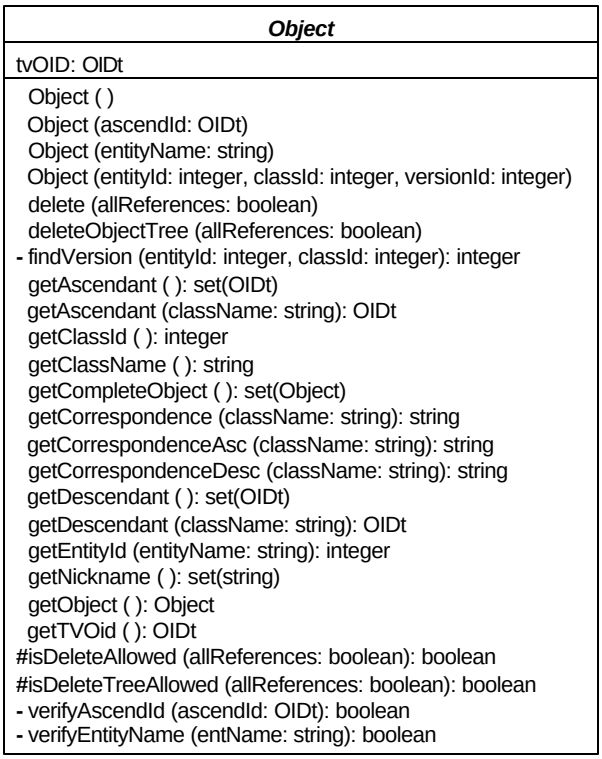


Figura 3.13 – Especificação da classe *Object*

Foram acrescentadas as seguintes operações:

- *findVersion* – operação particular, retorna o número da versão a ser criada;
- *getClassName* – consulta os metadados e retorna o nome da classe a qual o objeto pertence;
- *getClassId* – consulta os metadados e retorna o identificador da classe;
- *getCorrespondence* – consulta os metadados e retorna a correspondência entre o objeto e a classe recebida por parâmetro;
- *getCorrespondenceAsc* – consulta os metadados e retorna a cardinalidade do ascendente na correspondência entre o objeto e a classe recebida por parâmetro;

- *getCorrespondenceDesc* – consulta os metadados e retorna a cardinalidade do descendente na correspondência entre o objeto e a classe recebida por parâmetro;
- *getEntityId* – consulta os metadados e retorna o identificador da entidade;
- *isDeleteAllowed* – retorna se um objeto pode ser excluído;
- *isDeleteTreeAllowed* – retorna se um objeto e todos os seus descendentes podem ser excluídos;
- *verifyAscendId* – verifica se o ascendente recebido por parâmetro é válido;
- *verifyEntityName* – verifica se a entidade recebida por parâmetro é válida.

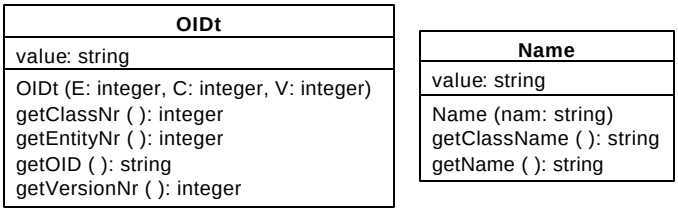


Figura 3.14 – Especificação das classes *OIDt* e *Name*

Object tem um relacionamento com a classe *Name*, chamado *nickname*, que mantém uma lista de apelidos para os objetos (opcional). Sua especificação e da classe *OIDt* são apresentadas na Figura 3.14.

Classe *TemporalObject*

A classe *TemporalObject* foi adicionada à hierarquia para representar os aspectos temporais do modelo (Figura 3.15). É nessa classe que está o atributo temporal *alive* que armazena se o objeto está ativo ("vivo") ou não. Esse atributo é modelado como temporal porque os objetos temporais podem ser restaurados. Desse modo, os tempos de vida inicial e final dos objetos são armazenados nos valores *alive.vTimei* e *alive.vTimef*, respectivamente.

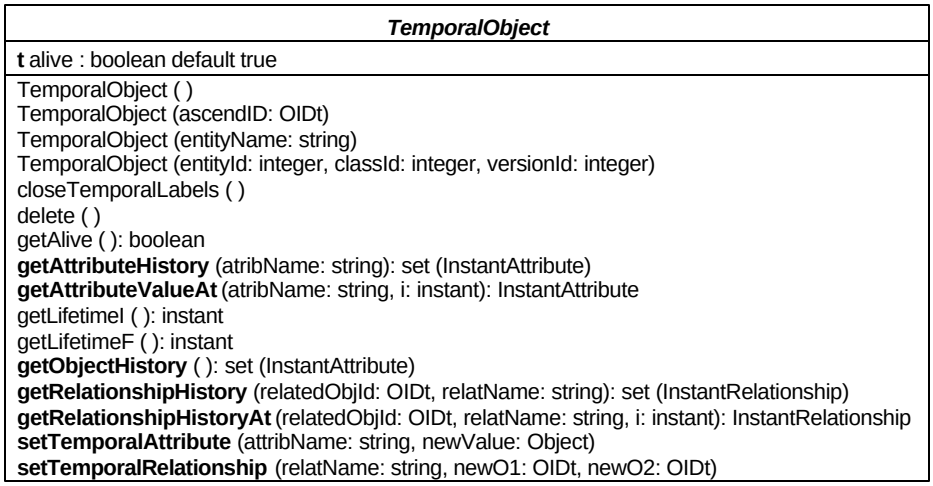


Figura 3.15 – Especificação da classe *TemporalObject*

São definidas as seguintes operações:

- *closeTemporalLabels* – percorre atributos e relacionamentos temporais atualizando os dados que possuem tempo de validade e/ou transação em aberto;

- *delete* – executa a exclusão lógica do objeto;
- *getAttributeHistory* – retorna a coleção de valores do seu histórico com os respectivos rótulos temporais se o atributo passado como parâmetro for temporal, e o valor atual com o rótulo temporal contendo *null* se o atributo for não-temporal;
- *getAttributeValueAt* – retorna o valor do atributo no instante recebido por parâmetro, e o valor atual com o rótulo temporal contendo *null* se o atributo for não-temporal;
- *getLifetimeI* – retorna o tempo de vida inicial do objeto ativo no momento;
- *getLifetimeF* – retorna o tempo de vida final do objeto. Se ele está ativo o valor é *null*, caso contrário é o valor mais recente (pode ter mais de um, no caso de ter sido excluído e restaurado);
- *getObjectHistory* – retorna o conjunto dos valores iniciais e finais de *alive*;
- *getRelationshipHistory* – retorna a coleção de valores do seu histórico com os respectivos rótulos temporais se o relacionamento passado como parâmetro for temporal, e o valor atual com o rótulo temporal contendo *null* se o relacionamento for não-temporal;
- *getRelationshipValueAt* – retorna os valores dos tvOIDs do relacionamento no instante recebido por parâmetro, e o valor atual com o rótulo temporal contendo *null* se o relacionamento for não-temporal;
- *setTemporalAttribute* – atualiza o valor do atributo temporal armazenando os respectivos valores no rótulo temporal;
- *setTemporalRelationship* – atualiza o valor do relacionamento temporal armazenando os respectivos valores no rótulo temporal.

Classe *VersionedObjectControl*

A classe *VersionedObjectControl* possui basicamente os mesmos atributos e operações da classe *VersionedObject* no Modelo de Versões, com as mesmas funções (Figura 3.16). A diferença é que os atributos *configurationCount*, *currentVersion*, *firstVersion*, *lastVersion*, *userCurrentFlag* e *versionCount* são definidos como temporais, uma vez que seus valores podem evoluir com o tempo e o histórico deve ficar armazenado.

Os atributos *configurationCount*, *nextVersionNumber* e *versionCount* contêm valores iniciais especificados como 0, 3 e 2, respectivamente. Esses valores dizem respeito à instanciação de *VersionedObjectControl.* Um controle para o objeto versionado é criado apenas na primeira derivação de versão. O objeto fica com a primeira versão, que era a única até a derivação, e a segunda, que é a nova derivada. Nesse momento, o objeto versionado tem duas versões e o número da próxima é três.

A operação *change_current* é renomeada para *setCurrentVersion* para manter o padrão estabelecido para o TVM. Essa operação permite que o usuário troque a versão corrente que, enquanto o usuário não a executa, é automaticamente mantida pelo sistema como a última versão, podendo ser uma versão configurada. Quando o usuário informa o tvOID da versão por parâmetro, essa passa a ser a corrente, e o atributo *userCurrentFlag* é atualizado para *true*. Quando o usuário não passa o valor por

parâmetro, o sistema assume de volta o controle da versão corrente que recebe o tvOID da última criada, e *userCurrentFlag* retorna a *false.*

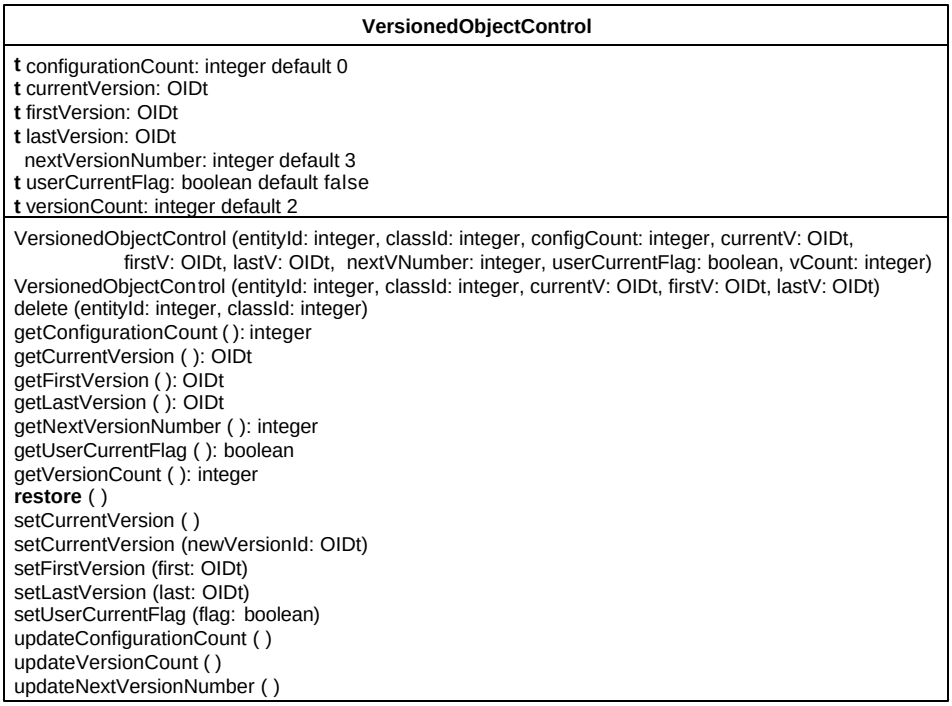


Figura 3.16 – Especificação da classe *VersionedObjectControl*

Classe *TemporalVersion*

A classe *TemporalVersion* possui basicamente os mesmos atributos e operações definidos na classe *Version* no Modelo de Versões, com as mesmas funções (Figura 3.17). Os atributos *status* e *successor* são definidos como temporais. Os atributos *ascendant* e *descendant* também são definidos como temporais e detalhados na seção sobre herança por extensão. Há uma realocação das operações *derive* (renomeada de *derive_version*), *getConfiguration* e *getOIDControl* (renomeada de *getVersionedObject*) que saem da classe *Object* do Modelo de Versões.

A operação *derive* permite que uma nova versão seja gerada para um objeto, derivada de uma ou mais versões ou objetos. A versão que serviu de base para a derivação é automaticamente promovida para *Stable* (se seu estado era *Working*). A nova versão é criada no estado *Working*. Pode ser fornecido um único tvOID ou um conjuntos de tvOIDs que indicam a partir de quais objetos a nova versão está sendo derivada. Se for um único, o tvOID pode ser de uma versão, objeto versionado ou sem versões. Se for fornecido um conjunto, os tvOIDs devem ser identificadores de versões.

A regra da derivação a partir de várias versões estabelecida por Golendziner continua a mesma, ou seja: é feita uma cópia da primeira versão fornecida por parâmetro, o usuário deve fazer um agrupamento das demais versões, e a versão derivada torna-se automaticamente sucessora das versões fornecidas. Apenas o método *derive* pode chamar o construtor (*TemporalVersion*) que recebe o predecessor como parâmetro, e exatamente por isso esse construtor é definido como particular. O mesmo acontece para os métodos *addSuccessor*, *setSuccessor* e *removeSuccessor*, pois um sucessor só é adicionado na derivação e removido na exclusão lógica.

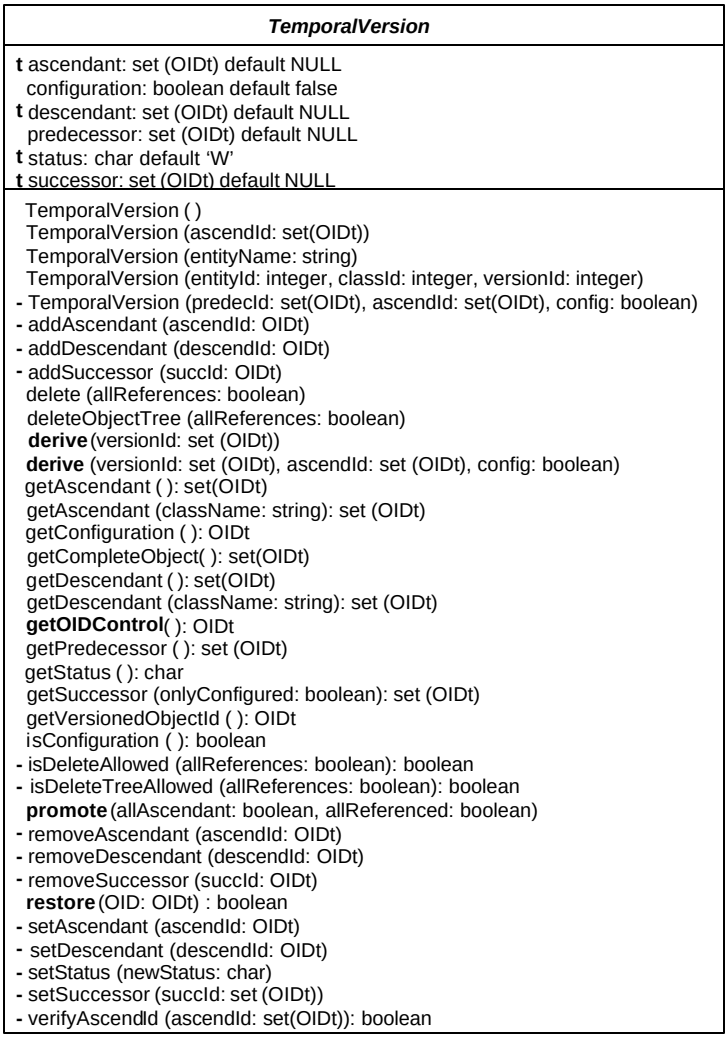


Figura 3.17 – Especificação da classe *TemporalVersion*

A operação *promote* continua com a mesma função, ou seja, atualiza o *status* da versão ou do objeto sem versões. Considerando que o *status* representa um estágio de desenvolvimento, ele deve ser o mesmo ou menos desenvolvido que os dos ascendentes. Por exemplo, se uma versão possui estado *Stable*, suas versões ascendentes só podem ter *Stable* ou *Consolidated*. Para garantir essa condição, a opção *allAscendants* permite que quando uma versão é promovida, seus ascendentes sejam automaticamente promovidos para o mesmo *status*, quando já não o possuírem. Se essa opção não for usada e houver ascendentes que violam a regra definida, a operação retorna erro. De forma análoga, *allReferences* permite que todos os objetos referenciados por essa versão sejam também promovidos. Essa opção é desejável, considerando-se que pode ser necessário promover uma versão juntamente com toda a hierarquia de agregação formada com essa versão raiz. A operação *promote* considera recursivamente os ascendentes até o objeto raiz.

A operação *setStatus* é particular (*private*) porque o estado *Working* é atribuído pelo construtor, e os demais pelos métodos *derive*, *promote*, *delete* e *restore*. As operações *derive*, *getOIDControl*, *promote* e *restore* são definidas como operações finais por serem funções-chave do Modelo. A operação *getOIDControl* retorna o tvOID da respectiva instância de *VersionedObjectControl*.

A operação *delete* é especializada na classe para verificar as condições de exclusão de versões e objetos versionados e armazenar os dados para a exclusão lógica. Essa operação recebe o parâmetro *allReferences* que, se verdadeiro, indica que as referências para o objeto são excluídas com o mesmo. É definido um método particular *isDeleteAllowed* que verifica se uma versão ou objeto versionado pode ser excluído. Condições a serem verificadas:

- se a versão (ou objeto) possui sucessora, descendente, faz parte de configuração ou objeto composto, a exclusão é negada;
- se a versão (ou objeto) possui referências e o parâmetro *allReferences* é *false*, a exclusão é negada. Nesse caso, o usuário deve excluir todas as referências, uma a uma, ou passar o parâmetro com valor *true*;
- se a versão (ou objeto) não possui predecessora, ela é a primeira e única versão do objeto versionado. Nesse caso, o objeto fica sem versões e mantém o valor de seus atributos e o relacionamento com *VersionedObjectControl*;
- se o objeto não possui versões, é excluído juntamente com a respectiva instância de *VersionedObjectControl* (se houver).

Alterações nos atributos e relacionamentos realizados pelo método *delete*:

- quando a versão (ou o objeto versionado) possui ascendente, se é o único descendente desse, o atributo *descendant* da ascendente recebe *null* (de todos os ascendentes no caso de ter mais de um), e se a versão pertence a uma lista de descendentes, é retirada da mesma (através do método *removeDescendant*);
- quando a versão possui predecessora, se é a única sucessora dessa, o atributo *successor* da predecessora recebe *null* (de todas as predecessoras no caso de ter mais de uma), e se a versão pertence a uma lista de sucessoras, é retirada da mesma (através do método *removeSuccessor*);
- o *status* da versão (ou do objeto versionado) é atualizado para *deactivated*;
- os atributos *alive.vTimef* e *alive.tTimef* da versão (ou do objeto versionado) recebem o valor do tempo de transação;
- a operação *closeTemporalLabels* é executada para encerrar os tempos de transação e validade em aberto.

A operação *deleteObjectTree* é definida para excluir o objeto e toda sua árvore descendente. Por motivos de segurança, a operação *delete* não permite que o usuário exclua uma versão e todos os seus descendentes, porque essa pode não ser exatamente a sua intenção. Antes de proceder à exclusão, deve-se verificar se todos os objetos, versões e objetos versionados descendentes satisfazem às condições de não possuir versão sucessora, fazer parte de configuração ou objeto composto. Essas verificações são realizadas pelo método particular *isDeleteTreeAllowed*. Se qualquer um dos descendentes não satisfizer essas condições, a exclusão é negada. Se a exclusão for executada, as alterações nos atributos e relacionamentos de cada um dos objetos seguem às regras estabelecidas por *delete*.

Além dessas operações, para que a realidade seja completamente modelada, o TVM permite a restauração de objetos que foram excluídos logicamente, através da operação *restore*. O exemplo mais simples da necessidade desse aspecto é a situação na qual algum objeto é excluído erroneamente. Outra situação é quando se deseja reavaliar

uma alternativa descartada anteriormente. Obviamente, o processo de restauração de um objeto com ou sem versões, ou de uma versão, implica no armazenamento de mais um valor de *alive.vTimei* no histórico do respectivo objeto. A restauração acontece para:

- uma versão folha que possui predecessora (não é a primeira versão do objeto). A Figura 3.18 apresenta o caso mais simples de restauração de uma versão excluída (Figura 3.18a: *V3*). Independente da derivação de outras versões (Figura 3.18b: *V4*) a partir da mesma versão (*V1*), qualquer versão folha pode ser restaurada com os valores dos atributos que possuía no momento da exclusão (Figura 3.18c) que recebem como validade inicial o tempo de transação no momento da restauração. A Figura 3.18d apresenta como podem ficar as linhas de tempo com os tempos de validade inicial e final do atributo *alive* dessas versões e do objeto versionado;

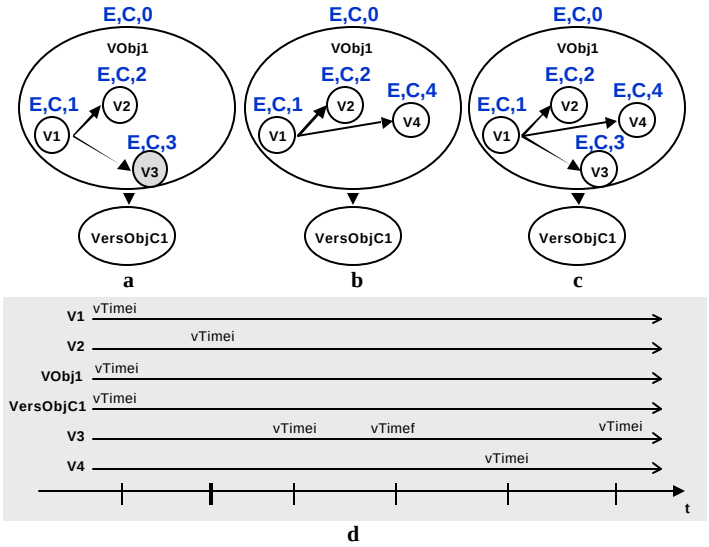


Figura 3.18 – Restauração de uma versão

- uma versão sem predecessora (primeira versão). Esse é um caso especial de restauração no qual a versão sem predecessora só pode ser restaurada se não existe uma primeira versão definida, ou seja, o objeto não tem versões. A Figura 3.19 ilustra essa situação. As versões do objeto foram excluídas (Figura 3.19a e Figura 3.19b), e o usuário criou uma versão (Figura 3.19c: *V4*). Para restaurar qualquer versão excluída anteriormente (*V3*), o usuário deve excluir a primeira versão atual (*V4*) e a sua hierarquia de derivação, se houver (Figura 3.19d). Os tempos das validades inicial e final do atributo *alive* são apresentados na Figura 3.19e;

- uma versão cuja predecessora foi excluída. A Figura 3.19d também ilustra essa situação, pois para poder restaurar a versão *V3*, deve-se restaurar sua predecessora (*V1*) e manter a sua seqüência de derivação;

- todas as versões do objeto. É possível também restaurar todas as versões do objeto, desde que não se tenha gerado outra seqüência de derivação. Novamente, a Figura 3.19 pode ilustrar essa situação, caso a versão *V2* seja restaurada juntamente com *V1* e *V3* na Figura 3.19d;

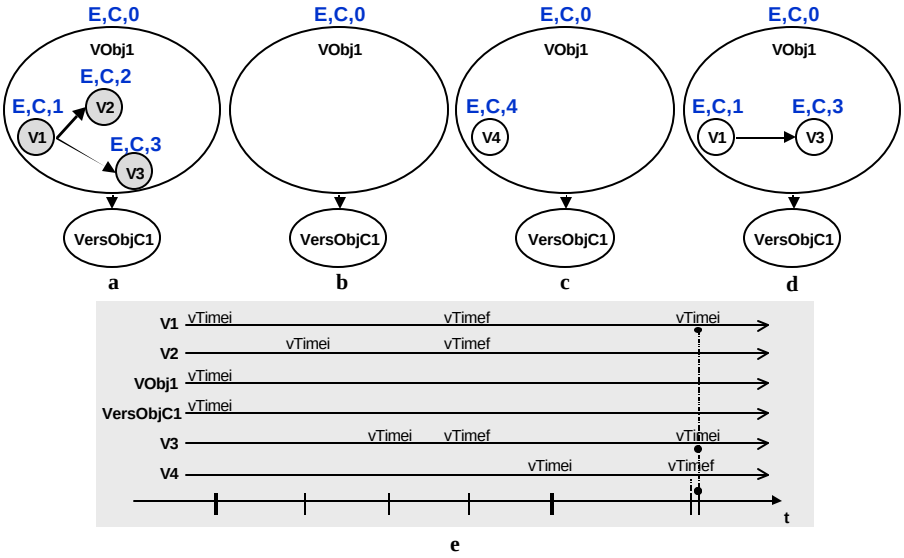


Figura 3.19 – Restauração de uma primeira versão e suas sucessoras

• um objeto versionado com versões. A Figura 3.20 apresenta a situação na qual um objeto versionado é excluído juntamente com suas versões (Figura 3.20b). A referência ao controle do objeto versionado e o objeto controle também são excluídos (Figura 3.20c), pois não faz sentido manter o controle de um objeto que não existe mais. A restauração ativa o objeto versionado como estava no momento da exclusão, ou seja, qualquer outra versão previamente excluída não é restaurada (Figura 3.20d). Os tempos das validades estão na Figura 3.20e.

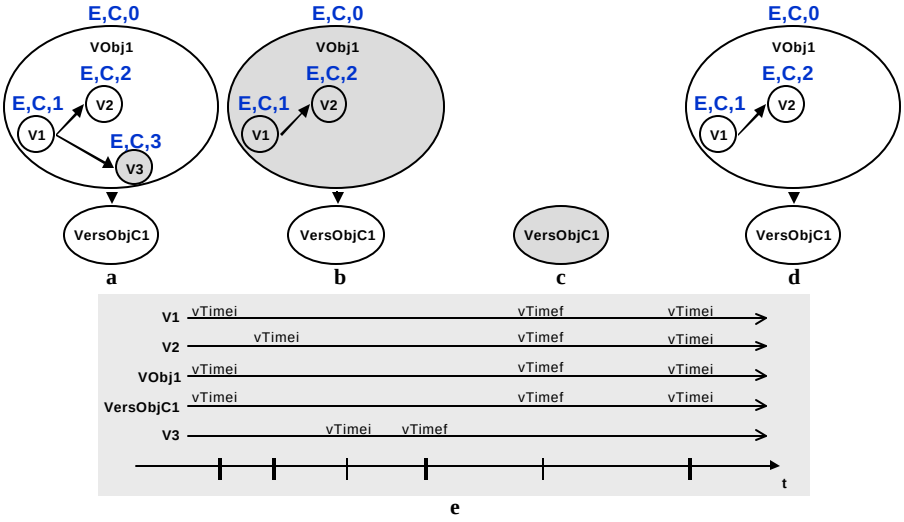


Figura 3.20 – Restauração de um objeto versionado

O método *restore* envolve muitas outras questões além das apresentadas. Entre elas, como restaurar os objetos ascendentes e se os relacionamentos para objetos excluídos também devem ser restaurados juntamente com os objetos.

3.2.4 Funcionamento da Hierarquia

Para tornar o entendimento da hierarquia de classes mais claro, é apresentado um conjunto de tabelas que mostra passo a passo a especificação de classes, criação de objetos, derivação de versões, e a atualização nos atributos que essas tarefas realizam.

A Tabela 3.7 apresenta a parte inicial do funcionamento com a especificação de uma classe pelo usuário. A Tabela 3.8 apresenta a criação de um objeto e a sucessiva derivação de uma versão. Com a derivação da versão, os dados do objeto anteriormente criado pelo usuário permanecem no mesmo objeto, no caso a primeira versão do objeto versionado. O objeto versionado é criado com os valores dos atributos iguais aos da versão corrente. Essa nova instância da classe é fundamental para o uso de referências dinâmicas que irão referenciar diretamente o seu tvOID.

Tabela 3.7 – Especificação da classe pelo usuário

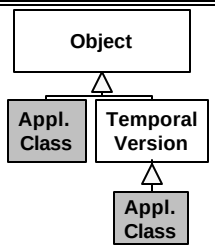
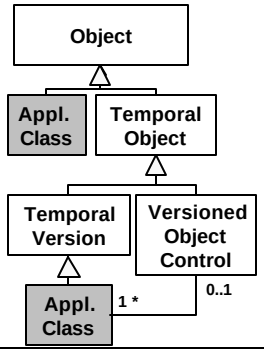
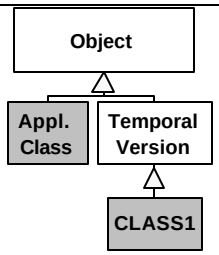
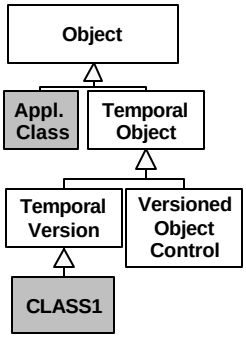
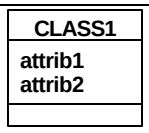
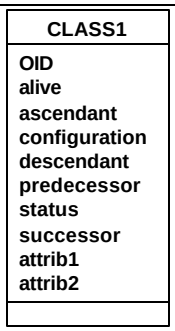
Usuário	Sistema	Explicação
		Como o usuário enxerga a hierarquia do sistema.
		O usuário define uma classe *CLASS1* como temporal versionada (subclasse de *TemporalVersion*).
		Definição da classe *CLASS1* com dois atributos: *attrib1* e *attrib2*, sendo que para o sistema a classe apresenta todos os atributos e operações definidos em *TemporalObject* e *TemporalVersion*. Visando uma apresentação gráfica mais clara, as operações são omitidas. Essa classe é definida com o estado *Working*.

