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

RAQUEL VIEIRA COELHO COSTA

Projeto de um DB2 *Extender* para Suporte aos Conceitos de Tempo e Versão

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

Prof. Dr. Clesio Saraiva dos Santos
Orientador

Profa. Dra. Nina Edelweiss
Co-orientadora

Porto Alegre, janeiro de 2004

CIP – CATALOGAÇÃO NA PUBLICAÇÃO

Costa, Raquel Vieira Coelho

Projeto de um DB2 Extender para Suporte aos Conceitos de Tempo e Versão / por Raquel Vieira Coelho Costa. – Porto Alegre : PPGC da UFRGS, 2004.

92 f. : il.

Dissertação (mestrado) – Universidade Federal do Rio Grande do Sul. Programa de Pós-Graduação em Computação, Porto Alegre, BR – RS, 2004. Orientador: Clesio Saraiva dos Santos; Co-Orientadora: Nina Edelweiss.

1. Banco : Dados temporais. 2. Modelo de Versões. 3. Extensões de Banco de Dados. I. Santos, Clesio Saraiva dos. II. Edelweiss, Nina. III. Título.

AGRADECIMENTOS

Em princípio agradeço ao Prof. Clesio e à Profa. Nina, meus orientadores, pela oportunidade e incentivo de realizar e concluir esta pesquisa. Eles souberam de maneira brilhante passar o conhecimento necessário e esclarecer minhas dúvidas.

Agradeço também aos colegas do meu grupo de pesquisa (Mirella, Fábio, Anelise, Fabrício) que estiveram presentes nas nossas reuniões com idéias e sugestões, em especial à Renata que me deu uma força muito grande com sua disposição em revisar meu trabalho e com suas palavras de incentivo.

Gostaria de agradecer, é claro, à minha mãe e meu irmão que sempre estão presentes na minha vida me incentivando e apoiando. Ao meu pai, que mesmo não estando mais por aqui, sempre será meu exemplo maior de dedicação e perseverança.

Um agradecimento sincero também aos meus colegas da Procuradoria, tanto de Porto Alegre, quanto de Brasília, por me incentivarem a concluir esta jornada, principalmente aos amigos: Luis Otávio, Doris, Mello, Léo, Joab, Genésio, Cláudia e Ernani.

Obrigada aos meus sogros e meus cunhados por sempre se interessarem em saber como estava o andamento da minha dissertação. Outro muito obrigado a todos aqueles que também participaram deste trabalho e por um lapso meu não foram citados aqui.

Por fim e principalmente, ao meu amor Rafael, pela sua presença ao meu lado em todos os momentos, me ajudando e incentivando de todas formas possíveis, e me dando idéias valiosas.

"Teu êxito depende muitas vezes do êxito das pessoas que te rodeiam."
– BENJAMIN FRANKLIN

SUMÁRIO

LISTA DE ABREVIATURAS

API	*Application Programming Interface*
BD	Banco de Dados
BDT	Banco de Dados Temporal
ER	Entidade-Relacionamento
IAV	*Image, Audio and Video*
LOB	*Large OBjects*
OID	*Object Identificator*
OO	Orientado a Objetos
SGBD	Sistema de Gerenciamento de Banco de Dados
SQL	*Structured Query Language*
TTF	Tempo de Transação Final
TTI	Tempo de Transação Inicial
TVF	Tempo de Validade Final
TVI	Tempo de Validade Inicial
TVM	*Temporal Version Model*
TVQL	*Temporal Versioned Query Language*
UDF	*User Defined function*
UDT	*User Defined Type*
UML	*Unified Modeling Language*
XML	*Extensible Markup Language*

LISTA DE FIGURAS

LISTA DE TABELAS

RESUMO

A utilização de versões permite o armazenamento de diferentes alternativas de projeto no desenvolvimento de uma aplicação. Entretanto, nem todo o histórico das alterações aplicadas sobre os dados é registrado. Modificações importantes podem ser realizadas e os valores anteriores são perdidos. O histórico completo somente é acessível através da junção de versões com um modelo temporal. Os conceitos de tempo e de versão aplicados em conjunto possibilitam a modelagem de aplicações complexas. Uma extensão que implemente simultaneamente estes dois conceitos em um banco de dados comercial não está disponível. O Modelo Temporal de Versões (TVM – *Temporal Version Model*) fornece a base para esta funcionalidade.

O objetivo deste trabalho é projetar um *extender* para oferecer suporte aos conceitos de tempo e versão no sistema DB2, utilizando como base o TVM. A extensão engloba o mapeamento da hierarquia do TVM; a criação de tabelas administrativas; procedimentos para especificação das classes, atributos e relacionamentos; a definição de gatilhos e restrições para a manipulação dos dados diretamente nas tabelas criadas; e a especificação de procedimentos e UDFs para controle de versões e valores temporais e de outras UDFs que permitem consultas envolvendo os dois conceitos. Apesar do SGBD não ser totalmente orientado a objetos, como é definido no modelo utilizado (TVM), oferece mecanismos que permitem o mapeamento para um modelo objeto-relacional. Através da utilização desta extensão, a união de tempo e de versões pode ser utilizada em aplicações reais.

Palavras-chave: modelo de dados temporal, modelo de versões, extensão, banco de dados, DB2

Project of a DB2 Extender to Support Time and Version Concepts

ABSTRACT

The use of versions allows the storage of different project choices in the development of an application, but not all the data changes are registered. Important modifications may be realized and the previous values are lost. The whole history is only accessible joining versions and a temporal model. Version and time concepts applied together make possible the modeling of complex applications. An extension implementing both concepts simultaneously in a commercial database is not available. The Temporal Versions Model (TVM) offers a base to this functionality.

The purpose of this work is to design an extender to offer time and version support to DB2 system, using the base of TVM. The extension includes the mapping of TVM hierarchy; the creation of administrative tables; procedures to specify classes, attributes and relationships; the definition of triggers and constraints to the manipulation of data directly in created tables; the specification of procedures and UDFs to control versions and temporal values; and UDFs to allow queries with both concepts. Although the DBMS is not object-oriented, as defined in the model used (TVM), it offers mechanisms that allow mapping to an object-relational model. By the use of this extension, time and version union can be used in real applications.

Keywords: temporal data model, versions model, extension, database, DB2

1 INTRODUÇÃO

A utilização de versões permite o armazenamento de alternativas de projeto, porém nem todo o histórico das alterações aplicadas sobre os dados é registrado. Modificações importantes podem ter sido realizadas sem que seja permitido o acesso a todos os valores. O histórico completo somente é acessível através da junção com um modelo temporal.

Juntando esses conceitos, alguns trabalhos propõem a união dos modelos temporais e de versões (MORO et al., 2001), (RODRÍGUEZ; OGATA; YANO, 1999) e (TANSEL, 1993). No modelo de dados proposto por Moro, particularmente, são armazenadas as versões de um objeto, e para cada versão, o histórico das alterações feitas nos valores de seus atributos e relacionamentos dinâmicos. Esse modelo denomina-se TVM (*Temporal Version Model*) e encontra-se detalhado em (MOROa, 2001).

Uma das características mais poderosas do DB2 é sua arquitetura extensível (RENNHACKKAMP, 1997). Com a inclusão dos *extenders* relacionais que exploram características objeto-relacionais, a IBM permite que os usuários estendam as funcionalidades dos servidores DB2 (CAREY et al., 1999; IBM, 2001) para qualquer domínio de aplicação (BISCHOFF, 1997). Cada *extender* deve definir os atributos que caracterizam os novos tipos de dados, bem como oferecer funções para a criação, atualização, eliminação e pesquisa sobre os dados armazenados.

As extensões são pacotes de UDTs (*User-Defined Type*), UDFs (*User-Defined Function*), *triggers*, *stored procedures* e *constraints* (IBMc, 2000) que satisfazem um determinado domínio (MATTOS, 1994). Utilizando os *extenders*, os usuários podem armazenar dados complexos. Este armazenamento pode ser realizado em colunas, tabelas, ou em arquivos externos (MATTOS, 1995; WANG; ZANIOLO, 2000).

Os *extenders* são armazenados de forma independente da base de dados em si. Isto torna possível que sejam definidas extensões por vários desenvolvedores e instaladas apenas as que forem de interesse do usuário. Existem inúmeros *extenders* disponíveis para serem instalados como: *Text Extender* (IBMa, 2000), *IAV (Image, Audio and Video) Extenders* (IBMb, 2000) e *XML Extender* (CHENG, 2000;IBMc, 2000). A IBM e as suas empresas parceiras desenvolveram inúmeros outros *extenders*, porém estes são os mais utilizados e que possuem estrutura melhor documentada.

Este trabalho propõe a definição de um gerenciador de tempo e versões sobre um banco de dados comercial. O TVM *Extender* visa proporcionar aos usuários a incorporação dos conceitos de tempo e versão em suas aplicações. Com o mapeamento proposto, o TVM é utilizado em um SGBD comercial e amplamente difundido atualmente como o DB2. A especificação proposta pode ser usada ainda em um mapeamento para outro banco de dados ou como base de manipulação de dados para outros trabalhos relacionados à evolução de esquemas e linguagens de consulta.

O objetivo principal é o projeto de uma extensão que ofereça suporte a tempo e versão neste banco de dados comercial. Este processo envolve a análise da estrutura do modelo TVM, bem como de todos os conceitos envolvidos no desenvolvimento de *extenders* do DB2.

Em uma primeira etapa são analisados os conceitos do modelo TVM, destacando as suas diversas características. A estrutura de armazenamento que o *extender* utiliza é definida, realizando um estudo sobre as formas disponíveis nas extensões existentes e as características objeto-relacionais do DB2. Para adicionar as funcionalidades de tempo e versão ao DB2 são definidos metadados que armazenam informações sobre o gerenciamento das classes, relacionamentos e versões. Como um *extender* é um conjunto de funcionalidades (UDTs, UDFs, *triggers*, *stored procedures* e *constraints*), em uma segunda etapa é realizado o mapeamento das classes e métodos do TVM, que é orientado-objeto, para estes mecanismos.

Os mecanismos definidos permitem que os usuários especifiquem suas classes e relacionamentos, manipulem os dados (criação, alteração e eliminação) e realizem consultas sobre eles. Na seqüência é realizada a seleção das principais características do TVM que são utilizadas no projeto do núcleo do *extender.* O produto final do trabalho engloba o mapeamento do TVM para um extender do DB2 e um protótipo de seu núcleo, integrando as principais características de tempo e versão. As definições deste projeto já levam em consideração futuras implementações a serem desenvolvidas que irão englobar o modelo TVM como um todo.

O texto está organizado como segue: o capítulo 2 apresenta uma síntese do modelo TVM, que foi mapeado para um *extender* do DB2. O capítulo 3 introduz as extensões de bancos de dados, detalhando a extensibilidade do DB2.

O capítulo 4 é a base do trabalho e apresenta a definição do *TVM Extender*. No capítulo 5, um estudo de caso ilustra a utilização do *TVM Extender* através de uma aplicação simples.

As conclusões obtidas sobre este trabalho, bem como as sugestões para trabalhos futuros constam no capítulo 6.

Os anexos apresentam os seguintes conteúdos:

- Anexo A: contém o *script* utilizado no protótipo para a criação dos tipos de dados no DB2, que representam a hierarquia do TVM e a criação das tabelas correspondentes aos metadados do *TVM Extender*, com algumas UDFs de apoio;

- Anexo B: mostra os *scripts* que definem os procedimentos armazenados que servem para especificar as classes, atributos e relacionamentos pelo protótipo.

2 CONCEITOS DE TEMPO E VERSÃO

Este capítulo resume os principais conceitos relacionados aos aspectos temporais bem como de versões em bancos de dados. Além disso, são citados alguns trabalhos que vêm sendo desenvolvidos envolvendo os aspectos temporais e de versionamento. Especificamente, é detalhado o Modelo Temporal de Versões, que foi utilizado como base para esta extensão.

2.1 Aspectos Temporais

Um modelo de dados temporal permite a representação de aspectos estáticos e dinâmicos de uma aplicação, bem como sua evolução temporal (EDELWEISS; OLIVEIRA, 1994; TANSEL, 1993). Bancos de Dados Temporais (BDT) armazenam todos os estados de uma aplicação (presente, passado e futuro), mantendo sua evolução com o passar do tempo. As informações temporais são associadas implicitamente aos dados, correspondendo ao tempo de validade (tempo no qual a informação é valida no mundo real) e/ou ao tempo de transação (tempo no qual a informação foi inserida no banco de dados).

Os modelos de dados tradicionais apresentam duas dimensões: (i) representando as instâncias dos dados (linhas de uma tabela), e (ii) representando os atributos de cada instância (colunas desta tabela). Cada atributo de uma instância apresenta um só valor e cada alteração desse valor implica na perda do anterior. Para os modelos temporais é acrescentada mais uma dimensão aos modelos tradicionais, chamada dimensão temporal, a qual associa uma informação temporal a cada valor. Nesse caso, se o valor de um atributo for alterado, o valor anterior não é removido do BD (EDELWEISS; OLIVEIRA, 1994).

O uso de um modelo de dados temporal para especificar uma aplicação não implica, necessariamente, na utilização de um SGBD específico para o modelo. Bancos de dados comerciais podem ser usados se existir um mapeamento adequado entre o modelo temporal e o banco de dados utilizado. Um BDT pode ser implementado sobre um banco de dados relacional, orientado a objetos, objeto-relacional e outros. Em cada um deles devem ser preservadas as características individuais, bem como as regras que regem cada banco de dados (BD).

A implementação de bancos de dados temporais em sistemas gerenciadores de banco de dados (SGBD) convencionais deve-se à inexistência de um SGBD totalmente temporal que tenha um amplo uso. Nesse caso, são feitos mapeamentos apropriados para representar modelos temporais necessitando uma estratégia específica para sua representação, para que o gerenciamento dos dados históricos seja independente da intervenção do usuário.

Algumas implementações de bancos de dados temporais em SGBDs relacionais, entidade-relacionamento e orientados a objetos são propostas nas referências

(EDELWEISS et al., 2000) (CAVALCANTI et al., 1995) (ANTUNES; HEUSER; EDELWEISS, 1997) (ELMASRI et al., 1993) (HELD et al., 1975) (JENSEN et al., 1998) (SIMONETTO, 1998) (TANSEL, 1993). Essas implementações, em sua maioria, estão baseadas em modelos de dados já publicados na literatura.

2.2 Versões

Com o passar do tempo o banco de dados pode sofrer alterações por vários motivos, como a representação de alternativas de projetos ou a própria evolução dos dados. Nesses casos, há a necessidade de guardar mais de um estado para um objeto e, para satisfazer esse requisito, o conceito de versões pode ser incorporado à semântica do sistema. Uma versão corresponde a um estado identificável de um objeto e deve ser tratada uniformemente em um modelo de dados.

Em modelos orientados a objetos, um objeto pode ser representado através de versões. Nesses modelos, uma versão é um objeto de primeira classe, possuindo seu próprio identificador (*Object Identifier* - OID), o que permite que seja diretamente manipulada ou consultada como qualquer outro objeto.

Objetos versionados são objetos que apresentam versões (GOLENDZINER, 1995). Um objeto versionado pode ter várias versões, organizadas em um grafo acíclico dirigido, representando a ordem de derivação.

Como o aspecto de versionamento está associado aos objetos, um objeto não versionado pode passar a ser versionado dinamicamente. Nesse processo, esse objeto não versionado passa a ser a primeira versão do objeto versionado e o antecessor da nova versão.

Todo objeto versionado possui uma versão corrente que é gerenciada pelo sistema, sendo esta atualizada sempre que novas versões forem criadas ou no caso do usuário especificar outra versão com sendo a versão corrente do objeto versionado. Nesse caso, esta versão permanece fixa.

A literatura apresenta alguns estudos de versões sobre bases de dados relacionais (DADAM; LUM; WERNER, 1984), mas a grande parte da pesquisa sobre esse conceito concentra-se na sua utilização de modelos de dados orientados a objetos (CHOU; KIM, 1986) (BEE; MAHBOD, 1988) (BJÖRNERSTEDT; HULTÉN, 1989) (KIM 89) (AGRAWAL,1991) (TALENS; OUSSALAH, 1993) (WUU; DAYAL, 1993) (GOLENDZINER, 1995) (CONRADI; WESTFECHTEL, 1998). O controle de versões pode ser utilizado para gerenciar documentos XML (CONRADI; WESTFECHTEL, 1998).

2.3 União dos Conceitos

Apesar dos modelos de dados versionados armazenarem alternativas de projeto, algumas modificações podem ter sido realizadas e perdidas por terem sido sobrepostas. O fato de que as informações evoluem com o tempo levou ao desenvolvimento de modelos de banco de dados temporais (EDELWEISS; OLIVEIRA, 1994) (TANSEL, 1993). Esses modelos temporais têm o objetivo de integrar em um sistema as informações referentes ao passado, presente e futuro, armazenando uniformemente os estados de um objeto com relação à sua evolução no decorrer do tempo. Para englobar todas as possibilidades é importante a utilização de um modelo que suporte os aspectos de versões bem como os temporais.

Alguns trabalhos propõem a união dos modelos temporais e de versões (MOROa, 2001) (RODRÍGUEZ; OGATA; YANO, 1999) (TANSEL, 1993). No modelo de dados

de Moro, particularmente, são armazenadas as versões de um objeto, e para cada versão, o histórico das alterações feitas nos valores de seus atributos e relacionamentos dinâmicos (MOROa, 2001). Esse modelo denomina-se Modelo Temporal de Versões (TVM - *Temporal Versions Model*). Experiências anteriores implementaram separadamente os conceitos de tempo (BRAYNER; MEDEIROS, 1994) (HUBLER, 2000) e versão (AGRAWAL,1991) (BEE; MAHBOD, 1988) (GOLENDZINER, 1995) de modelos de dados orientados a objetos em bancos de dados tradicionais.

O Modelo Temporal de Versões é um modelo de dados orientados a objetos que suporta: (i) versões: possibilitando a definição e a manipulação de objetos, versões e configurações, a navegação através da hierarquia de herança e permitindo manter transparente a manipulação de versões, quando necessário; e (ii) tempo: permitindo representar toda a história e a evolução dos dados da aplicação. Por representar o modelo dados base deste estudo, o TVM é mostrado com mais detalhes na próxima seção.

2.4 Modelo Temporal de Versões

O Modelo Temporal de Versões (TVM – *Temporal Versions Model*) é detalhado em (MOROa, 2001, MORO et al., 2001) foi definido para permitir o armazenamento das versões de objetos e, para cada versão, o histórico de suas propriedades dinâmicas e valores dos relacionamentos.

O TVM abrange a gerência de versões no nível de aplicação, dentre os dois tipos identificados por (BJÖRNERSTEDT; HULTÉN, 1989), que suporta a representação de informações dependentes de tempo e seqüenciamento, conforme definido pelo usuário da aplicação. Este nível pode evitar a sobrecarga da capacidade de armazenamento do banco de dados, preservando a sua *performance*. A outra abordagem seria o gerenciamento no nível de sistema, na qual o histórico reflete toda a seqüência de modificações detectadas pelo sistema, realizadas sobre um objeto.