Tabela 3.8 – Criação de objeto e versão pelo usuário

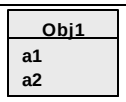
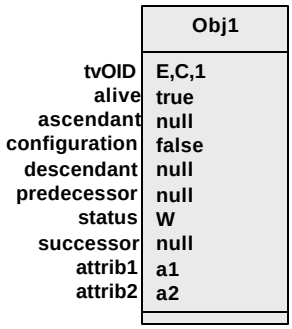
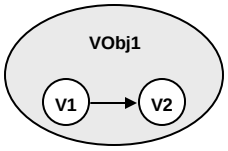
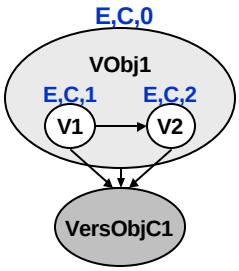
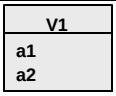
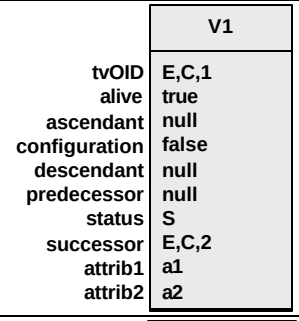
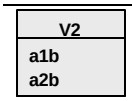
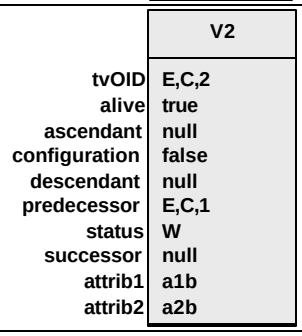
Usuário	Sistema	Explicação
		Através do método construtor, o usuário define o objeto *Obj1* de *CLASS1* sem versões, que recebe o tvOID **E,C,1** (**E** para o identificador da entidade e **C** da classe).
		Os valores do objeto (não versionado) são simples: *ascendant* é o identificador da classe ascendente (*null* porque é a classe raiz da hierarquia por extensão), *configuration* só é verdadeiro se o objeto participar de uma configuração (o que não é o caso), *descendant* e *predecessor*, e *successor* são null, o *status* é *Working*, e os atributos possuem os valores informados pelo usuário (*a1* e *a2*).
		Através da operação *derive* o usuário deriva uma versão de *Obj1*. A evolução do objeto não versionado para versionado provoca a criação de um objeto versionado *VObj1* (**E,C,0**), não influindo na identificação do objeto existente (**E,C,1**), que passa a ser a primeira versão (*V1*). A versão derivada passa a ser a segunda (*V2*). O sistema cria uma instância de *VersionedObjectControl* (*VersObjC1*) e as referências a partir de *V1*, *V2* e *VObj1*. O *status* de *V1* é atualizado para *Stable*.

Tabela 3.9 – Valores das instâncias das classes

Usuário	Sistema	Explicação
		Na derivação, o atributo *successor* recebe o valor na nova versão que está sendo criada, e o *status* passa para *Stable* (*S*)
		Quando deriva a versão, *V2* possui os valores dos atributos iguais aos de *V1*, depois o usuário através das operações de atualização modifica os valores como quiser. Nesse caso, os valores dos atributos foram atualizados para *a1b* e *a2b*. O atributo *predecessor* recebe o valor da versão da qual foi originada, no caso a *V1*.

Usuário	Sistema	Explicação
VObj1 a1b a2b	VObj1 tvOID E,C,0 alive true ascendant null configuration false descendant null predecessor null status W successor null attrib1 a1b attrib2 a2b ↓ VersObjC1 tvOID OID alive true configurationCount 0 currentVersion E,C,2 versionCount 2 firstVersion E,C,1 lastVersion E,C,2 nextVersionNumber 3 userCurrentFlag false	Como explicado anteriormente, o objeto versionado é criado com os valores dos atributos iguais aos da versão corrente. Seu *status* é sempre *Working* porque a cada modificação nas versões seus atributos podem variar. A instância de *VersionedObjectControl* recebe os seguintes valores em seus atributos: *configurationCount* e *versionCount* recebem a quantidade de configurações e versões do objeto, respectivamente; *currentVersion* recebe o tvOID de *V2*, a menos que o usuário tenha estabelecido explicitamente *V1* como corrente (*userCurrentFlag* seria verdadeiro); *firstVersion* e *lastVersion* recebem os tvOIDs de *V1* e *V2*, respectivamente; *nextVersionNumber* recebe o valor do número de versão do tvOID da versão mais recente (*V2*) e incrementa um.

Pode-se afirmar que o conceito de objeto versionado é apresentado na hierarquia pela união da instância da subclasse de *TemporalVersion* do objeto versionado (apresentado por *VObj1* na Tabela 3.8) com uma instância de *VersionedObjectControl* (apresentado por *VersObjC1* na Tabela 3.8). Exemplificando essa situação, a Tabela 3.9 ilustra as instâncias das versões e do objeto versionado com os valores para a situação apresentada. Na exclusão, o funcionamento segue as especificações da operação *delete* de *TemporalVersion.*

3.3 Relacionamento de Herança por Extensão

Uma das características mais importantes do Modelo de Versões é a definição de *herança por extensão* na qual as versões são admitidas nos vários níveis da hierarquia de herança. Com esse recurso, um objeto pode ser desenvolvido em um nível de abstração e posteriormente detalhado nos níveis inferiores da hierarquia. Isso possibilita que a modelagem de entidades do mundo real seja feita em vários níveis, projetando ou modificando características de um objeto em uma camada de cada vez.

No início do capítulo anterior (seção 2.1), o conceito de versões é definido através do exemplo de uma aplicação para engenharia de software. Usando esse mesmo exemplo, a Figura 3.21 apresenta duas classes: classe *Design* é super classe da hierarquia (ascendente) e possui *SourceCode* definida como sua subclasse (descendente).

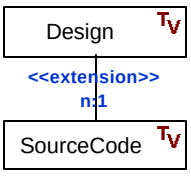


Figura 3.21 – Herança por extensão no projeto de software

Primeiramente os analistas projetam a aplicação, e depois os programadores constróem os códigos fonte das classes, regras de integridade e módulos especificados.

Um código fonte da classe *SourceCode* está diretamente relacionado à sua modelagem conceitual na classe *Design*, não existindo código fonte que não pertença a um projeto.

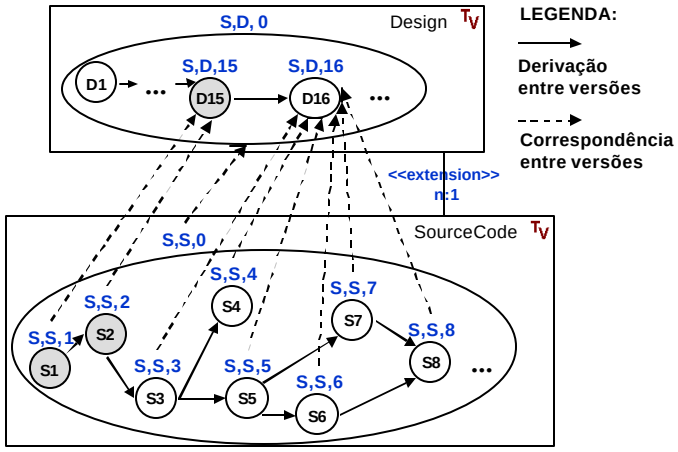


Figura 3.22 – Versões para projeto e implementação de software

Esse código deve ser testado e aprovado, e/ou rescrito. A Figura 3.22 ilustra uma seqüência possível do andamento da construção do software. Após o projeto conceitual ser trabalhado (*D1* a *D15*) pelos analistas, a equipe de programadores começa a escrever o código (*S1*, estado *Working*). O código é compartilhado com a equipe de teste (*S1* promovida para *Stable*) enquanto os programadores continuam trabalhando (*S2*). Algum tempo depois, a equipe de teste retorna aos programadores os erros e as possibilidades de melhoria. Então, os programadores reescrevem (*S2*) o código seguindo o ciclo.

A equipe de testes pode encontrar alguns problemas que devem ser corrigidos diretamente no projeto. Os analistas então derivam uma nova versão para poderem realizar as alterações (*D16*). Após o término das correções e testes, os programadores podem gerar novas versões dos códigos (*S3* em diante), as quais devem ser ligadas à nova versão do projeto.

Caso a equipe de teste sugira algoritmos diferentes para testar o desempenho, os programadores podem trabalhar em duas, ou mais, versões alternativas (*S4* e *S5*) para a equipe escolher a melhor (*S5*). Os programadores também podem dividir o programa em módulos e estabelecer grupos para trabalhar em partes diferentes (*S6* e *S7*). Em algum momento, os programadores devem juntar novamente as partes derivando uma nova versão a partir das mesmas (*S8*).

Resumindo, o relacionamento de herança por extensão é diferente de:

- associação – porque relaciona duas versões, dois objetos versionados ou uma versão e um objeto versionado. A cardinalidade do relacionamento também diz respeito às versões dos objetos e não aos objetos;
- agregação (relacionamento "parte de") – porque as classes descendentes não fazem parte necessariamente da classe ascendente. No exemplo, o projeto conceitual na classe ascendente não envolve a especificação do código, a qual fica na classe descendente;
- composição – porque o objeto pode pertencer a somente um todo na composição [FOW 00], o que não é o caso pois as mesmas classes descendentes podem se relacionar com outras ascendentes;
- herança por refinamento – justificado na seção 3.3.1.

3.3.1 Herança por Extensão x Herança por Refinamento

Para evitar confusões com a herança adotada nas linguagens e modelos orientados a objetos (denominada de *herança por refinamento*), o TVM propõe que o mecanismo de herança por extensão seja modelado como uma especialização do conceito de relacionamento. Esse mecanismo é chamado de *relacionamento de herança por extensão* (ou apenas *herança por extensão*). A hierarquia estabelecida entre as classes participantes desse relacionamento é chamada de *hierarquia por extensão*.

A herança por extensão não pode ser uma especialização da herança por refinamento por serem conceitos distintos e independentes. Enquanto uma subclasse na hierarquia por refinamento herda (recebe) o estado e o comportamento de sua superclasse, na hierarquia por extensão estado e comportamento da superclasse são mantidos na mesma, conforme ilustrado na Figura 3.23. Nesta, a instância de uma subclasse *B* na hierarquia por refinamento possui os seus atributos e aqueles herdados da superclasse *A* (*a* e *b*). A instância de uma subclasse *D* na hierarquia por extensão possui apenas seu próprio atributo (*d*), pois o atributo da superclasse *C* é mantido em suas próprias instâncias (*c*). Devido às suas definições, absolutamente nunca pode haver uma herança por refinamento e uma herança por extensão especificadas para a mesma dupla de classes.

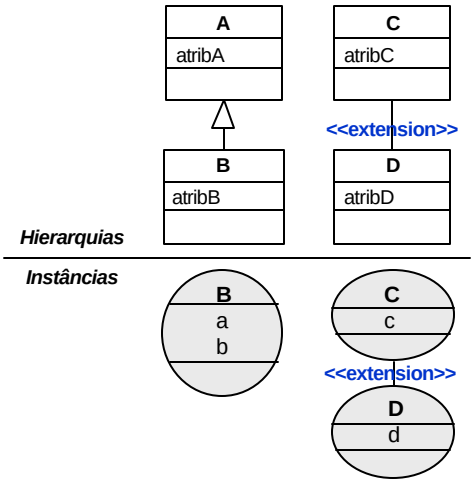


Figura 3.23 – Hierarquia por Refinamento e por Extensão

3.3.2 Correspondência entre os Objetos e Versões

Esta seção refina um pouco mais o conceito de herança por extensão, através de um novo exemplo.

Considerando a especificação apresentada na Figura 3.24 (*Automóvel* e *Caminhão* formam a hierarquia por extensão de *Veículo*), a Figura 3.25 ilustra algumas instâncias dessas classes (objetos versionados e versões) em mais de um nível da hierarquia.

A entidade *FiatUno* é modelada em dois níveis de abstração: veículo e automóvel. Em cada um dos níveis existem versões associadas a objetos versionados. Cada versão deve possuir pelo menos um ascendente que lhe corresponda, pois a criação de uma versão implica obrigatoriamente na ligação a um ascendente (a menos que seja raiz da hierarquia). Uma versão também pode apresentar mais de um ascendente, significando que as mesmas características definidas no seu nível podem ser ligadas a diferentes características no nível superior da hierarquia por extensão.

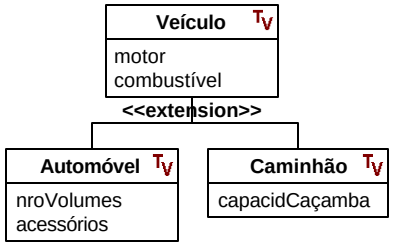


Figura 3.24 – Especificação de *Veículo* e sua hierarquia por extensão

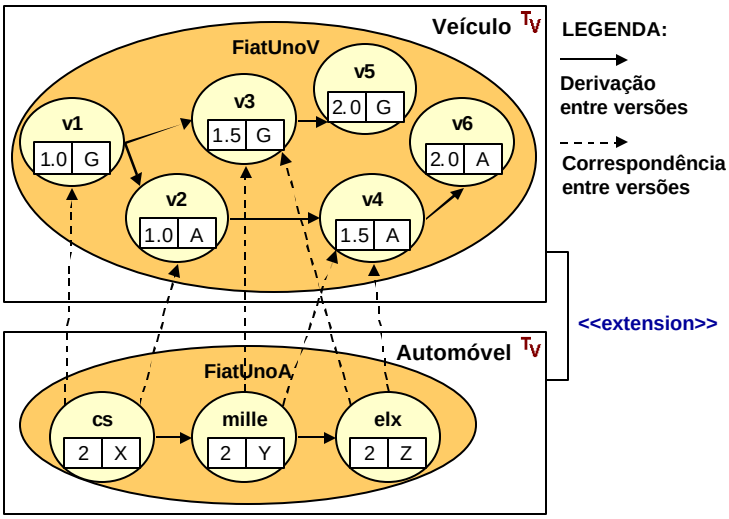


Figura 3.25 – Versões nos níveis da hierarquia por extensão e correspondências

A representação de versões nos diversos níveis de hierarquia permite a existência de múltiplos ascendentes para um objeto (versionado, sem versões ou versão) em uma subclasse, caso o objeto ascendente possua versões. É permitido ao usuário estabelecer *restrições de cardinalidade* (mapeamentos) entre versões de um objeto em uma classe e versões de seu ascendente na superclasse. Essas restrições são chamadas de *correspondências* e são modeladas como a cardinalidade do relacionamento de herança por extensão, podendo ser:

- *1:1* – cada versão na subclasse corresponde a exatamente uma versão na superclasse;
- *1:n* – cada versão na subclasse pode corresponder a várias versões na superclasse, e várias versões na superclasse podem corresponder somente a uma versão na subclasse;
- *n:1* – uma versão na subclasse pode corresponder a somente uma versão na superclasse, mas uma versão na superclasse pode corresponder a várias versões na subclasse;
- *n:m* – várias versões na subclasse podem estar relacionadas com uma versão na superclasse, e cada versão na subclasse pode corresponder a várias versões na superclasse.

É importante observar que as cardinalidades de não obrigatoriedade (*1:0* e *n:0*) não existem nas correspondências, pois sempre um descendente deve estar relacionado com um ascendente. Caso a cardinalidade seja omitida, é assumido *1:1*. Também deve ficar claro que a correspondência é uma cardinalidade estabelecida especificamente entre as versões dos objetos. A noção de cardinalidade de relacionamento de associação,

por exemplo, tem um significado totalmente diferente, uma vez que é estabelecido entre os objetos das classes. Esse segundo tipo de cardinalidade é desnecessário no relacionamento de herança por extensão porque obrigatoriamente o relacionamento acontece entre um objeto (com versões ou não) e seu objeto ascendente (com versões ou não). Em outras palavras, a cardinalidade entre os objetos da classe ascendente e descendente é sempre um para um.

Objetos não versionados e objetos versionados sem versões, são considerados como uma versão, para efeitos de verificação de restrições de cardinalidade imposta pela correspondência.

3.3.3 Representação de Versões em Diversos Níveis da Hierarquia

Com a possibilidade de representar versões de objetos em diversos níveis da hierarquia por extensão, a modelagem de uma entidade do mundo real pode ser feita em vários níveis de abstração. Considerando o exemplo apresentado na Figura 3.24, uma possibilidade é criar uma versão em *Veículo* que é refinada posteriormente, adicionando-se propriedades de automóvel e "migrando" para a subclasse *Automóvel*. A migração dos objetos de uma classe para outra, geralmente, não é permitida. O motivo para este impedimento é que um objeto não pode ter seu "tipo" mais especializado alterado [KIM 89]. Esse tipo é atribuído em tempo de criação do objeto e permanece o mesmo até que ele seja destruído. Além disso, se a versão a ser refinada não for uma folha no grafo de derivação, pode não ser possível removê-la, uma vez que já existem versões derivadas desta.

Outra possibilidade é criar versões diretamente na classe *Automóvel*, com os valores para as propriedades especificadas em *Veículo* (as propriedades especificadas em *Automóvel* ficariam indefinidas, por exemplo, com valores nulos). Nesse caso, para alterar uma versão, completando-a com as propriedades de automóvel, ela não pode possuir versões derivadas (na derivação a versão predecessora passa para o *status Stable*). A Figura 3.26 apresenta uma possível representação da situação modelada na Figura 3.24 com as versões representadas em um único nível.

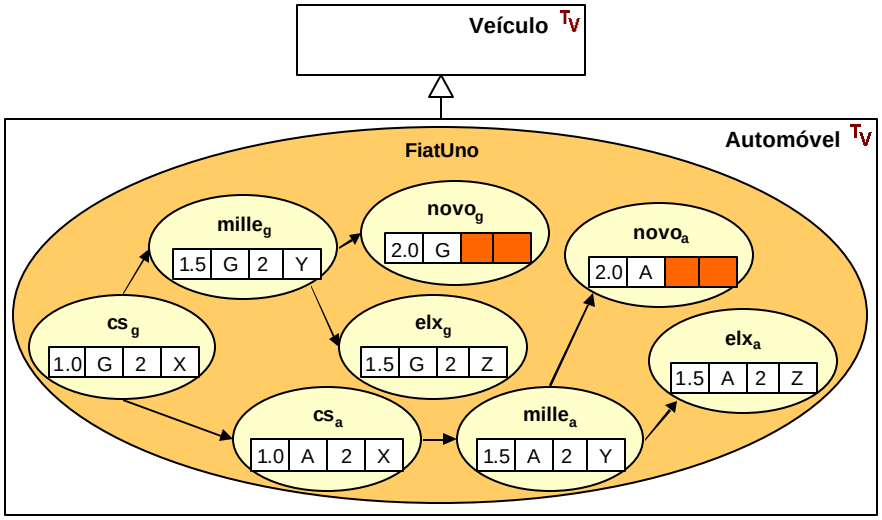


Figura 3.26 – Versões somente na classe mais especializada

Uma desvantagem dessa representação é a visualização. Quando versões são admitidas em um nível de hierarquia, as semelhanças e diferenças entre as versões não são visualizadas com a mesma facilidade como nas características definidas em

diferentes níveis da hierarquia. Para verificar, basta comparar a representação na Figura 3.25 (por níveis) e a da Figura 3.26 (tudo nas folhas).

Outros pontos a considerar são o agrupamento de versões de acordo com os valores armazenados na representação de versões em níveis e a diminuição de instâncias com valores não definidos. Por exemplo, considerando a Figura 3.25, a verificação de todas as versões *FiatUno* no nível de *Automóvel* que apresentam motor 1.5 e combustível gasolina é feita buscando os descendentes da versão *v3* (do nível de *Veículo*). Versões em um nível podem ser consideradas como alternativas para as quais são criadas versões nos níveis inferiores da hierarquia. Os veículos *v5* e *v6* ainda estão sem descendentes, mas já estão instanciadas em seus níveis. Instanciando essas mesmas versões na folha da hierarquia por refinamento, os valores dos atributos da classe descendente ficam com valores em branco ou *null*.

Com tudo isso, as principais vantagens da herança por extensão em relação à herança por refinamento são:

- modelagem em níveis de abstração independentes;
- visualização fácil e direta das diferenças entre os objetos;
- agrupamento das semelhanças das alternativas em versões;
- inibição (ou controle) da proliferação excessiva de versões, pois são instanciadas somente as versões realmente necessárias e nunca com valores nulos.

3.3.4 Operações sobre a Hierarquia por Extensão

As operações definidas para objetos e versões (também aplicadas a configurações) na hierarquia por extensão podem ser de criação ou navegação.

Criação

A criação de versões pode ocorrer em quatro situações:

- criação de um objeto versionado e de suas versões;
- derivação de uma versão a partir de outra existente;
- derivação de um objeto não versionado (o objeto sem versões tornar-se a primeira versão de um novo objeto versionado);
- criação de uma versão configurada (*getConfiguration*).

As operações definidas para criação de versões que envolvem a herança por extensão são:

- *TemporalVersion* – se não for raiz da hierarquia por extensão, quando uma versão é criada, ela deve ser ligada a uma (ou mais) versão ascendente. Se a versão ascendente não for informada, a nova versão será ligada à versão corrente do(s) objeto(s) ascendente(s). A versão na superclasse pode existir sem estar relacionada ainda com uma versão na subclasse;
- *derive* – pode receber o(s) ascendente(s) por parâmetro ou o ascendente é a versão corrente do objeto na superclasse;
- *getConfiguration* – estabelece uma versão configurada e está detalhado na seção sobre configuração (seção 3.4).

Além dessas, em certas operações é necessário adicionar ou remover um ascendente ou descendente do conjunto. Para tal, são definidas as seguintes operações em *TemporalVersion*:

- *addAscendant* – acrescenta um ascendente na lista do objeto ou da versão;
- *addDescendant* – acrescenta um descendente na lista do objeto ou da versão;
- *removeAscendant* – remove um dos ascendentes do objeto ou da versão;
- *removeDescendant* – remove um dos descendentes do objeto ou da versão.

Todos os métodos de gerenciamento de ascendentes e descendentes devem obedecer à correspondência estabelecida entre a classe e sua superclasse. O método *getCorrespondence* consulta os metadados do esquema e obtém a correspondência requisitada.

Navegação

As operações para navegação na hierarquia de herança permitem a recuperação de ascendentes e descendentes de um objeto. A classe *Object* possui as operações *getAscendant* e *getDescendant* para recuperar respectivamente o conjunto de ascendentes e descendentes, que são estabelecidos pela identificação da entidade no tvOID. Essas operações podem receber o nome da superclasse ou subclasse como parâmetro. Nesse caso, só retornam um tvOID porque, por definição, só pode haver um objeto por classe. Importante notar que essa classe não tem as operações *setAscendant* e *setDescendant* justamente por serem estabelecidos no tvOID e somente pelo construtor.

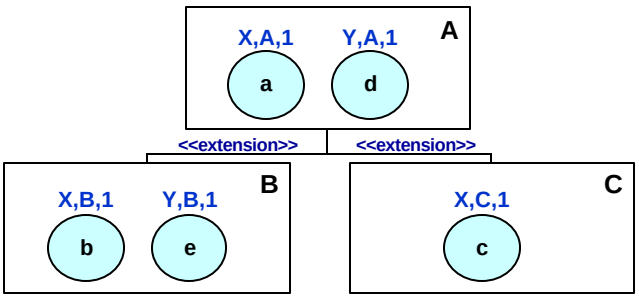


Figura 3.27 – Ascendentes e descendentes em classes sem tempo e versões

Por exemplo, a Figura 3.27 apresenta três classes sem tempo e versão (*A*, *B* e *C*), suas instâncias (*a* e *d*; *b* e *e*; *c*) associadas aos seus respectivos tvOIDs. Os resultados das operações *getAscendant* e *getDescendant* sobre esses objetos são:

- *a.getDescendant* retorna “X,B,1” e “X,C,1”;
- *a.getDescendant (B)* retorna “X,B,1”;
- *d.getDescendant* retorna “Y,B,1”;
- *b.getAscendant* retorna “X,A,1”;
- *e.getAscendant* retorna “Y,A,1”;
- *c.getAscendant* retorna “X,A,1”.

Esses dois métodos são especializados em *TemporalVersion* porque os ascendentes e descendentes são armazenados como atributos, uma vez que pode ter mais de uma versão ascendente/descendente em um objeto versionado. Na recuperação,

o ascendente pode ser especificado pelo seu tvOID, ou através de um dos critérios pré-definidos: *recent* (mais recente), *first* (primeiro) ou *current* (corrente). O critério é usado sempre que houver mais de uma versão ascendente associada à versão (ou objeto) desejada(o).

Os atributos *ascendant* e *descendant* também são definidos como temporais e continuam com as mesmas funções, ou seja, no momento da criação de um objeto se o seu ascendente não foi informado, o sistema gera um novo número para ser o identificador da entidade. Os construtores de *TemporalVersion* podem receber o conjunto de ascendentes como parâmetro.

A operação *getObject* permite buscar os valores de atributos de um objeto definidos em uma classe. Essa operação só retorna os atributos definidos em um nível da hierarquia de herança. Para obter todos os atributos de uma entidade do mundo real modelada, devem ser buscados todos os objetos que a representam nos vários níveis da hierarquia. A operação *getCompleteObject* retorna os ascendentes de um objeto, um para cada superclasse, e um objeto por classe agregada, se for o caso.

3.3.5 Atribuição de tvOID

O identificador da classe pode ser obtido através de uma consulta aos metadados. O número da versão sempre é *null* para objetos de classe sem tempo e versões, *1* para a primeira instância de um objeto temporal versionado, e *0* para o objeto versionado. Para as demais versões, o número deve ser obtido através do atributo *nextVersionNumber* em *VersionedObjectControl*. O identificador da entidade é gerado automaticamente, também através dos metadados, ou pode ser obtido pelo tvOID do objeto ascendente ou predecessor (se houver).

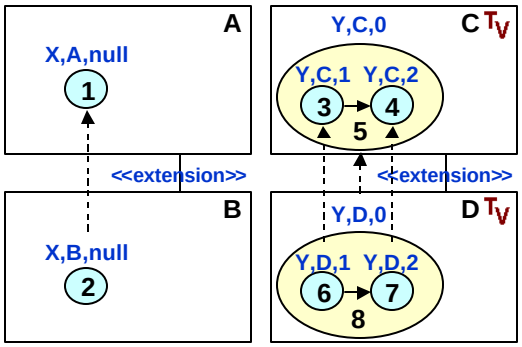


Figura 3.28 – Definição do tvOID

A Figura 3.28 apresenta as possíveis combinações para objetos e seus ascendentes. Os objetos *1* e *2* representam a entidade *X* nas classes *A* e *B*, respectivamente. As versões *3*, *4* e *5* representam a entidade *Y* na classe *C*, e as versões *6*, *7* e *8* representam essa mesma entidade na classe *D*. A Tabela 3.10 apresenta como são gerados os tvOIDs para esses objetos e versões (entre parênteses são os números das versões da figura).

Tabela 3.10 – Atribuição de tvOID ao objeto

Situação (versão)	Como gera o tvOID
Classe Normal	Verifica se é uma subclasse na hierarquia por extensão:
(1) *Classe raiz* Sem ascendente	Consulta metadados e obtém o número da próxima entidade; obtém o identificador da classe que está chamando; o número da versão é *null*
(2) *Subclasse* Com ascendente	Verifica se o tvOID do(s) ascendente(s) recebido(s) como parâmetro é válido (*verifyAscendId*); obtém identificador da entidade pelo tvOID ascendente; obtém o identificador da classe que está chamando; o número da versão é *null*
Classe Temporal Versionada	Verifica se é uma subclasse na hierarquia por extensão:
Classe raiz Sem ascendente	Se recebe a entidade por parâmetro, deve conferir nos metadados se esse é um valor válido (*verifyEntityId*)
(3) Sem predecessora	Consulta metadados e obtém o número da próxima entidade, se não recebeu entidade por parâmetro; obtém o identificador da classe que está chamando; o número da versão é obtido através da consulta ao controle do objeto versionado (*VersionedObjectControl nextVersionNumber*) ou é 1, caso não haja controle relacionado
(4) Com predecessora	Obtém o identificador da entidade através do tvOID da predecessora; obtém o identificador da classe que está chamando; o número da versão é obtido através da consulta ao controle do objeto versionado (*VersionedObjectControl nextVersionNumber*)
(5) Objeto versionado	Obtém o identificador da entidade através do tvOID da predecessora da versão; obtém o identificador da classe; o número da versão é 0
Subclasse Com ascendente	Verifica se o tvOID do ascendente (ou dos ascendentes, caso haja mais de um) é válido (*verifyAscendId*), ou a identificação da entidade. Se não recebe o ascendente por parâmetro nem a identificação da entidade, operação falha.
(6) Sem predecessora	Se recebe a identificação da entidade, essa é assumida, e a correspondência referencia a versão corrente da classe (referência dinâmica), ou obtém identificador da entidade pelo tvOID ascendente; obtém o identificador da classe que está chamando; o número da versão é obtido através da consulta ao controle do objeto versionado (*VersionedObjectControl nextVersionNumber*) ou é 1, caso não haja controle relacionado.
(7) Com predecessora	Se recebe a identificação da entidade, essa é assumida (referência dinâmica), ou obtém identificador da entidade pelo tvOID ascendente, ou obtém identificador da entidade pelo tvOID predecessor (referência dinâmica); obtém o identificador da classe que está chamando; o número da versão é obtido através da consulta ao controle do objeto versionado (*VersionedObjectControl nextVersionNumber*).
(8) Objeto versionado	Obtém o identificador da entidade através do tvOID da predecessora da versão que está derivando, e a correspondência referencia a versão corrente do ascendente (referência dinâmica); obtém o identificador da classe que está chamando; o número da versão é 0

3.4 Configuração

O sistema de configurações continua basicamente o mesmo estabelecido no Modelo de Versões. A única diferença é que, assim como as versões e os objetos versionados, a configuração possui o comportamento temporal estabelecido. Para evitar dúvidas, esta seção apresenta resumidamente o conceito de configuração e suas propriedades.

A construção de uma configuração é solicitada para uma versão (chamada *versão base*) através da operação específica *getConfiguration*. Deve ser definida uma versão para cada ascendente na hierarquia por extensão e uma versão para cada componente das classes agregadas. Caso alguma classe agregada seja especificada não temporal-versionada, o método retorna o tvOID correspondente à entidade requisitada. Podem ser necessárias várias escolhas para uma versão que apresenta uma ou mais referências dinâmicas e/ou múltiplos ascendentes em mais de uma superclasse. O processo de configuração é recursivo e consiste de dois passos para cada versão: (i) resolução de cada referência dinâmica em agregações, escolhendo uma das versões do objeto versionado referido; (ii) escolha da versão ascendente para cada uma das superclasses.

Por exemplo, considerando o cenário da Figura 3.29, a construção de uma configuração para a versão *b1* consiste dos seguintes passos:

1. como *b1* possui duas versões ascendentes correspondentes na superclasse *A*, uma delas deve ser escolhida (através de critérios pré-definidos ou expressões), nesse caso é *a1*;
2. *a1* possui uma agregação com o objeto versionado *Q* (referência dinâmica) que deve ser resolvida, sendo escolhida *q2*;
3. *q2* possui também duas versões ascendentes correspondentes na superclasse *P*, das quais *p2* é escolhida.

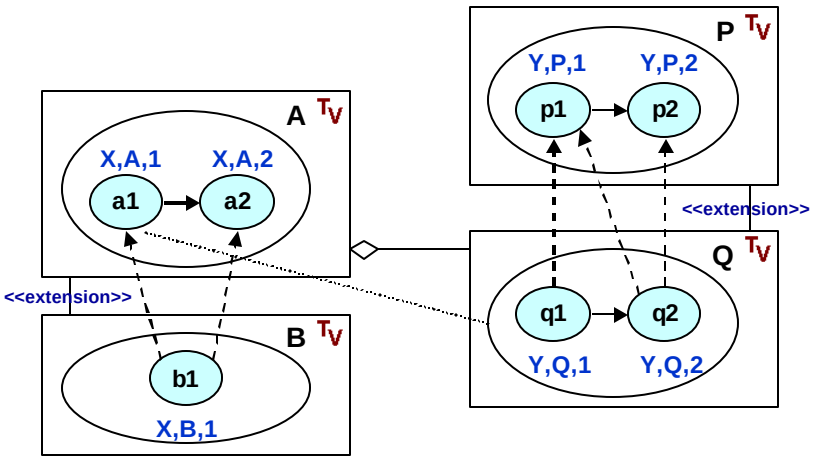


Figura 3.29 – Cenário para configuração

Cada escolha define uma "parte" da configuração. Quando a configuração está completamente especificada, é gerada uma versão configurada, como sucessora de sua versão base. A Figura 3.30 ilustra as versões configuradas geradas como resultado dos passos mencionados. Como a versão *a1* foi escolhida no passo 1, foi gerada a versão *c2* (sucessora de *a1*) que é referenciada a partir de *c1*. As versões *c3* e *c4* foram geradas como resultado das escolhas feitas nos passos 2 e 3, respectivamente.

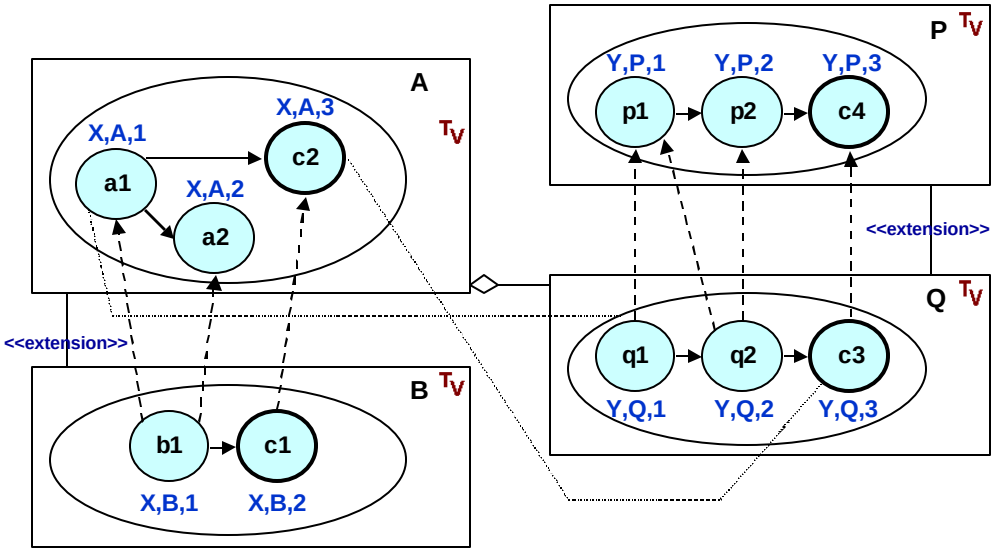


Figura 3.30 – Versões configuradas e seus relacionamentos

Um versão configurada só pode fazer referência a versões configuradas, que podem ser compartilhadas por outras versões. Essa restrição garante que versões configuradas sejam sempre completamente definidas. As versões *c2*, *c3* e *c4* são configurações parciais, podendo ser referenciadas por outras versões. A vantagem da definição de configurações parciais é que, muitas vezes, um componente (ou ascendente) é composto de muitos outros objetos e a utilização de uma configuração parcial permite aproveitar escolhas feitas previamente. Uma versão configurada é sempre criada a partir de uma versão existente, permanecendo ligada a essa como sucessora. Além das características já especificadas, uma configuração possui o atributo *alive*, ou seja, tem seu tempo de vida armazenado.

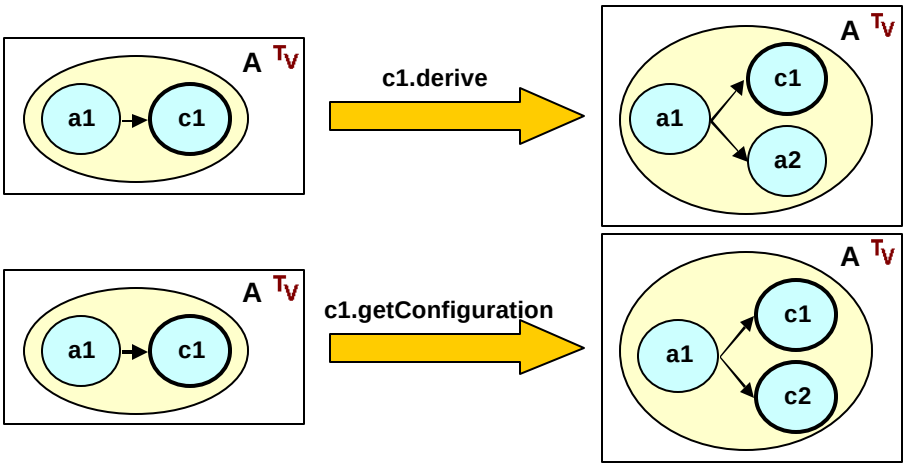


Figura 3.31 – Operações *derive* e *getconfiguration* em uma configuração

Das operações chave definidas (*derive*, *getObject*, *getCompleteObject*, *delete*, *getConfiguration*) apenas *derive* e *getConfiguration* possuem um significado diferente quando aplicadas a versões configuradas. O resultado de *derive* é uma cópia da versão configurada, conectada como sucessora da mesma versão base da versão copiada. O resultado de *getConfiguration* é outra versão configurada cópia da original. A Figura 3.31 ilustra essas duas situações.

3.5 Relações entre Classes “Normais” e Temporais Versionáveis

As especificações do TVM podem gerar uma certa confusão quanto aos tipos de relacionamentos que podem ser especificados entre uma classe sem tempo e versões e uma classe temporal versionada. Esta seção torna explícitas as situações nas quais os relacionamentos podem ocorrer ou não.

3.5.1 Associação

A associação entre classes normais e temporal versionadas é permitida desde que o relacionamento não seja temporal, como ilustra a Figura 3.32. Uma regra de integridade deve ser estabelecida para que as classes normais referenciem apenas objetos sem versões ou configurações (versões resolvidas).

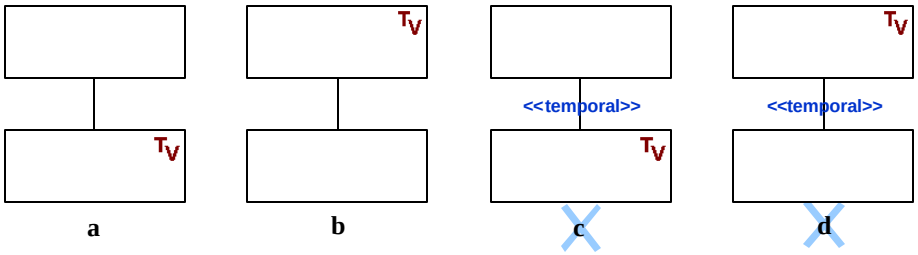


Figura 3.32 – Classe normal e temporal versionada: associação

3.5.2 Herança por Refinamento

Como o TVM é um modelo orientado a objetos, a herança por refinamento pode ser definida sem problemas entre duas classes normais (Figura 3.33a). Afim de manter a fidelidade às características principais do Modelo de Versões, entre duas classes temporais versionadas a herança obrigatoriamente deve ser por extensão (Figura 3.33b), assim como uma classe temporal versionada não pode ter uma superclasse normal (Figura 3.33c). Uma classe normal também não pode ter uma superclasse temporal versionada, pois isso implica diretamente no seu comportamento, devendo a classe se tornar temporal versionada (Figura 3.33d).

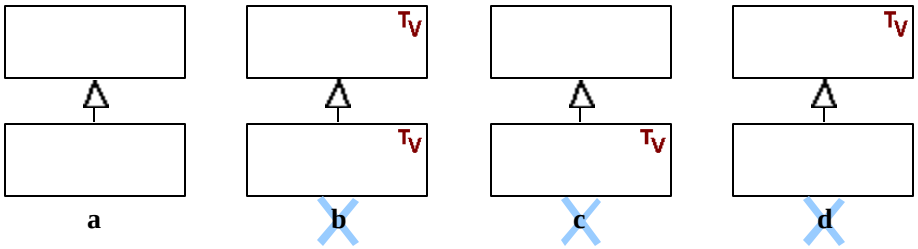


Figura 3.33 – Classe normal e temporal versionada: herança por refinamento

3.5.3 Herança por Extensão

Uma classe temporal versionada pode ter um ascendente em uma classe normal, sendo que a correspondência só pode ser *1:1* ou *n:1*, pois qualquer versão está associada a um único objeto ascendente (Figura 3.34a). Uma classe normal não pode ter um ascendente em uma classe temporal versionada porque seu método *getCompleteObject* busca um ascendente por classe com base na informação de entidade no tvOID (Figura 3.34b). Como um objeto versionado pode ter várias versões, obviamente existirão várias versões com o mesmo número de entidade. Em classes temporais versionadas essa

questão é resolvida com o método *getConfiguration*, o qual resolve as referências dinâmicas.

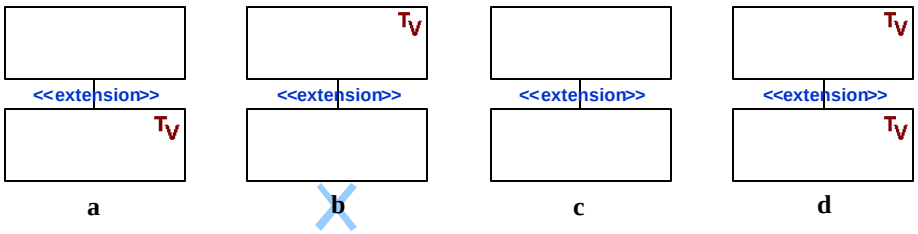


Figura 3.34 – Classe normal e temporal versionada: herança por extensão

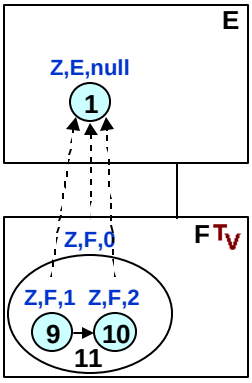


Figura 3.35 – Atribuição de tvOID

Para o caso de uma classe temporal versionada ter com o ascendente uma classe normal, a atribuição do tvOID (Figura 3.35) é realizada da seguinte maneira: verifica se o tvOID do ascendente (ou dos ascendentes caso haja mais de um) é válido (*verifyAscendId*); obtém identificador da entidade pelo tvOID ascendente; obtém o identificador da classe que está chamando; o número da versão é obtido através da consulta ao controle do objeto versionado (*VersionedObjectControl. nextVersionNumber*) ou é 1, caso não haja controle relacionado. Se o ascendente não é recebido por parâmetro (a menos que a superclasse tenha só um objeto instanciado) não é possível saber qual objeto representa a entidade, sendo retornado um erro.

3.5.4 Agregação

Uma classe normal não pode ter uma (ou mais) classe temporal versionada como agregada pois um objeto agregado pode ter várias versões e seria necessária a operação de configuração para resolver as referências dinâmicas (Figura 3.36a). Por outro lado, uma classe temporal versionada pode ter como componente uma (ou mais) classe normal, pois a operação de configuração obtém o único objeto definido para a entidade na classe (Figura 3.36b).

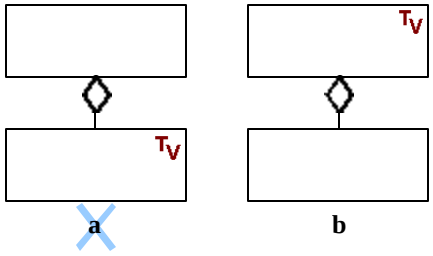


Figura 3.36 – Classe normal e temporal versionada: agregação

3.6 Linguagens

3.6.1 Linguagem de Definição de Classes

A linguagem para definição das classes do esquema foi estendida do Modelo de Versões, em Saggiorato [SAG 99], incluindo as cláusulas:

- **`hasVersions`** – indica se uma classe é versionável ou não;
- **`aggregateOf`** – define relacionamento de agregação;
- **`relationship`** – define relacionamento de associação.

No atual trabalho foi necessário estender novamente essa linguagem. Incluiu-se a cláusula **`temporal`** para indicar que o atributo ou relacionamento terá sua evolução armazenada, e o nome da relação inversa (**`inverse`**) para poder estabelecer as duas cardinalidades na definição do relacionamento.

Outro ponto a especificar na linguagem é a visibilidade, um assunto que é simples em princípio, mas possui sutilezas complexas. A idéia básica é que qualquer classe tem elementos públicos e particulares. Os elementos públicos podem ser usados por qualquer outra classe, e os particulares somente pela classe proprietária. Embora muitas linguagens usem termos como *public* (público), *private* (particular) e *protected* (protegido), eles têm significados diferentes em linguagens diferentes. Essas diferenças são pequenas, mas causam confusão principalmente para os que utilizam mais de uma linguagem [FOW 00].

Tentando estabelecer um padrão de visibilidade para o TVM, decidiu-se ter como base as notações da UML (*Unified Modeling Language*), pois ela aborda o tema sem entrar em detalhes. Essencialmente, na UML o usuário pode rotular qualquer atributo ou operação com um indicador de visibilidade, sendo adotadas três abreviações: + para público, – particular e # protegido. Porém, o significado é dependente da linguagem. A abstenção de visibilidade indica que a mesma é apenas não apresentada (não que seja indefinida ou pública). Uma ferramenta deveria atribuir visibilidade a novos atributos e operações mesmo se a visibilidade não fosse apresentada [OMG 00]. Desse modo, a linguagem de definição de classes do TVM adota os seguintes modificadores:

- de classes:
  - **`public`** – qualquer outra classe da aplicação pode acessá-la;
  - **`abstract`** – define uma classe abstrata, ou seja, uma classe que não pode ser diretamente instanciada;
  - **`final`** – a classe não pode ser especializada, ou seja, não podem ser definidas subclasses a partir dela. Obviamente não faz sentido que uma classe seja abstrata e final ao mesmo tempo, porque no primeiro caso é necessário uma subclasse para definir instâncias;
- de atributos:
  - **`public`** – visibilidade pública;
  - **`protected`** – visibilidade protegida;
  - **`private`** – visibilidade particular;

- **`static`** – atributo com escopo de classe;

- de operações:
  - **`public`** – visibilidade pública;
  - **`protected`** – visibilidade protegida;
  - **`private`** – visibilidade particular;
  - **`static`** – método com escopo de classe e só pode acessar atributos estáticos;
  - **`abstract`** – define somente a assinatura, sendo implementado nas subclasses;
  - **`final`** – método que não pode ser modificado pelas subclasses.

```
classDef ::= [ public ][ abstract | final ]
 class className [ hasVersions ] [ inherit
     [ byExtension] className [ correspondence (1:1 | 1:n | n:1 | n:m)] ]
     [[temporal] aggregateOf [n] className ( byValue | byReference )
     {, [temporal] aggregateOf [n] className ( byValue | byReference )} ]
      ( [ Properties:
            { [ public | private | protected ] [ static ]
                  [temporal] attributeName : attributeDomain ; }+
        [ Relationships:
            { [temporal] relationshipName (0:1 | 0:n | 1:1 | 1:n | n:m)
              [ inverse inverseRelationshipName ] relatedClassName ; }+
        [ Operations:
            { [ public | private | protected ] [ static ]
                  [ abstract | final ] operationDefinitions }+ ] ) ;
```

Figura 3.37 – Sintaxe simplificada da Linguagem de Definição para classe

```
classDef ::= [ public ][ abstract | final ] (normalClass | tempVersionClass)
 normalClass ::= class className
    [ inherit ( byExtension className | superClassName ) ] classDefinition ;
 tempVersionClass ::= class tempClassName hasVersions [ inherit
    ( tempClassName [correspondence (1:1 | 1:n | n:1 | n:m)]
    | className [correspondence (1:1 | n:1)]
    )] tempClassDefinition;
```

Figura 3.38 – BNF da definição dos tipos de classe

A Figura 3.37 apresenta a sintaxe geral e simplificada da linguagem para a definição de uma classe. A BNF completa da linguagem pode ser conferida no Anexo 1. A linguagem de definição propõe os dois tipos de classes já mencionados, como apresentado no início da BNF (Figura 3.38).

As classes não temporais e não versionáveis são representadas na BNF por `normalClass`. As classes temporais versionadas são denominadas como `tempVersionClass`. A cláusula **`correspondence`** permite estabelecer o tipo de cardinalidade entre a classe descendente e sua ascendente. A cardinalidade entre classes normais é omitida (*1:1*) pois em cada classe só pode haver um objeto ascendente ou descendente. As demais cardinalidades só podem ser utilizadas em subclasses da raiz temporal versionada (subclasse de *TemporalVersion*), sendo que se o usuário não especifica um valor, é assumido que a cardinalidade é um para um.

3.6.2 Linguagem de Consulta

A linguagem de consulta do Modelo Temporal de Versões deve permitir, além das consultas básicas realizadas pela linguagem padrão SQL, novas consultas que retornem valores específicos das características de tempo e versões. É desejável que a linguagem de consulta estabeleça um comportamento o mais homogêneo possível para elementos normais e temporais versionados.

Entre as características de tempo estão consultas sobre:

- valores atuais de atributos e relacionamentos;
- o histórico da vida de um objeto, ou seja, os tempos nos quais um objeto está "vivo";
- o histórico de atributos e relacionamentos dos objetos;
- os valores de atributos e relacionamentos em determinados instantes ou períodos de tempo de transação e/ou validade.

Como facilidades relativas a tempo, a cláusula de busca (`WHERE`) deve oferecer:

- comparativos entre instantes de tempo (igual, antes, depois);
- comparativos entre intervalos de tempo (igual, intersecção, sobreposição, ...);
- comparativos entre instantes e intervalos de tempo (antes, dentro, fora).

Entre as características de versionamento estão consultas sobre:

- os estados das versões;
- uma única versão ou o conjunto de versões de um objeto versionado;
- a navegação na hierarquia de ascendentes e descendentes;
- a navegação na hierarquia de derivação (predecessores, sucessores, primeiro, último, ...);
- versões correntes;
- configurações.

Juntando características de tempo e versão, podem ser definidas consultas sobre:

- os estados das versões em determinados instantes ou períodos;
- versões separadas ou o conjunto das versões em determinados instantes ou períodos;
- os ascendentes e descendentes em determinados instantes ou períodos;
- a hierarquia de derivação em determinados instantes ou períodos;
- versões correntes em determinados instantes ou períodos;
- configurações em determinados instantes ou períodos.

A descrição completa da linguagem de consulta básica pode ser encontrada na referência [GEL 01].

3.7 Comparativo com Outros Modelos

Diferentes áreas de aplicação que necessitam suporte a desenvolvimento evolutivo, tais como ferramentas CAD e CASE, motivaram as pesquisas relacionadas a versões. Mais recentemente, esse conceito tem sido espalhado para outras áreas, tais como sistemas de banco de dados e hipermídia. Uma excelente pesquisa sobre modelos de versões para gerenciamento de configuração de software foi realizada por Conradi em [CON 98]. Além disso, Hicks (et. al) apresenta uma lista classificando diferentes aplicações com versionamento e resumos específicos com versões na área de hipermídia [HIC 98].

O conceito de tempo pode estar presente não somente em modelagem de banco de dados temporais mas também em consultas, regras de integridade, aplicações de tempo real, banco de dados ativos e dedutivos, sincronização temporal de dados multimídia, entre outros. Sobre implementação e modelagem de banco de dados temporais, um bom conjunto de trabalhos é agrupado por Tansel (et. al) em [TAN 93]. Jensen também agrupou seu vasto conjunto de trabalhos, publicados e não publicados, sobre dados, consultas, implementação e projeto temporal em [JEN 99]. Aspectos de implementação são analisados detalhadamente por Snodgrass em [SNO 00].

Esses dois conceitos, versão e tempo, são tratados individualmente em grande parte da literatura, que é vasta com modelos para versões [AGR 91, BEE 88, BJÖ 89, CHO 86, GOL 95, KIM 89, ROD 99, WUU 93] e dados temporais [ANT 97, CLI 87, EDE 94, ELM 93, GAD 88, KAF 92, KAK 96, LOR 88, LOU 91, NAV 89, ROS 91, SAR 90, SNO 95, SU 91, TAU 91]. A novidade é colocá-los juntos, com tratamento igual e simultâneo.

Essa idéia apareceu anteriormente em uma extensão proposta por Wuu e Dayal para o modelo OODAPLEX, como um modelo uniforme para banco de dados orientado a objetos temporal e versionado [WUU 93]. Eles contam com o sistema rico de herança de tipos do OODAPLEX para modelar a informação temporal. Pontos de tempo são tratados como objetos abstratos, e uma hierarquia de tipos de tempo é definida para suportar várias noções de tempo (incluindo versões). Muitas funções temporais e regras de integridades são definidas para introduzir semântica adicional relacionada ao tempo no sistema. O retorno e a manipulação de informação temporal e não-temporal é expressa uniformemente. Entretanto, o conceito de versão não é definido explicitamente.

Uma definição formal de um modelo de dados orientado a objetos temporal versionado, chamado TVOO, é proposto por Rodrigues, Ogata e Yano [ROD 99]. Eles focaram na construção de um esqueleto (*framework*) formal que pode ser ajustado às necessidades de um dado sistema que varia com o tempo. Todos objetos e relacionamentos entre eles são temporais, e o histórico das mudanças é mantida em hierarquias de versões. Cada mudança feita em um objeto resulta em uma nova versão desse objeto. Um histórico da evolução é mantido para cada classe como um conjunto de versões, no qual cada instância de objeto tem sua correspondente hierarquia de versões. Versões comportam-se independentemente uma das outras, têm um tempo de vida, e são tipadas como uma árvore parcialmente ordenada.

As principais diferenças e semelhanças entre o TVM e esses dois modelos são:

- estrutura base – o TVM propõe uma hierarquia como base para a especificação de classes do usuário, enquanto o TVOO usa um *framework*

formal, e a extensão do OODAPLEX trabalha com uma hierarquia de tipos de tempo abstratos;

- objetos – o TVM e a extensão do OODAPLEX permitem objetos temporais versionados entre os objetos normais, enquanto o TVOO especifica que todos os objetos são temporais versionados;

- versionamento – cada mudança gera uma nova versão no TVOO, enquanto no TVM é o usuário quem cria explicitamente novas versões;

- dimensões temporais – o TVM e a extensão do OODAPLEX possuem os tempos de transação e validade, enquanto o TVOO permite somente o tempo de transação;

- hierarquia de derivação – os três modelos apresentam conceitos para gerenciar a ordem de derivação. Mas, enquanto essa hierarquia é representada por uma estrutura de árvore no TVOO, é representada por um grafo acíclico dirigido no TVM e na extensão do OODAPLEX;

- exclusão – os três modelos têm exclusão lógica. Objetos excluídos não podem ser recriados no TVOO.

A principal vantagem do TVOO sobre o OODAPLEX é que as características de versionamento estão completamente especificadas, e sobre o TVM é a especificação formal descrita para cada conceito. O TVM não é especificado formalmente porém é mais rico que o TVOO. Um exemplo é o fato de ser um modelo de dados bitemporal no qual o usuário especifica quais classes são temporais versionadas. Outros diferenciadores importantes do TVM são a herança por extensão e a ordem de tempo ramificada.

Além disso, o TVOO define o conceito de versão de uma forma um pouco diferente do conceito usualmente encontrado. Nele, uma versão não é equivalente a uma alternativa, pois um objeto só pode ter uma versão ativa em cada instante. Já o TVM (assim como os demais modelos de versões citados) especifica versão como uma das alternativas de projeto que podem coexistir.

3.8 Considerações Finais

Este capítulo apresentou o Modelo Temporal de Versões como alternativa para a união dos conceitos de tempo a um modelo de versões. No TVM é realizada uma releitura do Modelo de Versões adequando vários conceitos aos padrões hoje utilizados. Duas características ímpares são encontradas no TVM: a herança por extensão e a ordem temporal ramificada.

A linguagem de definição de classes e os aspectos a serem considerados na definição da linguagem de consulta também foram abordados. Uma das grandes vantagens da linguagem de definição de classes é que as restrições quanto aos tipos de relacionamentos que podem existir entre classes normais e classes temporais versionadas estão definidas explicitamente na sua BNF. Sob certo ponto de vista, pode-se afirmar que a BNF determina essas restrições semânticas além das léxicas e sintáticas. Como limitações, a linguagem de definição não permite a construção de tipos literais nem a definição de tipos literais temporais. Além disso, o único tipo de coleção definido é o *set* – a linguagem não estabelece outros tipos como *array*, *bag*, *list* e *dictionary*.

A linguagem de manipulação de dados não foi definida. Essa linguagem deve permitir que sejam gerenciadas todas as características presentes na linguagem de definição de classes. A sugestão mais simples é acrescentar as mesmas palavras reservadas, com os respectivos significados, a uma linguagem base como a SQL92 ou SQL3, por exemplo.

Esse capítulo também não aborda a relação dos rótulos de tempo com as propriedades ACID (*Atomicidade*, *Consistência*, *Isolamento* e *Durabilidade*). Essas propriedades são os critérios de corretude para transações instantâneas rodando em um nível de isolamento serializável. Assumindo que o SGBD apresente mecanismos necessários para fornecer as propriedades ACID para transações instantâneas, ele irá automaticamente mantê-las também para transações temporais. Entretanto, Torp, Jensen e Snodgrass apresentam um estudo de controvérsias que podem ocorrer quando o tempo de transação é manipulado [JEN 99]. Com base em vários problemas encontrados, é enumerado um conjunto de requisitos para uma semântica temporal de transações consistente.

Alguns desses requisitos são cobertos pelas regras já estabelecidas, porém outros ainda devem ser incorporados, como por exemplo, transações temporais devem conseguir visualizar suas próprias alterações imediatamente. Além desses, existem outros em relação ao tempo de bloqueio (*lock*) e execução (*commit*) para transações concorrentes. A conclusão principal é que os rótulos temporais devem ser gerenciados cuidadosamente para garantir que as propriedades ACID, mesmo estando sob controle do SGBD, sejam preservadas. Devido à sua complexidade, essa tarefa é deixada para uma extensão futura do Modelo.

4 Ambiente Temporal de Versões

Estabelecer uma arquitetura de SGBD integrada para implementar um modelo de dados que estende o SQL com suporte a tempo é uma tarefa de alto custo, que somente os maiores fabricantes de banco de dados podem financiar [JEN 99]. Então, o fato de já existir um SGBD que gerencie grandes quantidades de dados temporais sugere uma alternativa melhor: prover suporte para aplicações de tempo e versões embutido por meio de uma camada entre um SGBD existente e a aplicação (Figura 4.1).