O TVM difere dos outros modelos temporais, como (WUU; DAYAL, 1993) e (RODRÍGUEZ; OGATA; YANO, 1999), por apresentar: (i) o uso de duas ordens de tempo: ramificada para um objeto, e linear para a evolução de cada versão; (ii) a possibilidade do usuário especificar qual elemento terá suas informações temporais armazenadas. Essa segunda característica é importante para limitar o tamanho do espaço de dados e prover um desempenho melhor.

A seguir são apresentados os principais conceitos envolvidos no TVM, que foi mapeado para um *extender* do DB2, como forma de validação do TVM. As diferentes características do TVM devem ser analisadas buscando identificar os mecanismos adequados para a modelagem dos objetos para um banco de dados objeto-relacional.

2.4.1 Versões

Para o TVM, uma *versão* é a descrição de um objeto em um determinado momento de tempo, ou sob um determinado ponto de vista, cujo registro é importante para a aplicação. Num modelo orientado a objetos, uma versão é um objeto de primeira classe. Como tal, possui seu próprio identificador de objeto (OID), o que permite que seja diretamente manipulada e consultada (GOLENDZINER, 1995).

As versões de uma entidade do mundo real devem ficar agrupadas e constituem um *objeto versionado*, que é também um objeto de primeira classe (possui um OID), mantendo informações sobre as versões a ele associadas. Um objeto versionado pode apresentar propriedades que devem ser comuns a todas as suas versões. Cada versão só faz parte de um objeto versionado.

Como muitas vezes não é possível antecipar se objetos apresentarão versões ou não, eles podem dinamicamente passar de não versionados a versionados. Objetos (versionados ou não) que possuem as mesmas propriedades e comportamento podem ser agrupados em uma classe. A característica de ser ou não versionado é uma característica de cada objeto e não de sua classe. Assim, uma classe pode apresentar objetos não versionados e versionados.

As versões de um objeto versionado são organizadas formando um grafo acíclico dirigido, refletindo seus relacionamentos, onde apenas a primeira versão não possui uma antecessora. Uma referência a um objeto versionado é denominada referência dinâmica, uma vez que pode indicar qualquer uma das versões daquele objeto. O mecanismo de versão corrente permite determinar automaticamente a versão de um objeto versionado a ser utilizada. Apresenta também referências estáticas, onde uma versão específica do objeto versionado é indicada. As versões possuem ainda um *status*, que reflete seu estágio de desenvolvimento, podendo ser em trabalho (*Working*), estável (*Stable*), consolidada (*Consolidated*) ou desativada (*Deactivated*).

A relação entre os objetos, seus ascendentes e descendentes é realizada através do identificador do objeto, que tem estrutura apresentada na Figura 2.1.

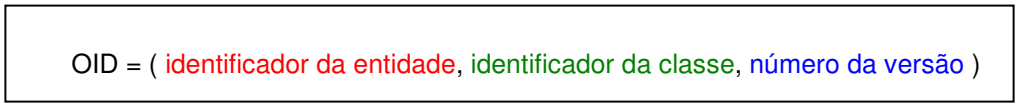


Figura 2.1: Estrutura do OID

O número da versão é representado por valores inteiros gerados seqüencialmente. Eles não podem ser reutilizados em casos de exclusão. Quando um objeto representar o objeto versionado, seu número de versão será nulo e, quando for um objeto não versionado, seu número de versão será 1. Assim, um objeto não versionado será identificado da mesma maneira que a primeira versão de um objeto versionado.

2.4.2 Aspectos Temporais

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. Como muitas versões de um mesmo objeto podem coexistir no tempo, pode ser necessário considerar duas linhas de tempo:

- *tempo ramificado*: definido para um objeto, devido às diferentes linhas de tempo de cada versão que são originadas da linha do objeto;
- *tempo linear*: determinado para cada versão.

A variação temporal é discreta, e a temporalidade é representada no TVM pelo rótulo de tempo elemento temporal (conjunto de intervalos temporais), bitemporal (tempos de transação e validade) ou implícito. Cada objeto possui um atributo pré-definido *alive* associado ao seu estado de vida. Na criação do objeto, *alive* recebe o valor *true*, armazenando o tempo de validade inicial. Quando ocorre a exclusão lógica do objeto, o atributo recebe o valor *false* e o tempo de validade final recebe o tempo de transação.

Já os atributos e relacionamentos do objeto podem ser definidos com *estáticos* (sem variação de valores armazenada) ou *temporalizados* (todas as variações são armazenadas formando o seu histórico). O usuário deve especificar quais atributos e relacionamentos serão temporalizados. Os valores dos objetos temporalizados estão

associados com os rótulos temporais que representam os tempos de validade inicial e final, e os tempos de transação inicial e final[1].

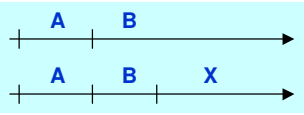
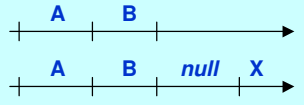
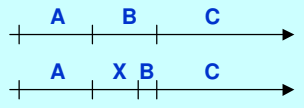
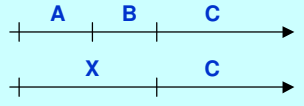
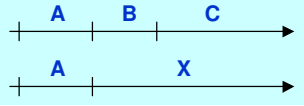
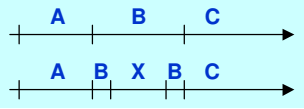
2.4.2.1 Inserção

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 os tempos de transação inicial e final, sendo que o final somente será fornecido quando uma nova instância desta informação for inserida, gerada por uma atualização da base de dados. 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).

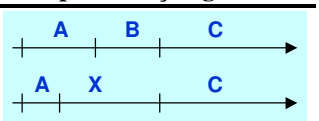
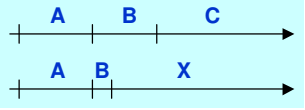
2.4.2.2 Atualização

A Tabela 2.1 apresenta as possíveis combinações para atualização de valores e rótulos temporais. Foram adotadas as seguintes abreviações: TVI para tempo de validade inicial, e TVF para tempo de validade final; as informações existentes são representadas por *A*, *B* e *C*, e a nova informação por *X*.

Tabela 2.1: Possibilidades de rótulos na atualização

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

[1] Tempo de transação é o tempo no qual é feita a atualização do banco de dados e tempo de validade representa o período de validade da informação armazenada.

Condição	Explicação e resultado	Representação gráfica
- TVF = TVF	O TVF da informação armazenada recebe o TVI da nova menos um instante	
- TVF infinito	O TVF da informação armazenada recebe o TVI da nova menos um instante que vale a infinito	

2.4.2.3 Exclusão

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. 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 registrados. O uso deste tipo de exclusão deve ser cuidadoso, pois proíbe que estados passados sejam reconstruídos. O TVM não define as regras para esse tipo de exclusão (MOROa, 2001).

2.4.3 Diagrama de Classes

A hierarquia do TVM, apresentada na Figura 2.6, permite definir dois tipos de classes da aplicação:

- *classe de aplicação não temporal e não versionável* - definida como subclasse de `Object`. A modelagem permite contemplar classes que representam tabelas de uma base já existente ou classes auxiliares nas quais os conceitos de tempo e versões não são necessários;

- *classe de aplicação temporal e versionável* - definidas 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.

Na hierarquia de classes do TVM, o controle das versões é feito pela classe `VersionedObjectControl`, sendo que cada versão ou objeto versionado tem um relacionamento estabelecido com essa classe. Além disso, na hierarquia do TVM a mudança de uma classe, de não versionada para versionada, deve ser feita pelo usuário em tempo de projeto. No entanto, é importante considerar que é possível que um objeto de uma classe temporal versionada não tenha versão associada.

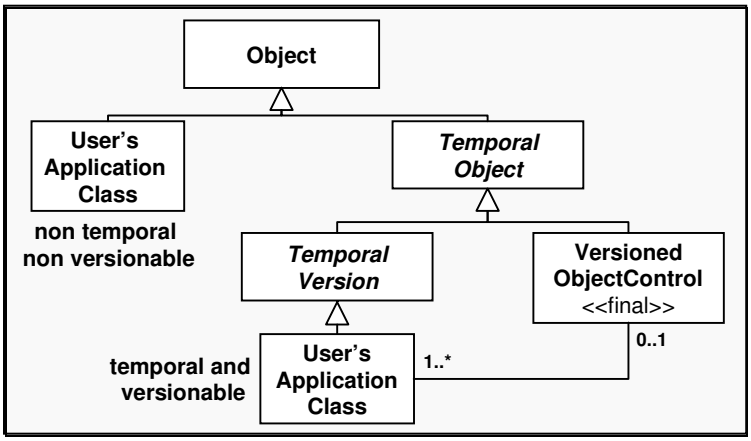


Figura 2.2: Hierarquia de Classes do TVM

A Figura 2.3 ilustra a forma como os objetos de uma classe temporal versionada são instanciados. Ela apresenta os objetos versionados `ObjVers1`, `Obj2` e `Obj3`, sendo que somente o `ObjVers1` possui versões associadas.

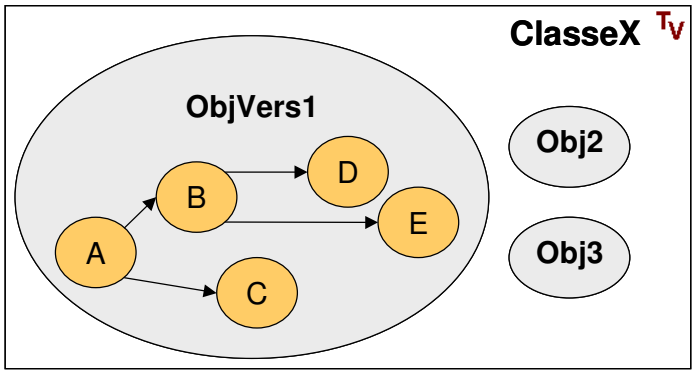


Figura 2.3: Objetos versionados e não versionados de uma classe TVM

Quando uma versão é derivada como A, dando origem a duas novas versões B e C., são utilizados os conceitos de versões predecessoras e sucessoras. No exemplo da Figura 2.3, B e C aparecem como sucessoras de A. As versões B e C têm como predecessora a versão A; D e E têm B como predecessora; e D e E são sucessoras de B.

2.4.4 Estados de uma Versão

As versões em uma determinada aplicação representam as alternativas de projeto e a evolução dos seus dados. Cada versão possui um estado que reflete seu estágio de desenvolvimento e/ou consistência durante seu tempo de vida. Uma versão pode passar por alguns estados (*Working*, *Stable*, *Consolidated* e *Deactivated*), que definem o conjunto de operações que podem ser aplicadas sobre uma versão, conforme a Tabela 2.2.

Tabela 2.2: Estados das versões e respectivas operações

Estado	Operações
Working (W)	Pode servir como base de uma derivação, ser promovida para *Stable*, ser alterada, consultada ou ainda excluída (logicamente)
Stable (S)	Pode servir como base de uma derivação, ser promovida para *Consolidated*, consultada, compartilhada por outros usuários, excluída desde que não possua nenhuma versão sucessora, e não

Estado	Operações
	pode ser alterada
Consolidated (C)	Pode servir como base para derivação, ser consultada, compartilhada por outros usuários, e não pode ser alterada nem excluída
Deaactivated (D)	Pode ser apenas consultada e restaurada

As transições entre esses estados bem como as operações que ocasionam tais transições estão representadas no diagrama de estados na Figura 2.4.

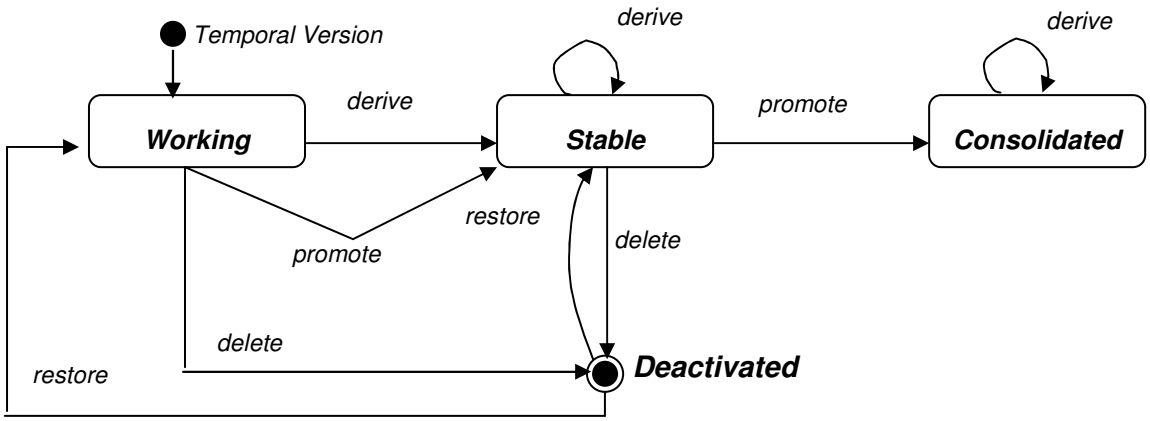


Figura 2.4: Estados das Versões

2.4.5 Relacionamentos

Os relacionamentos de associação podem ser estabelecidos entre duas classes normais, ou entre classes normais e temporais versionáveis, ou entre duas temporais versionáveis. Entretanto, os de agregação são permitidos apenas entre classes temporais versionáveis.

É preciso diferenciar os relacionamentos de associação e os de agregação, principalmente por causa das configurações que utilizam esse último tipo. As classes normais não podem ter esse tipo de relacionamento porque há restrições na exclusão de objetos que estão envolvidos em relacionamentos de agregação, e não é possível aplicar essas restrições para as classes normais. Porém, isso não impõe uma grande perda, pois é importante ressaltar que os objetos das classes normais não podem ser configurações; apenas os objetos temporais versionáveis é que podem porque herdam da classe *TemporalVersion*, na qual são definidas as características para dar suporte a configurações.

Tanto relacionamentos de associação quanto de agregação podem ser definidos como temporais, sendo que, nesse caso, os seus inversos também devem ser tratados como temporais. Porém, apenas classes temporais versionáveis aceitam declarações desse tipo. As classes normais não podem apresentar relacionamentos temporais porque não possuem suporte ao tempo.

2.4.6 Relacionamento Herança por Extensão

Outra característica relevante do TVM é a definição do **relacionamento de herança por extensão**, onde versões são admitidas em 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. 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. Essa hierarquia formada pelas classes que participam da herança por extensão é chamada hierarquia de extensão.

A herança por extensão difere da herança adotada nas linguagens e modelos orientados a objetos, denominada de herança por refinamento. A primeira diferença entre essas heranças está na finalidade da modelagem. Na especificação de sistemas ou na modelagem de dados, essas heranças são definidas em tempo de análise ou projeto. A herança por refinamento traz as vantagens de extensibilidade, pela redefinição de estado e comportamento nas subclasses durante o projeto, e de reuso de código na implementação. A herança por extensão é definida quando é detectada a necessidade de instanciar a entidade em diferentes níveis de detalhamento (em tempo de execução). Além disso, a herança por extensão permite a modelagem dinâmica das aplicações. Durante a fase de projeto o esquema raramente apresenta-se estável ou completamente definido, sendo possível realizar modificações em cada nível separadamente sem afetar as instâncias existentes.

Outra diferença está no aspecto de funcionamento das hierarquias. Na herança por refinamento, as instâncias de Y mantêm seu próprio armazenamento para os atributos definidos em Y e X (Figura 2.5a). Já na herança por extensão, todos os atributos de x são compartilhados por y (Figura 2.5b). Esse conceito é apresentado por (BILIRIS,1990).

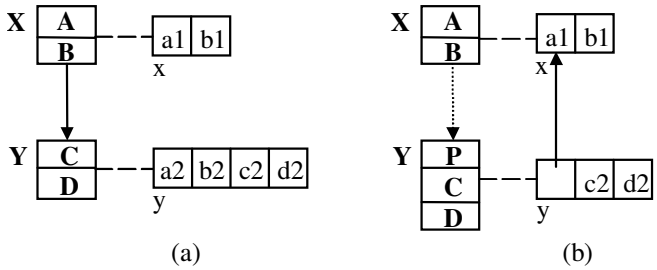


Figura 2.5: Armazenamento na Herança por Refinamento e por Extensão

Dessa forma, na hierarquia de extensão, cada versão deve estar necessariamente associada a pelo menos um ascendente (objeto versionado, versão ou objeto temporal versionável que ainda não possui versões) na superclasse, caso não pertença a uma classe raiz na hierarquia. Esse ascendente (ou grupo de ascendentes) tem que pertencer a mesma entidade da versão em questão, como será visto adiante. Assim, no momento da criação de uma versão, deve ser feita a ligação dessa com um ascendente. Entretanto, versões na superclasse podem existir sem estarem relacionadas ainda com versões nas subclasses. Essa obrigatoriedade de um ascendente também é imposta aos objetos versionados e não versionados.

A referência a um ascendente pode ser feita de forma estática, quando é indicada uma versão específica, ou dinâmica, quando a referência é ao objeto versionado e não a uma determinada versão dele. Na referência dinâmica, a versão corrente será utilizada quando o ascendente for solicitado.

É possível que uma versão apresente mais de um ascendente. Na Figura 2.6, é apresentada a correspondência entre versões em diferentes níveis. Considerando que existe a classe Veículo e a classe Automóvel, a qual é uma especialização da anterior, a entidade Palio aparece no nível de Veículo, onde apresenta os atributos motor e combustível, e no nível de Automóvel, com atributos número de volumes e acessórios. Tomando como exemplo a versão ELX no nível Automóvel, têm-se como antecessoras

em Veículo, as versões v3 e v4, o que significa que é possível ter um Palio ELX 1.6 a álcool ou à gasolina.

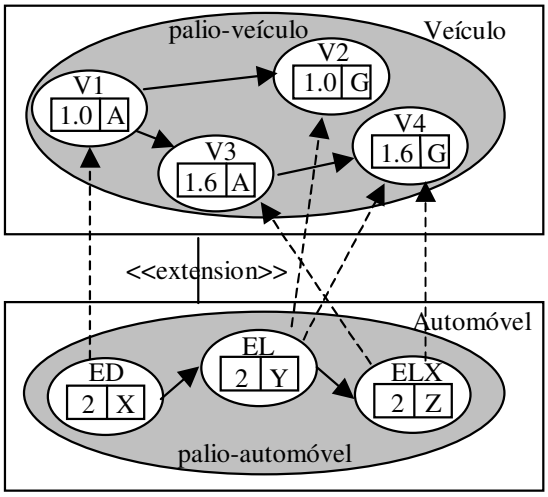


Figura 2.6: Correspondências de versões em diferentes níveis de hierarquia

Podem ser estabelecidas restrições de cardinalidade (mapeamento) entre versões de um objeto em uma classe e as versões de seus ascendentes na superclasse, podendo ser 1:1, 1:n, n:1 ou n:m. É importante lembrar que, independente da cardinalidade, os ascendentes de cada versão devem pertencer a mesma entidade da versão.