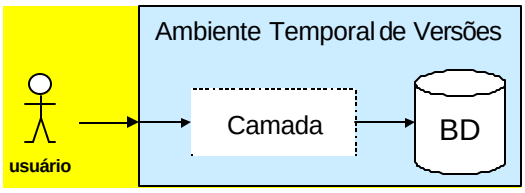


Figura 4.1 – Ambiente Temporal de Versões

Seguindo essa idéia, este capítulo apresenta as camadas da arquitetura e as funcionalidades do Ambiente Temporal de Versões que implementa o TVM sobre um banco de dados convencional. O mapeamento das classes da hierarquia do TVM é proposto com maiores detalhes. Por fim, é apresentada uma ferramenta de apoio à especificação de classes.

4.1 Visão Geral

A camada intermediária explora os serviços já fornecidos pelo SGBD para o suporte a aplicações com tempo. Através dela, é possível maximizar o reuso de tecnologias existentes e apresentar um SGBD com todas as características necessárias. Na primeira subseção é proposta uma arquitetura com reutilização dos serviços do SGBD existente, que por si só é considerado uma "caixa preta". Na segunda são apresentadas as funcionalidades com mais detalhes.

4.1.1 Arquitetura

Detalhando essa idéia, a Figura 4.2 apresenta como é a interação da interface do Ambiente com a camada e com o banco de dados suporte. Primeiramente, a hierarquia de classes base do TVM e sua estrutura de metadados devem estar previamente criadas na base suporte. O usuário pode requisitar as funcionalidades do Ambiente através das Ferramentas e *Wizards* bem como através de arquivos próprios de entrada. A informação do usuário passa por um *Scanner* e um *Parser*, os quais podem detectar erros, que são retornados ao usuário pela interface. O *Parser*, se necessário, recebe dados armazenados na base através do Gerenciador de Metadados. As informações do *Parser* passam para o Gerador de Código que juntamente com as Regras de Integridade (de tempo e versões) compõe um comando ou uma consulta SQL a ser executada na base suporte. O Processador de Saída gerencia a apresentação dos dados para o usuário (texto, tabela, imagem ou outro tipo de mídia). Do ponto de vista do usuário, a camada encapsula o SGBD que realiza as funções nos dados.

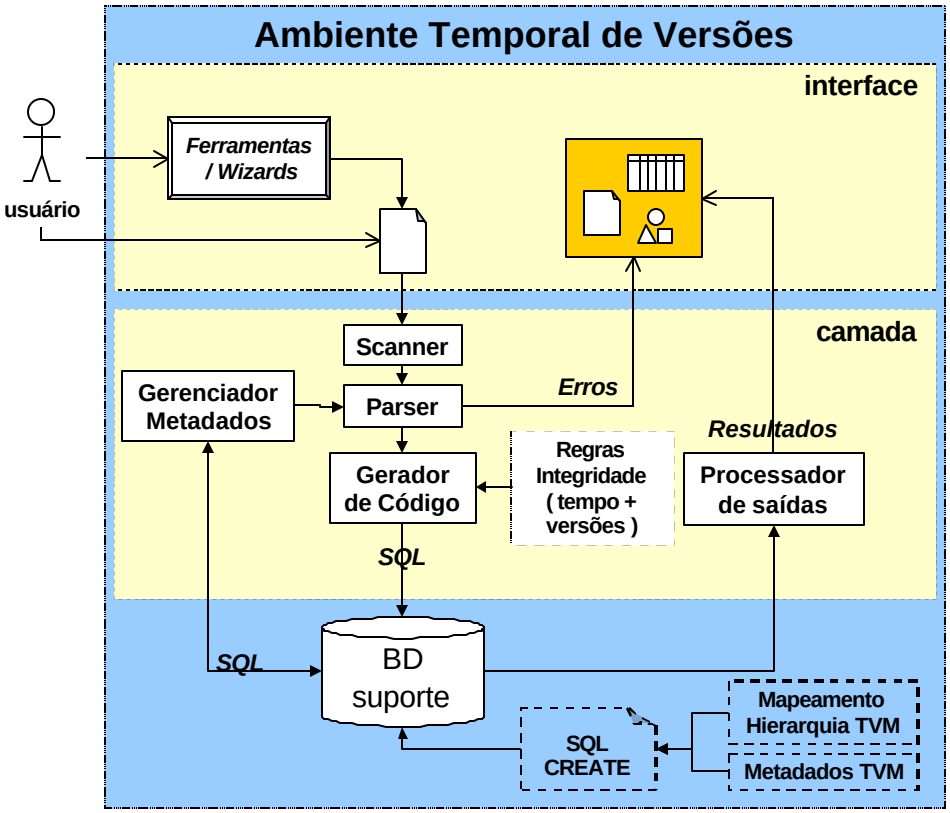


Figura 4.2 – Ambiente Temporal de Versões: interface, camada e BD

4.1.2 Funcionalidades

O Ambiente Temporal de Versões proporciona ao usuário as funcionalidades básicas de um SGBD divididas nos seguintes módulos:

- Especificação de Classes – o usuário pode especificar suas classes através da ferramenta de apoio ou fornecer um arquivo com as definições. Posteriormente, o respectivo arquivo gerado em SQL pode ser executado na base;
- Gerenciamento de Classes – alterações nas definições das classes (atributos, relacionamentos, cardinalidades, ...) podem ser realizadas;
- Gerenciamento de Dados – o usuário pode instanciar seus objetos e gerenciá-los;
- Consulta a Dados – o usuário pode consultar a base e obter resultados em texto, tabela ou gráficos.

Cada uma dessas funções é mapeada pela camada para o seu respectivo conjunto de comandos ou consultas SQL, como ilustra a Figura 4.3.

Dando início à implementação do Ambiente, este trabalho propõe a realização do mapeamento da hierarquia do TVM para o banco de dados objeto-relacional. Adicionalmente, é apresentada uma ferramenta para auxílio na especificação de classes de aplicação, simplificando o trabalho do usuário durante a definição de classes. Desse modo, o usuário não precisa ter conhecimento prévio da linguagem de definição de dados do Modelo.

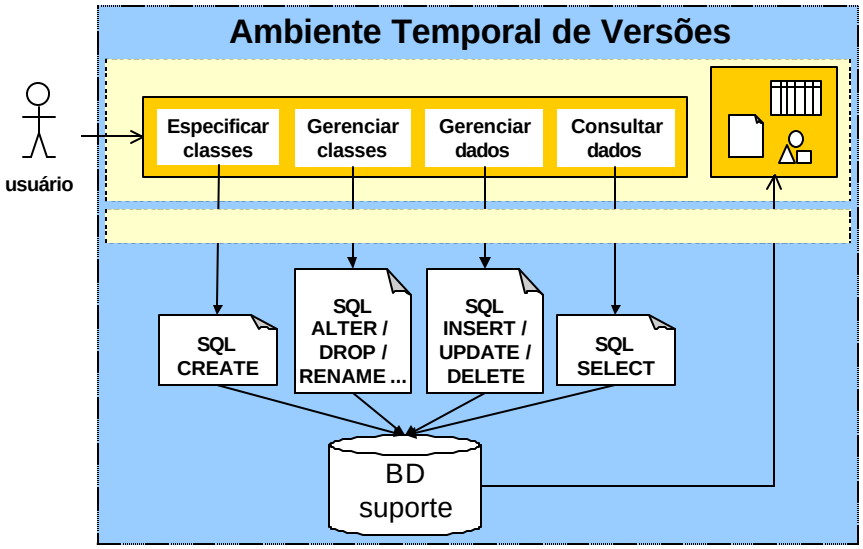


Figura 4.3 – Interações do usuário com o Ambiente

4.2 Mapeamento da Hierarquia

A criação das classes em um banco de dados depende do mapeamento prévio da hierarquia do TVM para o respectivo banco, como apresentado na Figura 4.4.

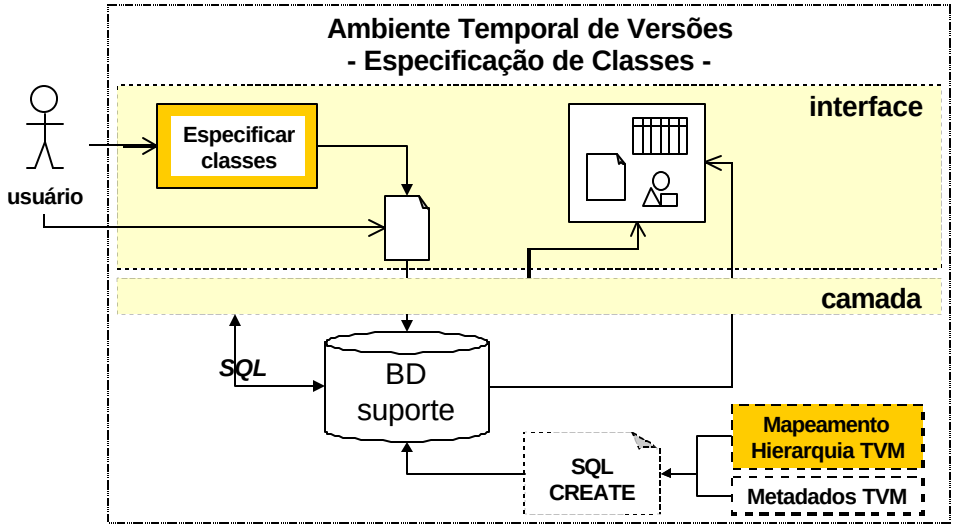


Figura 4.4 – Especificação de Classes no Ambiente

A hierarquia de classes do Modelo Temporal de Versões pode ser mapeada para uma hierarquia de classes ou tabelas equivalente no SGBD escolhido. O TVM define métodos para todas as operações que podem ser aplicadas aos objetos, e muitos deles podem ser realizados através de comandos SQL. O mapeamento dos métodos é apresentado na forma de um pseudo-algoritmo que combina atribuição (:=), variáveis (*varNome*), chamadas a métodos, condições (*SE condição: SENÃO:*), *loops* (*PARA CADA* e *ENQUANTO..FAÇA*), comandos, consultas SQL, comentários (//), acesso à instância de *VersionedObjectControl* do objeto (*refVOC*) e ocorrência de erros (*ERRO*). A execução de métodos que possuem um valor de saída pára no momento que é executada uma operação *retorna.* Início e fim dos comandos são indicados pela identação à esquerda.

Adicionalmente, as questões de direitos de acesso às instâncias de *VersionedObjectControl* devem ser especificadas pelo administrador da base após sua criação. A seguir estão os mapeamentos das classes da hierarquia base. Em princípio, as operações estão ordenadas alfabeticamente. Porém, para aproveitar melhor o espaço, algumas podem estar fora de ordem.

4.2.1 Classe *Object*

A classe *Object* é mapeada para uma classe não instanciável. O atributo tvOID é mapeado para um tipo *string*. A Tabela 4.1 apresenta o mapeamento dos principais comandos dos métodos.

Tabela 4.1 – Mapeamento dos métodos da classe *Object*

Operação Mapeamento
Object ( ) varE := SQL SELECT: nos metadados, obtém o número da próxima entidade varC := getClassId varV := findVersion (varE, varC) tvOID := new OIDt (varE,varC,varV)
Object (ascendId: OIDt) SE verifyAscendId(ascendId) *true*: varE := ascendId.getTvOid.getEntityNr varC := getClassId varV := findVersion (varE, varC) tvOID := new OIDt (varE,varC,varV) SENÃO: ERRO // ascendente inexistente
Object (entityId: integer, classId: integer, versionId: integer) SQL SELECT: na classe classId com entidade entityId e versão versionId, SE obtém vazio: SQL SELECT: nos metadados, SE entityId é entidade válida E classId é classe válida: varE := entityId varC := classId SE versionId = 0: // objeto versionado SQL SELECT: SE existem 2 versões na classe varC com entidade varE e são as versões 1 e 2: // só insere na primeira derivação varV := 0 tvOID := new OIDt (varE,varC,varV) SENÃO: ERRO // só insere na primeira derivação SENÃO: SE versionId null: // controle do obj. Versionado ou normal SQL SELECT: nos metadados, SE classId identifica classe normal ou VersionedObjectControl: varV := null tvOID := new OIDt (varE,varC,varV) SENÃO: ERRO // não é normal ou VOC SENÃO: // versão qualquer SE findVersion(varE,varC) = varV: tvOID := new OIDt (varE,varC,varV) SENÃO: ERRO // número de versão impróprio SENÃO: ERRO // número entidade ou classe inválido SENÃO: ERRO // objeto existente

Operação Mapeamento
Object (entityName) SE verifyEntityName(entityName) *true*: varE := getEntityId (entityName) varC := getClassId varV := findVersion (varE, varC) tvOID := new OIDt (varE,varC,varV) SENÃO: ERRO // entidade inexistente
delete (allReferences: boolean) SE isDeleteAllowed(allReferences) *true*: SE allReferences *true*: SQL SELECT: nos metadados, todas as classes q referenciam SQL UPDATE: os atributos q referenciam o objeto recebem null // allReferences = associação E agregação SQL DROP SENÃO: ERRO
deleteObjectTree (allReferences: boolean) SE isDeleteTreeAllowed(allReferences) *true*: PARA CADA objeto descendente: objeto.deleteObjectTree(allReferences) PARA o objeto: delete(allReferences) SENÃO: ERRO
findVersion (entityId: integer, classId: integer): integer SQL SELECT: nos metadados, obtém tipo da classe SE classe normal: retorna null SENÃO: SE getClasseName = VersionedObjectControl E não existe objeto versionado com entidade entityId e classe classId: retorna null SENÃO: SQL SELECT: na classe, SE não existe objeto com entidade entityId: retorna 1 SENÃO: SE existe refVOC <> null retorna refVOC.getNextVersionNumber SENÃO: SQL SELECT: SE existe só um objeto com entityId: retorna 2 SENÃO: ERRO // não é possível encontrar identificação para a versão
getAscendant ( ): set(OIDt) SQL SELECT: nos metadados, obtém classes ascendentes SQL SELECT: nas classes ascendentes, retorna os objetos com base no tvOID
getAscendant (className: string): OIDt SQL SELECT: na classe className, retorna o ascendente com base no tvOID
getClassId ( ): integer SQL SELECT: nos metadados, retorna identificador da classe
getClassName ( ): string SQL SELECT: nos metadados, retorna o nome da classe do objeto
getCorrespondence (className: string): string SQL SELECT: nos metadados, retorna a correspondência da classe com className
getCorrespondenceAsc (className: string): string SQL SELECT: nos metadados, retorna a correspondência da classe com className, só a cardinalidade da ascendente // parte depois dos :
getCorrespondenceDesc (className: string): string SQL SELECT: nos metadados, retorna a correspondência da classe com className, só a cardinalidade da descendente // parte antes dos :

Operação Mapeamento
getCompleteObject ( ): set(Object) varX := new (set(null)) varX := self // PRIMEIRO: percorre agregações: SQL SELECT: nos metadados, obtém SE objeto possui relacionamento agreg. por referência: SQL SELECT: na classe, obtém oids que referencia na agregação: varX := varX + objeto.getCompleteObject // SEGUNDO: percorre ascendentes: PARA CADA objeto ascendente: varX := varX + objeto.getCompleteObject retorna varX
getDescendant ( ): set(OIDt) SQL SELECT: nos metadados, obtém classes descendentes SQL SELECT: nas classes descendentes, retorna os objetos da entidade com base no tvOID
getDescendant (className: string): OIDt SQL SELECT: na classe descendente, retorna o descendente com base no tvOID
getEntityId (entityName: string): integer SQL SELECT: nos metadados, retorna o identificador da entidade
getNickname ( ): set(string) SQL SELECT: na classe *Name,* retorna os apelidos do objeto
getObject ( ): Object SQL SELECT: retorna o objeto
getTvOid ( ): OIDt SQL SELECT: retorna tvOID do objeto
isDeleteAllowed (allReferences: boolean): boolean SE getDescendant () diferente de vazio: retorna *false* SQL SELECT: nos metadados, obtém SE objeto faz parte de classe agregada: retorna *false* SE allReferences *true*: retorna *true* SENÃO: SQL SELECT: nos metadados obtém SE existe referência para ou a partir do objeto: retorna *false* SENÃO: retorna *true*
isDeleteTreeAllowed (allReferences: boolean): boolean SQL SELECT: nos metadados, obtém SE objeto faz parte de classe agregada: retorna *false* SE allReferences *false*: SQL SELECT: nos metadados, obtém SE existe referência para ou a partir do objeto: retorna *false* SE getDescendant( ) diferente de vazio: PARA CADA objeto descendente: SE isDeleteTreeAllowed (allReferences) *false*: retorna *false* retorna *true* SENÃO: retorna *true*
verifyAscendId (ascendId: OIDt): boolean SQL SELECT: nos metadados, obtém classes ascendentes SQL SELECT: nas classes ascendentes, SE existe um tvOID com o mesmo valor passado por parâmetro na classe: retorna *true* SENÃO: retorna *false*
verifyEntityName (entName: string): boolean SE getEntityId (entName) <> null: retorna *true* SENÃO: retorna *false*

4.2.2 Classe *TemporalObject*

A classe *TemporalObject* é mapeada para uma classe não instanciável, contendo o atributo booleano *alive*, o qual possui valor inicial *true*. A Tabela 4.2 apresenta o mapeamento dos métodos com seus principais comandos.

Tabela 4.2 – Mapeamento dos métodos da classe *TemporalObject*

Operação Mapeamento
TemporalObject ( ) alive := *true* super ( )
TemporalObject (ascendId: OIDt) alive := *true* super (ascendId)
TemporalObject (entityName: string) alive := *true* super (entityName)
TemporalObject (entityId: integer, classId: integer, versionId: integer) alive := *true* super (entityId, classId, versionId)

closeTemporalLabels ( )
SQL SELECT: nos metadados, obtém atributos e relacionamentos temporais
PARA CADA um retornado: close( )
// *observação:* o método Object.close() pode ter o seguinte código:
obtém histórico (getHistory):
SE TTF null: // valor do estado atual da base (TTF em aberto)
// para: valor presente, valor presente com TVF, e valor futuro (TVI, TVF > now)
SQL UPDATE: TTF := tempo de transação
SE TVI < now E TVF null: // valor atual da base (TVF e TTF em aberto)
// observação: SQL INSERT = new InstantAttribute ou new InstantRelationship
SQL INSERT: mesmo valor da informação, mesmo TVI, TVF = tempo de transação, TTI = tempo de transação, TTF = null

delete ( ) setTemporalAttribute(alive, false) closeTemporalLabels ( )
getAlive ( ): boolean SQL SELECT: retorna o valor do atributo *alive*
getAttributeHistory (atribName: string): set (InstantAttribute) // getHistory SQL SELECT: nos metadados, obtém SE *atribName* é temporal: SQL SELECT: retorna todos os *InstantAttribute* de *atribName* // atribName.getHistory SENÃO: retorna valor de *atribName* e rótulo temporal com *null* **getAttributeValueAt (atribName: string, i: instant): InstantAttribute** SQL SELECT: nos metadados, obtém SE *atribName* é temporal: SQL SELECT: retorna o *InstantAttribute* de *atribName* cujo tempo de validade contém o instante *I* // atribName.getValueAt(i) SENÃO: retorna valor de *atribName* e rótulo temporal com *null*
getLifetimeI ( ): instant SQL SELECT: retorna o TVI do atributo alive *true* mais recente
getLifetimeF ( ): instant SE getAlive false: retorna o TVF do atributo *alive true* mais recente SENÃO: retorna *null*

Operação Mapeamento
getObjectHistory ( ): set (InstantAttribute) SQL SELECT: retorna os valores do atributo *alive* com os respectivos rótulos de tempo // alive.getHistory

getRelationshipHistory (relatedObjId: OIDt, relatName: string): set (InstantRelationship)
// getHistory
SQL SELECT: nos metadados, obtém SE relacionamento é temporal:
SQL SELECT: retorna todos os *InstantRelationship* de *relatName* com o objeto *relatedObjId*
// relatName.getHistory
SENÃO: retorna valor de *relatName* com o objeto *relatedObjId* rótulo temporal com *null*

getRelationshipHistoryAt (relatedObjId: OIDt, relatName: string, i: instant): InstantRelationship SQL SELECT: nos metadados, obtém SE relacionamento é temporal: SQL SELECT: retorna o *InstantRelationship* de *relatName* com o objeto *relatedObjId* cujo tempo de validade contém o instante *I* // relatName.getValueAt(i) SENÃO: retorna valor de *relatName* com o objeto *relatedObjId* e rótulo temporal com *null*
setTemporalAttribute (attribName: string, newValue: Object) SQL SELECT: nos metadados, obtém SE attribName é atributo temporal: SQL UPDATE: atribName.TTF recebe o tempo de transação // observação: SQL INSERT = new InstantAttribute SQL INSERT: insere uma nova tupla com atribName atual e TVI igual ao da atual, TVF com o tempo de transação menos um instante, e TTI com o tempo da transação, TTF *null* SQL INSERT: insere uma nova tupla com atribName newValue, TVI e TTI com o tempo da transação, TVF e TTF *null* SENÃO: ERRO // não pode atualizar histórico de atributo sem tempo
setTemporalRelationship (relatName: string, newO1: OIDt, newO2: OIDt) SQL SELECT: nos metadados, obtém SE relatName é relacionamento temporal: SQL UPDATE: relatName.TTF recebe o tempo de transação // observação: SQL INSERT = new InstantRelationship SQL INSERT: insere uma nova tupla com o1 e o2 atuais e TVI igual ao da atual, TVF com o tempo de transação menos um instante, e TTI com o tempo da transação, TTF *null* SQL INSERT: insere uma nova tupla com relatName newO1 e newO2, TVI e TTI com o tempo da transação, TVF e TTF *null* SENÃO: ERRO// não pode atualizar histórico de relacionamento sem tempo

4.2.3 Classe *TemporalVersion*

A classe *TemporalVersion* é mapeada para uma classe contendo os atributos definidos no modelo. Os atributos do tipo *set* (*ascendant*, *descendant*, *predecessor* e *successor*) devem ser mapeados para um tipo de coleção de OIDt que não admite repetições e não está ordenado. O atributo *configuration* é mapeado para um tipo booleano, e *status* para um tipo caracter. A Tabela 4.3 mostra como os principais comandos dos métodos da classe *TemporalVersion* são tratados pelo mapeamento.

Os métodos que envolvem conjuntos (*set*) usam os operadores + e – para indicar a entrada e a saída de um elemento do conjunto. Por exemplo, *getAscendant + ascendId* une um novo ascendente ao conjunto de ascendentes existentes. Os métodos *getConfiguration* e *restore* não estão mapeados devido às suas complexidades, sendo deixados para estudos futuros.

Tabela 4.3 – Mapeamento dos métodos da classe *TemporalVersion*

Operação Mapeamento
TemporalVersion ( ) SQL SELECT: nos metadados, SE subclasse: ERRO // ascendente não especificado SENÃO: ascendant := null configuraltion := false descendant := null predecessor := null status := 'W' successor := null super ( )
TemporalVersion (entityName: string) SQL SELECT: nos metadados, SE existe entidade = entityName: SQL SELECT: na classe, se existe objeto com entidade = entityName: ERRO // entidade existente, informar predecessor SENÃO: SQL SELECT: nos metadados, obtém SE subclasse na hierarquia: SQL SELECT: nos metadados, obtém se existe objeto em todas classes ascendentes com entidade = entityName: varX := new set(null) PARA CADA classe ascendente: varC := objeto.getAscendant (classe) varX := varX + varC.getFirstElement.getVersionedObjectId ascendant := varX SENÃO: ERRO // se é subclasse precisa instanciar primeiro os ascendentes SENÃO: // classe raiz ascendant := null configuration := false descendant := null predecessor := null status : ='W' successor := null super (entityName) SE ascendant <> null: PARA CADA objeto em varX: SE objeto.getDescendant = null: objeto.setDescendant (novoOID) SENÃO: objeto.addDescendant(novoOID) SENÃO: ERRO // entidade inexistente
addAscendant (ascendId: OIDt) varC := ascendId.getClassName SE getAscendant.length = getCorrespondenceAsc(varC): ERRO // excedeu correspondência SENÃO: setTemporalAttribute (ascendant, getAscendant + ascendId)
addDescendant (descendId: OIDt) varC := descendId.getClassName SE getDescendant.length = getCorrespondenceDesc(varC): ERRO // excedeu correspondência SENÃO: setTemporalAttribute (descendant, getDescendant + descendId)
addSuccessor (succId: OIDt) setTemporalAttribute (successor, getSuccessor + succId)

Operação
Mapeamento
TemporalVersion (ascendId: set(OIDt))
SE verifyAscendId (ascendId) *true*: PARA CADA objeto em ascendId: SE objeto.getDescendant.length = objeto.getCorrespondenceDesc (getClassId): ERRO // não pode acrescentar mais descendentes ascendant := ascendId configuration := false descendant := null varC := getClassId varE := ascendId.getFirstElement.getTVOID.getEntityNr SQL SELECT: na classe varC com entidade varE, SE obtém vazio: predecesssor : = null SENÃO: ERRO // deve informar predecessor status : = 'W' successor := null super (ascendId) PARA CADA objeto em ascendId: SE objeto.getDescendant = null: objeto.setDescendant (novoOID) SENÃO: objeto.addDescendant(novoOID) SENÃO: ERRO // ascendente inválido
TemporalVersion (entityId: integer, classId: integer, versionId: integer)
SQL SELECT: na classe classId, SE existe objeto com entidade entityId e versão versionId: ERRO // objeto existente SENÃO: SQL SELECT: nos metadados SE existe entidade entityId E classe classId SQL SELECT: nos metadados, obtém SE subclasse na hierarquia: SQL SELECT: nos metadados, obtém se existe objeto em todas classes ascendentes com entidade = entityName: varX := new set(null) PARA CADA classe ascendente varC := objeto.getAscendant (classe) varX := varX + varC.getFirstElement.getVersionedObjectId PARA CADA objeto em varX SE objeto.getDescendant.length = objeto.getCorrespondenceDesc(getClassName): ERRO // não pode acrescentar descendente ascendant := varX SENÃO: ERRO // se é subclasse PR ECISA de ascendente SENÃO: // classe raiz ascendant := null configuration := false descendant := null predecessor := null status : ='W' successor := null super (entityId, classId, versionId) SE ascendant <> null: PARA CADA objeto em ascendant: SE objeto.getDescendant = null: objeto.setDescendant (novoOID) SENÃO: objeto.addDescendant(novoOID) SENÃO: ERRO // não existe entidade e classe

Operação Mapeamento

TemporalVersion (predecId: set(OIDt), ascendId: set(OIDt), config: boolean)
SE verifyAscendId(ascendId)
PARA CADA objeto em ascendId
SE objeto.getDescendant.length = objeto.getCorrespondenceDesc (getClassId):
ERRO // excedeu correspondência
ascendant := ascendId
configuration := config
descendant := null
predecessor := predecId
status := 'W'
successor := null
super (predecId.getFirstElement.getEntityName)
SE predecId <> null:
PARA CADA objeto em predecId // atualiza sucessores dos predecessores
objeto.addSuccessor(novoOID)
SE objeto.getStatus( ) = 'W': objeto.setTemporalAttribute (status, 'S')
SE ascendant <> null:
PARA CADA objeto em ascendId:
SE objeto.getDescendant = null: objeto.setDescendant (novoOID)
SENÃO: objeto.addDescendant(novoOID)
SENÃO: ERRO // ascendente inexistente

getAscendant (): set(OIDt) SQL SELECT: retorna o(s) valor(es) do atributo *ascendant*
getAscendant (className: string): set(OIDt) SQL SELECT: retorna o(s) *valor*(es) do atributo *ascendant* na classe especificada
getDescendant ( ): set(OIDt) SQL SELECT: retorna o(s) valor(es) do atributo *descendant*
getDescendant (className: string): set(OIDt) SQL SELECT: retorna o(s) valor(es) do atributo *descendant* na classe especificada
getOIDControl ( ): OIDt SQL SELECT: retorna o tvOID da respectiva instância em *VersionedObjectControl*
getPredecessor ( ): set (OIDt) SQL SELECT: retorna o(s) valor(es) do atributo *predecessor*

getCompleteObject ( ): set (OIDt)
varX := new (set(null))
varX := objeto
// primeiro: percorre agregações:
SQL SELECT: nos metadados, obtém SE objeto possui relacionamento de agregação por refer.
SQL SELECT: na classe, obtém oids que referencia na agregação:
varX := varX + getCompleteObject (objetosReferenciados)
// segundo: percorre objetos ascendentes
PARA CADA classe ascendente
varY := getVersionedObject
varX := varX + getCompleteObject (varY)
retorna varX

Operação
Mapeamento

delete (allReferences: boolean)

```
SE isDeleteAllowed (allReferences) true
    SE getTvOID.getVersionNr = 0          // objeto versionado: apaga todas versões
        SQL SELECT: na classe, obtém todas as versões
            PARA CADA versão:
                SE versão.isDeleteAllowed (allReferences) false:
                    ERRO // não pode excluir as versões do objeto versionado
            PARA CADA versão (da última para a primeira na árvore de derivação):
                versão.delete (allReferences)
        // exclui normalmente
        SE getAscendant <> null
            PARA CADA objeto ascendente:
                SE objeto.getDescendant.length =1: objeto.setDescendant (null)
                SENÃO: objeto.removeDescendant (self.getTvOid)
        SE getPredecessor <> null
            PARA CADA objeto predecessor:
                SE objeto.getSuccessor.length = 1: objeto.setSuccessor (null)
                SENÃO: objeto.removeSuccessor (self.getTvOid)
        setStatus( 'D')
        SE allReferences true:
            PARA CADA objeto relacionado:
                apaga a referência a ele          // setTemporalRelationship
        // atualiza refVOC
        se refVOC <> null
            refVOC.updateConfigurationCount
            refVOC.updateVersionCount
            SE refVOC.getCurrentVersion = self.getTvOid:
                varAt := refVOC.currentVersion.getCurrent.getTempLabel.getVTimei – 1
                varI := refVOC.currentVersion.getValueAt (varAt)
                varLabel := varI.getTempLabel
                ENQUANTO varLabel.getTTimef <> null FAÇA:
                    // percorre o histórico do atributo currentVersion até encontrar uma
                    // ex-versão corrente que não tenha sido excluída
                    varAt := varLabel.getVTimei – 1
                    varI := refVOC.currentVersion.getValueAt (varAt)
                    varLabel := varI.getTempLabel
                refVOC.setCurrentVersion (varI.getValue.getTvOid)
            SE refVOC.getLastVersion = self.getTvOid:
                varAt := refVOC.lastVersion.getCurrent.getTempLabel.getVTimei – 1
                varI := refVOC.lastVersion.getValueAt (varAt)
                varLabel := varI.getTempLabel
                ENQUANTO varLabel.getTTimef <> null FAÇA:
                    // percorre o histórico do atributo lastVersion até encontrar uma
                    // ex-última versão que não tenha sido excluída
                    varAt := varLabel.getVTimei – 1
                    varI := refVOC.lastVersion.getValueAt (varAt)
                    varLabel := varI.getTempLabel
                refVOC.setLastVersion (varI.getValue.getTvOid)
        // atualiza alive
        super.delete( )
        SE refVOC = null OU refVOC.versionCount = 0:
            getVersionedObjectControl.delete () // apaga o controle
    SENÃO: ERRO              // não pode excluir a versão
```

Operação Mapeamento
deleteObjectTree (allReferences: boolean) SE isDeleteTreeAllowed *true*: SE getDescendant <> null: // possui descendentes PARA CADA objeto descendente: SE objeto.isDeleteTreeAllowed(allReferences) *false*: ERRO // não pode excluir PARA CADA objeto descendente: objeto.deleteObjectTree(allReferences) SENÃO: delete(allReferences) SENÃO: ERRO // não pode excluir objeto

derive (versionId: set (OIDt))
SE versionId.length = 0: ERRO // derivação precisa de predecessores
SENÃO:
SE existe configuração em versionId: // tem de atualizar os predecessores
PARA CADA objeto em versionId:
SE objeto.isConfiguration
versionId := versionId – objeto.getTVOid
versionId := versionId + objeto.getPredecessor
derive (versionId)
SENÃO: // não é configuração
varV := new set (null)
SQL SELECT: nos metadados, SE subclasse:
varA := new set (null)
PARA CADA objeto em versionId:
SE objeto.getAscendant NOT IN varA:
varA := varA + objeto.getAscendant
PARA CADA objeto em varA:
SE objeto.getVersionedObjectId NOT IN varV:
varV := varV + objeto.getVersionedObjectId
varY := new TemporalVersion (versionId, varV, false)
SE versionId.length = 1 E versionId.refVOC = null // PRIMEIRA derivação!
varE := versionId.getFirstElement.getEntityNr
varC := versionId.getFirstElement.getClassNr
SE getStatus( ) = 'W': setTemporalAttribute (status, 'S')
// cria controle:
varX := new VersionedObjectControl
(varE, varC, varY.getTvOid, versionId.getTvOid, varY.getTvOid)
versionId.refVOC := varX
varY.refVOC := varX
varZ := new TemporalVersion (varE, varC, 0) // objeto versionado V=0
varZ.refVOC := varX
SENÃO: // derivação qualquer
varY.refVOC := versionId.getFirstElement.refVOC
// atualiza refVOC:
SE refVOC.userCurrentFlag false:
refVOC.setTemporalAttribute(currentVersion, varY.getTvOid)
refVOC.setTemporalAttribute(lastVersion, varY.getTvOid)
refVOC.updateVersionCount
refVOC.updateNextVersionNumber

Operação Mapeamento

derive (versionId: set (OIDt), ascendId: set(OIDt), config: boolean): OIDt
// observação: o parâmetro config é true quando executado por getConfiguration
SE versionId.length = 0: ERRO // derivação precisa de predecessores
SENÃO:
SE existe configuração em versionId: // tem de atualizar os predecessores
PARA CADA objeto em versionId:
SE objeto.isConfiguration
versionId := versionId – objeto.getTVOid
versionId := versionId + objeto.getPredecessor
derive (versionId)
SENÃO: // não é configuração
varY := new TemporalVersion (versionId, ascendId, false)
SE versionId.length = 1 E versionId.refVOC = null // PRIMEIRA derivação!
varE := versionId.getFirstElement.getEntityNr
varC := versionId.getFirstElement.getClassNr
SE getStatus( ) = 'W': setTemporalAttribute (status, 'S')
// cria controle:
varX := new VersionedObjectControl
(varE, varC, varY.getTvOid, versionId.getTvOid, varY.getTvOid)
versionId.refVOC := varX
varY.refVOC := varX
varZ := new TemporalVersion (varE, varC, 0) // objeto versionado V=0
varZ.refVOC := varX
SENÃO: // derivação qualquer
varY.refVOC := versionId.getFirstElement.refVOC
// atualiza refVOC:
SE refVOC.userCurrentFlag *false*:
refVOC.setTemporalAttribute(currentVersion, varY.getTvOid)
refVOC.setTemporalAttribute(lastVersion, varY.getTvOid)
refVOC.updateVersionCount
refVOC.updateNextVersionNumber

getStatus ( ): char SQL SELECT: retorna o valor do atributo *status*
getSuccessor (onlyConfigured: boolean): set (OIDt) SE *onlyConfigured false*: SQL SELECT: retorna o(s) valor(es) do atributo *successor* SE *onlyConfigured true*: SQL SELECT: retorna o(s) valor(es) do atributo *successor* cujo atributo *configuration* é *true*
getVersionedObjectId ( ): OIDt SQL SELECT: na classe verifica SE existe objeto com tvOID que possui entidade e classe igual ao do objeto com versão 0, retorna o tvOID SENÃO: SQL SELECT: na classe verifica SE existe objeto com tvOID que possui entidade e classe igual ao do objeto e versão null OU 1, retorna o tvOID SENÃO : ERRO // não existe objeto versionado
isConfiguration ( ): boolean SQL SELECT: retorna o valor do atributo *configuration*

Operação Mapeamento
isDeleteAllowed (allReferences: boolean): boolean SE getStatus() = 'C' OU getStatus() = 'D': retorna *false* SE getDescendant() diferente de vazio: retorna *false* SE getSuccessor() diferente de vazio: retorna *false* SE getConfiguration *true*: retorna *false* SQL SELECT: nos metadados, obtém SE objeto faz parte de classe agregada: retorna *false* SE allReferences *true*: retorna *true* SENÃO: SQL SELECT: nos metadados obtém SE existe referência para ou a partir do objeto: retorna *false* SENÃO: retorna *true*
isDeleteTreeAllowed (allReferences: boolean): boolean SE getSuccessor() diferente de vazio: retorna *false* SE getConfiguration *true*: retorna *false* SQL SELECT: nos metadados, obtém SE objeto faz parte de classe agregada: retorna *false* SE allReferences *false*: SQL SELECT: nos metadados recupera SE existe referência para ou a partir do objeto: Retorna *false* SE getDescendant( ) diferente de vazio: PARA CADA objeto descendente: SE objeto.isDeleteTreeAllowed (allReferences) *false*: retorna *false* SENÃO: retorna *true*
promote (allAscendant: boolean, allReferenced: boolean) SE getStatus() = 'W': setTemporalAttribute (status, 'S') SE allAscendant *true*: PARA CADA obejto ascendente: SE objeto.getStatus() = 'W': objeto.promote (allAscendant, allReferenced) SE allReferenced *true*: SQL SELECT: nos metadados obtém objetos das classes relacionadas PARA CADA objeto: SE objeto.getStatus() = 'W': objeto.setTemporalAttribute (status,'S') SE getStatus() ='S': setTemporalAttribute (status, 'C') SE allAscendant *true*: PARA CADA objeto ascendente: SE objeto.getStatus () = 'W' OU objeto.getStatus() = 'S': objeto.setTemporalAttribute (status, 'C') SE allReferenced *true*: SQL SELECT: nos metadados obtém objetos das classes relacionadas PARA CADA objeto: SE objeto.getStatus() = 'S': objeto.promote (allAscendant, allReferenced)
removeAscendant (ascendId: OIDt) SE getAscendant.length = 1: ERRO // não pode excluir ascendente SENÃO: setTemporalAttribute (ascendant, getAscendant – ascendId)
removeDescendant (descendId: OIDt) setTemporalAttribute (descendant, getDescendant – descendId)
removeSuccessor (succId: OIDt) setTemporalAttribute (successor, getSuccessor – succId)

Operação Mapeamento
setDescendant (descendId: OIDt) setTemporalAttribute (descendant, descendId)
setStatus (newStatus: char) setTemporalAttribute (status, newStatus)
setSuccessor (succId: OIDt) setTemporalAttribute (successor, succId)
verifyAscendId (ascendId: set (OIDt): boolean SQL SELECT: nos metadados, obtém classes ascendentes PARA CADA elemento do conjunto ascendId: SQL SELECT: nas classes ascendentes, SE NÃO existe um tvOID com o mesmo valor passado por parâmetro na classe: retorna *false* retorna *true*

4.2.4 Classe *VersionedObjectControl*

A classe *VersionedObjectControl* é mapeada para um tipo estruturado contendo todos os atributos definidos no modelo. Os atributos *configurationCount*, *nextVersionNumber* e *versionCount* são mapeados para um tipo inteiro. O atributo *userCurrentFlag* é mapeado para um tipo booleano. Os atributos *currentVersion*, *firstVersion* e *lastVersion* são mapeados para OIDt. A Tabela 4.4 apresenta os principais comandos dos métodos da classe *VersionedObjectControl*.

Tabela 4.4 – Mapeamento dos métodos da classe *VersionedObjectControl*

Operação Mapeamento
VersionedObjectControl (entityId: integer, classId: integer, configCount: integer, currentV: OIDt, firstV: OIDt, lastV: OIDt, nextVNumber: integer, userCurrentFlag: boolean, vCount: integer) configurationCount := configCount currentVersion := currentV firstVersion := firstV lastVersion := lastV nextVersionNumber := nextVNumber userCurrentFlag := false versionCount := vCount super (entityId, classId, null)
VersionedObjectControl (entityId: integer, classId: integer, currentV: OIDt, firstV: OIDt, lastV: OIDt) configurationCount := 0 currentVersion := currentV firstVersion := firstV lastVersion := lastV nextVersionNumber := 3 userCurrentFlag := false versionCount := 2 super (entityId, classId, null)
delete (entityId: integer, classId: integer) SQL SELECT: nos metadados, SE existe referência a partir de outro objeto ou versão: ERRO // não pode excluir controle de versões SENÃO: super.delete ( )
getConfigurationCount ( ): integer SQL SELECT: retorna o valor do atributo *configurationCount*

Operação Mapeamento
getCurrentVersion ( ): OIDt SQL SELECT: retorna o valor do atributo *currentVersion*
getFirstVersion ( ): OIDt SQL SELECT: retorna o valor do atributo *firstVersion*
getLastVersion ( ): OIDt SQL SELECT: retorna o valor do atributo *lastVersion*
getNextVersionNumber ( ): integer SQL SELECT: retorna o valor do atributo *nextVersionNumber*
getUserCurrentFlag ( ): boolean SQL SELECT: retorna o valor do atributo *userCurrentFlag* **getVersionCount ( ): integer** SQL SELECT: retorna o valor do atributo *versionCount*
setCurrentVersion ( ) setTemporalAttribute(userCurrentFlag, NOT userCurrentFlag)
setCurrentVersion (newVersionId: OIDt) SE newVersionId existe: setTemporalAttribute(currentVersion, newVersionId) setUserCurrentFlag(*true*) SENÃO: ERRO
setFirstVersion (first: OIDt) setTemporalAttribute(firstVersion, first)
setLastVersion (last: OIDt) setTemporalAttribute(lastVersion, last)
setUserCurrentFlag (flag: boolean) setTemporalAttribute (userCurrentFlag, NOT getUserCurrentFlag)
updateConfigurationCount ( ) varX := SQL SELECT: retorna count (versões com isConfiguration *true*) SE getConfigurationCount <> varX: setTemporalAttribute (configurationCount, varX)
updateVersionCount ( ) varX:= SQL SELECT: retorna count (versões referenciadas com getVersionNr <> 0) SE getVersionCount <> varX setTemporalAttribute (versionCount, varX)
updateNextVersionNumber ( ) nextVersionNumber := getNextVersionNumber + 1

As informações a respeito dos objetos versionados são armazenadas em uma única tabela chamada *VersionedObjectControls* criada a partir do tipo *VersionedObjectControl.* Toda a classe de aplicação temporal versionada tem um atributo para representar o seu relacionamento com essa classe. Então, os objetos versionados e as versões possuem uma referência para um objeto de *VersionedObjectControls*, que contém os dados do controle do objeto versionado ao qual pertencem. Os objetos sem versões possuem valor *null* para a referência.

4.3 Ferramenta de Apoio à Especificação

No Ambiente Temporal de Versões o usuário pode especificar sua aplicação, com todos os conceitos definidos pelo TVM, através da Ferramenta de Apoio à Especificação. A Figura 4.5 apresenta a interface do Ambiente com o menu de acesso aos seus quatro módulos principais: *Especificar Aplicações*, *Manipular Dados*, *Consultar* e *Administrar Base*.

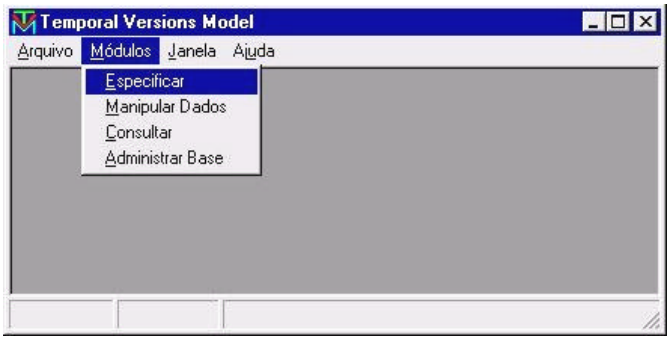


Figura 4.5 – Interface do Ambiente Temporal de Versões

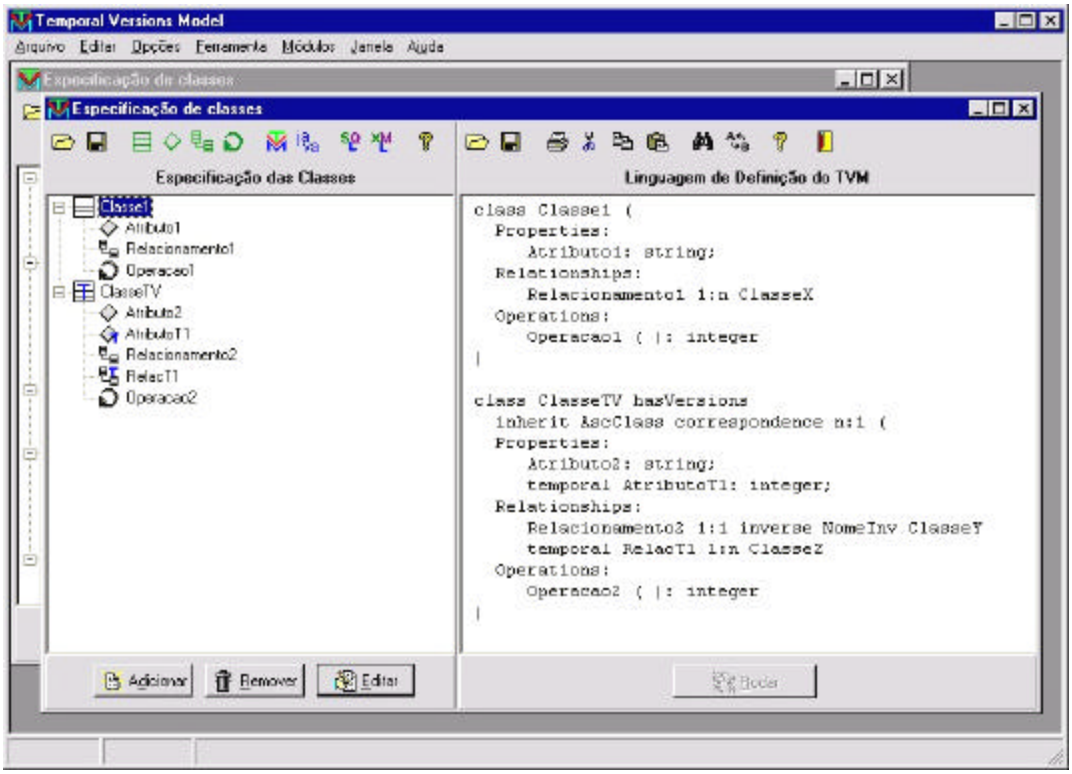


Figura 4.6 – Ferramenta de Apoio à Especificação

A Ferramenta de Apoio à Especificação de Classes é mostrada na Figura 4.6. Ao lado esquerdo está a estrutura em forma de árvore que apresenta classes, atributos, relacionamentos e operações, cada elemento identificado através de um ícone (Tabela 4.1). Ao lado direito está a linguagem de definição do TVM para a aplicação.

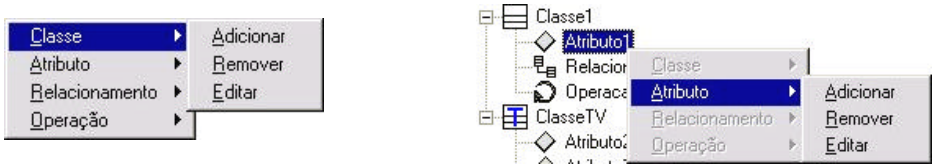
Tabela 4.5 – Ícones na árvore de especificação

Ícone na árvore	Significado
	Classe normal
	Classe temporal
	Atributo
	Atributo temporal
	Relacionamento
	Relacionamento temporal
	Operação

Figura 4.7 – Botões de acesso rápido

Na parte superior, a interface apresenta botões de acesso rápido (Figura 4.7) para respectivamente:

- adicionar uma classe à especificação;
- adicionar um atributo à classe selecionada;
- adicionar um relacionamento à classe selecionada;
- adicionar uma operação à classe selecionada;
- gerar a especificação na linguagem de definição do TVM;
- gerar a especificação em java;
- gerar *script* de criação no banco de dados em SQL;
- gerar a especificação em XML.



a **b**

Figura 4.8 – Menus específicos

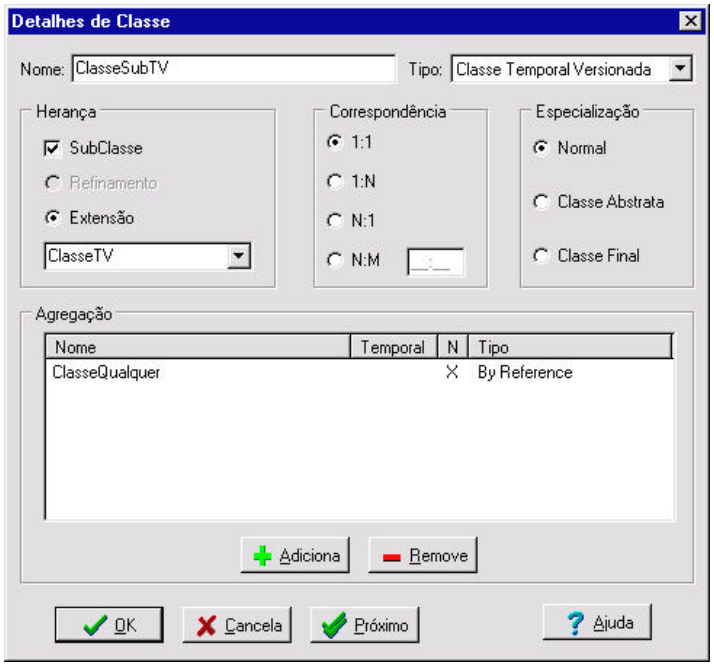


Figura 4.9 – Detalhes da especificação de classes

Quando usuário clica no botão direito do *mouse* sobre a árvore com a especificação das classes, um menu relativo ao elemento selecionado é aberto. A Figura 4.8a ilustra o menu apresentado quando o elemento selecionado é uma classe, e a Figura 4.8b ilustra quando o elemento selecionado é um atributo.

O usuário especifica os detalhes da classe em uma tela própria ilustrada na Figura 4.9. Os detalhes da classe são equivalentes às características apresentadas na linguagem de definição: nome da classe, tipo (normal ou temporal versionada), herança, correspondência, classe abstrata ou final, e agregação. Quando o tipo "Classe Temporal Versionada" é selecionado, a herança por refinamento fica automaticamente desabilitada. O botão "Próximo" insere a classe sendo editada e limpa os campos para adicionar outra classe.

O usuário especifica os detalhes dos atributos na tela ilustrada na Figura 4.10. São apresentados os campos: nome do atributo, tipo, visibilidade, escopo, valor padrão e se o mesmo é temporal. Novamente, o botão "Próximo" insere o atributo sendo editado e limpa os campos para adicionar outro.

A Figura 4.11 ilustra a tela na qual são especificados os relacionamentos. São apresentados os seguintes campos: nome do relacionamento, nome da classe relacionada, nome do relacionamento inverso, cardinalidade e se o relacionamento é temporal. No campo de "Classe Relacionada" o usuário pode selecionar o nome de uma das classes especificadas ou definir o nome de uma nova classe.

A Figura 4.12 ilustra a tela de detalhes de operações, apresentando os seguintes campos: nome da operação, visibilidade, escopo, tipo de especialização, parâmetros e tipo retornado.

Figura 4.10 – Detalhes da especificação de atributos

Figura 4.11 – Detalhes da especificação de relacionamentos

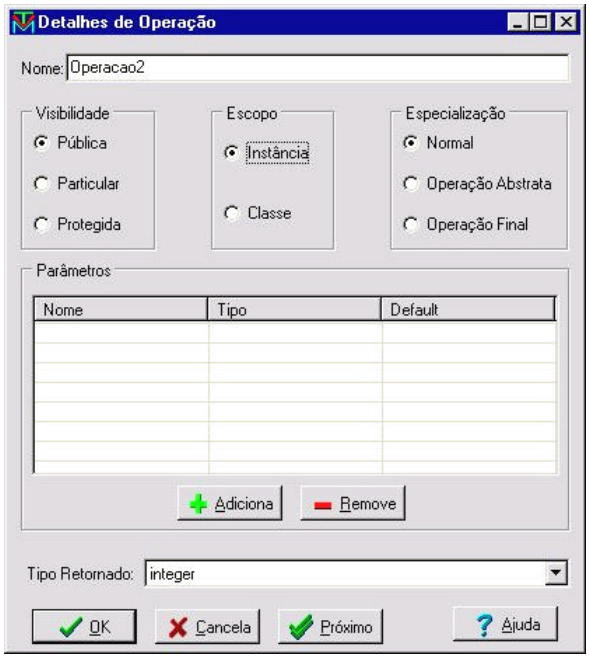


Figura 4.12 – Detalhes da especificação de operações

4.4 Considerações Finais

Esse capítulo apresentou uma arquitetura para desenvolver o funcionamento do Modelo Temporal de Versões sobre um SGBD convencional. Apenas o mapeamento da hierarquia do Modelo foi proposto com detalhes. Para mapear as classes de aplicação normais e temporal versionadas para o SGBD é necessário adicionar os conceitos definidos pelo TVM na especificação das classes quando for gerado o *script* de criação e quando essas classes forem instanciadas.

Saggiorato apresenta as regras de mapeamento para criar classes de aplicação versionadas sobre banco de dados objeto-relacionais em [SAG 99]. O resumo dessas regras é apresentado na Tabela 4.6. Com base nesse estudo, é possível incorporar os conceitos de tempo e realizar o mapeamento das classes de aplicação.

Tabela 4.6 – Regras de mapeamento de Saggiorato

Elemento	**Regra de Mapeamento**
Atributo	Atributo de um tipo objeto
Método	Método de um tipo objeto
Classe não versionável	São gerados: - Um tipo objeto que descreve a estrutura da classe - Uma tabela de objetos que armazena as instâncias da classe - Um gatilho associado à operação de inserção para gerar os OIDs dos objetos - Um gatilho associado à operação de exclusão para manter a integridade do banco de dados
Herança	Adicionar um atributo na classe descendente que referencia a sua classe ascendente. Se houver mais de um ascendente (herança múltipla), adiciona-se um atributo PARA CADA ascendente

Elemento	Regra de Mapeamento
Classe versionável	São gerados: - Um tipo objeto que descreve a estrutura da classe - Uma tabela de objetos que armazena as instâncias da classe - Uma tabela que armazena informações sobre os objetos versionados - Um gatilho associado à operação de inserção para gerar os OIDs dos objetos. - Um gatilho associado à operação de exclusão para manter a integridade do banco de dados, bem como dos objetos versionados e suas versões - Um gatilho associado à operação de atualização para impedir a atualização de versões cujo *status* não seja *em trabalho* - Um gatilho associado à operação de atualização do atributo *Versions* para impedir a atualização dos atributos de controle de versões
Relacionamento de associação [1:1], [1:n]e [n:1]	Adiciona-se um atributo (do tipo REF) no tipo objeto que mapeia a classe da aplicação que efetua a referência
Relacionamento de associação n:m	Cria-se um tipo *nested table* como um conjunto de referências a classe referenciada Adiciona-se um atributo no tipo objeto que mapeia a classe da aplicação que efetua a referência, cujo tipo de dados e a coleção de referência
Relacionamento de agregação por valor	Adiciona-se à classe composta: - Um atributo cujo tipo de dados é o tipo objeto que mapeia a classe componente, se a cardinalidade for 1 - Um atributo cujo tipo de dados é um conjunto do tipo objeto que mapeia a classe componente, se a cardinalidade for n
Relacionamento de agregação por referência	Se a cardinalidade for 1, adiciona-se na classe composta um atributo do tipo REF, que referencia a classe componente Se a cardinalidade for n, cria-se um tipo como um conjunto de referências a classe componente e adiciona-se um atributo na classe composta deste tipo

Finalizando esse capítulo, foi apresentada uma ferramenta para auxílio na especificação de classes do TVM. Nessa interface, o usuário especifica as classes com atributos, relacionamentos e operações em uma estrutura de níveis. Uma vez completada a definição de classes, o usuário pode gerar a especificação na linguagem de definição do TVM.

Pretende-se estender as facilidades da ferramenta permitindo que o usuário entre com outros tipos de arquivos e gere a especificação na linguagem de definição. Uma sugestão é permitir que o usuário especifique suas classes em uma ferramenta própria para modelagem orientada a objetos, como o *Rational Rose.* Através do uso de estereótipos (*stereotypes*), o usuário definiria classes temporais versionadas, relacionamentos de herança por extensão bem como atributos e relacionamentos temporais. A partir do diagrama de classes, o usuário poderá gerar um arquivo que servirá de entrada para a ferramenta. Por exemplo, pode-se estender a ferramenta de modo que transforme as classes escritas em Java (podem ser geradas a partir do diagrama de classes do *Rational Rose*) para a definição do Modelo.

Outras extensões da ferramenta dizem respeito ao tipo de arquivo de saída. Por enquanto, a ferramenta só gera especificação na linguagem de definição do Modelo. A seguir, deverá ser gerado um *script* em SQL para ser rodado no SGBD base. Outras sugestões são gerar a especificação em XML, para integração, e em Java, para programação.

5 Estudo de Caso

Neste capítulo é apresentado um estudo de caso em três partes:

- Representação do Mapeamento no DB2 – considerações sobre a implementação do TVM em um banco de dados objeto-relacional comercial;
- Modelagem da Aplicação – modelagem conceitual de uma aplicação com o uso do TVM;
- Representação Gráfica das Instâncias – representação gráfica do funcionamento da aplicação modelada sobre o banco de dados comercial.

5.1 Representação do Mapeamento no DB2

Como experiência para a implementação do mapeamento do TVM em um banco de dados objeto-relacional foi escolhido o banco de dados *DB2 Universal Database* (*UDB*). Um dos motivos que levaram à escolha desse SGBD é o fato de ser o mais próximo ao padrão SQL-92 no seu suporte aos tipos de dados temporais [SNO 00].

5.1.1 Características do DB2

Esta seção está dividida em duas partes que apresentam: (i) características gerais do DB2 UDB; e (ii) os tipos de dados do DB2, com ênfase nos tipos temporais [BER 00, CHA 98]. A sintaxe dos principais comandos necessários para a implementação do mapeamento estão no Anexo 3.