2.4.7 Configuração

A definição de uma configuração para uma determinada versão é dada pela escolha de uma única versão (pois pode haver vários ascendentes) para cada nível ascendente em que a entidade está representada na hierarquia de extensão e uma única versão para cada componente na hierarquia de agregação. Se o componente também possui outros componentes, o mesmo processo é feito de forma recursiva para todos estes na hierarquia de agregação.

É possível criar diferentes configurações para o mesmo objeto mediante a escolha de versões para componentes e/ou ascendentes. Dessa forma, uma configuração pode ser dita uma versão especial de um objeto, chamada versão configurada.

2.4.8 Linguagem de definição

A Figura 2.7 apresenta a sintaxe geral e simplificada da linguagem para definição de uma classe proposta para o TVM. A BNF completa pode ser obtida em (MOROa, 2001). Quando uma classe é temporal versionado é utilizada a cláusula `hasVersions`. A herança por extensão pode ser definida pela cláusula `byExtension`. A cláusula `correspondence` permite estabelecer o tipo de cardinalidade entre a classe descendente e sua ascendente. A cardinalidade entre as 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, a cardinalidade assumida é de um para um. A cláusula `temporal` indica que um atributo ou relacionamento terá sua evolução armazenada.

```
class ::= [public] [ abstract | final]
class className [hasVersions] [ inherit
[byExtension] className [correspondence (1:1 | 1:n | n:1 | n:m)]]
[[temporal] aggregate_of[n] className (by value | by reference)
{,[temporal]aggregate_of[n] className (by value | by reference)}]
( [ Properties:
{ [ public | private | protected] [static]
[temporal] attributeName: attributeDomain;}+]
[ Relationships:
{ [temporal] relationshipName (0:1 | 0:n | 1:1 | 1:n | n:m)
[inverse inverseRelationshipName ] relatedClass;}+ ]
[ Operations:
{ [ public | private | protected] [static]
[abstract | final] operationDefinitions }+ ] )
```

Figura 2.7: Sintaxe simplificada da Linguagem de Definição para uma classe

2.4.9 Linguagem de Consulta

A linguagem de consulta específica para o TVM, chamada TVQL – *Temporal Versioned Query Language*, permite a realização de consultas básicas (SQL padrão) e novas consultas que retornam valores característicos de tempo e versões. A TVQL estabelece um comportamento homogêneo para elementos normais e temporais versionados e foi descrita em detalhes em (ZAUPA, 2002). A sintaxe geral da TVQL é apresentada na Figura 2.8, onde os colchetes indicam um argumento opcional, as chaves representam argumento repetitivo opcional, os parênteses indicam um conjunto de opções, os argumentos separados pelo caracter 'l' indicam que um deles deve ser utilizado e os argumentos em negrito representam as palavras reservadas da TVQL.

```
query ::= SELECT [EVER][DISTINCT] targetC,targetC
FROM identificC,identificC [WHERE[EVER]searchC]
[GROUP BY groupC,groupC [HAVING logicalExpr]]
[ORDER BY orderC,orderC [setOp query];
targetC ::= (*|propertyName|aggregationFunctions|preDefInterval
|preDefInstant) [AS identifier]
identificC ::= className [.VERSIONS] [aliasName]
searchC ::= logicalExpr|tempExpr
groupC ::= propertyName|preDefInterval|preDefInstant
logicalExpr ::= AnyLogicalExpression
tempExpr ::= logicalExpr|PRESENT (logicalExpr)
orderC ::= groupC [ASC|DESC]
setOp ::= UNION | INTERSECTION | DIFFERENCE
preDefInterval ::= [propertyName.](tInterval|vInterval)
preDefInstant ::= [propertyName.](tiInstant|tfinstant|viInstant|vfInstant)
|[className.](iLifeTime)|(fLifeTime)
```

Figura 2.8: Sintaxe geral da TVQL

2.4.9.1 Suporte a Tempo

Com relação ao aspecto temporal, a TVQL possibilita a realização de consultas sobre o histórico de um objeto, de atributos e relacionamentos desse objeto, além de

permitir comparativos entre instantes e intervalos de tempo. Foi definido também um conjunto de operadores de comparação específicos para condições temporais. Para considerar todos os valores do histórico da vida de objetos e versões, a palavra reservada `EVER` é utilizada. Para anular a condição temporal mencionada no comando SQL foi estabelecida a função `PRESENT`, a qual considera os valores atuais das propriedades.

As consultas da Figura 2.9 ilustram as alternativas de consulta com o uso do `EVER` no `SELECT`, ou seja, considerando o retorno das propriedades temporais selecionadas. Na Figura 2.9a a cláusula `WHERE` considera também o histórico dos dados, retornando o nome de todas as pessoas que algum dia moraram na rua *Onze de maio*. Já na Figura 2.9b a cláusula `WHERE` considera apenas os dados atuais, pois a palavra reservada `PRESENT` anula o `EVER` anteriormente mencionado no `SELECT`. Nesse caso, a consulta retorna o histórico dos nomes das pessoas que atualmente moram na rua *Onze de maio*. Por último a Figura 2.9c considera o histórico das propriedades alterando o período de validade (ou transação), retornando o histórico dos nomes das pessoas que moraram na rua *Onze de maio*, a partir de 01/01/2002.

<table>
<tr><td rowspan="3">SELECT EVER nome
FROM pessoa</td><td>WHERE rua LIKE ‘Onze de maio%’;
(a)</td></tr>
<tr><td>WHERE PRESENT (rua LIKE ‘Onze de maio%’);
(b)</td></tr>
<tr><td>WHERE rua.viInstant >= ‘01/01/2002’;
(c)</td></tr>
</table>

Figura 2.9: Alternativas de consulta com `EVER` no `SELECT`

Por outro lado, se o usuário acrescentar o `EVER` somente na cláusula `WHERE`, a consulta retorna os valores atuais. No entanto a restrição pode considerar tanto o histórico dos valores, como também os valores atuais da base dependendo de como esta cláusula é apresentada.

2.4.9.2 Suporte a Versões

Para as características de versionamento, a linguagem oferece consultas sobre os estados das versões, versão corrente, sobre uma ou várias versões de um objeto, sobre a navegação na hierarquia (ascendentes/descendentes, primeiro/último, predecessores/sucessores, etc) e também sobre as configurações. Considerando a união das características temporais e de versões, as consultas são realizadas sobre as versões, seus estados, configurações e hierarquia, associando isso a um determinado instante ou período de tempo.

Nesse contexto, o usuário define na cláusula `FROM` os objetos e versões a serem consultados. Todas as versões dos objetos são consideradas acrescentando na cláusula `FROM` o nome da classe, seguido da palavra reservada `VERSIONS`. Se somente o nome da classe constar na cláusula `FROM`, a consulta considera os objetos e versões correntes.

Por exemplo, considerando a classe `Automóvel` com seus objetos (`Palio`, `Tempra`, `Marea`) e versões (`EX`, `XL`, `EL`, `EM`) conforme apresentados na Figura 2.10.

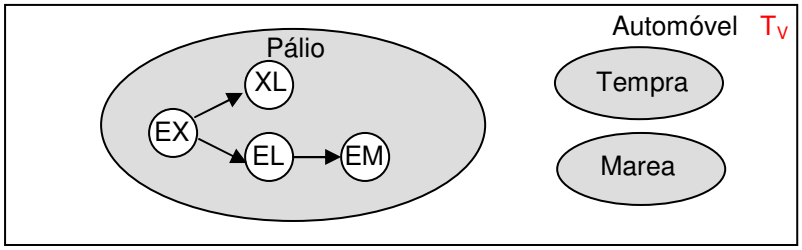


Figura 2.10: Exemplo de classe Temporal Versionada

Considerando as duas cláusulas `FROM` abaixo pertencentes a duas consultas:

```
FROM Automóvel
FROM Automóvel.VERSIONS
```

A primeira consulta é realizada sobre os valores dos objetos `Tempra` e `Marea` e a versão corrente do objeto `Palio`. A segunda consulta é realizada sobre os valores dos objetos `Tempra` e `Marea` e sobre as versões do objeto `Palio`: `EX`, `XL`, `EL` e `EM`.

As funções definidas para a recuperação específica das informações de versões e objetos versionados possuem o retorno do tipo `boolean` e podem ser divididas em 3 grupos:

- funções que recuperam o estado (*isWorking*, *isStable*, *isConsolidated* e *isDeactivated*);
- funções da navegação na hierarquia de herança (*isAscendantOf* e *isDescendantOf*);
- funções da navegação na hierarquia de derivação (*isFirst*, *isLast*, *isSuccessorOf*, *isPredecessorOf*, *isCurrent*, *isUserCurrent*, *isConfiguration*).

Para cada função definida para recuperação de estado e para navegação na hierarquia de derivação, existe outra com o mesmo nome e com o sufixo `At`, retornando o valor da informação relativo a um determinado ponto no tempo de validade.

As propriedades para controle do objeto versionado foram definidas para que esses objetos pudessem ser consultados de maneira transparente e podem ser utilizadas tanto no `SELECT` como no `WHERE` do comando TVQL. São elas: *configurationCount*, *currentVersion*, *firstVersion*, *lastVersion*, *nextVersionNumber*, *userCurrentFlag* e *versionCount*.

Um exemplo de consulta que envolva tanto as características de tempo e versões é apresentado a seguir. Ela obtém o histórico dos nomes das versões de `Pessoa` cujo estado era consolidado em 18/05/2002.

```
SELECT EVER nome
FROM pessoa.VERSIONS p
WHERE p.isConsolidatedAt('18/05/2002');
```

2.5 Considerações Finais

Nesse capítulo foram citados alguns modelos de dados com suporte a aspectos temporais e de versões existentes na literatura. Em especial, foi apresentado um modelo que une as características de tempo e de versões chamado TVM. Esse modelo permite a representação de todo o histórico e evolução dos valores dos atributos e relacionamentos das instâncias dos objetos de uma aplicação, bem como as alternativas de projeto.

3 EXTENSÕES DE BANCOS DE DADOS

Apesar da evolução marcante dos sistemas de gerenciamento de banco de dados comerciais nos últimos anos, a cada dia surgem novas aplicações de maior complexidade, impondo requisitos específicos que, muitas vezes, não fazem parte da funcionalidade do banco de dados. Como conseqüência, o SGBD deve prover mecanismos de pesquisar, recuperar e manipular os novos tipos de dados, utilizando instruções SQL (WANG; ZANIOLO, 2000).

Entretanto, não é viável que cada fabricante de SGBD implemente todos os tipos de dados necessários e os métodos que permitam a sua manipulação. Mesmo que fosse possível, pode não ser de seu interesse disponibilizar características muito específicas de determinadas aplicações. A solução vislumbrada por alguns e detalhada em (RENNHACKKAMP, 1997) foi tornar seus bancos de dados extensíveis, permitindo que os próprios desenvolvedores de aplicações implementem seus tipos de dados específicos. Surge, então, o conceito de *extensão* de um banco de dados objeto-relacional, que abrange um conjunto de atributos, estruturas e regras para gerar novos tipos de dados complexos.

Este capítulo aborda algumas extensões disponíveis nos bancos de dados comerciais, detalhando as extensões do DB2.

3.1 Extensões Disponíveis nos Bancos de Dados Comerciais

Os SGBDs comerciais estão buscando diferentes alternativas para manter o padrão SQL ao mesmo tempo que suportam novos tipos de dados e mecanismos para desenvolver aplicações avançadas. Em conseqüência disso, surgiram diferentes pacotes para novos tipos de dados e áreas de aplicação, como texto, imagens, vídeo, áudio, séries de tempo, dados espaciais e outros. A nova geração de servidores da *Oracle*, *Informix*, IBM e *Sybase* oferecem muito mais funcionalidades que o SQL puro. As capacidades objeto-relacionais introduzem novas interfaces de programação de aplicações (APIs), novos tipos, métodos e tecnologias de estender os servidores. As extensões dos servidores, como: *DB2 Extenders*, *Oracle Data Cartridges*, *Informix DataBlades* e *Sybase Snap-Ins* enriquecem as formas de desenvolver aplicações e sites Web que utilizem bancos de dados. (YOUNG, 1997 , NORTH, 1998).

A IBM implementou este conceito em seu sistema de banco de dados DB2 (MATTOS, 1995). As extensões foram denominadas *DB2 Extenders*, e permitem que os usuários definam novos tipos de dados complexos e os métodos para manipulá-los, além de melhorarem a produtividade no desenvolvimento de aplicações avançadas. Esta solução encaixou-se perfeitamente ao objetivo de adicionar as funcionalidades do TVM a um banco de dados comercial e amplamente utilizado como o DB2.

3.2 Extenders do DB2

A possibilidade de estender o SGBD para inteligentemente manipular tipos de dados complexos é um dos principais objetivos dos desenvolvedores de banco de dados. Entre os tipos complexos de dados estão: texto, imagens, vídeos, clipes de áudio, além de tipos de dados definidos pelos usuários específicos para determinados requisitos de negócio. O objetivo é incrementar o SGBD e sua linguagem de consulta, de modo a prover acesso integrado a todos os dados, bem como enriquecer o banco com uma semântica específica às aplicações (DAVIS, 1996).

Uma vez definido os novos tipos de dados, denominados *user-defined types* (UDTs), é necessário prover formas de manipulação e consultas eficazes nos seus dados, que podem ser realizadas através de mecanismos como:

- *user-defined functions* (UDFs) – permitem que o desenvolvedor defina novos métodos pelos quais as aplicações possam criar e manipular dados armazenados nos UDTs.
- índices definidos pelo usuário que permitam retornar de forma eficiente o conteúdo dos dados complexos.

Outro importante aspecto objeto-relacional do SGBD abrange a capacidade de manipular todos os dados e funções (sejam definidos pelo usuário ou predefinidos) em uma única sentença SQL. É essencial que as extensões não prejudiquem as funcionalidades originais do SGBD. Pelo contrário, a manipulação dos novos tipos é que tem de aproveitar características inerentes aos bancos de dados como: gerenciamento de transações, mecanismos de verificação de integridade, *backup* e *recovery*, entre outras.

A IBM, uma das empresas líderes na comercialização de SGBDs, tem empenhado grandes esforços de pesquisa, há vários anos, para desenvolver uma infra-estrutura para um banco de dados relacional extensível. O resultado destas pesquisas já aparece nitidamente na família de produtos do DB2.

Os *extenders* têm como base a infra-estrutura objeto-relacional do DB2. Cada extensão é um pacote de UDTs, UDFs, *triggers*, *stored procedures* e *constraints* que satisfaçam um determinado domínio de aplicação. Utilizando os *extenders*, os usuários podem armazenar dados complexos. Este armazenamento pode ser realizado dentro de tabelas ou em arquivos externos (MATTOS, 1995).

Cada *extender* deve definir os atributos que caracterizam os novos tipos de dados, bem como oferecer funções para a criação, atualização, eliminação e pesquisa sobre os dados armazenados. Assim, o usuário poderá incluir estes novos tipos e funções em sentenças SQL.

3.2.1 Definição

A IBM, a partir da versão 2 de seu sistema de gerenciamento de banco de dados DB2, introduziu uma infra-estrutura para a criação de tipos de dados e funções definidos pelos usuários (*user-defined types* e *user-defined functions*). Estes elementos permitem a definição e o armazenamento de objetos com estados e comportamentos complexos. As extensões exploram estes e outros conceitos, abordados em (MATTOS, 1994, CHAMBERLIM, 1996), como: gatilhos (*triggers),* procedimentos (*stored procedures*), restrições (*constraints)* e suporte para *Large Objects* (LOBs).

Os *extenders* do DB2 são utilizados, então, na especificação de novos tipos complexos de dados. Eles encapsulam os atributos, as estruturas e o comportamento dos tipos definidos, armazenando-os em colunas de tabelas do banco. Os dados neles

contidos podem ser acessados através de instruções SQL, como ocorre com os tipos padrões.

3.2.2 Integração dos mecanismos

Os mecanismos de extensão devem funcionar de forma integrada. A Figura 3.1 apresenta um cenário que mostra um possível inter-relacionamento destes mecanismos. Trata-se do armazenamento de *e-mails*. Foi definido um UDT *e-mail*, que é inserido em uma tabela, com os atributos: data de envio, data de chegada, remetente, assunto, destinatário e conteúdo. Uma *trigger* é responsável por ativar UDFs que dividem o texto nos atributos das tabela. Pode ser observada uma *constraint* que restringe que a data de chegada deve ser maior que a data de envio do *e-mail* e outra que impede o armazenamento de *e-mails* que retornaram por algum problema, caracterizados pelo conteúdo do assunto ser igual à "*undelivered"* (MATTOS, 1994).

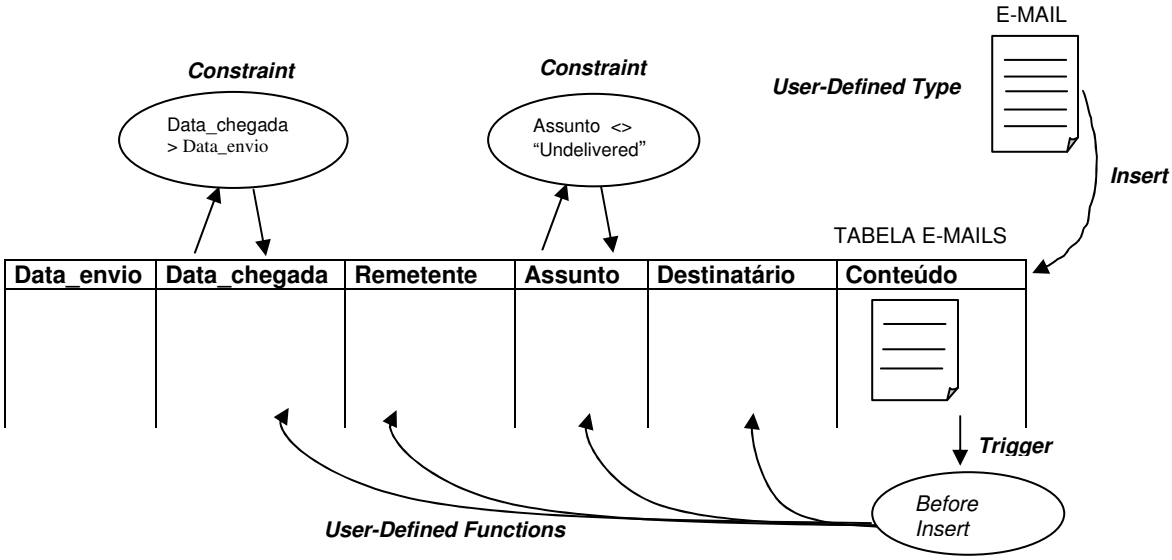


Figura 3.1: Sinergia entre os mecanismos

3.2.3 Funcionamento

O desenvolvedor une as facilidades dos mecanismos descritos para implementar um *extender* (RENNHACKKAMP, 1997). São definidos um ou mais UDTs obedecendo uma elaborada estrutura interna, com múltiplos atributos.