Características Gerais

Entre as facilidades e características que o DB2 UDB (versão 7) oferece estão:

- tipos de dados definidos pelo usuário (*User-Defined Data Types*) – o usuário define os dados em termos do negócio e do comportamento no banco de dados, em vez de na aplicação;
- funções definidas pelo usuário (UDF, *User-Defined Functions*) – estende a linguagem SQL para permitir funções específicas para o processamento de dados, porém não pode conter comandos SQL;
- SQL recursivo – permite modelar um relacionamento de dados hierárquico em um simples comando SQL, em vez de escrever na aplicação;
- integridade referencial declarativa – assegura que os valores de dados críticos no funcionamento de um banco de dados relacional são mantido em sincronia pela designação do relacionamento quando uma tabela é criada;
- *constraint* – permite definir escala válida e valores padrão para os dados;
- gatilho (*trigger*) – permite executar automaticamente uma função quando um valor é adicionado, excluído ou atualizado na base de dados;
- tipo estruturado – suporta tipos estruturados e permite definir uma hierarquia de tipos com herança simples. Esses tipos podem ser usados para criar tabelas nas quais os atributos do tipo tornam-se colunas de uma tabela com uma coluna

de identificador de objeto. As tabelas também podem ser criadas com base em uma hierarquia de tipos com herança de dados;

- procedimento armazenado (*stored procedure*) – suporta procedimentos que podem ser escritos em várias linguagens (C, JAVA, C++, Cobol, ...);

- tabela temporária (*declared temporary tables*) – uma tabela temporária é acessível somente pela aplicação que a criou, sendo automaticamente excluída ao término dessa aplicação;

- coluna identidade – fornece um tipo de dados coluna identidade (*identity column*) que provê um caminho para o DB2 automaticamente gerar um valor numérico garantido único para cada linha que é adicionada à tabela;

- aprimoramento da usabilidade OLAP – permite agrupar, consolidar e visualizar colunas múltiplas em um resultado simples com o uso de *ROLLUP* e *CUBE*;

- métodos – são definidos implícita ou explicitamente, como parte da definição de um tipo estruturado do usuário;

    - métodos definidos implicitamente são criados para cada tipo estruturado. Métodos *Observer* são definidos para cada atributo do tipo estruturado e permitem as aplicações recuperar o valor do atributo para uma instância do tipo. Métodos *Mutator* são definidos para cada atributo, permitindo a aplicação atualizar a instância do tipo trocando o valor de um atributo;

    - métodos definidos explicitamente, ou métodos definidos pelo usuário, são métodos registrados para um banco de dados em *SYSCAT.FUNCTIONS*, usando uma combinação das expressões *CREATE TYPE* (ou *ALTER TYPE ADD METHOD*) e *CREATE METHOD*. Todos os métodos definidos para um tipo estruturado são definidos no mesmo esquema do tipo.

Para modelagem de objetos complexos, o UDB fornece um conjunto estendido de tipos de dados embutidos e também permite que o usuário defina seus próprios tipos e funções.

Tipos de Dados Básicos e Temporais

O DB2 apresenta os seguintes tipos de dados básicos:

- numéricos – *Smallint*, *Integer*, *Decimal(p,s)* ou *Numeric(p,s)*, *Real*, *Double*;

- caracter – *Char(n)*, *Varchar(n)*, *Long Varchar*;

- gráficos – *Graphic(n)*, *Vargraphic(n)*, *Long Vargraphic(n)*;

- objetos grandes (*large objects*) – *Dbclob(n)*, *Clob(n)*, *Blob(n)*.

As grandes restrições do DB2 em relação aos tipos oferecidos são a ausência de tipos de coleção (*set*, *bag*, *list*, *array*) e tipo lógico (booleano).

Os tipos de dados temporais definidos para instantes são:

- *DATE* – armazena os valores de ano, mês e dia de um instante;

- *TIMESTAMP* – armazena os valores de ano, mês e dia de um instante, como em *DATE*, juntamente com hora, minuto, segundo e um número de dígitos de fração de segundo. O padrão é seis dígitos fracionais;

- *TIME* – armazena os valores de hora, minuto e segundo, com um número opcional de dígitos de fração de segundo (precisão). O valor da precisão zero corresponde à ausência de fração e a granularidade é o segundo;

A hierarquia para os tipos definidos pelo DB2 é apresentada na Figura 5.1.

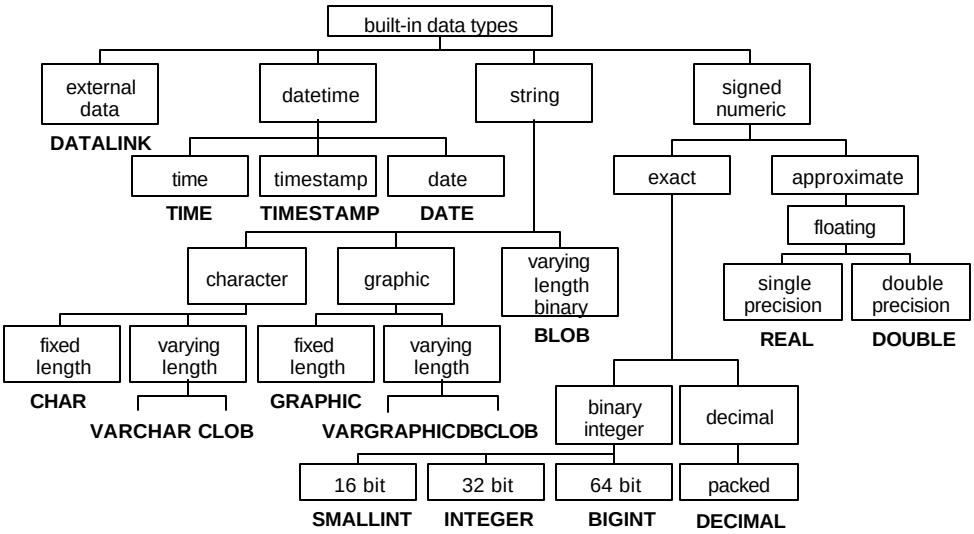


Figura 5.1 – Hierarquia de tipos do DB2

O DB2 não define o tipo *INTERVAL* (presente no SQL92), o qual é um tipo complexo definido a partir da combinação dos campos ano, mês, dia, hora, minuto, e segundo. Porém, o DB2 suporta versões especializadas do tipo de dados *DECIMAL*, chamadas *durações* (*durations*):

- *date duration* – no formato *YYYYMMDD*, é um número *DECIMAL(8,0)* que representa um intervalo de dias, com alcance de 10000 anos;
- *time duration* – no formato *HHMMSS*, é um número *DECIMAL(6,0)* que representa um intervalo de segundos, com alcance de um dia;
- *timestamp duration* – no formato *YYYYMMDD.HHMMSSZZZZZZ*, é um número *DECIMAL(20,6)* que representa um intervalo de microsegundos, com alcance de 10000anos.

Esses valores podem ser armazenados em colunas do tipo *DECIMAL* e representadas por constantes *DECIMAL*. Então,

```
DATE ('1997-11-08') + 00010101
```

adiciona um ano, um mês e um dia ao instante, resultando em 1998-12-09.

O DB2 também suporta um tipo literal altamente restrito, chamado *duração rotulada* (*labeled duration*), que é uma expressão numérica seguida por uma unidade de tempo (singular ou plural). Durações rotuladas podem ser usadas somente na adição e subtração com tipos de instante. Por exemplo,

```
DATE ('1997-11-08') + 1 MONTH
```

As unidades disponíveis são *YEAR*, *MONTH*, *DAY*, *HOUR*, *MINUTE*, *SECOND*, *MICROSECOND*, e plurais dessas palavras chave. A Tabela 5.1 mostra como as facilidades do SQL-92 podem ser simuladas no DB2. Na coluna do SQL-92, *d* denota um valor *datetime*, *i* um intervalo, e *n* um numérico. Na coluna do DB2, *d* denota um valor *datetime*, *i* um *duration timestamp*, e *itype* um inteiro denotando um tipo *interval*.

Tabela 5.1 – Tabela comparativa entre SQL 92 e IBM DB2 UDB [2]

SQL-92	Equivalência no IBM DB2 UDB
Literais:	
DATE '1997-01-01'	DATE ('1997-01-01')
TIME '12:34:56'	TIME ('12:34:56')
TIMESTAMP '1997-01-01 12:34:56'	TIMESTAMP ('1997-01-01-12.34.56.000000')
INTERVAL '3-4' YEAR TO MONTH	40 MONTHS, 000030400 (somente expressões)
INTERVAL '1 23:45:12' DAY TO SECOND	000000012345612.000000, 171312 SECONDS(somente expressões)
Tipos:	
DATE	DATE
TIME	TIME (precisão fixa em 0)
TIMESTAMP	TIMESTAMP (precisão fixa em 6)
TIME WITH TIME ZONE	sem equivalência
TIMESTAMP WITH TIME ZONE	sem equivalência
INTERVAL YEAR TO MONTH	duração de data (DECIMAL (8,0)) com campo DAY zero
INTERVAL DAY TO SECOND	duração timestamp: YEAR, MONTH, e MICROSECOND zero
Predicados:	
$d_1 = d_2$	$d_1 = d_2$
$d_1 < d_2$	$d_1 < d_2$
$d_1 <> d_2$	$d_1 <> d_2$
d_1 BETWEEN d_2 AND d_3	d_1 BETWEEN d_2 AND d_3
$i_1 = i_2$	$i_1 = i_2$
$i_1 < i_2$	$i_1 < i_2$
$i_1 <> i_2$	$i_1 <> i_2$
i_1 BETWEEN i_2 AND i_3	i_1 BETWEEN i_2 AND i_3
d IS NULL	d IS NULL
i IS NULL	i IS NULL
(d_1 , i) OVERLAPS (d_2 , d_3)	$d_1 < d_3$ AND $d_2 <$ (d_1 + i)
Construtores de datetime:	
d + i	d + i
i + d	i + d
d - i	d - i
d AT i	d + i
d AT LOCAL	d + CURRENT TIMEZONE
CURRENT_DATE	CURRENT DATE
CURRENT_TIME	CURRENT TIME
CURRENT_TIMESTAMP	CURRENT TIMESTAMP
Construtores de intervalos:	
$i_1 + i_2$	não é possível
$i_1 - i_2$	não é possível
($d_1 - d_2$) qual	TIMESTAMPDIFF (itype, CHAR ($d_1 - d_2$))
($d_1 - d_2$) MONTH	TIMESTAMPDIFF (64, CHAR ($d_1 - d_2$))
i * n	não é possível
n * i	não é possível
i_1 / i_2	não é possível
+ i	i
- i	não é possível
Outros operadores:	
CAST (d AS DATE)	CAST (d AS DATE)
CAST (d AS TIME)	CAST (d AS TIME)
CAST (d AS TIMESTAMP)	CAST (d AS TIMESTAMP)
CAST (i AS INTERVAL YEAR TO MONTH)	não é possível
CAST (i AS INTERVAL DAY TO SECOND)	não é possível
CAST (d AS CHAR)	CHAR (d)
CAST (i AS CHAR)	não é possível
CAST (i AS INTEGER)	
i is YEAR TO DAY	JULIAN_DAY(DATE (' 001-01-01') + i) -JULIAN_DAY (DA TE (' 001-01-01-00'))
i is DAY TO HOUR	24 * DAY (i) + HOUR (i)
i is DAY TO MINUTE	1440 * DAY (i) + 60 * HOUR (i) + 60 * MINUTE (i) + SECOND (i)
i is DAY TO SECOND	86400*DAY(i) + 3600*HOUR (i) + 60 * MINUTE (i) + SECOND (i)
EXTRACT (DAY FROM d)	DAY (d)
EXTRACT (DAY FROM i)	DAY (i)
EXTRACT (HOUR FROM i)	HOUR (i)

[2] Tabela traduzida de [SNO 00], páginas 45 e 46

5.1.2 Simplificações Realizadas no TVM

Para adaptar o TVM às possibilidades de implementação que o DB2 oferece, as seguintes restrições são estabelecidas:

- herança por refinamento simples – o DB2, assim como a maioria dos SGBDs comerciais, não oferece herança múltipla;
- tamanho – os nomes de classes, atributos, relacionamentos e operações devem ser adaptados para o tamanho máximo de 18 caracteres;
- *boolean* – os dados de tipo booleano devem ser adaptados, pois esse tipo não está disponível ainda no DB2. A sugestão é mapear para tipo *Char(1)* e acrescentar a restrição na coluna (*constraint*) que só pode aceitar os valores 'T' e 'F';
- operações – as operações de recuperar e atualizar atributos são realizadas automaticamente pelos métodos *Observer* e *Mutator*.

Como o DB2 não oferece um tipo de dados para coleção de valores, os atributos de tipo *set* devem ser atualizados para tipos comuns. Essa restrição interfere diretamente nas seguintes características do Modelo:

- *ascendant* e *descendant* – os objetos só podem ter um objeto ascendente e um descendente, o que limita a herança por extensão;
- *successor* e *predecessor* – a derivação entre versões passa a ser representada por uma lista e não mais por um grafo acíclico dirigido.

5.1.3 Implementação da Hierarquia

Antes de entrar nos detalhes da implementação da hierarquia do TVM, dois conceitos fundamentais do DB2 devem ser explicados melhor: *Observer* e *Mutator.*

Observer e *Mutator*

Seja um tipo estruturado simples para endereço [YOU 99]:

```
CREATE TYPE Address
AS (street  Char (30),
    city    Char (20),
    state   Char (2),
    zip     Integer
   ) NOT FINAL;
```

Cada tipo estruturado possui um conjunto de métodos *Observer*/*Mutator* para acesso e atualização do valor dos atributos. Para *Address*, os seguinte métodos são gerados automaticamente pelo sistema:

- *Observer* – cada atributo possui um método *observer* correspondente que retorna o valor do atributo, passando um valor do tipo estruturado.

```
Address.street -> Char (30)
Address.city -> Char (20)
Address.state -> Char (2)
Address.zip -> Integer
```

A notação de ponto simples é para a chamada do método e é associativa à esquerda;

- *Mutator* – cada atributo possui um método *mutator* correspondente para atualizar o seu valor.

```
Address.street(Char (30)) -> Address
Address.city(Char (20)) -> Address
Address.state(Char (2)) -> Address
Address.zip(Integer) -> Address
```

Mapeamento Geral

Os tipos definidos pelo TVM são mapeados para o DB2 conforme a Tabela 5.2. Para o tipo *boolean* é criado um tipo derivado de *Char*, chamado *boo*, que aceita apenas os valores 'T' e 'F'. Para *oidt* também é criado um tipo definido pelo usuário, chamado OIDt, detalhado a seguir.

Tabela 5.2 – Mapeamento dos tipos de dados

TVM	DB2
integer	integer
real	real
string	varchar
char	char
boolean	CREATE DISTINCT TYPE boo
date	date
time	time
instant	timestamp
oidt	CREATE TYPE OIDt
set	-

Com base nos métodos *Observer* e *Mutator*, no uso de funções definidas pelo usuário (UDF), procedimentos armazenados e gatilhos, é possível generalizar o mapeamento das operações da hierarquia do Modelo para o DB2:

- operações de recuperação de valores dos atributos – respectivo método *Observer*;
- operações para atualização de valores de atributos e relacionamentos normais – respectivo método *Mutator*;
- operações para atualização de valores de atributos e relacionamentos temporais – UDFs ou procedimentos armazenados que executam a atualização dos rótulos temporais;
  - operações sem SQL – UDF;
  - operações com SQL – procedimento armazenado;
- operações com retorno *set* – geralmente, os valores de retorno são resultados de uma consulta SQL. Desse modo, essas operações podem retornar os valores direto da consulta, sem armazenar em alguma estrutura;
- atributos e relacionamentos temporais – devem ser definidos gatilhos associados às três operações de manipulação (inserir, atualizar e excluir) para realizar os controles sobre os rótulos temporais de cada elemento temporal.

Classe *Object*

A classe *Object* é mapeada para um tipo estruturado não instanciável (*NOT INSTANTIABLE*), com o atributo tvOID do tipo OIDt. A operação *getTvOid* não precisa ser mapeada porque é realizada pelo *Observer*. As demais operações são mapeadas para procedimentos armazenados que executam os algoritmos especificados na seção anterior. Em vez de retornar um tipo de dados *set*, as operações *getAscendant()*, *getCompleteObject*, *getDescendant()* e *getNickname* retornam uma tabela ou arquivo com o resultado da consulta SQL. As operações *getCorrespondence*, *getCorrespondenceAsc*, *getCorrespondenceDesc* e *isDeleteTreeAllowed* são renomeadas, respectivamente para *getCorresp*, *getCorrespAsc*, *getCorrespDesc* e *isDeleteAllowe*.

Classe *TemporalObject*

A classe *TemporalObject* é mapeada para um tipo estruturado não instanciável, contendo o atributo *alive* do tipo *boo*. Os construtores e a operação *delete* são mapeadas para UDFs, e as demais operações para procedimentos armazenados. As operações *getAttributeHistory*, *getObjectHistory*, *getRelationshipHistory* retornam uma tabela ou arquivo com o resultado da consulta SQL. As seguintes operações são renomeados para atender à limitação de 18 caracteres:

- *getAttributeHistory* para *getAttrHistory*;
- *getAttributeValueAt* para *getAttrValueAt*;
- *getRelationshipHistory* para *getRelatHistory*;
- *getRelationshipValueAt* para *getRelatValueAt*;
- *setTemporalAttribute* para *setTempAttribute*;
- *setTemporalRelationship* para *setTempRelation.*

Classe *TemporalVersion*

A classe *TemporalVersion* é mapeada para um tipo estruturado não instanciável, contendo os atributos definidos no modelo. Como já mencionado, os atributos *ascendant*, *descendant*, *predecessor* e *successor* são mapeados para o tipo OIDt. O atributo *configuration* é mapeado para o tipo *boo*. O atributo *status* é mapeado para um tipo *Char(1)*, e a definição da tabela deve conter uma restrição (*constraint*) assegurando que os valores só podem ser 'W', 'S', 'C' e 'D'.

A operação *isDeleteTreeAllowed* é renomeada para *isDeleteTreeAllowe*, e *getVersionedObjectId* para *getVersObjectId*. As operações *isConfiguration*, *getAscendant()*, *getDescendant()*, *getPredecessor*, *getStatus*, *getSuccessor* não precisam ser mapeadas porque são realizadas pelo *Observer*. As operações *addAscendant*, *addDescendant*, *addSuccessor*, *derive*, *removeAscendant*, *removeDescendant*, *removeSuccessor*, *setAscendant*, *setDescendant*, *setStatus*, *setSuccessor* são mapeadas para UDFs. As demais operações são mapeadas para procedimentos armazenados.

Classe *VersionedObjectControl*

A classe *VersionedObjectControl* é renomeada para *VersionedObjCtrl* e mapeada para um tipo estruturado contendo todos os atributos definidos no modelo. As informações a respeito dos objetos versionados são armazenadas em uma única tabela

chamada *VersionedObjCtrls*, criada a partir do tipo estruturado *VersionedObjCtrl* adicionando colunas para os identificadores da entidade e da classe. Desse modo, toda a classe da aplicação temporal e versionável tem um atributo para representar o seu relacionamento com essa classe (*refVOC*). Ou seja, todo objeto da classe da aplicação, seja ele uma versão ou o próprio objeto versionado, tem uma referência a um objeto de *VersionedObjCtrls*, que contém os dados a respeito do objeto versionado ao qual pertence.

Os atributos são mapeados conforme a Tabela 5.2. Com exceção de *nextVersionNumber*, todos os atributos devem possuir gatilhos de controle dos rótulos temporais. As operações de recuperar valor dos atributos não precisam ser mapeadas por serem realizadas pelo método *Observer*. As operações de atualização de valores são mapeadas para UDFs. As seguintes operações são renomeadas:

- *VersionedObjectControl* para *VersionedObjCtrl*;
- *getConfigurationCount* para *getConfigCount*;
- *getNextVersionNumber* para *getNextVersionNum*;
- *updateConfigurationCount* para *updateConfigCount*;
- *updateVersionCount* para *updateVCount*;
- *updateNextVersionNumber* para *updateNextVersion*.

Rótulo Temporal

O rótulo temporal associado aos atributos e relacionamentos temporalizados é mapeado para um tipo estruturado chamado *TEMPORALLABEL* com os quatro atributos do rótulo temporal. Os tempos de validade são do tipo *DATE*, e os de transação do tipo *TIMESTAMP*.

5.2 Modelagem da Aplicação

O estudo de caso consiste da modelagem de uma empresa de criação de *websites*. Além dos *sites* de seus clientes, a empresa mantém as páginas profissionais de seus empregados. Cada *website* apresenta as seguintes características:

- é composto de uma ou mais páginas, sendo que uma delas deve ser considerada como página inicial. Por exemplo, o cliente pode querer manter em páginas diferentes seus dados pessoais, suas informações profissionais, uma relação de seus *sites* preferidos, e assim por diante, mas todas referenciadas a partir da mesma página principal;
- tem um padrão de página associado composto pela cor e imagem de fundo da página, por um *banner*, e pelas especificações *default* da fonte. Esse padrão é usado como controle do *layout* da página de todos os empregados. Segundo especificação da empresa, esse padrão varia conforme as estações do ano e datas comemorativas. Por exemplo, no mês de aniversário da empresa, a imagem de fundo apresenta um bolo com velas acesas, e o *banner* mostra as ofertas especiais do mês (e desse mesmo modo nas épocas de Páscoa, Carnaval, datas cívicas, Natal, etc);

- está sob a responsabilidade de um grupo de trabalho, que é composto por pelo menos um projetista e um programador;

- pode possuir *links* associados e recursos de multimídia, os quais estão divididos em dois tipos: recursos de imagem e de áudio.

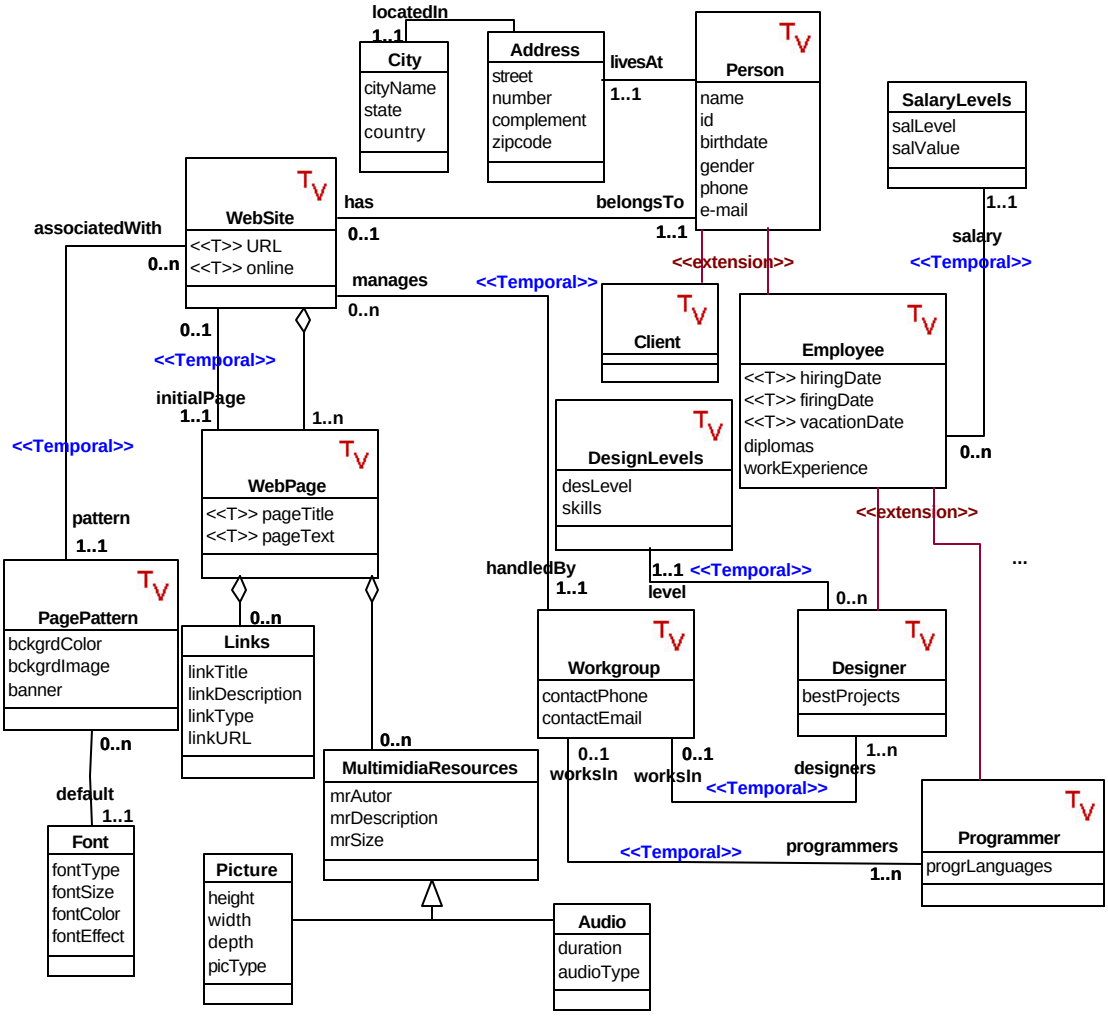


Figura 5.2 – Diagrama de classes do estudo de caso

A Figura 5.2 apresenta apenas uma parte simplificada do diagrama de classes do modelo. As especificações do cliente não são apresentadas, bem como as outras categorias de empregado além de projetista e programador. As operações estão omitidas em todas as classes.

As características de versão e tempo são usadas em:

- classes *WebSite* e *WebPage* – são armazenadas todas as alternativas do *site* e de suas páginas, bem como o histórico de seus atributos;

- relacionamento *pattern* (inverso *associatedWith*) – pode ser considerado um relacionamento chave do modelo. É através dele que a empresa troca o padrão dos *websites* de seus empregados para estarem de acordo com a estação do ano ou data comemorativa;

- relacionamento *initialPage* – é armazenado o histórico das páginas iniciais;

- relacionamento *handledBy* (inverso *manages*) – no caso de algum *site* ter seu grupo responsável modificado, essa mudança deve ser armazenada. Por exemplo, pode acontecer de um cliente preferir o trabalho feito por outro grupo;

- relacionamentos *worksIn* (inversos *designers* e *programmers*) – tanto os projetistas quanto os programadores pertencem a um grupo de trabalho, o que não impede de trocarem de grupo ao longo do desenvolvimento dos projetos;

- relacionamento *salary* – o empregado tem armazenado o histórico de seus níveis salariais para fins de se recuperar a evolução de sua carreira;

- relacionamento *level* – estabelece o nível do projetista de acordo com suas habilidades específicas. Esse conceito é usado para classificar o projetista conforme a complexidade de projeto que pode realizar. O armazenamento de seu histórico é necessário para fins de se ter a evolução de carreira;

- atributos *hiringDate*, *firingDate*, *vacationDate* – armazenam respectivamente as datas de contratação, demissão e início de férias do empregado. Desse modo, todo o histórico de tempo de trabalho do empregado é armazenado;

- classes *Person* e *Workgroup* – essas classes foram definidas como versionadas e temporais a fim de permitir a existência de relacionamentos e atributos temporais.

Entre as vantagens de usar o TVM para modelar essa aplicação, o projetista:

- simplifica a tarefa de trocar o padrão associado às páginas de todos os empregados, a qual pode ser realizada através da adição de um novo valor no relacionamento *pattern*;

- garante que todos os períodos de cada alternativa são armazenados;

- assegura que a evolução dos *websites* dos cliente seja armazenada juntamente com os períodos. Se um cliente desejar habilitar alguma página antiga, basta que o usuário modifique os tempos de validade da atual e da página a ser recuperada;

- garante que os históricos do tempo de trabalho e do salário dos empregados sejam armazenados.

A empresa tem várias possibilidades de usar as informações armazenadas na base de dados construída a partir desse modelo. Uma das mais importantes é a chance de descobrir padrões e perfis dos clientes através de técnicas de mineração de dados (*data mining*).

Esse exemplo usa o conceito de versionamento para ilustrar principalmente diferentes versões definidas de acordo com o tempo no qual são válidas. Essa mesma especificação pode ser usada para modelar outros aspectos. Por exemplo, o projetista pode criar diferentes versões de acordo com a tecnologia presente dentro das páginas. Então, pode-se ter versões com *HTML*, *XML*, *java script*, *asp*, *php*, *flash*, etc.

5.2.1 Linguagem de Definição

A seguir, é apresentada a definição de todas as classes do modelo exemplo conforme sintaxe definida no anexo 1. As cardinalidades dos relacionamentos de herança por extensão estão omitidas, possuindo o valor padrão (*1:1*).

```
Class WebSite hasVersions
       aggregateOf n WebPage byReference
       ( Properties:
              temporal URL : string ;
              temporal online : boolean;
         Relationships:
              belongsTo 1:1 inverse has Person;
       temporal initialPage 1:1 WebPage;
              temporal handledBy 1:1 inverse manages Workgroup;
              temporal pattern 1:1 inverse associatedWith PagePattern; ) ;
Class WebPage hasVersions
       aggregateOf n Links byReference;
       aggregateOf n MultimidiaResources byReference;
       ( Properties:
              temporal pageTitle : string;
              temporal pageText : text;                              ) ;
Class Workgroup hasVersions
       ( Properties:
              contactPhone : string;
              contactEmail : string;
         Relationships:
              temporal designers 1:n inverse worksIn Designer;
              temporal programmers 1:n inverse worksIn Programmer;
              temporal manages 0:n inverse handleBy Website;         ) ;
Class Person hasVersions
       ( Properties:
              name : string;
              id : string;
              birthdate : date;
              gender : char;
              phone : string;
              e-mail : string;
         Relationships:
              has 0:1 WebSite;
              livesAt 1:1 Address;                                   ) ;
Class Address
       ( Properties:
              street : string;
              number : integer;
              complement : string;
              zipcode : integer;
         Relationships:
              locatedIn 1:1 City;                                    ) ;
Class City
       ( Properties:
              cityName : string;
              state : string;
              country : string;                                      ) ;
Class Client hasVersions inherit Person ( );
Class Employee hasVersions inherit Person
       ( Properties:
              temporal hiringDate : date;
              temporal firingDate : date;
              temporal vacationDate : date;
              diplomas : string;
              workExperience : text;
         Relationships:
              temporal salary 1:1 SalaryLevels;                      ) ;
Class SalaryLevels
       ( Properties:
              salLevel : char;
              salValue : real;                                       ) ;
```

```
Class Designer hasVersions inherit Employee
       ( Properties:
              bestProjects : text;
         Relationships
              temporal worksIn 1:1 inverse designers Workgroup;
              temporal level 1:1 DesignLevels;                    ) ;
Class DesignLevels hasVersions
       ( Properties:
              desLevel : char;
              skills : text;                                      ) ;
Class Programmer hasVersions inherit Employee
       ( Properties:
              progrLanguages : text;
         Relationships:
              temporal worksIn 1:1 inverse programmers Workgroup;  ) ;
Class PagePattern hasVersions
       ( Properties:
              bckgrdColor : string;
              bckgrdImage : string;
              banner : string;
         Relationships:
              default 1:1 Font;
              temporal associatedWith 0:n inverse pattern Website; ) ;
Class Font
       ( Properties:
              fontType : string;
              fontSize : integer;
              fontColor : string;
              fontEffect : string;                                ) ;
Class Links
       ( Properties:
              linkTitle : string;
              linkDescription : text;
              linkType : string;
              linkURL : string;                                   ) ;
Class MultimidaResources
       ( Properties:
              mrAutor : string;
              mrDescription : string;                             ) ;
Class Picture inherit MultimidiaResources
       ( Properties:
              height : integer;
              width : integer;
              depth : string;
              picType : string;                                   ) ;
Class Audio inherit MultimidiaResources
       ( Properties:
              duration : string;
              audioType : string;                                 ) ;
```

5.3 Representação Gráfica das Instâncias

A Figura 5.3 apresenta a classe *PagePattern* com seus objetos versionados *Primavera*, *Verao*, *Outono* e *Inverno*. Além disso, o objeto versionado *Verao* é apresentado com suas versões: *Verao* (9,15,1) é o padrão de página para o verão; *Natal* (9,15,2) e *NatalB* (9,15,3) foram derivadas como alternativas para o período de Natal; *NovoAno* (9,15,4) foi derivada como versão do novo ano; e *NovoMilenio* (9,15,5) foi derivada da versão de ano novo, como alternativa para o novo milênio. Observa-se que

a mudança no valor do atributo *banner* (*natalAd* para *23DezAd*) não cria nova versão, apenas gera mais um valor para o seu histórico.

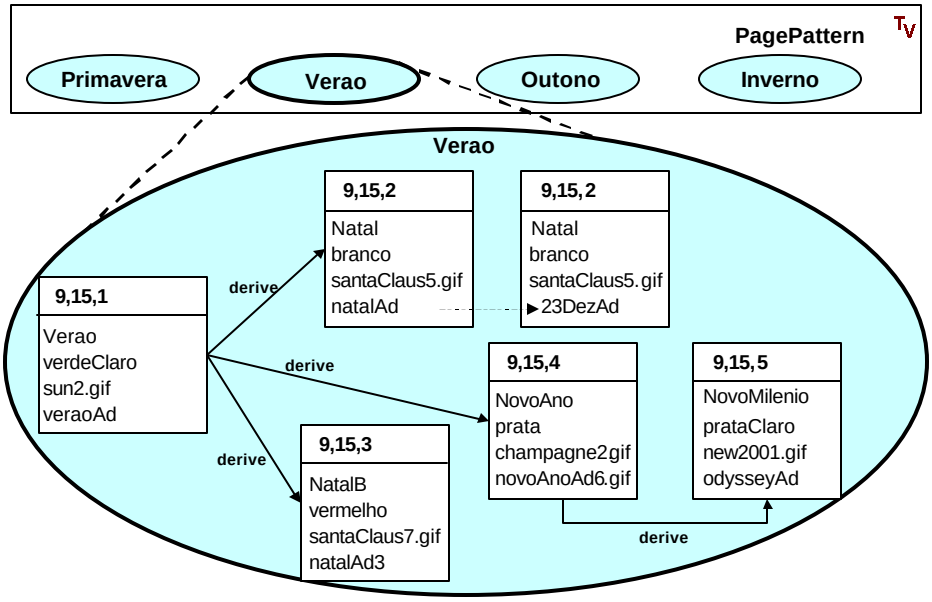


Figura 5.3 – Representação gráfica do objeto versionado *Summer*

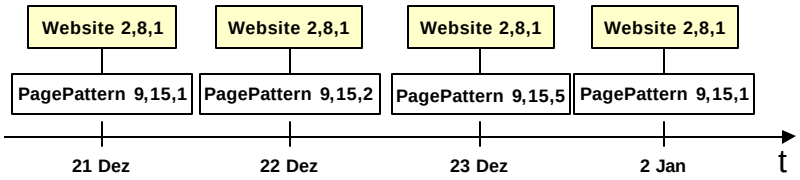


Figura 5.4 – Relacionamento temporal *pattern/associatedWith*

A Figura 5.4 ilustra a representação gráfica da evolução do relacionamento *pattern/associatedWith* (entre as classes *WebSite* e *PagePattern*) para o *website* de um dos empregados. A versão de *WebSite* é sempre a mesma (2,8,1), porém a associação com o padrão de página muda de acordo com a data comemorativa dentro do período de verão. A Tabela 5.3 apresenta os valores do rótulo temporal para o relacionamento, sendo que toda a variação foi definida pelo usuário no início do mês de dezembro.

Tabela 5.3 – Variação temporal do relacionamento *pattern/associatedWith*

PagePattern* no relacionamento *pattern	**Tempo validade inicial**	**Tempo validade final**	**Tempo transação inicial**	**Tempo transação final**
9,15,1	21/12/2000	21/12/2000	01/12/2000	null
9,15,2	22/12/2000	25/12/2000	01/12/2000	null
9,15,5	26/12/2000	01/01/2001	01/12/2000	null
9,15,1	02/01/2001	null	01/12/2000	null

A Figura 5.5 ilustra a classe *PagePattern*, o objeto versionado *Verao* com suas versões (*Verao*, *Natal*, *NatalB*, *NovoAno*, *NovoMilenio*), e as respectivas linhas de tempo ($T_{\text{objeto Verao}}$, $t_{\text{versão Verao}}$, t_{Natal}, t_{NatalB}, t_{NovoAno}, e $t_{\text{NovoMilenio}}$). São apresentados os instantes iniciais do atributo *alive* (= 'T') para o objeto versionado (T0) e cada versão (t1, t2, t3, t4, e t5), tornando claro o conceito de tempo ramificado apresentado na seção 4.1. O atributo *alive* final é mostrado para a versão *NatalB*, ou seja, essa versão foi excluída do sistema no instante *t3End* (*alive* = 'F').

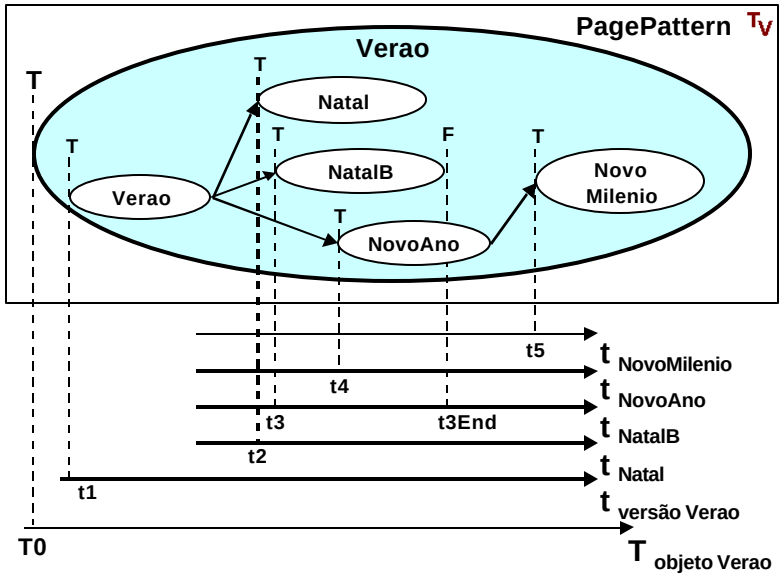


Figura 5.5 – Classe, objeto versionado, versões, e atributo *alive*

A Figura 5.6 apresenta como as classes são mapeadas para o DB2. A Figura 5.6a mostra a classe de aplicação normal *MultimidiaResources* com seus atributos. A Figura 5.6b mostra a classe temporal versionada *Person* com os atributos herdados (de *Object*, *TemporalObject* e *TemporalVersion*) juntamente com seus atributos e relacionamentos. De mesmo modo, a Figura 5.6c mostra a subclasse temporal versionada *Employee*. Uma maneira de implementar os atributos e relacionamentos temporais é criar uma tabela auxiliar para armazenar seu histórico. O valor atual é mantido na tabela principal da classe. O atributo *refSalary* na tabela *Employee* contém a referência para a tabela do histórico do relacionamento *salary*.

A Figura 5.7 ilustra como a herança por extensão é mapeada para o DB2 através do atributo *ascendant*. A alternativa apresentada é acrescentar o atributo em todas as classes, tanto normais quanto temporais versionadas. As classes raiz possuem valor *null*, e as subclasses referenciam seu(s) objeto(s) ascendente(s). A Figura 5.7a ilustra a hierarquia especificada entre as classes de aplicação temporais versionadas *Person* e *Employee* com o respectivo mapeamento na Figura 5.7b.

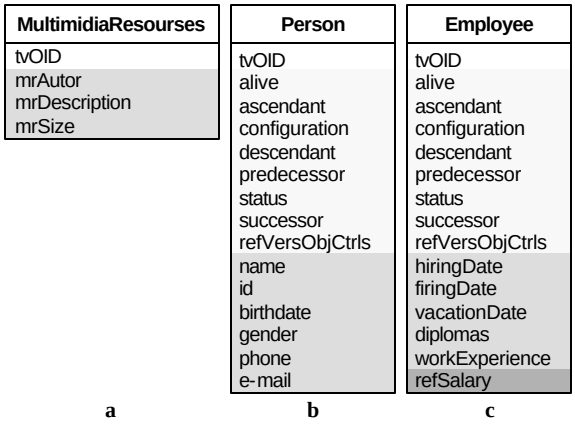


Figura 5.6 – Mapeamento das classes de aplicação para o DB2

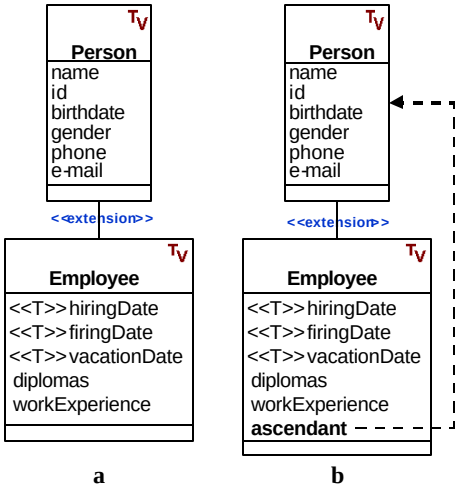


Figura 5.7 – Herança por extensão, classe temporal versionada

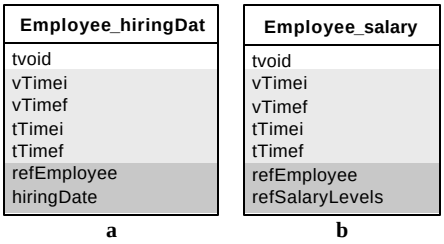


Figura 5.8 – Atributo e relacionamento temporais no DB2

A Figura 5.8 ilustra a representação do mapeamento do atributo temporal *hiringDate* (Figura 5.8a) e do relacionamento temporal *salary* (Figura 5.8b), ambos da classe *Employee*. A figura mostra uma alternativa para a nomenclatura da classe que armazena o histórico do atributo temporal (NomeClasse_NomeAtributo) e daquela que armazena o do relacionamento temporal (NomeClasse_NomeRelacionamento). Também são ilustrados os rótulos temporais e as referências para a classe que contém o atributo, e para as classes que fazem parte do relacionamento.

5.4 Considerações Finais

Esse capítulo apresentou o mapeamento da hierarquia de classes base do TVM para o banco de dados DB2, a modelagem de uma aplicação através do TVM e a representação gráfica de algumas instâncias dessa aplicação.

Quanto à implementação da hierarquia do TVM, uma vantagem do DB2 é a proximidade dos seus tipos de dados aos definidos na SQL 92. São definidos os tipos *DATE*, *TIME* e *TIMESTAMP* juntamente com funções próprias de manipulação, também semelhantes à SQL92. Mesmo não apresentando um tipo *INTERVAL*, o DB2 suporta durações como versões especializadas do tipo de dados *DECIMAL*. As restrições do DB2 são as ausências de tipos de coleção (*set*, *bag*, *list*, *array*) e tipo booleano. A ausência de tipo de coleção interfere nas características do Modelo porque limita o potencial da hierarquia por extensão e das derivações. O estudo de como simular os tipos de coleção no DB2 pode ser tratado em um trabalho futuro.

Outra vantagem é a forma como o DB2 implementa tipos estruturados, nos quais o próprio sistema oferece os métodos para acessar e atualizar os atributos (*Observer* e *Mutator*).

A modelagem da aplicação mostra como as facilidades do TVM podem ser usadas em um sistema do mundo real. As vantagens do TVM são realmente visíveis em aplicações que usam projeto. No estudo de caso, a possibilidade de possuir versões e tempo associados aos projetos das páginas, facilita o uso cíclico de padrões que variam de acordo com a época do ano. As informações armazenadas no banco de dados podem servir de base para outras técnicas como mineração de dados e descoberta de padrões e tendências nas páginas dos clientes da empresa.

Um ponto que poderia ser estudado na especificação do TVM é quanto aos relacionamentos entre classes normais e temporais versionadas. Por exemplo, na aplicação modelada, a classe *SalaryLevels* tem de ser especificada como temporal somente para poder existir o relacionamento temporal com *Person*, uma vez que não possui atributos temporais. Uma sugestão é permitir o relacionamento temporal entre classes normais e temporal versionadas com caminho de acesso apenas da temporal para a normal. Na aplicação, *Person* teria um relacionamento temporal unidirecional com *SalaryLevels*.

6 Conclusões

O Modelo Temporal de Versões é a união de um modelo de versões com informações temporais. Dessa união, três vantagens são oferecidas: (i) armazenamento de alternativas de projeto; (ii) armazenamento da história dos dados evolutivos; (iii) possibilidade de reconstruir o estado da base em qualquer data passada, sem usar operações complexas de *backup* e *recovery*. O Modelo apresenta seu diferencial nas características de relacionamento de herança por extensão, ordem temporal ramificada e por permitir classes sem tempo e versões entre as classes temporais versionadas.

O Modelo é apresentado com riqueza de detalhes que incluem: regras de integridade temporal, exclusão lógica, hierarquia de tipos temporais, diagrama de estado das versões, hierarquia de classes base com seus atributos, relacionamentos, operações e funcionamento, herança por extensão, configuração, relacionamentos entre classes normais e temporais versionadas, linguagens de definição de classes e de consulta. Também é apresentado um breve estudo que compara o TVM com outros dois modelos existentes.

É proposta a arquitetura de uma camada intermediária para o funcionamento do TVM sobre um SGBD convencional. Essa camada é responsável pelo gerenciamento das características específicas do Modelo. Um ambiente (Ambiente Temporal de Versões) é projetado para encapsular a camada e o SGBD, tornando-os transparentes ao usuário. A hierarquia base do TVM e seus metadados devem estar previamente criados no SGBD convencional para que o Ambiente funcione adequadamente. Desse modo, o mapeamento da hierarquia para um banco de dados objeto-relacional é apresentado em detalhes. A definição e o mapeamento dos metadados serão propostos em um trabalho futuro.

Entre as demais funcionalidades do Ambiente, esse trabalho detalha a Ferramenta de Apoio à Especificação de classes. Através de sua interface, o usuário especifica classes, atributos, relacionamentos e operações sem ser necessário o conhecimento prévio da linguagem de definição do TVM. A partir da especificação, a ferramenta pode gerar o *script* para criação na base de dados.

Para mostrar a viabilidade do Ambiente, o mapeamento da hierarquia é realizado sobre o banco de dados DB2. As vantagens desse SGBD são a proximidade dos tipos temporais definidos na linguagem SQL92 e a implementação de tipos estruturados com métodos próprios de acesso e atualização. As desvantagens são a ausência de tipo booleano e tipo para coleções, o qual limita o potencial da hierarquia por extensão e das derivações. O estudo de caso de uma aplicação real também é ilustrado mostrando as vantagens de modelar versões e dados temporais homogeneamente. A desvantagem é a grande quantidade de informações extra armazenadas, a qual pode ser amenizada através da escolha, pelo usuário, de quais classes e dados serão temporais.

Mesmo com a especificação do Modelo, algumas questões ou sugestões de aprimoramento continuam em aberto: construção de novos tipos literais e estruturados a partir da linguagem de definição, linguagem de manipulação de dados, regras para garantir as propriedades ACID, definição completa dos metadados, mapeamento das classes de aplicação, extensões possíveis da ferramenta de apoio à especificação de classes, novas possibilidades de relacionamento temporal entre classes normais e temporais versionadas, operações para gerar configurações e para restaurar objetos

logicamente excluídos, especificação de uma operação para criar uma nova entidade baseada em uma já existente, entre outras.

Outros trabalho estão sendo desenvolvidos com o Modelo Temporal de Versões, entre eles: extensão para o padrão ODMG, especificação do Modelo e da linguagem de consulta em XML, extensão do Modelo para o suporte à evolução de esquemas [GAL 98, ROM 00], extensão e mapeamento da linguagem de consulta para SQL, e implementação de um *extender* para o DB2 com as características do TVM.

Como sugestões para trabalhos futuros estão:

- análise comparativa do desempenho em um banco de dados padrão e outro que implementa as características do Modelo. Um estudo nesse estilo foi realizado por Jensen, em [JEN 99], mostrando o custo de manipular os tempos de transação;
- estudo de métodos de otimização de consultas;
- estudo de métodos de indexação para melhorar o desempenho;
- uso de outras ferramentas próprias para modelagem orientada a objetos, como o Rational Rose, em conjunto com o Ambiente;
- mapeamento do Modelo para um banco de dados orientado a objetos.

Como produção científica, a partir deste trabalho estão publicados os seguintes artigos:

1. Moro, Mirella Moura; Saggiorato, Silvia Maria; Edelweiss, Nina; Santos, Clesio Saraiva dos. **Adding Time to an Object-Oriented Versions Model**. In: DEXA 2001 - 12th International Conference on Database and Expert Systems Applications, September 2001, Munich, Germany, LNCS;
2. Moro, Mirella Moura; Saggiorato, Silvia Maria; Edelweiss, Nina; Santos, Clesio Saraiva dos. **Dynamic Systems Specification using Versions and Time**. In: IDEAS - International Database Engineering and Applications, July 2001, Grenoble, France. IEEE Press. p. 99-107;
3. Moro, Mirella Moura; Saggiorato, Silvia Maria; Edelweiss, Nina; Santos, Clesio Saraiva dos. **A Temporal Versions Model for Time-Evolving Systems Specification**. In: Proceedings of SEKE - 13th International Conference on Software Engineering & Knowledge Engineering, June 2001, Buenos Aires, Argentina. p.252-259;
4. Moro, Mirella Moura; Saggiorato, Silvia Maria; Edelweiss, Nina; Santos, Clesio Saraiva dos. **Um Modelo Temporal de Versões para Especificação de Aplicações**. In: Memorias IDEAS - 4th Iberoamerican Workshop on Requirements Engineering and Software Environments, April 2001, Santo Domingo, Costa Rica. p.336-347.

Anexo 1 – BNF da Linguagem de Definição de Classes

A BNF da linguagem de definição de classes é apresentada nesse anexo conforme a seguinte notação:

- [ ] para itens opcionais;
- { } para itens opcionais repetitivos (zero ou mais vezes);
- { }+ para itens repetitivos (uma ou mais vezes);
- ( ) para conjunto de opções
- símbolos terminais e não-terminais são diferenciados pela apresentação em negrito dos terminais.

Alguns aspectos específicos encontrados em linguagens de programação orientada a objetos não estão incorporados à BNF da linguagem. Por exemplo, a possibilidade da classe implementar uma interface não é considerado.

As informações de classes sem tempo e versões são diferenciadas das informações de classes temporal versionadas. As informações relativas a classes temporal versionadas têm o prefixo **`temp`**, as de classes normais não têm prefixo, e as de classes que podem ser de ambos os tipos têm o prefixo **`both`**.

```
class ::=   [ public ][ abstract | final ] ( normalClass | tempVersionClass )
normalClass ::= class className
     [ inherit ( byExtension className | superClassName ) ] classDefinition ;
tempVersionClass ::= class tempClassName hasVersions
                    [ inherit ( ascendant | tempAscendant) ]
     tempClassDefinition ;
className ::= identifier
superClassName ::= identifier
ascendant ::= className [correspondence (1:1 | n:1)]
tempAscendant ::= tempClassName [correspondence (1:1 | 1:n | n:1 | n:m)
tempClassName ::= identifier
classDefinition ::= [ aggregationList ]
            ( [ Properties: attributeList ]
            [ Relationships: relationshipList ]
            [ Operations: operationDefinitions ] )
tempClassDefinition ::= [ aggregationList | tempAggregationList ]
            ( [ Properties: tempAttributeList ]
            [ Relationships: tempRelationshipList ]
            [ Operations: operationDefinitions ] )
aggregationList ::= aggregateOf [n] className aggregationType
                  {, aggregateOf [n] className aggregationType }
attributeList ::= { [ visibility ] [ static ]
                  attributeName : attributeDomain [ default value ] ; }+
relationshipList ::= { relationshipName cardinality
                 [ inverse inverseRelationshipName ] relatedClassName; }+
visibility ::= public | private | protected
operationDefinitions ::= [ visibility ] [ static ] [ abstract | final ]
     operationName ( [ parameterList ] )  [ : returnedValueDomain ]
     { ; [ visibility ] [ static ] [ abstract | final ]
       operationName ( [ parameterList ] ) [ : returnedValueDomain ] }
tempAggregationList ::=
     [temporal] aggregateOf [n] className aggregationType
```

```
      {,[temporal] aggregateOf [n] className aggregationType }
  tempAttributeList ::=
      { ( [visibility] [static]
                  attributeName : attributeDomain [ default value ] ; )
      | ( [temporal] [visibility] [static]
                  attributeName : className [ default value ] ; ) }+
  tempRelationshipList ::=
      { [temporal] relationshipName cardinality
                  [ inverse inverseRelationshipName ] relatedClassName; }+
  aggregationType ::= byValue | byReference
  attributeName ::=  identifier
  attributeDomain ::= domain
  relationshipName ::= identifier
  cardinality ::= 0:1 | 0:n | 1:1 | 1:n | n:m
  relatedClassName ::= identifier
  operationName ::= identifier
  parameterList ::= parameterName : parameterDomain [default value]
              { ; parameterName : parameterDomain  [default value] }
  returnedValueDomain ::= domain
  identifier ::= letter | _ | identifier { letter | digit | _ }
  domain ::= integer | real | string | char | boolean | date | time
            | instant | OIDt | set | className
  parameterName ::= identifier
  parameterDomain ::= domain
  value ::= integerNumber | realNumber | stringVal | charVal | booleanVal |
instantVal | dateVal | hourVal | OIDval | null
  letter ::= A | B | C | D | E | F | G | H | I | J | K | L | M | N | O | P | Q
      | R | S | T | U | V | W | X | Y | Z | a | b| c | d | e | f | g | h | i
      | j | k | l | m | n | o | p | q | r | s | t | u | v | w | x | y | z
  digit ::=  0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
  set ::= set ( domain )
  integerNumber ::= digit { digit }
  realNumber ::= digit { digit } , digit { digit }
  stringVal ::= “ { any character including blanck }  ”
  charVal ::= ‘letter’ | ‘digit’
  booleanVal ::= true | false
  instantVal ::= dateVal , hourVal
  OIDval ::= pointer
  dateVal ::= number / number / number number
  hourVal ::= number : number
  pointer ::= { any character including blanck }
  number ::= digit digit
```

Anexo 2 – Definição das Classes da Hierarquia

Diferenciando entre as funções das visibilidades, a definição de classes do modelo aqui proposta segue o estabelecido pela linguagem C++ (pois essa foi a base do uso-padrão da UML [FOW 00]):

- **public** é visível em qualquer lugar na aplicação e pode ser acessado ou chamado por qualquer objeto no sistema;
- **protected** pode ser usado (ou alterado) somente pela classe que o define ou uma subclasse dessa;
- **private** pode ser usado (ou alterado) somente pela classe que o define.