A visão lógica dos dados não precisa ter necessariamente a mesma forma da estrutura física de armazenamento. Os atributos internos e a estrutura podem ser camuflados através de uma interface. A interface consiste em um conjunto de UDFs, que recebem argumentos, e executam a recuperação, o armazenamento, a pesquisa e a manipulação dos atributos e da estrutura do UDT.

Como pode ser observado na Figura 3.2, os *extenders* constituem uma camada intermediária entre as aplicações do usuário e o os dados armazenados no SGBD. A *interface* e algumas funções formam a parte cliente das extensões. Já *triggers,* UDTs e UDFs ficam localizados junto ao servidor.

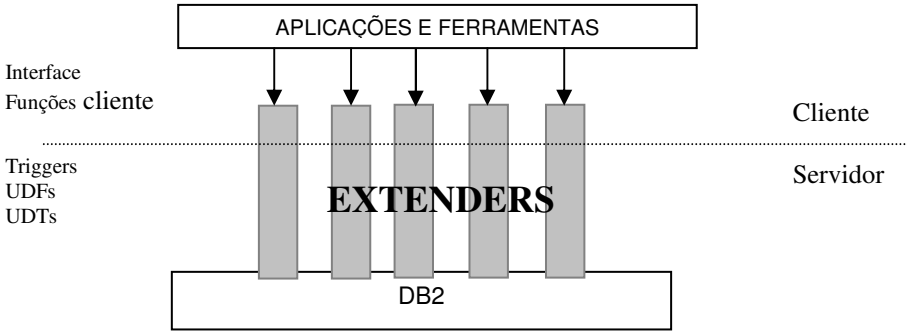


Figura 3.2: DB2 Extenders

Os *extenders* são armazenados de forma independente do banco de dados em si. Isto torna possível que sejam definidas extensões por vários desenvolvedores e instaladas apenas as que forem de interesse do usuário. Existem inúmeros *extenders* disponíveis para serem instalados como: *Text Extender*, *IAV (Image, Audio and Video) Extenders* e *XML Extender*.

Quando um *extender* é instalado no DB2, são criadas tabelas administrativas e apontadores para armazenar e acessar os tipos de dados por ele definidos. Estas tabelas contêm as informações necessárias para que o extender processe a requisição do usuário.

Algumas delas identificam as tabelas de usuário e colunas que estão habilitadas para um *extender*. Nelas são mantidas informações sobre atributos que são únicos. Por exemplo, no *Image Extender* são armazenadas informações como: largura, altura e número de cores de uma imagem, bem como dados de atributos como: a pessoa que importou o objeto, quem por último o alterou. LOBs podem ser armazenados nas próprias tabelas administrativas ou mantidos em arquivos separados e apenas com apontadores nestas tabelas.

3.3 Considerações Finais

O capítulo 3 apresentou alguns aspectos sobre o conceito de extensões de sistemas de gerenciamento de dados, principalmente os *extenders* do DB2. O usuário consegue adicionar novas funcionalidades ao banco de dados, utilizando apenas as características objeto-relacionais do banco.

4 TVM EXTENDER

A extensão do Modelo Temporal de Versões para o DB2 foi denominada *TVM Extender*. Através dessa extensão, o usuário pode especificar classes versionadas, com atributos e relacionamentos temporais, manipular os dados e fazer consultas que abrangem estes dois conceitos. O *TVM Extender* consiste de um conjunto de mecanismos (UDTs, UDFs, gatilhos, procedimentos armazenados e restrições) e metadados que mapeiam a hierarquia do TVM e gerenciam os aspectos de tempo e versão dos dados.

Primeiramente, as classes da hierarquia apresentada na Figura 2.2, além da hierarquia temporal, são mapeadas para tipos definidos pelo usuário, e os métodos definidos pelo TVM para UDFs. Isso possibilita que as classes especificadas pelo usuário possam herdar os atributos das classes predefinidas, aproveitando as características objeto-relacionais do DB2.

4.1 Interação do Usuário

A Figura 4.1 ilustra as funções exercidas pelo *extender* e suas respectivas interações com o usuário. Primeiramente, o usuário habilita o extender que implica na criação dos tipos de dados, funções e tabelas administrativas. Pode ser feita, então, a especificação das classes, atributos e relacionamentos por UDFs. Com as tabelas criadas, o usuário tem condições de manipular os dados diretamente nas tabelas criadas ou através de UDFs para versões e valores temporais. Por fim, outras UDFs que mapeiam métodos da hierarquia do modelo permitem a realização de consultas envolvendo os conceitos de tempo e versão.

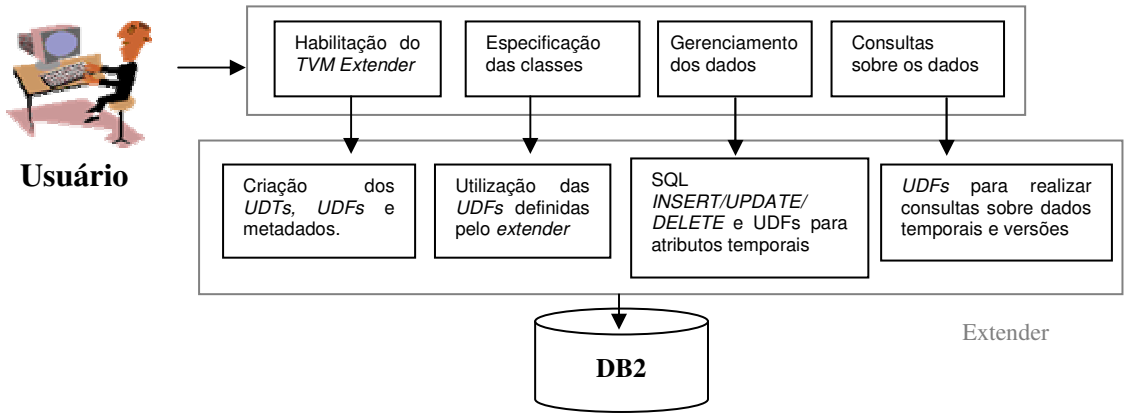


Figura 4.1: Estrutura do TVM Extender e Interações com o Usuário

Para que a tarefa de definição das classes, atributos e relacionamentos não se torne muito trabalhosa, ela pode ser realizada através de um ambiente gráfico denominado Ferramenta de Apoio à Especificação das Classes (MOROa, 2001) que fará o mapeamento da árvore das classes para as UDFs definidas. As demais manipulações devem ser efetuadas diretamente através das funções e tabelas geradas pelo *extender*.

4.2 Metadados

Inicialmente foram definidos os metadados que servem para o gerenciamento do TVM. Na Figura 4.2 é apresentado o diagrama UML que representa os metadados e permitem o armazenamento das informações sobre os esquemas, classes e seus atributos/relacionamentos. Estas classes foram mapeadas para tabelas, como pode ser observado no Anexo A.

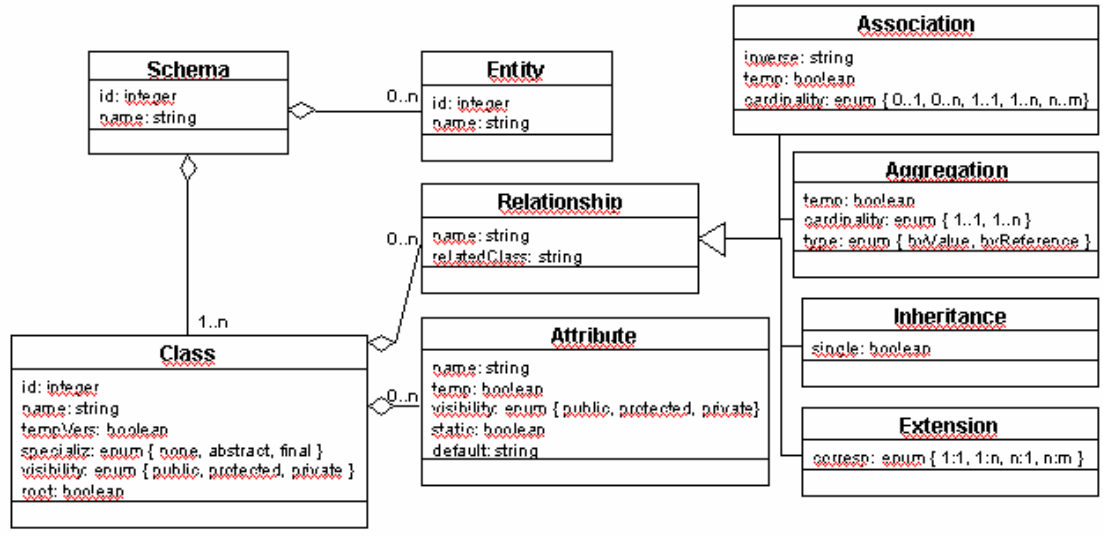


Figura 4.2: Metadados para Gerenciamento do TVM

Esses metadados correspondem a tabelas relacionais do *extender*. Nessa estrutura, a tabela `Entity` contém atributos para o identificador e nome da entidade. A tabela `Class` possui atributos para o identificador e o nome da classe, bem como para gerenciar se a classe é temporal versionada. Possui o atributo `generated` que determina se as tabelas que vão armazenar os dados da classe já foram geradas. Possui também um atributo para indicar se é uma raiz de uma hierarquia de herança e o identificador do esquema a que pertence (`schemaId`). A tabela `Relationship` possui atributos para seu nome, o identificador da classe a que pertence (`classId`) e da classe a que se relaciona (`relatedId`). Possui um campo para identificar o tipo de relacionamento (*Association, Aggregation, Inheritance* e *Extension*). De acordo com este tipo pode utilizar alguns dos outros campos definidos:

- *Associação (Association):* utiliza atributos para nome da referência inversa, se é temporal, e cardinalidade (*0..1*, *0..n*, *1..1*, *1..n*, *n..m*);
- *Agregação (Aggregation)*: utiliza atributos para armazenar se é um relacionamento temporal, cardinalidade (*1..1*, *1..n*) e tipo (por valor – `byValue`, ou por referência – `byReference`);
- *Herança (Inheritance)*: utiliza atributo para indicar se é herança simples ou múltipla;

- *Herança por Extensão (Extension)*: usa atributo para cardinalidade do relacionamento (*1:1*, *1:n*, *n:1*, *n:m*)

A tabela `Attribute` possui atributos para o nome, para indicar se é atributo temporal, para visibilidade, para determinar se é estático e valor *default*. Além destas, a classe `VersionedObjectControl` da hierarquia do TVM também exige um mapeamento com a finalidade de realizar o controle da ligação entre as versões de um objeto versionado. Essa classe é mapeada para uma tabela denominada VOC, com a estrutura apresentada na Figura 4.3, com as respectivas tabelas auxiliares para armazenar o histórico dos atributos temporais. Na seção 4.3.2, o mapeamento destes atributos temporais é especificado em detalhes.

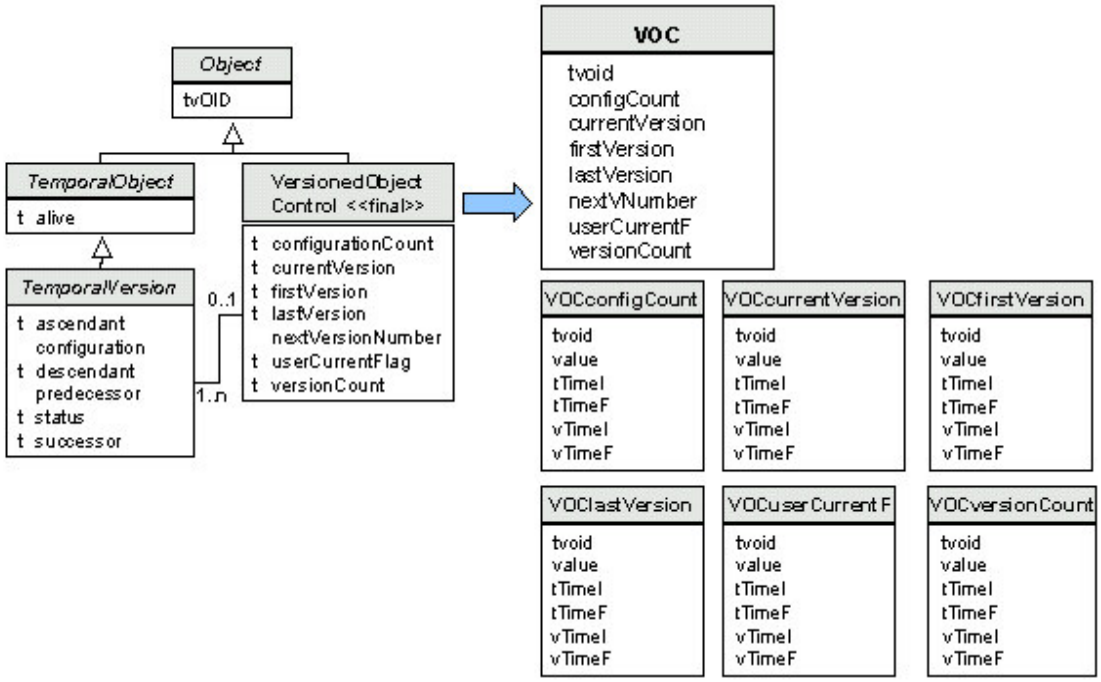


Figura 4.3: Estrutura da Tabela VOC e Tabelas Auxiliares

No Anexo A podem ser encontrados os comandos SQL necessários para a criação das tabelas resultantes do mapeamento da classe `VersionedObjectControl` e dos outros metadados apresentados na Figura 4.2.

4.3 Mapeamento da Hierarquia do TVM

Para realizar o mapeamento da hierarquia inicialmente os tipos de dados básicos utilizados pelo TVM são mapeados para os tipos de dados do DB2, conforme a Tabela 4.1.

Tabela 4.1: Mapeamento dos tipos de dados

TVM	DB2
Integer	Integer
Real	Real
String	Varchar
Char	Char
boolean	char (1) CONSTRAINT cname CHECK (NomeColuna IN ('T', 'F'))
Date	Date

TVM	DB2
Time	Time
Instant	Timestamp
Oidt	Varchar (20)
Set	-

Em seguida, a hierarquia de classes do Modelo Temporal de Versões é mapeada para tipos equivalentes no DB2. Isso permite que sejam criados UDTs para as classes, baseados na hierarquia e sejam geradas tabelas tipadas através destes UDTs. Assim, as classes de aplicação, através do *extender*, podem utilizar as características de tempo e versão. O TVM define métodos para todas as operações que podem ser aplicadas a um objeto, porém muitas delas podem ser realizadas através de comandos SQL. Os principais métodos das classe da hierarquia são apresentados na Figura 4.4. Desse modo, apenas os métodos referentes ao tratamento das características de tempo e versões associadas aos objetos devem ser mapeados.

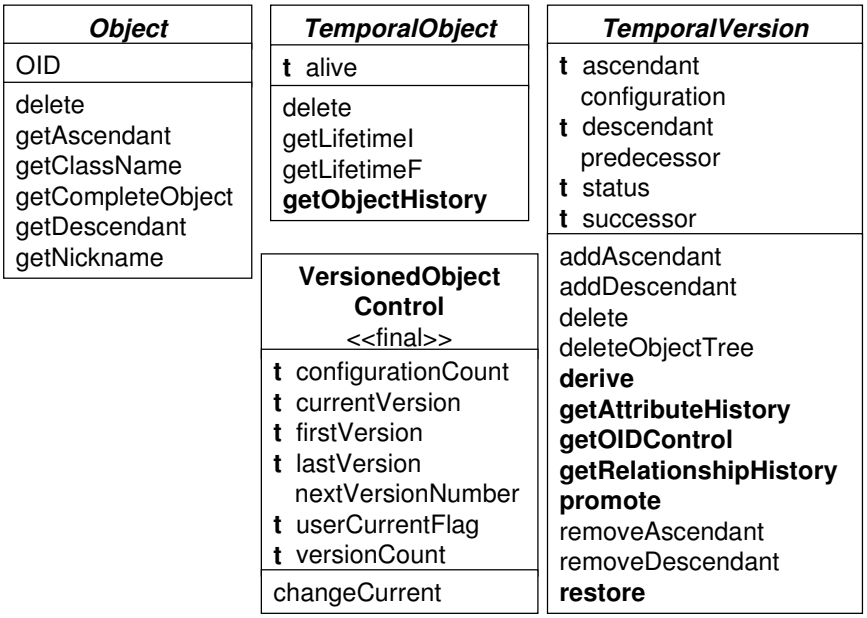


Figura 4.4: Classes da hierarquia com principais atributos e operações

Os comandos de criação dos principais tipos da hierarquia do TVM são apresentados na Figura 4.5. O identificador `tvOID` só é adicionado ao tipo de dados no instante a criação da tabela tipada, conforme exige a sintaxe do DB2.

```
CREATE TYPE Object NOT INSTANTIABLE

CREATE TYPE TemporalObject UNDER Object ( alive  char(1))  NOT INSTANTIABLE

CREATE TYPE TemporalVersion UNDER TemporalObject (status  char(1), refVOC OIDt )
NOT INSTANTIABLE
```

Figura 4.5: Comandos para criar os UDTs principais da hierarquia

4.3.1 Identificadores de Objetos

O DB2 exige que tabelas baseadas em tipos (`typed tables`) tenham um identificador único do tipo `REFERENCE` para o tipo da tabela (IBMc, 2000). O valor do identificador é inserido através do construtor do UDT e armazenado de forma tipada na tabela. A estrutura de OID utilizada no TVM foi mapeada, de acordo com o proposto, para uma string que é inserida utilizando o construtor do tipo de dados da sua classe. As suas três partes <id entidade, id classe, nro versão> são armazenadas em um único atributo do `REFERENCE`, que por sua vez mapeia um `varchar(16)` com os valores separados por vírgulas. Por exemplo, na Figura 4.6 (a) é criado um tipo de dados para a classe ENDERECO, com os atributos RUA e NUMERO; em (b) é criada a `typed table` que armazenará as instâncias da classe e tem como identificador `tvOID`; já em (c) é realizada uma operação de inserção na tabela, que deve utilizar o tipo (`endereco_t`) para o valor e `tvOID`; e pr fim em (d) é apresentado um exemplo de consulta a esta tabela para recuperar a instância que tem um determinado `tvOID`, para isso é preciso converter o valor com o uso da função `varchar()`.

(a)

```
CREATE TYPE endereco_T UNDER Object
ALTER TYPE ADD ATTRIBUTE rua
varchar(60)
```

(b)

```
CREATE TABLE endereco OF endereco_T
(REF is tvOID USER GENERATED)
MODE DB2SQL
```

(c)

```
INSERT INTO endereco (tvOID, rua,
numero)VALUES (endereço_T('1,4,1'),
'Av. Sete de Setembro', 454)
```

(d)

```
SELECT * FROM endereco WHERE
varchar(tvOID) = '1,4,1'
```

Figura 4.6: Exemplo do OID em uma *Typed Table*

O identificador da entidade é gerado pelo sistema. Em hierarquias de herança ele é igual para representar o mesmo objeto nos vários níveis da respectiva hierarquia. O identificador da classe também é uma informação mantida pelo sistema, através da tabela de controle contendo informações sobre as classes da aplicação. Assim, quando um objeto não versionado[2], versionado ou uma versão for criada, dependendo da classe a qual pertencer, é utilizado o identificador da classe contido nesta tabela de controle para gerar o `tvOID`. O número de versão é controlado pela classe `VersionedObjectControl`, através do atributo `nextVersion`.

4.3.2 Mapeamento das Classes de Aplicação e Valores Temporais

Depois de realizado o mapeamento das classes do TVM, a próxima etapa envolve o mapeamento das classes de aplicação. Nesse caso, para manter as características temporais dos atributos e relacionamentos, foram vislumbradas três maneiras para mapear as classes: (i) armazenando os rótulos temporais por tuplas; (ii) gerando todo o histórico em uma mesma tabela; ou (iii) criando uma tabela auxiliar para representar cada um dos atributos e relacionamentos temporais. Essas alternativas serão detalhadas a seguir.

[2] O conceito de objeto versionado, não versionado e versões foi apresentado na seção 2.4.3.

4.3.2.1 Armazenamento dos rótulos temporais por tuplas

Na alternativa (i) a idéia é deixar os rótulos temporais dos atributos na tabela à qual pertencem. Assim, para cada atributo/relacionamento é criado um conjunto de rótulos temporais para registrar seus tempos de validade e transação (inicial e final) na tabela principal. Na Figura 4.7 é mostrada como seria a tabela de uma classe `Aluno` com dois atributos temporais (`nome` e `endereco`), que possuem rótulos temporais.

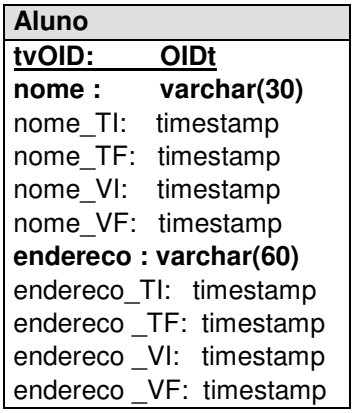


Figura 4.7: Armazenamento dos Rótulos Temporais na Tabela Principal

Nessa alternativa, muitas informações redundantes são armazenadas na tabela, pois cada vez que um atributo temporal mudar de valor, toda a tupla é copiada. Como a maior parte das consultas é realizada sobre os valores atuais, essa alternativa pode comprometer o desempenho no caso das consultas atuais, por aumentar significativamente o volume de informação armazenada na tabela mapeada.

4.3.2.2 Armazenamento dos rótulos temporais em somente uma tabela auxiliar

Já na alternativa (ii), a idéia é criar uma tabela auxiliar onde todos os dados históricos ficariam armazenados, independentemente da classe à qual pertencem e do tipo da propriedade. A Figura 4.8 ilustra esta forma de mapeamento. A tabela `Temporal` é composta pelo `tvOID`, o nome do atributo temporal (`nomeAtributo`), o valor do atributo e os rótulos temporais, e é responsável por armazenar todos os valores de todos os atributos temporais da aplicação.

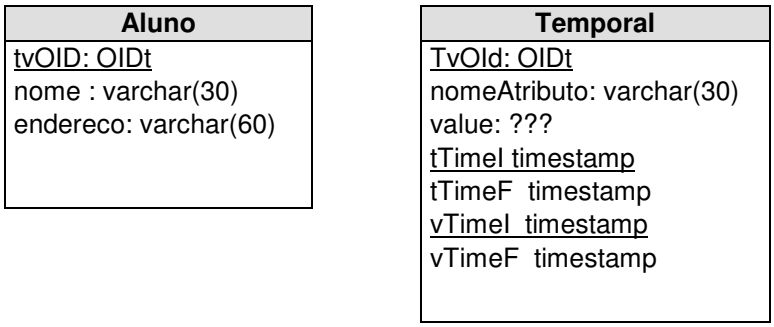


Figura 4.8: Armazenamento de Todos os Históricos na Mesma Tabela

Entretanto essa alternativa também é inviável, principalmente porque o domínio dos tipos dos atributos varia. Assim o atributo `value` da tabela `Temporal` teria que ser de um tipo genérico o bastante para suportar qualquer um dos tipos de atributos existentes no DB2. Além disso, esta alternativa poderia comprometer o desempenho quando a consulta envolvesse informações históricas do banco de dados devido ao volume de

informações armazenadas na tabela `Temporal`, uma vez que o histórico de toda a aplicação estaria armazenado nesta tabela.

4.3.2.3 Armazenamento em Tabelas Auxiliares para cada Atributo/Relacionamento

A terceira alternativa consiste também na criação de uma tabela para cada classe da aplicação, no entanto é criada uma tabela auxiliar para cada atributo temporal presente nas classes a serem mapeadas. A tabela auxiliar armazena o `tvOID` da tabela principal, o valor histórico do atributo e os rótulos temporais. Para facilitar o gerenciamento, seu nome é formado pelo nome da tabela principal mais o nome do atributo temporal (Figura 4.9).

Dessa forma, a tabela principal possui colunas que representam os atributos (temporais ou não temporais), além dos atributos necessários para o controle das versões. Como para cada atributo temporal existe uma tabela auxiliar relacionada, o problema de domínio de valores, apresentado na alternativa anterior, está solucionado.

Aluno
tvOID: OIDt
nome : varchar(30)
endereco: varchar(60)

AlunoNome
tvOID: OIDt
value: varchar(30)
tTimeI timestamp
tTimeF timestamp
vTimeI timestamp
vTimeF timestamp

AlunoEndereco
tvOID: OIDt
value: varchar(60)
tTimeI timestamp
tTimeF timestamp
vTimeI timestamp
vTimeF timestamp

Figura 4.9: Rótulo temporal representado por tabelas auxiliares

Quanto ao armazenamento das informações atuais, foram vislumbradas duas alternativas:

a) armazenar o valor atual do atributo em um campo de mesmo nome na tabela principal e os demais valores na tabela auxiliar; ou

b) armazenar os valores do atributo apenas na tabela auxiliar. A tabela principal nem teria o campo associado ao atributo, uma vez que os valores são recuperados por UDFs.

Utilizando a opção (a) não é possível garantir que o valor que está na tabela seja realmente o valor atual. Como o usuário pode definir que valores comecem a valer em um tempo futuro, pode ser que uma informação se torne válida e o campo na tabela principal não tenha sido atualizado. Entretanto, o valor atual estando na tabela principal facilitaria a performance das pesquisas, uma vez que a maioria das consultas é feita sobre esses valores. Assim, para valores atuais, desconsiderando o histórico, o usuário poderia utilizar um simples `SELECT` na tabela para obter os dados.

Se existissem gatilhos que pudessem ser ativadas em um determinado tempo, esta alternativa seria viável, pois estes seriam utilizados para alterar o valor atual do atributo temporal na tabela principal quando o mesmo começasse a valer. Assim, os valores atuais representariam corretamente o contexto sem correr o risco de se tornarem inconsistentes. Mas, não foram encontradas referências no DB2 a um tipo similar de gatilho que não seja disparado por um evento (`INSERT, UPDATE, DELETE`), e sim em um determinado tempo.

Já com a alternativa (b) é possível verificar exatamente o valor que está valendo no momento em que a tabela auxiliar for consultada. No entanto, as tabelas só podem ser acessadas através de funções, o que deve acarretar uma performance mais deficiente em nas consultas.

Levando em consideração estes aspectos, o TVM *Extender* utiliza uma solução híbrida. Os valores são armazenados apenas na tabela auxiliar, mas o atributo temporal também existe na tabela principal mesmo que sem receber um valor (nulo). Este campo serve para ativar gatilhos que serão ativados quando o usuário tentar inserir, atualizar e apagar valores deste atributo temporal. Para realizar uma consulta, entretanto, é necessário utilizar uma UDF que retorne o valor corrente do atributo ou realizar um `SELECT` na tabela principal em junção com as tabelas auxiliares pelo campo `tvOID`, fazendo as restrições necessárias.

4.3.3 Atributos para Controle das Versões

Para o atributo `status` é criada uma tabela auxiliar. Os atributos `ascendant`, `descendant`, `successor` e `predecessor`[3] não foram mapeados para atributos, pois o DB2 não suporta coleções. Estes atributos foram mapeados para tabelas de controle `AscDesc` e `PredSuc`, que armazenam os pares de OIDs correspondentes, além de todo o histórico das modificações. Essas tabelas armazenam os pares predecessor-sucessor e ascendente-descendente de todas as classes especificadas, com os respectivos rótulos temporais. As chaves primárias são compostas dos pares e dos tempos de validade e de transação iniciais.

O controle destas tabelas é realizado pelo sistema. Somente na instanciação e na derivação são armazenados os predecessores e sucessores, bem como os ascendentes e descendentes. Depois de incluídos na tabela, esses valores só podem ser alterados no momento da exclusão da versão, encerrando os tempos de validade dos pares nos quais a versão excluída está presente. Essas tabelas só podem ser alteradas pelo sistema, ou seja, pelos gatilhos e procedimentos armazenados de gerenciamento. A Figura 4.10 mostra a estrutura das tabelas `PredSucc` e `AscDesc`.

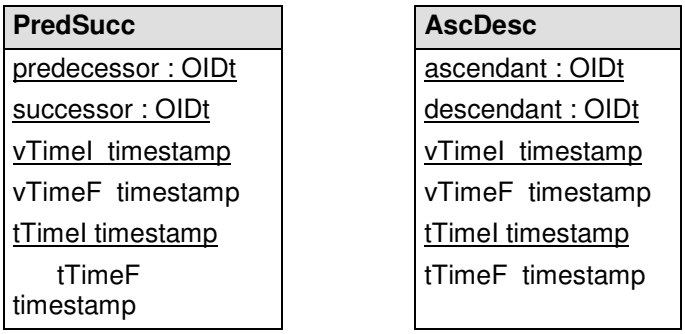


Figura 4.10: Estrutura das Tabelas PredSucc e AscDesc

Foi necessário ainda realizar um mapeamento dos relacionamentos disponíveis em um modelo orientado-objeto para um banco de dados relacional, que permite apenas relacionamentos normais, variando apenas a cardinalidade. Os relacionamentos de agregação/composição e herança são mapeados para um relacionamento normal, no qual a classe agregada ou subclasse recebe a chave primária da classe principal como chave estrangeira. Na herança simples, as subclasses herdam os atributos da superclasse, ou seja, todos os atributos e relacionamentos da superclasse são adicionados na subclasse. Se a superclasse for abstrata, ela não é instanciada e, portanto, não gera tabela no esquema relacional. Por outro lado, se a superclasse não for abstrata, o seu

[3] Os conceitos de `ascendant`, `descendant`, `successor` e `predecessor` foram definidos nas seções 2.4.3 e 2.4.6.

mapeamento segue as regras definidas para as classes, conforme descritos anteriormente.

Para cada atributo ou relacionamento temporal é criada uma tabela auxiliar para armazenar o histórico dos valores. A Figura 4.11 ilustra a estrutura básica destas tabelas. O nome da tabela é composto pelo nome da classe e o nome do atributo/relacionamento. Cada linha na tabela armazena o identificar do objeto (que referencia o objeto na tabela principal), o valor, os tempos de transação (inicial e final) e os tempos de validade (inicial e final). A chave que identifica cada registro é formada pelo identificador e pelos tempos iniciais de transação e de validade.

Na tabela principal de cada classe ainda é inserido um atributo `refVOC` (chave estrangeira) para referenciar o identificador (`tvOID`) da tabela de controle dos objetos versionados (VOC).

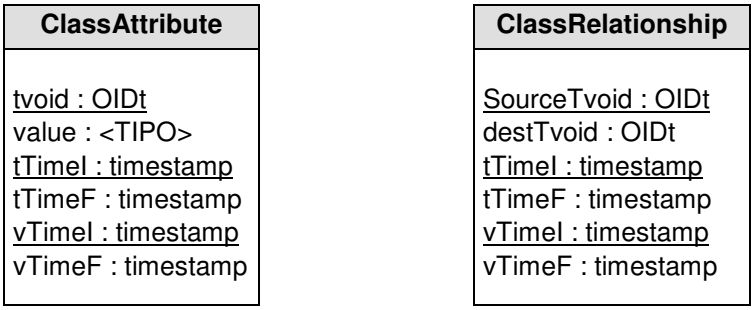


Figura 4.11: Tabelas Auxiliares de Atributos/Relacionamentos Temporais

4.4 Especificação das Classes

Para a especificação dos esquemas, das classes e dos relacionamentos do TVM são utilizados procedimentos armazenados, pois estes devem ser invocados independentemente de uma instrução SQL como é o caso das UDFs.

O DB2 oferece uma funcionalidade que possibilita criar uma tabela especial, denominada `typed table`, a partir de um tipo de dados definido pelo usuário. Os atributos do UDT são colunas da tabela e podem ser manipulados como informações relacionais. As funções descritas nesta seção, além de inserir as informações necessárias nos metadados, criam as `typed tables` para as classes, incluindo todos os gatilhos, restrições e, ainda, tabelas auxiliares para os atributos e relacionamentos temporais.

No DB2 depois que uma `typed table` é gerada, baseada em um UDT, sua estrutura não pode mais ser modificada. Devido a este fato, restringimos que o usuário defina a sua classe, com os atributos e relacionamentos e execute um comando para gerá-la. A partir deste momento ela não pode mais ter sua estrutura modificada.

Um trabalho futuro poderia estender o que está sendo definido, permitindo que o usuário modifique uma classe depois que as tabelas tenham sido geradas. Para isso, deverá realizar alguns estudos de como modificar o UDT, eliminar a `typed table`, criá-la novamente, sem perder os dados nela já incluídos utilizando uma tabela auxiliar.

4.4.1 Criação de Esquemas

O usuário pode especificar um esquema que reunirá todos as classes de uma determinada aplicação. Para realizar esta tarefa, ele deve utilizar o procedimento armazenado `createSchema`. Quando é executado, ocorre uma verificação da próxima identificação disponível de um esquema, através da tabela `Schema` nos metadados. Os

dados do esquema são inseridos nesta tabela e o valor do identificador `schemaId` é retornado ao usuário.

4.4.2 Definição de Classes

O primeiro passo para a especificação de uma classe é a definição de suas características básicas. Quando o usuário for definir a classe terá de especificar se ela é temporal versionada, a qual esquema ela pertence e se é raiz de alguma hierarquia. O sistema verifica se a classe foi definida como temporal versionada. Em caso positivo, cria um tipo de dados que represente aquela classe como subclasse de `temporalVersion`. Caso contrário, a classe é criada abaixo de `Object`. Obtém o próximo identificador de classe realizando uma consulta à tabela `Class`. E por fim, insere os dados da classe nesta tabela. O atributo `generated` foi adicionado para o sistema controlar quais classes já foram geradas ou não, no momento de invocação do procedimento `generateClasses`. Inicialmente o atributo `generated` recebe o valor *false.*

O próximo passo é a definição dos atributos e relacionamentos de uma classe. Para isso, o usuário deve utilizar os procedimentos `addAttribute` e `addRelationship`, que são descrito em detalhes nas seções seguintes.

4.4.2.1 Inclusão de Atributos

Quando o usuário invoca o procedimento `addAttribute` deve especificar o tipo do atributo, se é temporal e o seu valor *default*. Primeiramente é verificado se a classe está definida, executando uma consulta na tabela `Class` com o identificador de classe recebido por parâmetro. Se o resultado for vazio, a classe ainda não está definida. É necessário verificar ainda se a classe não foi gerada, para isso basta consultar o atributo `generated` da tabela `Class` que deverá ser *false*. Como o UDT desta classe já vai estar criado é necessário executar um comando `Alter Type add attribute` para incluir o atributo que está sendo definido. E por fim, os dados do atributo são incluídos nos metadados (tabela `Attribute`). A diferença de um atributo temporal e não temporal é concretizada no procedimento `generateClasses` definido na seção 4.4.3, que gerará tabelas auxiliares para os atributos temporais.

4.4.2.2 Inclusão de Relacionamentos

O procedimento `addRelationship` permite que o usuário adicione um relacionamento de uma determinada classe para outra. Dentre as características de um relacionamento estão: o nome do relacionamento, seu tipo, se é temporal ou não e sua cardinalidade. Ao ser invocado, o procedimento executa as mesmas verificações do procedimento `addAttribute` para certificar-se de que as classes estão definidas e ainda não foram geradas. Utiliza o comando `Alter Type add Attribute` para incluir o campo que fará o relacionamento entre as tabelas. No final, as características do relacionamento são inseridas na tabela `Relationship` nos metadados, para que possam ser utilizadas no momento de geração das classes.

4.4.3 Geração das Classes

O procedimento `generateClasses` é responsável por gerar as tabelas referentes às classes e tabelas auxiliares para atributos e relacionamentos temporais. Quando este procedimento é executado são criadas as tabelas baseadas nos tipos já definidos pelas

outras *stored procedures*, bem como as tabelas auxiliares para os atributos e relacionamentos temporais.

Primeiramente, é feita uma verificação de quais classes não foram geradas, criando as tabelas destas, bem como as tabelas auxiliares para atributos e relacionamentos temporais. É necessário ainda criar gatilhos e restrições para que o usuário possa utilizar apenas a tabela principal e os dados sejam inseridos/atualizados e eliminados das tabelas auxiliares.

Depois que as classes são geradas, elas não podem mais ser alvo de modificações em sua estrutura devido à sintaxe do DB2. Entretanto, novas classes podem ser definidas e o procedimento `generateClasses` executado novamente.

Para cada classe definida, é necessário:

gerar a tabela baseada no UDT da classe;

verificar se existem atributos com o campo `TEMP` ativado, e para cada um deles gerar uma tabela auxiliar identificada pelo nome da classe + nome do atributo. Esta tabela auxiliar conterá o `tvOID`, o valor, os rótulos temporais de validade e de transação;

gerar os gatilhos necessários para restringir que o usuário manipule diretamente os dados das tabelas auxiliares, além dos gatilhos nas tabelas principais responsáveis para a cada modificação nos atributos e relacionamentos temporais, os respectivos valores sejam incluídos nas tabelas auxiliares, inicializando os rótulos temporais de forma coerente;

alterar a tabela `Class` nos metadados, atribuindo *true* para o campo `generated` da respectiva classe, que não poderá mais ser alteradas.

4.4.4 Sintaxe

Na Tabela 4.2, é possível observar a sintaxe dos procedimentos armazenados utilizados na especificação das classes e descritos nas seções anteriores.