Os atributos das classes da hierarquia são definidos por omissão como particulares (*private*), caso contrário a visibilidade será definida explicitamente.

```
public abstract class Object (
   Properties:
     tvOID: OIDt;
   Relationships:
     nickname 0:n inverse nicknameOf Name;
   Operations:
     Object ();
     Object (ascendId: OIDt);
     Object (entityName: string);
     Object (entityId: integer, classId: integer, versionId: integer);
     delete (allReferences: boolean);
     deleteObjectTree (allReferences: boolean);
     private findVersion (entityId: integer, classId: integer): integer;
     getAscendant (): set(OIDt);
     getAscendant (className: string): OIDt;
     getClassId (): integer;
     getClassName (): string;
     getCompleteObject (): set(Object);
     getCorrespondence (className: string): string;
     getCorrespondenceAsc (className: string): string;
     getCorrespondenceDesc (className: string): string;
     getDescendant (): set(OIDt);
     getDescendant (className: string): OIDt;
     getEntityId (entityName: string): integer;
     getNickname (): set(string);
     getObject (): Object;
     getTvOid (): OIDt;
     protected isDeleteAllowed (allReferences: boolean): boolean;
     protected isDeleteTreeAllowed (allReferences: boolean): boolean;
     private verifyAscendId (ascendId: OIDt): boolean;
     private verifyEntityName (entName: string): boolean
);

public abstract class TemporalObject inherit Object (
   Properties:
     temporal alive: boolean default true;
   Operations:
     TemporalObject ();
     TemporalObject (ascendID: OIDt);
     TemporalObject (entityName: string);
     TemporalObject (entityId: integer, classId: integer, versionId: integer);
     private closeTemporalLabels ();
```

```
     delete ();
     getAlive (): boolean;
     final getAttributeHistory (atribName: string): set (InstantAttribute);
     final getAttributeValueAt (atribName: string, i: instant): InstantAttribute;
     getLifetimeI (): instant;
     getLifetimeF (): instant;
     final getObjectHistory (): set (InstantAttribute);
     final getRelationshipHistory (relatedObjId: OIDt, relatName: string): set
(InstantRelationship);
     final getRelationshipHistoryAt (relatedObjId: OIDt, relatName: string, i:
instant): InstantRelationship;
     final setTemporalAttribute (attribName: string, newValue: Object);
     final setTemporalRelationship (relatName: string, newO1: OIDt, newO2: OIDt)
);

  public final class VersionedObjectControl inherit TemporalObject (
    Properties:
     temporal configurationCount: integer default 0;
     temporal currentVersion: OIDt;
     temporal firstVersion: OIDt;
     temporal lastVersion: OIDt;
     nextVersionNumber: integer default 3;
     temporal userCurrentFlag: boolean default false;
     temporal versionCount: integer default 2;
    Operations:
     VersionedObjectControl (entityId: integer, classId: integer, configCount: integer,
currentV: OIDt, firstV: OIDt, lastV: OIDt, nextVNumber: integer, userCurrentFlag:
boolean, vCount: integer);
     VersionedObjectControl (entityId: integer, classId: integer, currentV: OIDt,
firstV: OIDt, lastV: OIDt);
     delete (entityId: integer, classId: integer);
     getConfigurationCount (): integer;
     getCurrentVersion (): OIDt;
     getFirstVersion (): OIDt;
     getLastVersion (): OIDt;
     getNextVersionNumber (): integer;
     getUserCurrentFlag (): boolean;
     getVersionCount (): integer;
     final restore ();
     setCurrentVersion ();
     setCurrentVersion (newVersionId: OIDt);
     setFirstVersion (first: OIDt);
     setLastVersion (last: OIDt);
     setUserCurrentFlag (flag: boolean);
     updateConfigurationCount ();
     updateVersionCount ();
     updateNextVersionNumber ()
  );

  public class TemporalVersion inherit TemporalObject (
    Properties:
     temporal ascendant: set (OIDt) default NULL;
     configuration: boolean default false;
     temporal descendant: set (OIDt) default NULL;
     predecessor: set (OIDt) default NULL;
     temporal status: char default 'W';
     temporal successor: set (OIDt) default NULL;
    Operations:
```

```
     TemporalVersion ();
     TemporalVersion (ascendId: set(OIDt));
     TemporalVersion (entityName: string);
     TemporalVersion (entityId: integer, classId: integer, versionId: integer);
     private TemporalVersion (predecId: set(OIDt), ascendId: set(OIDt), config:
boolean);
     private addAscendant (ascendId: OIDt);
     private addDescendant (descendId: OIDt);
     private addSuccessor (succId: OIDt);
     delete (allReferences: boolean);
     deleteObjectTree (allReferences: boolean);
     final derive (versionId: set (OIDt));
     final derive (versionId: set (OIDt), ascendId: set(OIDt), config: boolean);
     getAscendant (): set(OIDt);
     getAscendant (className: string): set(OIDt);
     getConfiguration (): OIDt;
     getCompleteObject(): set(OIDt);
     getDescendant (): set(OIDt);
     getDescendant (className: string): set(OIDt);
     final getOIDControl (): OIDt;
     getPredecessor (): set (OIDt);
     getStatus (): char ;
     getSuccessor (onlyConfigured: boolean): set (OIDt);
     getVersionedObjectId (): OIDt;
     isConfiguration (): boolean
     private isDeleteAllowed (allReferences: boolean): boolean;
     private isDeleteTreeAllowed (allReferences: boolean): boolean;
     final promote (allAscendant: boolean, allReferenced: boolean);
     private removeAscendant (ascendId: OIDt);
     private removeDescendant (descendId: OIDt);
     private removeSuccessor (succId: OIDt);
     final restore (OID: OIDt) : boolean;
     private setAscendant (ascendId: OIDt);
     private setDescendant (descendId: OIDt);
     private setStatus (newStatus: char);
     private setSuccessor (succId: OIDt);
     private verifyAscendId (ascendId: set(OIDt));
);

  public class TemporalLabel (
    Properties:
     tTimei: instant;
     tTimef: instant;
     vTimei: instant;
     vTimef: instant;
    Operations:
     TemporalLabel (iValidTime: instant, fValidTime: instant, iTransTime: instant,
fTransTime: instant);
     getTTimei (): instant;
     getTTimef (): instant;
     getVTimei (): instant;
     getVTimei (): instant;
     setTTimef (i: instant);
     setVTimef (i: instant)
);
```

```
 public class InstantAttribute (
   Properties:
    value: Object;
    tempLabel : TemporalLabel;
   Operations:
    InstantAttribute (val: Object);
    InstantAttribute (val: Object, iValidTime: instant, fValidTime: instant);
    getValue (): Object;
    getTempLabel (): TemporalLabel;
    getTempLabelOf (val: Object): set (TemporalLabel);
    setFTransactionTime (i: instant);
    setFValidTime (i: instant)
);

 public class InstantRelationship (
   Properties:
    obj1: OIDt;
    obj2: OIDt;
    tempLabel : TemporalLabel;
   Operations:
    InstantRelationship (o1: OIDt, o2: OIDt);
    InstantRelationship (o1: OIDt, o2: OIDt, iValidTime: instant, fValidTime:
instant);
    getObj1 (): OIDt;
    getObj2 (): OIDt;
    getTempLabel (): TemporalLabel;
    getTempLabelOf (o1: OIDt, o2: OIDt): set (TemporalLabel);
    setFTransactionTime (i: instant);
    setFValidTime (i: instant)
);

 public class TemporalAttribute
      aggregateOf n InstantAttribute byReference (
   Properties:
    type: string;
   Relationships:
    currentAttr 1:1 InstantAttribute;
   Operations:
    TemporalAttribute (typ: string, val: Object);
    TemporalAttribute (typ: string, val: Object, iValidTime: instant, fValidTime:
instant);
    close ();
    getCurrent (): InstantAttribute;
    getHistory (): set (InstantAttribute);
    getHistoryOfValue (val: Object): set (TemporalLabel);
    getValueAt (at : instant): InstantAttribute;
    getType (): string;
    private setFTransactionTime (validAt: instant, fTransTime: instant);
    private setFValidTime (validAt: instant, fValidTime: instant);
    updateValue (at: instant, newVal: Object, iValidTime: instant, fValidTime:
instant)
);

 public class TemporalRelationship
 aggregateOf n InstantRelationship byReference (
   Relationships:
    currentRel 1:1 InstantRelationship;
   Operations:
```

```
     TemporalRelationship (o1: OIDt, o2: OIDt);
     TemporalRelationship (o1: OIDt, o2: OIDt, iValidTime: instant, fValidTime:
instant);
     close ();
     getCurrent ():InstantRelationship;
     getHistory (): set (InstantRelationship);
     getHistoryOfValue (o1: OIDt, o2: OIDt): set (TemporalLabel);
     getValueAt (at : instant): InstantRelationship;
     private setFTransactionTime (validAt: instant, fTransTime: instant);
     private setFValidTime (validAt: instant, fValidTime: instant);
     updateValue (at: instant, newO1: OIDt, newO2: OIDt, iValidTime: instant,
fValidTime: instant)
);

  public class OIDt (
    Properties:
     value: string;
    Operations:
     OIDt (E: integer, C: integer, V: integer);
     getClassNr (): integer;
     getEntityNr (): integer;
     getOID (): string;
     getVersionNr (): integer
);

  public class Name (
    Properties:
     value: string;
    Relationships:
     nicknameOf 1:1 inverse nickname Object;
    Operations:
     Name (nam: string);
     getClassName (): string;
     getName (): string
);
```

Anexo 3 – Sintaxe dos Principais Comandos do DB2

Esta seção apresenta apenas as sintaxes resumidas dos principais comandos executados durante a implementação no DB2. Maiores detalhes sobre os comandos apresentados e os demais podem ser encontrados em [CHA 98].

A BNF dos comandos é apresentada neste anexo conforme a seguinte notação:

- [ ] para itens opcionais;
- { } para itens opcionais repetitivos (zero ou mais vezes);
- { }+ para itens repetitivos (uma ou mais vezes);
- ( ) para conjunto de opções;
- palavras-chave terminais em maiúsculo, e não-terminais começando por minúsculo;
- símbolos terminais em negrito.

A. Tipos

```
CREATE TYPE typeName [ UNDER superTypeName ]
[ AS ( attributeDefinition { , attributeDefinition } ) ]
[ [ NOT ] INSTANTIABLE ]
[ INLINE LENGTH integer ]
[ WITHOUT COMPARISON ]
[ NOT FINAL ]
MODE DB2SQL
[ WITH FUNCTION ACCESS ]
[ REF USING refType ]
[ CAST ( SOURCE AS REF ) WITH functionName1 ]
[ CAST ( SOURCE AS REF ) WITH functionName2 ]
[ methodSpecification { , methodSpecification } ]
```

Observação: `mode DB2SQL` tem a função de proteger as aplicações do UDB versão 5.2 das mudanças ocasionadas pela adição de conceitos de orientação a objetos, podendo ser ignorado.

A.1. Definição de atributos (`attributeDefinition`)

```
attributeName [ dataType ]
[ ( lobOptions | datalinkOption ) ]
```

A.2. Especificação de métodos (`methodSpecification`)

```
METHOD methodName
( [ parameterName ] datatype1 [ AS LOCATOR ]
  { , [ parameterName ] datatype1 [ AS LOCATOR ] } )
RETURNS
( datatype2 [ AS LOCATOR ]
 | datatype3 CAST FROM datatype4 [ AS LOCATOR ] )
[ SPECIFIC specificName ]
[ SELF AS RESULT ]
[ ( SQLroutineCharacteristics
   | externalRoutineCharacteristics ) ]
```

B. Tabelas

```
CREATE [ SUMMARY ] TABLE tableName
( ( { columnDefinitions | tableConstraints } )
 | OF typeName [ typedTableOptions ]
 | summaryTableDefinitions
 | LIKE ( tableName1 | viewName | nickname ) [ copyOptions ]
)
[ IN tableSpaceName
     [ INDEX IN tableSpaceName] [ LOG IN tableSpaceName ] ]
```

```
[ ( DATA CAPTURE NONE | DATA CAPTURE CHANGES ) ]
[ NOT LOGGED INITIALLY ]
[ partioningKeyDefinition ]
```

B.1. Colunas (`columnDefinitions`)

```
columnName [ dataType ]
[ NOT NULL ]
[ lobOptions ]
[ datalinkOptions ]
[ SCOPE ( typedTableName | typedViewName ) ]
{  [ CONSTRAINT constraintName ]
   [ referencesClause ]
   [ CHECK checkConditions ]
   [ PRIMARY KEY | UNIQUE ]       }
[ columnDefaultSpecification ]
[ INLINE LENGTH integer ]
```

B.2. Opções de tabelas tipadas (`TypedTableOptions`)

```
[ ( HIERARCHY hierarchyName
  | UNDER subtableName INHERIT SELECT PRIVILEGES ) ]
[ ( typedElementList ) ]

typedElementList ::=
     ( OIDcolumnDefinition | withOptions
      | uniqueConstraints | checkConstraints )
      { , ( OIDcolumnDefinition | withOptions
      | uniqueConstraints | checkConstraints ) }
```

C. Tipos definidos pelo usuário

```
CREATE DISTINCT TYPE typeName
AS sourceDataTypeName [ WITH COMPARISONS ];
```

D. Funções definidas pelo usuário

Podem ser de três tipos: funções *sourced*, funções escalares externa, e funções de tabela externas. A sintaxe apresentada é a do tipo mais simples (*sourced*) que usa como base uma outra função pré-estabelecida.

```
CREATE FUNCTION functionName
( parameterDataTypes ) RETURNS datatype [ SPECIFIC specificName]
SOURCE ( functionName
       | SPECIFIC specificName
       | funcionName ( [parameterDataTypes] ) )  ;
```

E. Procedimentos armazenados (*stored procedures*)

De acordo com as convenções do UDB, um procedimento armazenado possui quatro parâmetros, mas usa realmente somente dois deles, que são ponteiros para as estruturas SQLDA e SQLCA. A declaração do procedimento no servidor deve ser como segue.

```
SQL_API_RC SQL_API_FN procName (
void              dummy1,       /* not used */
void              dummy2 ,      /* not used */
struct sqlda      *exchange_da,/* in / out */
struct sqlca      *out_sqlca,   /* out */
)
{ commands } ;
```

Depois, é necessário registrá-lo na base de dados:

```
CREATE PROCEDURE procName
( { [ ( IN | OUT | INOUT ) ] paramName datatype }+ )
[ SPECIFIC specificName ]
[ ( DYNAMIC RESULT SETS 0 | DYNAMIC RESULT SETS integer ) ]
[ ( MODIFIES SQL DATA | NO SQL | CONTAINS SQL | READS SQL DATA)]
```

```
[ [ NOT ] DETERMINISTIC  ]
[ CALLED ON NULL INPUT ]

[ ( LANGUAGE
      ( C | JAVA | COBOL | OLE ) [ externalProcedureOptions ]
    | LANGUAGE SQL [ SQLprocedureBody ] )
]
```

O cliente chama o procedimento da seguinte forma:

```
CALL ( procName
[ ( (  hostVariables )
  | USING DESCRIPTOR [ hostVariables ] ) ]
```

F. Gatilhos (*triggers*)

```
CREATE TRIGGER triggerName
( NO CASCADE BEFORE | AFTER )
( INSERT | DELETE | UPDATE [ OF columnNames ] ) ON tableName
[ REFERENCING
     { ( OLD | NEW | OLD_TABLE | NEW_TABLE )
           [ AS ] transitionVariable  }+ ]
( FOR EACH STATEMENT | FOR EACH ROW ) MODE DB2SQL
[ WHEN ( triggerCondition ) ]
( triggeredSqlStatement
 | BEGIN ATOMIC
     { triggeredSqlStatement; }+
   END )
```

Referências Bibliográficas

[AGR 91] AGRAWAL, R.; BUROFF, S.; GEHANI, N.; SHASHA, D. Object versioning in Ode. In: INTERNATIONAL CONFERENCE ON DATA ENGINEERING, 7., 1991, Tokyo. **Proceedings**... Tokio, Japan, 1991. p. 446-455.

[ANT 97] ANTUNES, D; HEUSER, C; EDELWEISS, N. TempER: uma abordagem para modelagem temporal de banco de dados. **Revista de Informática Teórica e Aplicada**. Porto Alegre: Instituto de Informática, Universidade Federal do Rio Grande do Sul. v. IV, n.1, 1997. p. 49-85.

[ARR 94] ARRUDA, E.H.P. **Um Modelo de Implementação da Linguagem TF-ORM para o SGBD O2**. Porto Alegre: CPGCC da UFRGS, 1994. (TI - 408).

[BEE 88] BEECH, D.; MAHBOD, B. Generalized Version Control in an Object-Oriented Database. In: IEEE INTERNATIONAL CONFERENCE ON DATA ENGINEERING, 4., 1988, Los Angeles. **Proceedings**... Los Angeles, USA, 1988. p.14-22.

[BER 00] BERGMAN, R; TSOUNIS, C. **DB2 Universal Database Version 7 Features and Facilities**. IBM Data Management Solutions White Paper, April, 2000. Disponível em http://www-4.ibm.com/software/data/pubs/papers/. Acesso em 22/02/2001.

[BIE 98] BIELIKOVÁ, M.; NAVRÁT, P. Modelling Versioned Hypertext Documents. In: B. MAGNUSSON, ed. *System Configuration Management*, ECOOP'98, SCM-8 Symposium, 1996, Bruxelas. **Proceedings…** Dordrecht: Springer-Verlag, 1996. p. 188-197.

[BJÖ 89] BJÖRNERSTEDT, A.; HULTÉN, C. Version control in an object-oriented architecture. In: KIM, W.; LOCHOVSKY, F.H. (eds.) **Object-Oriented Concepts, Databases, and Applications**. New York: ACM Press, 1989, p. 451-485.

[BRA 94] BRAYNER, Â.R.A.; MEDEIROS, C.B. Incorporação do tempo em um SGBD orientado a objetos. In: SIMPÓSIO BRASILEIRO DE BANCOS DE DADOS, 9.,1994, São Carlos, SP. **Anais**... São Carlos: [s.n.], 1994.

[CAV 95] CAVALCANTI, J.M.B et al. Uma abordagem para a implementação de um modelo temporal orientado a objetos usando SGBDs relacionais. In: SIMPÓSIO BRASILEIRO DE BANCOS DE DADOS, 10.,1995, Recife. **Anais**... Recife: UFPE, 1995.

[CHA 98] CHAMBERLIN, D.D. **Using the new DB2 – IBM's Object-Relational Database System**. San Francisco: Morgan Kaufmann Publishers, 1998. 682p.

[CHO 86] CHOU, H. T.; KIM, W. A Unifying Framework for Version Control in a CAD Environment. In: INTERNATIONAL CONFERENCE ON MANAGEMENT OF DATA, 12., 1986, Kyoto. **Proceedings**... Kyoto, Japan, 1986. p. 336-344.

[CLI 87] CLIFFORD, J.; CROCKER, A. The historical relational data model (HRDM) and algebra based on lifespans. **IEEE Transactions on Software Engineering**, New York, v.14, n.7, July. 1987.

[CON 98] CONRADI, R.; WESTFECHTEL, B. Version Models for Software Configuration Management. **ACM COMPUTING SURVEYS**, v. 30, n. 2, p. 232-282, June 1998 .

[DAD 84] DADAM, P.; LUM, V.; WERNER, H. Integration of Time Versions into a Relational Database System. In: CONFERENCE ON VERY LARGE DATA BASES – VLDB, 10., 1984, Singapore. **Proceedings**... Singapore, 1984. p. 509-522.

[DAT 96] DATTOLO, A.; LOIA, V. Collaborative version control in an agent-based hypertext environment. **Information Systems**, v. 21, n. 2, p. 127-145, April 1996.

[EDE 94] EDELWEISS, N; OLIVEIRA, J. P. M. de. **Modelagem de Aspectos Temporais de Sistemas de Informação**. In: IX ESCOLA DE COMPUTAÇÃO, julho 1994, Recife-PE.

[ELM 93] ELMASRI, R.; WUU, G.T.J.; KOURAMAJIAN, V. A temporal model and query language for EER databases. In: TANSEL, A. et al (eds.) **Temporal Databases: Theory, Design, and Implementation**. Redwood City: The Benjamin/Cummings Publishing Company, Inc., 1993. chap. 9, p. 212-229.

[ELM 00] ELMASRI, R; NAVATHE, S.B. **Fundamentals of Database Systems**. 3ed. Redwood City: Addison-Wesley Longman, 2000. 955 p.

[FOW 00] FOWLER, M.; SCOTT, K. **UML Essencial – um breve guia para a linguagem-padrão de modelagem a objetos**. Porto Alegre: Bookman, 2000. 169p.

[GAD 88] GADIA, S.K.; YEUNG, C.S. A Generalized model for a relational temporal database. In: ACM SIGMOD INTERNATIONAL CONFERENCE ON MANAGEMENT OF DATA, Chicago, 1988. **Proceedings**... Chicago, USA, 1988. p.251-259.

[GAL 98] GALANTE, R. de M.; SANTOS, C. S. dos; RUIZ, D.D.A. Um Modelo de Evolução de Esquemas Conceituais para Bancos de Dados Orientados a Objetos com o Emprego de Versões. In: SIMPÓSIO BRASILEIRO DE BANCO DE DADOS, 13., Maringá-PR. **Anais**... Maringá: UEM-DIN, 1998. p. 303-318.

[GEL 01] GELATTI, P.; GOMES, C. H. P.; MORO, M. M.; ROSSETI, L. L. F.; ZAUPA, A. P. **Linguagem de Consulta para o Modelo Temporal de Versões**. Porto Alegre: PGCC da UFRGS, 2001. Relatório de Pesquisa. R.P. 308. (*a ser publicado*)

[GOL 95] GOLENDZINER, L. **Um Modelo de Versões para Banco de Dados Orientados a Objetos**. Porto Alegre: CPGCC da UFRGS, 1995. Tese de Doutorado.

[HAA 92] HAAKE, A. CoVer: A contextual version server for hypertext applications. In: EUROPEAN CONFERENCE ON HYPERTEXT TECHNOLOGY - ECHT, 1992, Milano. **Proceedings**... Milano, Italia, 1992. p.43-52.

[HIC 98] HICKS, D.L.; LEGGETT, J.J.; NÜRNBERG, P.J.; SCHNASE, J.L. A Hypermedia Version Control Framework. **ACM Transactions on Information Systems**. v. 16, n. 2, April. 1998. p. 127-160.

[HÜB 99] HÜBLER, P.N.; EDELWEISS, N.; CARVALHO, T.P. Implementação de um banco de dados temporal utilizando um SGBD convencional. In: XXV CONFERENCIA LATINOAMERICANA DE INFORMATICA - CLEI'99, Assuncion, 1999. **Anais**… Assuncion, Paraguay, 1999. p.99-110.

[HÜB 00] HÜBLER, P.N. **Definição de um Gerenciador para o Modelo de Dados Temporal TF-ORM**. Porto Alegre: PGCC da UFRGS, 2000. Dissertação de Mestrado.

[JEN 98] JENSEN, C.S. et al. The Consensus Glossary of Temporal Database Concepts - February 1998 Version. In: ETZION, O; JAJODIA, S; SRIPADA, S (eds.) **Temporal Databases Research and Practice**. Berlin-Heidelberg: Springer-Verlag, 1998. p. 367-405.

[JEN 99] JENSEN, C.S. **Temporal Database Management**. Aalborg: Department of Computer Science, Aalborg University, 1999. Dr. technical thesis. Disponível em http://www.cs.auc.dk/~csj/Thesis/. Acesso em 13/03/2001.

[KAF 92] KÄFER, W.; SCHÖNING, H. Realizing a temporal complex-object data model. In: ACM SIGMOD INTERNATIONAL CONFERENCE ON MANAGEMENT OF DATA, June, 1992, San Diego. **Proceedings**... San Diego, USA, 1992. p.266-275.

[KAK 96] KAKOUDAKIS, I.; THEODOULIDIS, B. **The Tau Temporal Object Model**. Timelab Technical Report, Departament of Computation, UMIST, UK. Oct. 1996. Disponível em http://www.crim.org.uk/download/ Files/TechReps/ TR9604.pdf. Acesso em 05/08/2000.

[KIM 89] KIM, W.; BERTINO, E.; GARZA, J. F. Composite objects revisted. In: ACM SIGMOD INTERNATIONAL CONFERENCE ON MANAGEMENT OF DATA, 1989, Oregon. **Proceedings**... New York: ACM Press, 1989. p.337-347.

[LOR 88] LORENTZOS, N. A.; JOHNSON, R. G. Extending relational algebra to manipulate temporal data. **Information Systems**, v.13, n.3, p.289-296, 1988.

[LOU 91] LOUCOPOULOS, P.; THEODOULIDIS, C.; WANGLER, B. The entity relationship time model and conceptual rule language. In: INTERNACIONAL CONFERENCE ON THE ENTITY RELATIONSHIP APPROACH, 10., 1991, San Mateo. **Proceedings**... [S.l.: s.n.], 1991.

[MOE 94] MOELLER, H. Versioning structured technical documentation. In: Workshop in Hypertext Systems, ACM EUROPEAN CONFERENCE ON HYPERMEDIA TECHNOLOGY, Edinburgh, 1994. **Proceedings**… Disponível em http://cs-people.bu.edu/dgd/workshop/moeller.html. Acesso em 22/02/2001.

[MOR 00] MORO, M.M. **Estudo da união de um modelo de versões orientado a objetos e conceitos temporais**. Porto Alegre: PPGC da UFRGS, 2000. Trabalho Individual – TI. 953.

[NAV 89] NAVATHE, S.B.; AHMED, R. A Temporal Relational Model and a Query Language. **Information Sciences**, v. 47, n. 2. p. 147-175, March. 1989.

[NOR 98] NORONHA, M.A.; GOLENDZINER, L.G.; SANTOS, C.S. dos. Extending a Structured Document Model with Version Control. In: INTERNATIONAL DATABASE ENGINEERING & APPLICATIONS SYMPOSIUM, IEEE Computer Society. Cardiff, 1998. **Proceedings**... Cardiff, UK, 1998. p. 234-242.

[OMG 00] OMG, Object Management Group. **OMG Unified Modeling Language Specification**. Version 1.3. First Edition: March 2000. Disponível em ftp://ftp.omg.org/pub/docs/formal/00-03-01.pdf. Acesso em 08/05/2001.

[OST 92] OSTERBYE, K. Structural and cognitive problems in providing version control for hypertext. In: EUROPEAN CONFERENCE ON HYPERTEXT - ECHT, 1992, Milano. **Proceedings**... Milano, Italia, 1992. p. 33-42.

[ROD 99] RODRÍGUEZ, L; OGATA, H.; YANO, Y. TVOO: A Temporal Versioned Object-Oriented data model. In: **Information Sciences**, Elsevier Science Inc, 114 (1999), p. 281-300.

[ROM 00] ROMA, A.B. da S.; GALANTE, R. de M.; SANTOS, C.S. dos. Operações Primitivas e Complexas na Evolução de Esquemas em Bancos de Dados Orientados a Objetos. In: WORKSHOP IBERO-AMERICANO DE ENGENHARIA DE REQUISITOS E AMBIENTES DE SOFTWARE, 3., 2000, Cancun, México. **Proceedings**... [S.l.:s.n.], 2000.

[ROS 91] ROSE, E.; SEGEV, A. TOODM: A Temporal object-oriented data model with temporal constraints. In: INTERNATIONAL CONFERENCE ON THE ENTITY RELATIONSHIP APPROACH, 10., 1991, San Mateo. **Proceedings**... San Mateo, USA, 1991. p. 205-229.

[SAG 99] SAGGIORATO, S. **Mapeamento de Esquemas Orientado a Objetos com Versões para Esquemas Objeto-Relacionais**. Porto Alegre: PPGC da UFRGS, 2000. Dissertação de Mestrado.

[SAN 99] SANTOS, C.P.; SANTOS, C.S dos. Configuration of Versioned Documents. In: INTERNATIONAL CONFERENCE OF THE CHILEAN COMPUTER SCIENCE SOCIETY. Talca, 1999. **Proceedings**… Talca, Chile, 1999.

[SAR 90] SARDA, N.L. Extensions to SQL for historical databases. **IEEE Transaction on Knowledge and Data Engineering**, v.2, n.2, p.220-230, June. 1990.

[SIL 99] SILBERSCHATZ, A.; KORTH, H.F; SUDARSHAN, S. **Sistemas de Banco de Dados**. 3ed. São Paulo: Makron Books, 1999. 778 p.

[SIM 98] SIMONETTO, Eugênio de Oliveira**. Uma Proposta para a Incorporação de Aspectos Temporais, no Projeto Lógico de Bancos de Dados, em SGBDs Relacionais**. Porto Alegre: Pontifícia Universidade Católica do Rio Grande do Sul. 1998. Dissertação de Mestrado.

[SNO 87] SNODGRASS, R. The Temporal query language Tquel. **ACM Transactions on Database Systems**, v.12, n.2, p. 247-298. June. 1987.

[SNO 95] SNODGRASS, R. Temporal Object-Oriented Databases: A Critical Comparison. In: **Modern Database Systems – The Objetct Model,**

Interoperability, and Beyond. New York: ACM Press, 1995. chap. 19. p.386-408.

[SNO 00] SNODGRASS, R.T. **Developing Time-Oriented Database Applications in SQL**. San Francisco: Morgan Kaufmann, 2000.

[SOA 95] SOARES, L.F.G.; RODRIGUEZ, N.L.R.; CASANOVA, M.ª Nested composite nodes and version control in an open hypermedia system. **Information Systems**, v. 20, n. 6, p. 501-591, 1995.

[SU 91] SU, S. Y. W.; CHEN, H. H. M. A Temporal knowledge representation model OSAM*/T and its query language OQL/T. In: INTERNATIONAL CONFERENCE ON VERY LARGE DATA BASES, 17., 1991, Barcelona. **Proceedings**… Barcelona: VLDB, 1991. p.431-442.

[TAL 93] TALENS, G.; OUSSALAH, C. Versions of Simple and Composite Objects. In: CONFERENCE ON VERY LARGE DATA BASES - VLDB, 19., 1993, Dublin. **Proceedings**... Dublin, Ireland, 1993. p. 62-72.

[TAN 86] TANSEL, A. U. Adding time dimension to relational model and extending relational algebra. **Information Systems**, v.11, n.4, p.343-355, 1986.

[TAN 93] TANSEL, C. G.; et al. (editores). **Temporal Databases - theory, design and implementation** RedwoodCity: The Benjamin/Cummings Publishing Company, Inc., 1993.

[TAU 91] TAUZOVICH, B. Towards Temporal Extensions to the Entity-Relationship Model. In: INTERNACIONAL CONFERENCE ON THE ENTITY RELATIONSHIP APPROACH, 10., 1991, San Mateo. **Proceedings**... San Mateo, California: ER Institute, 1991. 793p. p.163-179.

[THE 94] THEODOULIDIS, B.; AIT-BRAHAM, A.; ANDRIANOPOULOS, G.; CHAUDHARY, J.; KARVELIS, G.; SOU, S. ORES, Temporal Databases System. In: ACM SIGMOD INTERNATIONAL CONFERENCE ON MANAGEMENT OF DATA, 1994, Minneapolis. **Proceedings**... Minneapolis, USA, 1994.

[TIM 99] **TIME DB, A Temporal Relational DBMS**. Disponível em http://www.timeconsult.com/Software/Software.html. Acesso em 08/2000.

[WUU 93] WUU, G. T. J; DAYAL, U. A Uniform Model for Temporal and Versioned Object-Oriented Databases. In: TANSEL, A. et al (eds.) **Temporal Databases: Theory, Design, and Implementation** Redwood City: The Benjamin/Cummings Publishing Company, Inc., 1993. chap. 10, p. 230-247.

[YOU 99] YOU-CHIN, F.; DESSLOCH, S.; CHEN, W.; MATTOS, N.; TRANS, B.; LINDSAY, B.; DEMICHIEL, L.; RIELAU, S.; MANNHAUPT, D. Implementation of SQL3 Structured Types with Inheritance and Value Substitutability. In: INTERNATIONAL CONFERENCE ON VERY LARGE DATA BASES, 25., 1999, Edinburgh. **Proceedings**… Edinburgh: VLDB, 1999. p.565-574.

[ZEL 95] ZELLER, A. A Unified Versions Model for Configuration Manage ment. In: ACM SIGSOFT Symposium on the Foundations of Software Engineering, 3., 1995, Washington. **Proceedings**... New York: ACM Press, 1995. p. 151-160.