Tabela 4.2: Sintaxe dos Procedimentos de Especificação das Classes

Função	Sintaxe
Criar um esquema	`createSchema (IN schemaName VARCHAR(30),` `OUT schemaId)`

Definir uma classe	`defineClass(IN className VARCHAR(30),`
	`IN schemaID integer,`
	`IN tempVers char(1),`
	`IN root char(1),`
	`OUT classId integer)`

Adicionar um atributo à classe	`addAttribute (IN classId INTEGER,`
	`IN name VARCHAR(30),`
	`IN type VARCHAR(30),`
	`IN temp CHAR(1),`
	`IN visibility VARCHAR(30),`
	`IN static VARCHAR(4),`
	`IN default VARCHAR(30))`

Adicionar um relacionamento à classe	`addRelationship (IN sourceClass INTEGER,`
	`IN destClass INTEGER,`
	`IN name VARCHAR(30),`
	`IN type VARCHAR(30),`
	`IN temp CHAR(1),`

Função	Sintaxe
	IN cardinality VARCHAR(4), IN inverse VARCHAR(30), IN single VARCHAR(1))
Gerar as classes definidas	`generateClasses()`

4.5 Manipulação dos Dados

Com as classes especificadas, o usuário manipula seus dados diretamente nas tabelas geradas pelo *extender*, utilizando UDFs (que modelam os métodos das classes do TVM) para manipular as versões. Para atributos e relacionamentos temporais o usuário deve utilizar outras funções que recebem além do valor, os rótulos temporais. Estas UDFs são responsáveis por verificar os tempos de validade para inserir/atualizar o valor apenas na tabela auxiliar ou também na tabela principal.

Quando o usuário tenta inserir ou atualizar um valor de um atributo/relacionamento temporal, as verificações expostas nas seções 2.4.2.1 e 2.4.2.2 devem ser executas. Cada uma das condições apresentadas na Tabela 2.1 foi mapeada para uma *trigger* do tipo `AFTER INSERT` ou `AFTER UPDATE`. Somente gatilhos deste tipo permitem executar comandos de `UPDATE` para atualizar os tempos de validade dos valores antigos para que não se tornem incoerentes.

4.5.1 Atributos e Relacionamentos Temporais

A Figura 4.12 ilustra as classes para manipular atributos e relacionamentos temporais definidos no TVM.

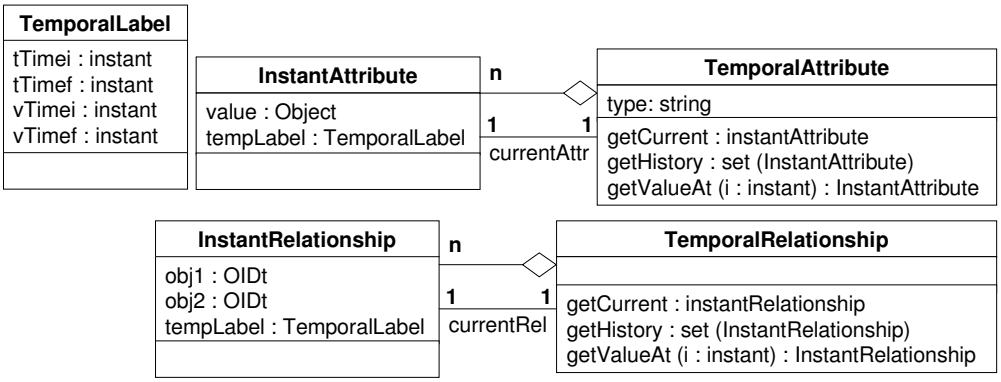
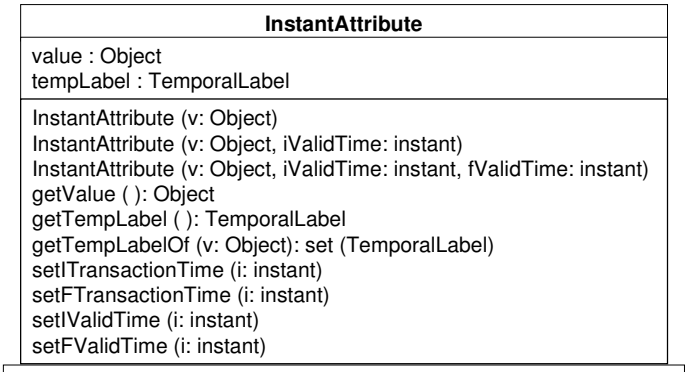


Figura 4.12: Classes de atributos e relacionamentos temporais do TVM

No mapeamento para o *TVM Extender,* a classe `TemporalLabel` foi mapeada para conjuntos de atributos do tipo `timestamp` (`vTimeI, vTimeF, tTimei, tTimeF`) que são incluídos nas tabelas auxiliares que armazenam o histórico dos atributos e relacionamentos.

As classes `InstantAttribute` e `TemporalAttribute` (Figura 4.13) foram mapeadas para os campos dos atributos na tabelas principal da classe e para as tabelas auxilares que armazenam o histórico dos atributos criadas pelo procedimento `generateClasses`.

(a)



(b)

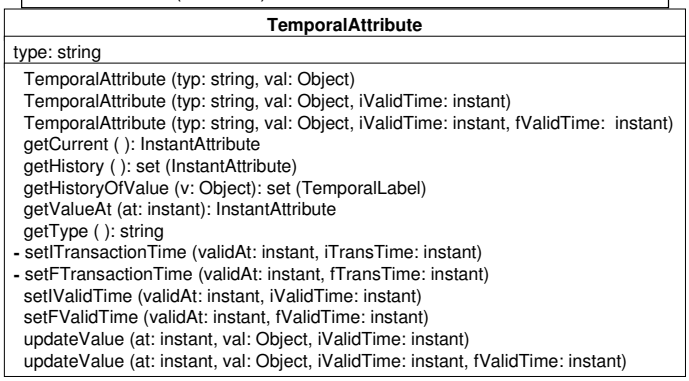


Figura 4.13: Classes *InstantAttribute* e *TemporalAttribute* do TVM

Para permitir que o usuário especifique os valores temporais sem ter que manipular diretamente as tabelas auxiliares é necessário mapear os métodos para inserir os valores com os rótulos temporais. Este mapeamento pode ser observado na Tabela 4.3.

Tabela 4.3: Mapeamento dos Métodos para Atributos Temporais

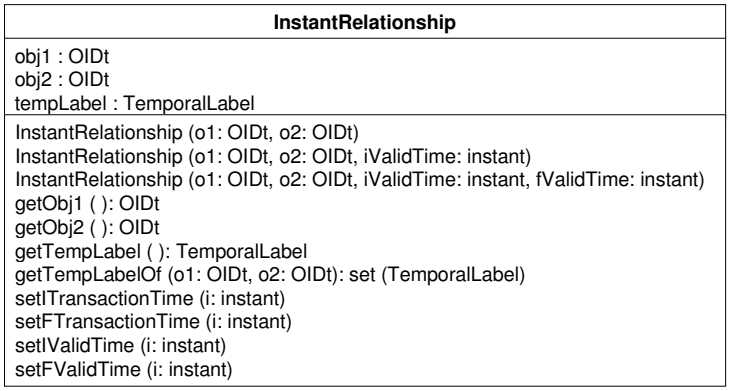
Método no TVM	**Mapeamento no *TVM Extender***
`InstantAttribute(..)`	Mapeado para o comando de inserção informando o valor do atributo que corresponde a uma coluna da tabela da classe.
`TemporalAttribute(..)`	Mapeado para o comando de inserção, tentando inserir o valor e rótulos temporais de validade na coluna do atributo temporal. O campo do atributo temporal é do tipo *string* e deve obedecer ao seguinte formato: **"value|vTimeI|vTimeF|"**. Uma *trigger* `AFTER INSERT` é responsável por converter o valor para seu tipo definido nos metadados e inseri-lo na tabela auxiliar, com os rótulos temporais de validade, encerrando os tempos finais se já houver um valor anterior. O rótulo de transação inicial é inserido com o instante atual. Os rótulos temporais são opcionais. Seus valores podem ser deixados em branco (Exemplo: "João||"). Se `vTimeI` não for informado, será considerado o instante atual e se `vTimeF` não for informado será considerado como nulo(infinito). Os tempos de transação não são modificados pelo usuário, apenas pelo sistema.
`getValue()`	Não foi mapeado, pois só faz sentido em um modelo OO.
`getCurrent()`	**`getCurrent (tvoid, attribute)`** – UDF que retorna o valor corrente de determinado atributo ou um `SELECT` na tabela principal da classe com uma junção da tabela auxiliar pelo atributo `tvOID` com a restrição `vTimef` igual a nulo.
`getHistory()` `getHistoryOfValue()`	**`getHistory (tvOID, attribute)`** – UDF que retorna todos os valores do atributo de um determinado objeto ou um `SELECT` na tabela principal da classe com uma junção da tabela auxiliar pelo atributo `tvOID` sem restringir os tempos de validade.

Método no TVM	Mapeamento no *TVM Extender*
`getValueAt()`	**`getValueAt (tvOID, attribute, vTimeI, vTimeF)`** – UDF que retorna o valor de um atributo que era válido em determinado tempo ou um `SELECT` na tabela principal da classe com uma junção da tabela auxiliar pelo atributo `tvOID` com a restrição do período dos tempos de validade compreender o instante desejado.
`getTempLabel()` `getTempLabelOf()`	**`getTempLabel (tvOID, attribute)`** – UDF que retorna os rótulos temporais de um atributo retornando os valores concatenados, ou um `SELECT` na tabela principal da classe com uma junção da tabela auxiliar pelo atributo `tvOID` com a restrição do valor desejado.
`getType()`	**`getType (class, attribute)`** – Procedimento armazenado que retorna o tipo do atributo temporal, consultando os metadados em um parâmetro do tipo saída.

`setIValidTime()`
`setFValidTime()`
`updateValue()`
	Estes métodos foram mapeados para o comando de `UPDATE` sobre o campo do atributo temporal passando um valor no seguinte formato : **`"value|vTimeI|vTimeF|"`**. Um gatilho do tipo `AFTER UPDATE` é responsável por separar os valores e atualizar a tabela que armazena o histórico do campo. Os tempos de transação não são modificados pelo usuário, apenas pelo sistema. Foi convencionado que se o usuário não deseja modificar o valor de alguma destas propriedades, deve deixa-la em branco. O usuário pode apenas querer modificar um dos rótulos temporais (Ex: `"|10/05/2003||"`). Se o valor do atributo for informado, é obrigatório informar também pelo menos o `vTimeI`.

As funções que atualizam os rótulos temporais devem seguir as regras temporais apresentadas na Tabela 2.1. Os métodos `getHistoryOfValue` e `getHistory` forma mapeados para uma função única `getHistory` que também faz o papel da primeira, pois recebe um objeto por parâmetro. O mesmo ocorre para os métodos `getTempLabel` e `getTempLabelOf`.

(a)



(b)

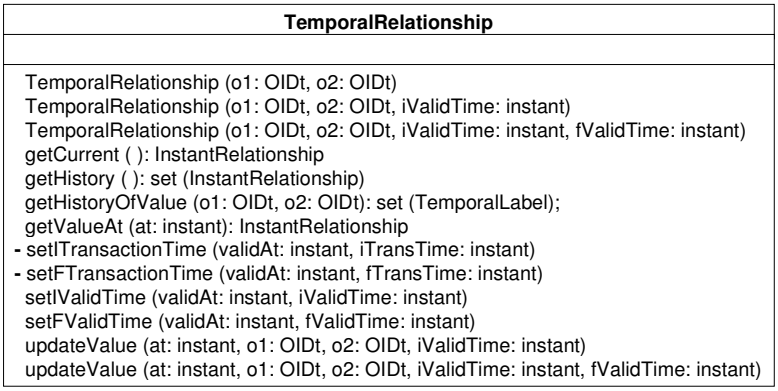


Figura 4.14: Classes *InstantRelationship* e *TemporalRelationship* do TVM

Analogamente, `InstantRelationship` e `TemporalRelationship` (Figura 4.14) foram mapeados os campos que representam os relacionamento na tabela principal e principalmente, para as tabelas auxiliares que registram o histórico daquele relacionamento.

Também de forma similar, devem ser mapeados os métodos essenciais para permitir a definição, atualização e obtenção dos valores ou rótulos temporais os relacionamentos, como pode ser observado na Tabela 4.4.

Tabela 4.4: Mapeamento dos Métodos para Relacionamentos Temporais

Método no TVM	**Procedimento ou função no *TVM Extender***
`InstantRelationshp(..)`	Mapeado para o comando de inserção informando o valor do relacionamento que corresponde a uma coluna da tabela da classe.
`TemporalRelationship(..)`	Mapeado para o comando de inserção, tentando inserir o valor e rótulos temporais na coluna do relacionamento temporal. O campo do atributo temporal é do tipo *string* e deve obedecer ao seguinte formato: **`"value|vTimeI|vTimeF|".`** Uma *trigger* `AFTER INSERT` é responsável por converter o valor para seu tipo definido nos metadados e inseri-lo na tabela auxiliar, com os rótulos temporais de validade, encerrando os tempos finais se já houver um valor anterior. Os rótulos de transação correspondentes são inseridos com o instante atual. Os rótulos temporais são opcionais. Se `vTimeI` não for informado, será considerado o instante atual e se `vTimeF` não for informado será considerado como nulo(infinito).
`getObj1()` `getObj2()`	Não foram mapeados, pois são basicamente métodos para uma modelagem OO.
`getCurrent()`	**`getCurrent (tvoid, relationship)`** – UDF que retorna o valor corrente do relacionamento ou um `SELECT` na tabela principal da classe com uma junção da tabela auxiliar pelo atributo `tvOID` com a restrição `vTimef` igual a nulo.
`getHistory()` `getHistoryOfValue()`	**`getHistory (tvOID, relationship)`** – UDF que retorna todos os valores do relacionamento de um determinado objeto objeto ou um `SELECT` na tabela principal da classe com uma junção da tabela auxiliar pelo atributo `tvOID` sem restringir os tempos de validade.
`getValueAt()`	**`getValueAt (tvOID, relationship, vTimeI, vTimeF)`** – UDF que retorna o valor de um atributo que era válido em determinado tempo ou um `SELECT` na tabela principal da classe com uma junção da tabela auxiliar pelo atributo `tvOID` com a restrição do período dos tempos de validade compreender o instante desejado.
`getTempLabel()` `getTempLabelOf()`	**`getTempLabel (sourceTvOID, destTvOID, attribute)`** – UDF que retorna os rótulos temporais de um relacionamento, ou um `SELECT` na tabela principal da classe com uma junção da tabela auxiliar pelo atributo `tvOID` com a restrição do valor desejado.
`getType()`	***`getType (class, relationship)`*** – Procedimento armazenado que retorna o tipo do relacionamento temporal através de um parâmetro de saída, consultando os metadados.

Método no TVM	Procedimento ou função no *TVM Extender*

`setFValidTime()`
`setFValidTime()`
`updateValue()`
	Estes métodos foram mapeados para o comando de `UPDATE` sobre o campo do relacionamento temporal passando um valor no seguinte formato: **`"value|vTimeI|vTimeF|"`**. Um gatilho do tipo `AFTER UPDATE` é responsável por separar os valores e atualizar a tabela que armazena o histórico do campo. Foi convencionado que se o usuário não deseja modificar o valor de alguma destas propriedades, deve deixa-la em branco. O usuário pode apenas querer modificar um dos rótulos temporais (Ex: `"|10/05/2003||"`). Se o valor do relacionamento for informado, é obrigatório informar também pelo menos o `vTimeI`.

4.5.2 Instanciação das Classes

No *TVM Extender* os métodos construtores das classes `Object` e `TemporalVersion`, que criam respectivamente instâncias de objetos não versionados e versionados, são mapeados para comandos de inserção nas tabelas criadas pela função `generateClasses()` da especificação de classes, com os gatilhos necessários.

Para criar uma instância de um objeto não versionado através do Modelo TVM, o método construtor `Object()` é executado, passando os parâmetros necessários. Originalmente no TVM o construtor poderia ser invocado de quatro formas diferentes, como apresentado na Tabela 4.5.

Tabela 4.5: Métodos Construtores de Objetos Não Versionados

Item	Construtor	Descrição
a)	`Object()`	Cria um objeto obtendo o próximo valor do OIDt dos metadados.
b)	`Object(ascendantID: OIDt)`	Cria um objeto relacionando-o com o seu ascendente na hierarquia.

c)	`Object(entityId: integer,`
	`classId: integer,`
	`versionId: integer)`
		Cria um objeto com o OIDt especificado realizando as verificações necessárias para cada um de seus componentes.

d)	`Object(entityName)`	Cria um objeto obtendo o próximo OIDt da entidade entityName, se esta for válida.

Estes construtores foram mapeados para comando de `INSERT` na tabela principal da classe gerada. Existe um gatilho que analisa o valor a ser inserido em `tvOID`, que é utilizado para diferenciar um método de outro. O valor do campo `tvOID` deve seguir o formato correspondente a cada método construtor, conforme é descrito na Tabela 4.6.

Tabela 4.6: Mapeamento dos Construtores de *Object*

Item	Construtor	Valor passado para tvOID no comando de INSERT
a)	`Object()`	`'obj()'`
b)	`Object(ascendantID: OIDt)`	`'obj(|aOIDt|)'`

c)	`Object(entityId: integer,`
	`classId: integer,`
	`versionId: integer)`
		`'obj(eID,cID,vID)'`

d)	`Object(entityName: string)`	`'obj(eName)'`

Para a primeira opção, que recebe um `OIDt` com valor `'obj()'`, o gatilho seleciona nos metadados o próximo valor de `entityId` e `classId`, e com estes dois valores encontra o valor de `versionId`. O valor deste `OIDt` é inserido então no respectivo atributo da tabela.

No método (b), que no valor de `OIDt` recebe o identificador do ascendente `'obj(|aOIDt|)'`, é realizado o mesmo procedimento da primeira opção e adicionalmente é estabelecido um relacionamento com seu ascendente na hierarquia, atualizando a tabela `AscDesc`. O número da entidade é obtido de seu ascendente. O `OIDt` é colocado entre barras `'||'`, para diferenciar da alternativa (c) que recebe um OIDt para o objeto a ser criado e não do seu ascendente.

A alternativa (c) recebe por parâmetro os identificadores da entidade, da classe e da versão (`'obj(eID,cID,vID)'`). O mesmo gatilho verifica todos os campos deste OIDt para ver se eles estão consistentes com os metadados, como está especificado em (MORO et al., 2001).

Na última opção, no lugar do valor de `OIDt` é passado por parâmetro o nome da entidade `'obj(eName)'`. Primeiramente, é verificado se o nome da entidade já existe nos metadados. Se não existir, o próximo valor de entidade é obtido e inserido nos metadados com aquele nome de entidade, senão gera um erro dizendo que a entidade não existe. A partir daí é realizado o mesmo procedimento da primeira opção.

Para criar versões de objetos existe o construtor `TemporalVersion`. A Tabela 4.7 apresenta as diferentes formas de invocar este método.

Tabela 4.7: Métodos Construtores de Versões

Item	Construtor	Descrição
a)	`TemporalVersion()`	Cria uma versão obtendo o próximo valor do OIDt dos metadados, incluindo a tabela VOC para obter o próximo valor de *versionId*..
b)	`TemporalVersion(` `  ascendantID: set(OIDt))`	Cria uma versão relacionando-a com seus ascendentes na hierarquia.

c)	`TemporalVersion(`
	`  entityId: integer,`
	`  classId: integer,`
	`  versionId: integer)`
		Cria uma versão com o OIDt especificado realizando as verificações necessárias para cada um de seus componentes.

d)	`TemporalVersion(entityName)`	Cria uma versão obtendo o próximo OIDt da entidade entityName

e)	`TemporalVersion(`
	`  predecId: set(OIDt),`
	`  ascendId: set(OIDt),`
	`  config: boolean)`
		Cria uma versão relacionando-a com suas predecessoras, seus ascendentes e inicializa o valor do atributo *config*.

Todas as alternativas da Tabela 4.7 foram mapeadas para comandos INSERT na tabela da classe temporal versionada, tentando inserir no valor de OIDt os valores correspondentes de acordo com a Tabela 4.8.

Tabela 4.8: Mapeamento dos Construtores de *Temporal Version*

Item	Construtor	Valor passado para tvOID no comando de INSERT
a)	`TemporalVersion()`	`'tv()'`
b)	`TemporalVersion(` `  ascendantID: set(OIDt))`	`'tv(|aOIDt|)'`

c)	`TemporalVersion(`
	`  entityId: integer,`
	`  classId: integer,`
	`  versionId: integer)`
		`'tv(eID,cID,vID)'`

d)	`TemporalVersion(` `  EntityName: string)`	`'tv(eName)'`

e)	`TemporalVersion(`
	`  predecId: set(OIDt),`
	`  ascendId: set(OIDt),`
	`  config: boolean)`
		`'tv(|pOIDt|,|aOIDt|,config)'`

A alternativa (a) tenta inserir no valor de OIDt o valor 'tv()'. O gatilho BEFORE INSERT preenche o valor de OIDt realizando as devidas verificações nos metadados. A opção (b) recebe os ascendentes por parâmetro, mas por simplificação no mapeamento só aceito um ascendente. Primeiramente, o gatilho verifica se o ascendente passado por parâmetro é válido, obtém o valor da entidade e da próxima versão disponível para preencher o valor de OIDt. Um trabalho futuro poderá receber vários ascendentes concatenando-os na *string* de entrada.

O terceiro método é muito semelhante ao primeiro, porém o usuário já pode inicializar o valor de OIDt, que também sofrerá as verificações necessárias. Os identificadores de entidade e classe são conferidos nas tabelas ENTITY e CLASS, respectivamente. Já o número de *versionID* é verificado nas tabela VOC e auxiliares. A alternativa (d) realiza a mesma operação da primeira depois de realizar a verificação do nome da entidade que foi passado por parâmetro.

A última opção foi simplificada para receber somente um ascendente e um predecessor, que são primeiramente verificados se estão coerentes com os metadados. O novo OIDt é gerado utilizando o identificador de entidade do predecessor. O valor do sucessor do predecessor recebido por parâmetro passa a apontar para a nova versão, que anda é adicionada como descendente do ascendente recebido.

Os comandos de INSERT passando OIDt no formato apresentado na Tabela 4.8 não são executados diretamente pelo usuário. Eles são executados pelo comando de derivação que o usuário deve utilizar para criar uma nova versão, conforme é definido na seção 4.5.4.

4.5.3 Gerenciamento dos Objetos

Para permitir o gerenciamento de objetos, foi necessário definir para a extensão o mapeamento de métodos da classe *Object* do TVM.

4.5.3.1 Método findVersion

Este método é responsável por retornar a próxima versão de uma determinada classe e entidade. Foi mapeado para um procedimento armazenado que recebe por parâmetro o identificador da entidade e da classe e retorna um valor inteiro que identifica a próxima versão. Para realizar esta tarefa, a extensão verifica o valor do atributo `currentVersion` na tabela VOC e auxiliares, utilizando o `OIDt` definido por '*classId*, *entityId*, 0' e a data corrente.

4.5.3.2 Métodos getAscendant e getDescendant

Os métodos `getAscendant` e `getDescendant` foram mapeados para UDFs que realizam consultas à tabela `AscDesc` apresentadas na seção 4.3.2. Recebendo por parâmetro a coluna com o `tvOID`, a função `getAscendant` retorna os identificadores das classes ascendentes de uma determinada instância comparando o valor de `entityId`. E o método `getDescendant` tem comportamento análogo, porém retorna os descendentes de um objeto ou versão. Não precisa receber os rótulos temporais, pois sempre existe apenas um valor corrente válido.

4.5.3.3 Métodos getTVOid, getClassId e getEntityId

Para estas operações que buscam informações do identificador (`getTVOid`, `getClassId` e `getEntityId`) de uma instância basta que o usuário realize um `SELECT` na tabela referente ao objeto, recuperando o valor de seu `OIDt` e isolando o item que desejar.

4.5.3.4 Métodos getClassName e getNickname

Os métodos `getClassName` e `getNickname` são mapeadas para funções que recebem por parâmetro a coluna `tvOID`, isolam o identificador da classe e realizam um consulta sobre a tabela `Class` dos metadados para obter o valor do nome ou do apelido da classe.

4.5.3.5 Método getObject

O método `getObject` não precisou ser mapeado porque para realizar uma operação de recuperar um determinado objeto no *TVM Extender* é necessário apenas realizar um `SELECT` na tabela referente a classe do objeto com o seu `tvOID`.

4.5.3.6 Métodos getCorrespondence, getCorrespondenceAsc e getCorrespondenceDesc

Esses métodos foram mapeados para procedimentos e são utilizados para obter a correspondência entre duas classes, recebidas por parâmetro, na hierarquia de herança por extensão. O primeiro método retorna correspondência do relacionamento entre as duas classes. O segundo retorna apenas a cardinalidade da classe ascendente (parte depois dos ‘:’). E o terceiro retorna a cardinalidade da classe descendente (parte ates dos ‘:’).

4.5.3.7 Método delete

Um objeto não temporal versionado pode ser excluído fisicamente através de um comando `DELETE` sobre a tabela da classe especificando o `OIDt` do objeto. Um gatilho do tipo `BEFORE DELETE` é responsável por fazer as verificações nos metadados e excluir o registro da entidade eliminada, se não possuir nenhum outro objeto associado.

4.5.3.8 Métodos isDeleteAllowed e isDeleteTreeAllowed

Estes métodos não foram mapeados porque as verificações pelas quais eles são responsáveis já estão sendo realizadas pelos gatilhos `BEFORE DELETE`.

4.5.3.9 Método verifyAscendId

O método `verifyAscendId` foi mapeado para um procedimento armazenado que recebe dois `OIDts`, realizando um `SELECT` na tabela `AscDesc` até o final da hierarquia, verificando se o segundo `tvOID` é ascendente do primeiro (retornando verdadeiro), caso contrário retorna falso.

4.5.3.10 Método verifyEntityName

O método `verifyEntityName`, mapeado para um procedimento, recebe o nome de uma entidade por parâmetro, realiza uma consulta na tabela `Entity` dos metadados. Se a entidade existe, retorna verdadeiro, caso contrário retorna falso.

4.5.4 Gerenciamento das Versões

A extensão também teve de realizar o mapeamento dos métodos da classe `TemporalVersion` do TVM que servem para o gerenciamento das versões.

4.5.4.1 *Métodos addAscendant, addDescendant e addPredecessor*

Para que o usuário consiga definir os ascendentes/descendentes e predecessores de uma versão, os métodos `addAscendant`, `addDescendant` e `addPredecessor` foram mapeados para procedimentos armazenados que inserem os pares de identificadores nas tabelas `AscDesc` e `PredSucc`, conforme o caso, ajustando os rótulos temporais. Os tempos de transação são preenchidos com a data atual.

4.5.4.2 *Métodos getAscendant, getDescendant, getPredecessor e getSuccessor*

Os métodos `getAscendant`, `getDescendant`, `getPredecessor` e `getSuccessor` foram definidos como UDFs que através do `tvOID` recebido por parâmetro consulta nas tabelas `AscDesc` e `PredSucc` o valor correspondente ao(s) ascendente(s)/descendente(s) ou sucessor(es)/predecessor(es) de uma determinada instância.

4.5.4.3 *Método derive*

O método `derive` permite que uma nova versão seja gerada baseada em uma ou mais versões e objetos. Se o estado da versão que serviu de base para a nova for *Working,* ela é automaticamente promovida para *Stable*, como pode ser observado na Figura 2.4. No TVM, esta operação pode receber um ou vários `tvOIDs`, ou seja, permite a derivação de mais de uma versão, porém só estabelece regras para derivar da primeira versão selecionada no processo de derivação. Como este método foi mapeado para uma *stored procedure* que recebe um número exato de parâmetros, o *TVM Extender* assumirá que uma versão pode ser derivada de apenas uma versão, objeto versionado ou objeto sem versões.

Este procedimento recebe por parâmetro: o nome da classe a que o objeto pertence, e o `tvOID` do objeto que servirá como base para a derivação. Primeiramente são feitas algumas verificações sobre os parâmetros e depois é executado um comando de `INSERT` na tabela da classe tomando como base a versão/objeto originário e atribuindo a `tvOID` a convenção para gerar um `OIDt` para uma versão temporal versionado como foi descrito na seção 4.5.2.

Para a criação da nova versão é passado o valor **`'tv(|tvOID|,|asctvOID|,F)'`** para `OIDt` e a cópia dos demais atributos/relacionamentos do objeto que serviu como base da derivação. Se for a primeira derivação, o procedimento ainda cria uma entrada na tabela VOC e auxiliares para o novo objeto versionado. O novo `OIDt` gerado é retornado no parâmetro de saída `newTvOID`.

4.5.4.4 *Método promote*

O método `promote` modifica o estado de desenvolvimento de uma determinada versão, atualizando o seu atributo *status*, obedecendo o diagrama de transição de estados da Figura 2.4. Foi mapeado para um procedimento armazenado que promove uma versão para o estado subseqüente. Recebe por parâmetro o `tvOID` e a classe da versão, retornando no parâmetro de saída *result* se a operação foi bem sucedida.

Se o status da versão recebida for *Working*, é modificado para *Stable;* e o *status* de seus ascendentes deve ser automaticamente atualizado para o mesmo valor, quando já não o possuírem. Se o *status* atual for *Stable,* passa para o estado *Consolidate* e os que tiverem *status 'W'* ou *'S'* são modificados para *'C'* também. Os valores do *status* são atualizados na tabela auxiliar que mapeia o histórico do atributo, definindo os rótulos temporais.

4.5.4.5 Métodos delete

Uma versão não pode ser excluída fisicamente, apenas sofrer uma exclusão lógica. Quando ela é excluída logicamente, passa para o estado *deactivated*, seu atributo *alive* é atualizado para *F* e seu tempo de vida finalizado. A operação `delete` foi mapeada para um procedimento armazenado que verifica as condições de exclusão de versões e objetos versionados e atualiza os dados para a exclusão lógica. Recebe por parâmetro o `tvOID` da versão e a classe que ele pertence. Realiza as verificações que o método `isDeleteAllowed` faria, só permitindo apagar o objeto se: o *status* da versão for diferente de 'C' ou 'D', não possuir descendente ou sucessor e que não faça par te de uma classe agregada.

Se for um objeto versionado, estas verificações são realizadas para todas as suas versões, que são igualmente apagadas. Se a versão (ou objeto) não possui predecessora, ela é a primeira e única versão do objeto versionado, então o objeto fica sem versões e mantém o valor de seus atributos e o relacionamento com `VersionedObjectControl`. E se o objeto não possui versões, é excluído juntamente com a respectiva instância de `VersionedObjectControl` (se houver).

Depois de realizar as verificações, o método `delete` executa os seguintes passos:

- Quando a versão (ou o objeto versionado) possui ascendente, o registro correspondente na tabela `AscDesc` é finalizado (ou de todos os ascendentes no caso de ter mais de um), ou seja, seus tempos finais são encerrados com a data inicial menos um dia;

- Quando a versão (ou o objeto versionado) possui predecessor, o registro correspondente é encerrado na tabela `PredSucc` (de todos os predecessores no caso de ter mais de um) , ou seja, seus tempos finais são encerrados com a data inicial menos um dia;

- O *status* da versão (ou do objeto versionado) é atualizado para *Deactivated (D)*;

- O atributo *alive.vTimef* da versão (ou do objeto versionado) recebe o valor do tempo de transação, ou seja, o registro correspondente à versão na tabela que armazena o histórico de *alive* é atualizado;

- Se a versão (ou o objeto versionado) possui algum atributo ou relacionamento temporal com o valor de `vTimef` em aberto, esse recebe o valor de *alive.vTimef* da versão.

- O controle do objeto versionado (registro da tabela VOC e auxiliares) deve ser atualizado. Se a versão for a corrente (`currentVersion`), deve voltar a valer a versão corrente anterior. O mesmo deve ocorrer para o atributo `lastVersion` de VOC. O valor de `versionCount` também deve ser decrementado.

A exclusão física é utilizada quando se deseja remover fisicamente as informações. Pode ser útil quando existir restrição de espaço ou por questões de segurança. Esta operação é conhecida como *vaccuming* (MOROa, 2001), porém não está sendo abordada neste trabalho.

4.5.4.6 Método restore

O método `restore` permite a restauração de objetos que foram excluídos logicamente. Também foi mapeado para um procedimento armazenado que recebe por parâmetro o `tvOID` e a classe da versão, que passa a valer a partir do momento em que for restaurada. No parâmetro de saída é retornado se a operação de restauração foi bem sucedida. O exemplo mais simples da necessidade desse procedimento é a situação na qual algum objeto é excluído erroneamente. Outra situação comum é a de um funcionário que saiu da empresa, após ter seus dados apagados, ser readmitido. A restauração acontece para:

- uma versão folha que possui predecessora (não é a primeira versão do objeto). Qualquer versão folha pode ser restaurada com os valores dos atributos que possuía no momento da exclusão que recebem como validade inicial o tempo de transação no momento da restauração.

- 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. Para restaurar qualquer versão excluída anteriormente, o usuário deve excluir a primeira versão atual e a sua hierarquia de derivação, se houver;

- uma versão cuja predecessora também foi excluída.

- todas as versões do objeto. É possível também restaurar todas as versões do objeto, desde que não se tenha gerado outra seqüência de derivação;

- um objeto versionado com versões. Nesse caso, a referência ao controle do objeto versionado e o objeto controle também são excluídos, uma vez que 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.

4.5.4.7 Métodos getStatus, getOIDControl e getVersionedObjectId

Os métodos `getStatus`, `getOIDControl` e `getVersionedObjectId` não foram mapeados pois suas funções podem ser realizadas por consultas (`SELECT`) diretamente nas tabelas das classes e na tabela VOC que controla os objetos versionados.

4.5.4.8 Métodos removeAscendant, removeDescendant e removeSuccessor

Os métodos `removeAscendant`, `removeDescendant` e `removeSuccessor` também foram mapeados para procedimentos armazenados que fecham os tempos finais dos registros necessários das tabelas `PredSucc` e `AscDesc`. Os procedimentos recebem por parâmetro: o `OIDt` do objeto e o `OIDt` do objeto a ser removido (ascendente, descendente ou sucessor). Esse par de valores é localizado na tabela correspondente (`AscDesc` ou `PredSucc`) e seus tempos finais são encerrados com a data atual menos um dia.

4.5.4.9 Método setStatus

Este método foi mapeado para um procedimento armazenado que recebe por parâmetro o `tvOID`, a classe do objeto, o novo valor do *status* e o tempo de validade

inicial. É utilizado um comando `UPDATE` para atualizar o valor do registro correspondente ao `tvOID` na tabela da que armazena o histórico do atributo *alive*. No parâmetro de saída é retornado se a operação foi bem sucedida.

4.5.4.10 Métodos isDeleteAllowed e isDeleteTreeAllowed

`IsDeleteAllowed` e `isDeleteTreeAllowed` são métodos auxiliares no TVM e não foram mapeados no *TVM Extender*, pois já estão inseridos no método `delete`.

4.5.4.11 Método verifyAscendId

Foi mapeado para um procedimento armazenado de maneira semelhante ao método `verifyAscendId` dos objetos definido na seção 4.5.3.

4.5.5 Gerenciamento dos Objetos Versionados

Além do gerenciamento de objetos e versões o modelo oferece alguns métodos para controlar os objetos versionados. O mapeamento destes métodos está descrito nesta seção.

4.5.5.1 Construtor VersionedObjectControl

Esta operação pode ser realizada através de um comando `INSERT` na tabela VOC. Na tabela existem gatilhos para receber os valores dos atributos temporais. O valor a ser inserido na coluna do atributo temporal deve seguir o formato **`'value|vTimeI|vTimeF|'.`** O gatilho `AFTER INSERT` é responsável por inserir o valor e os rótulos temporais iniciais na tabela auxiliar que guarda o histórico do atributo, encerrando os tempos finais se já houver um valor anterior.

4.5.5.2 Método setCurrentVersion

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. Foi mapeada para um procedimento que recebe por parâmetro o `tvOID` da versão que passa a ser a corrente, e o atributo `userCurrentFlag` é atualizado para *true*. Um outro procedimento é definido sem receber parâmetros que é responsável por voltar o controle de gerenciar a versão corrente ao *TVM Extender*. Neste caso, a versão corrente passa a ser a última criada e o atributo `userCurrentFlag` é modificado para *F*.

4.5.5.3 Métodos getCurrentVersion, getFirstVersion, getLastVersion, getVersionCount, getNextVersionNumber, getUserCurrentFlag

Estes métodos são utilizados para recuperar os valores das propriedades do objeto `VersionedObjectControl`. Como no *TVM Extender* estes objetos estão armazenados na tabela VOC, basta que o usuário execute uma consulta (`SELECT`) a esta tabela selecionando o atributo desejado, fazendo as restrições necessárias.

4.5.5.4 Métodos setFirstVersion, setLastVersion, setUserCurrentFlag

Os métodos `setFirstVersion, setLastVersion, setUserCurrentFlag` apenas atualizam os valores dos atributos da tabela VOC. Assim, não necessitam ser mapeados, pois podem ser realizados através de um comando `UPDATE` nos atributos específicos. Como tratam-se de atributos temporais, o valor que deve ser repassado para a coluna deve seguir o formato: **`'value|vTimeI|vTimeF|'`** . Um gatilho do tipo

AFTER UPDATE é responsável por separar os valores e atualizar a tabela que armazena o histórico do campo. O tempo de transação é preenchido com a data atual. Maiores detalhes podem ser obtidos na seção 4.5.1.

4.5.5.5 Método updateVersionCount

O método updateVersionCount, que verifica se o atributo versionCount realmente está refletindo o número total de versões de um objeto versionado. Foi mapeado para um procedimento armazenado que recebe por parâmetro o nome da classe e o OIDt do objeto versionado. A tabela da classe é pesquisada e o número de versões do objeto versionado é contado. Em seguida, o atributo versionCount é atualizado se estiver diferente na tabela auxiliar que armazena seu histórico. Os rótulos temporais são devidamente preenchidos.

4.5.5.6 Método updateNextVersionNumber

Este método incrementa em uma unidade o valor da próxima versão de um objeto versionado. Foi mapeado para um procedimento armazenado que recebe por parâmetro o OIDt do objeto versionado. Com este valor, o procedimento faz um UPDATE na tabela VOC acrescentando um ao atributo nextVersionNumber.

4.5.6 Sintaxe

A sintaxe do mapeamento dos métodos apresentados nas seções anteriores é apresentado na Tabela 4.9

Tabela 4.9: Mapeamento dos Construtores de *Object*

Método	**Sintaxe**
findVersion	findVersion(IN entityId INTEGER, IN classId INTEGER, OUT versionId INTEGER)
getCorrespondence	getCorrespondence(IN className1 VARCHAR(30), IN ClassName2 VARCHAR(30), OUT corresp CHAR(1))
verifyAscendId	verifyAscendId (IN tvOID1 OIDt, IN tvOID2 OIDt, OUT ascend CHAR(1))
addAscendant	addAscendant(IN tvOID1 OIDt, IN tvOID2 OIDt, OUT result CHAR(1))
addDescendant	addDescendant(IN tvOID1 OIDt, IN tvOID2 OIDt, OUT result CHAR(1))
addPredecessor	addPredecessor(IN tvOID1 OIDt, IN tvOID2 OIDt, OUT result CHAR(1))
derive	derive(IN className VARCHAR(30), IN tvOID OIDt, OUT newTvOID OIDt)
promote	promote(IN className VARCHAR(30), IN tvOID OIDt, OUT result CHAR(1))
delete	delete (IN className VARCHAR(30), IN tvOID OIDt, OUT result CHAR(1))

Método	Sintaxe
restore	restore(IN className VARCHAR(30), IN tvOID OIDt, OUT result CHAR(1))
removeAscendant	removeAscendant(IN tvOID OIDt, IN ascTvOID OIDt, OUT result CHAR(1))
removeSucessor	removeSucessor(IN tvOID OIDt, IN sucTvOID OIDt, OUT result CHAR(1))
setStatus	setStatus(IN className VARCHAR(30), IN tvOID OIDt, IN state CHAR(1), IN vTimeI TIMESTAMP, OUT newTvOID OIDt)
updateVersionCount	updateVersionCount(IN className VARCHAR(30), IN tvOID OIDt)

4.6 Consulta sobre os Dados

A linguagem de consulta do Modelo Temporal de Versões permite além da realização de consultas básicas através da linguagem padrão SQL, novas consultas que retornam valores específicos das características de tempo e versões (MORO et al., 2001) Ela estabelece um comportamento 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 estava “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.

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, estão consultas sobre:

- os estados das versões em determinados instantes ou períodos;

- as versões separadas ou o conjunto 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.

Em (ZAUPA, 2002) foi definida uma linguagem de consulta para o TVM, a TVQL (detalhada na seção 2.4.9), baseada no padrão SQL, porém modificando sua sintaxe. Como o objetivo do *TVM Extender* é não usar um interpretador externo para uma sintaxe modificada do SQL, e sim utilizar apenas os mecanismos de extensão do DB2 (principalmente UDFs) não será retratada neste trabalho a sintaxe da linguagem de consulta TVQL. Entretanto, a maioria das consultas sobre valores temporais e versões poderá ser obtida através das funções e procedimentos definidos na seção 4.5. As funções podem ser utilizadas no SQL seja na cláusula do `SELECT`, `FROM` ou `WHERE` e os procedimentos podem ser chamados de forma independente.

No *TVM Extender*, ao contrário da TVQL, a intenção é definir funções e procedimentos que podem ser utilizadas diretamente no banco de dados sem exigir um interpretador específico para a linguagem. Quando o *extender* é habilitado, estes métodos são reconhecidos pelas aplicações.

4.6.1 Suporte a Tempo

A Tabela 4.10 ilustra os métodos que foram mapeados e podem ser utilizados para retornar informações referentes ao conceito de tempo.

Tabela 4.10: Métodos que Retornam Informações sobre Tempo

Método	Descrição
`getCurrent`	Retorna o valor corrente de um atributo ou relacionamento.
`getHistory`	Retorna o histórico dos valores de um atributo ou relacionamento.
`getValueAt`	Retorna o valor de um atributo ou relacionamento em um determinado instante de tempo.
`getTempLabel`	Retorna os rótulos temporais da versão corrente de determinado objeto.

Estes métodos foram mapeados para funções/procedimentos, mas se o usuário desejar ele pode utilizar comandos `SELECT` fazendo uma junção da tabela principal da classe com as tabelas auxiliares dos atributos/relacionamentos temporais através do campo `tvOID`. As restrições necessárias devem ser especificadas na sentença do `WHERE`. Dessa forma a performance das consulta tende a ser mais eficiente do que a utilização de UDFs, pois são acionados os índices do banco de dados. Pode ser realizado no futuro, ainda, um estudo para analisar e comparar as duas performances.

4.6.2 Suporte a Versões

Dentre os métodos que podem ser utilizados para obter informações sobre as versões dos objetos estão os apresentados na Tabela 4.11, que foram previamente mapeados na seção 4.5.4:

Tabela 4.11: Métodos que Retornam Informações sobre as Versões

Método	Descrição
`GetCurrentVersion`	Retorna a versão corrente de um objeto.
`GetFirstVersion`	Retorna a primeira versão de um objeto.

GetLastVersion	Retorna a última versão de um objeto.
GetVersionCount	Retorna o número de versões de um objeto.
getNextVersionNumber	Obtém o próximo número de versão disponível para um objeto.
getUserCurrentFlag	Mostra se é o usuário ou o sistema que está controlando a versão corrente.
GetAscendant	Retorna o objeto ascendente na hierarquia de extensão.
GetDescendant	Retorna o objeto descendente na hierarquia de extensão.
getPredecessor	Retorna a versão predecessora.
GetSuccessor	Retorna a versão sucessora.
GetStatus	Retorna o status da versão.
getClassName	Retorna o nome da classe do objeto.

A maioria dos métodos apresentados na Tabela 4.10 e na Tabela 4.11 foram mapeados para UDFs e, assim podem ser utilizadas em uma consulta, seja na expressão do SELECT, do FROM ou do WHERE de acordo com o tipo de informação que elas retornam.

No cenário descrito na Figura 4.15, é possível observar a utilização da função *getCurrent()*, que é responsável por retornar o valor corrente da atributo temporal. Esta UDF, recebe por parâmetros o nome do atributo e o OIDt do objeto e faz as verificações necessárias na tabela auxiliar. Considerando a existência de uma classe ALUNO com dois atributos: nome e telefone e, sendo este último temporal, é apresentada uma consulta para recuperar os dados do aluno 'João', com seu telefone atual.

```
SELECT nome, getCurrent('telefone', tvOID) FROM aluno
WHERE varchar(tvOID) = '3,13,1'
```

Aluno

TvOID	nome	telefone
3, 12, 1	Maria	-
3, 13, 1	João	-
3, 23, 1	Luís	-

AlunoTelefone

tvOID	value	vTimeI	tTimeI	vTimeF	tTimeF
3, 12, 1	45640001	01/12/2002	01/12/2002	-	-
3, 13, 1	22323454	05/01/2003	05/01/2003	11/09/2003	11/09/2003
3, 23, 1	33425555	06/08/2003	06/08/2003	-	-
3, 13, 1	21219898	12/09/2003	12/09/2003	-	-

Figura 4.15: Exemplo de Consulta do Valor Atual de um Atributo

Se fosse utilizada a alternativa de fazer o SELECT da tabela principal (aluno) com a tabela auxiliar (alunoTelefone), a consulta para retornar o valor corrente do telefone se resumiria a:

```
SELECT nome, telefone FROM aluno A, alunoTelefone AT
WHERE (A.tvOID = AT.tvOID)  AND (AT.vTimeF is null)
```

5 ESTUDO DE CASO

Esse capítulo aborda a modelagem de um estudo de caso e a utilização do *TVM Extender* para implementá-lo. Ainda são apresentadas algumas instâncias de exemplo para ilustrar o gerenciamento dos dados e consultas, considerando o mapeamento proposto nessa dissertação.

5.1 A Aplicação

O estudo de caso utilizado consiste da modelagem de um sistema que gerencia o desenvolvimento de sistemas. Os sistemas são divididos em módulos, que podem ser implementados por diferentes programadores. O programador pode inserir um novo módulo ou modificar os módulos existentes, criando novas versões.

Cada sistema consiste em um *projeto*, que pode ser composto de um ou mais *módulos* e possui uma *equipe* de trabalho, que é responsável pelo seu desenvolvimento. A equipe é composta por vários desenvolvedores (*funcionários*), sendo que um desempenha a função de *liderança*. Cada funcionário está vinculado com uma *função* dentro da empresa.

Os módulos do sistema podem ser atualizados por um membro da equipe responsável por desenvolvê-lo. Essas atualizações são armazenadas através do controle de versões e tempo do TVM, de modo que todo o histórico delas seja mantido. Além disso, é armazenado o histórico para a *documentação* do projeto, bem como da rotatividade dos componentes da equipe e as mudanças de função de cada funcionário.

A Figura 5.1 apresenta apenas o diagrama de classes do modelo deste estudo de caso. As operações não estão sendo abordadas neste diagrama.

As características de versão e tempo são usadas em:

- classe *Projeto*: armazenando o histórico de sua documentação;
- classe *Modulo*: armazenando as alternativas dos módulos, bem como o histórico de suas alterações com os autores e suas observações;
- classe *Funcionario*: armazenado o histórico do endereço do funcionario;
- relacionamentos *trabalha* e *chefe*: mantém o histórico da rotatividade da equipe de trabalho, bem como de quem exerceu a função de chefia;
- relacionamento *pertence*: armazena os vínculos dos módulos com os projetos ao longo do tempo, como for definido pelo desenvolvedor;

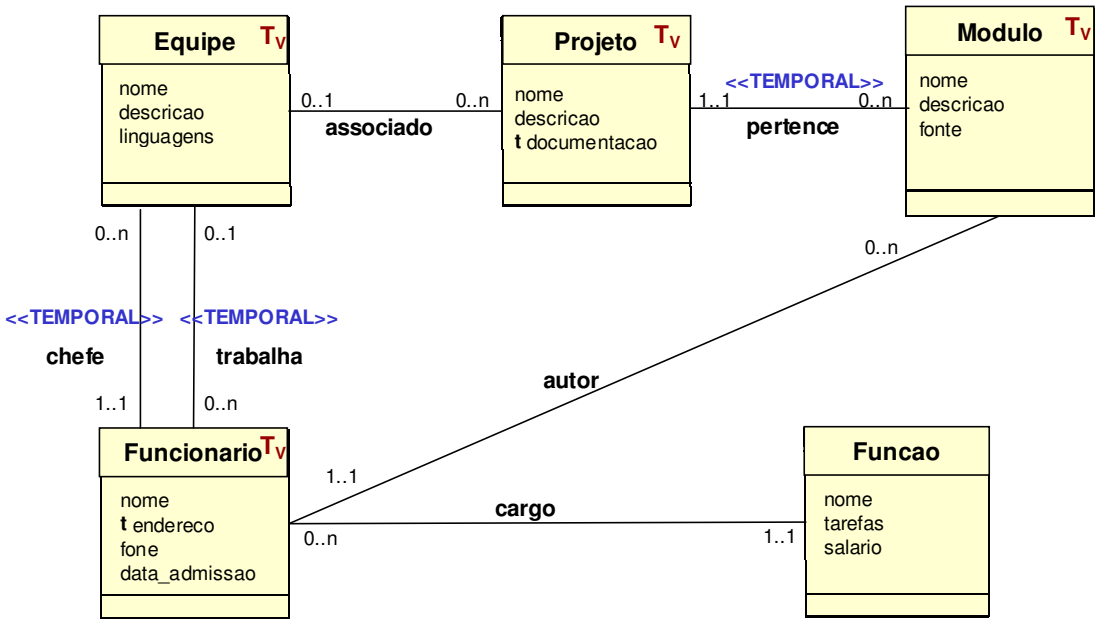


Figura 5.1: Diagrama de classes do estudo de caso

5.2 Especificação das Classes, Atributos e Relacionamentos

Primeiramente é criado o esquema, que agrupa as classes. Este esquema foi denominado `SisProj`, representando o Sistema de Gerenciamento de Projetos, e criado através do comando apresentado na Tabela 5.1. A seguir são apresentados os comandos para especificar as classes e seus atributos do estudo de caso, segundo as definições descritas na seção 4.4.

Tabela 5.1: Comando para criar o esquema da aplicação

Esquema	**Mapeamento**
`SistProj`	`createSchema('SisProj', schemaId);`

5.2.1 Classes e Atributos

O próximo passo é a definição de cada classe e seus atributos. Nessa fase é necessário especificar quais as classes temporais versionadas e quais atributos são temporais. A Tabela 5.2 apresenta os comandos de definição de cada classe e seus atributos.

Tabela 5.2: Comandos para a definição das classes e atributos

Classe Mapeamento
Equipe

`defineClass('Equipe', schemaId, 'T','T', classId3);`
`addAttribute(classId3, 'nome', 'varchar(30)', 'F', ' ', 'T', '');`
`addAttribute(classId3, 'descricao', 'varchar(60)', 'F', ' ', 'T', '');`
`addAttribute(classId3, 'linguagens', 'varchar(60)', 'F', ' ', 'T', '');`

Classe Mapeamento
Funcao

`defineClass('Funcao', schemaId, 'F', 'T', classId5);`
`addAttribute(classId5, 'nome', 'varchar(30)', 'F', ' ', 'T', '');`
`addAttribute(classId5, 'Tarefas', 'varchar(60)', 'F', ' ', 'T', '');`
`addAttribute(classId5, 'salario', 'float', 'F', ' ', 'T', '');`

Funcionario

`defineClass('Funcionario', schemaId, 'T', 'T', classId4);`
`addAttribute(classId4, 'nome', 'varchar(30)', 'F', ' ', 'T', '');`
`addAttribute(classId4, 'endereco', 'varchar(60)', 'T', ' ', 'T', '');`
`addAttribute(classId4, 'fone', 'varchar(15)', 'F', ' ', 'T', '');`
`addAttribute(classId4, 'data_admissao', 'timestamp', 'F', ' ', 'T', '');`

Projeto

`defineClass('Projeto', schemaId, 'T', 'T', classId1);`
`addAttribute(classId1, 'nome', 'varchar(30)', 'F', ' ', 'T', '');`
`addAttribute(classId1, 'descricao', 'varchar(60)', 'F', ' ', 'T', '');`
`addAttribute(classId1, 'documentation', 'file', 'T', ' ', 'T', '');`

Modulo

`defineClass('Modulo', schemaId, 'T', 'T', classId2);`
`addAttribute(classId2, 'nome', 'varchar(30)', 'F', ' ', 'T', '');`
`addAttribute(classId2, 'descricao', 'varchar(60)', 'F', ' ', 'T', '');`
`addAttribute(classId2, 'fonte', 'file', 'T', ' ', 'T', '');`

5.2.2 Relacionamentos

Os comandos apresentados na Tabela 5.3 são utilizados para criar os relacionamentos da aplicação no *extender*, conforme foi especificado na seção 4.4 :

Tabela 5.3: Comandos para a Especificação dos Relacionamentos

Relacionamento Mapeamento
pertence
`addRelationship(classId2, classId1,'pertence','','T', '0..n', 'F', '');`
associado
`addRelationship(classId1,classId3,'associado','','F','1..n','F','');`
trabalha
`addRelationship(classId4, classId3, 'trabalha', '', 'T', '0..n', 'F', '');`
chefe
`addRelationship(classId4, classId3, 'chefe', '','T', '0..1', 'F', '');`
autor
`addRelationship(classId4, classId2, 'autor', '','F', '0..n', 'F', '');`
cargo
`addRelationship(classId4, classId5, 'cargo', '','F', '1..n', 'F', '');`

5.2.3 Geração das Classes

Após a execução de todos os comandos de especificações das classes, relacionamentos e atributos, é necessário utilizar o comando para gerar as classes: `generateClasses`. Quando este comando é executado, o *TVM Extender* cria todas as tabelas referentes às classes baseadas nos tipos de dados da hierarquia do TVM e nos dados que foram inseridos nos metadados. Além disso, são criadas as tabelas auxiliares para os atributos e relacionamentos temporais.

A partir deste momento, o usuário não pode mais modificar a estrutura das classes. Outros trabalhos estudam as formas de atualizar os esquemas temporais e versionados.

5.2.4 Metadados

Depois das classes especificadas e geradas, os metadados estão preenchidos como pode ser observado na Figura 5.2. A tabela `Schema` contém o esquema `SistProj` que engloba toda a aplicação. Na tabela `Class` estão todas as classes especificadas. Os atributos e relacionamentos (temporais ou não) são armazenados nas tabelas `Attribute` e `Relationship`, com suas características específicas.

Schema

id	nome
1	SistProj

Class

id	nome	tempVers	generated	schemaId
1	equipe	T	T	1
2	funcao	F	T	1
3	funcionario	T	T	1
4	projeto	T	T	1
5	modulo	T	T	1

Attribute

id	nome	temp	type	static	default	classId
1	nome	F	varchar 30			1
2	descricao	F	varchar 60			1
3	linguagens	F	varchar 60			1
4	nome	F	varchar 30			2
5	tarefas	F	varchar 60			2
6	salario	F	float			2
7	nome	F	varchar 30			3
8	endereco	T	varchar 60			3
9	fone	F	varchar 15			3
10	data_admissao	F	Timestamp			3
11	nome	F	varchar 30			4
12	descricao	F	varchar 60			4
13	documentacao	T	File			4
14	nome	F	varchar 30			5
15	descricao	F	varchar 60			5
16	fonte	F	file			5

Relationship

id	nome	temp	type	inverse	static	cardinality	single	classId	relatedClass
1	pertence	T		F		0..n		6	4
2	associado	F		F		1..n		4	1
3	trabalha	T		F		0..n		3	1
4	chefe	T		F		0..n		1	3
5	autor	F		F		0..n		5	3
5	cargo	F		F		1..n		3	2

Figura 5.2: Tabelas dos Metadados após a Especificação das Classes

5.3 Manipulação dos Dados

Após a especificação e geração das classes, todos os metadados e outros mecanismos necessários para o gerenciamento dos conceitos de tempo e versões já foram devidamente criados. O usuário pode, então, instanciar as classes e manipular as instâncias como se fizer necessário.

5.3.1 Instanciação das Classes

Através dos comandos de inserção de dados do *TVM Extender*, definidos na seção 4.5, o usuário pode popular cada uma das classes definidas, como pode ser observado na Tabela 5.4.

Tabela 5.4: Comandos de Inserção dos Dados

Classe Comandos de Inserção
Equipe

`INSERT INTO Equipe (nome, descricao, linguagens, chefe)`
`VALUES ('equipe1', 'Equipe 1',' Desenvolvimento de programas cliente-servidor',' Delphi, C++')`
 	
`INSERT INTO Equipe (nome, descricao, linguagens, chefe)`
`VALUES ('equipe2', 'Equipe 2',' Desenvolvimento de aplicações Web',' Java, PHP, javascript')`

Função

`INSERT INTO Funcao (nome, tarefas, salario)`
`VALUES ('programador', 'Desenvolver sistemas', 1000)`
 	
`INSERT INTO Funcao (nome, tarefas, salario)`
`VALUES ('analista', 'Analisar, projetar e desenvolver aplicações.', 2500)`

Funcionário

`INSERT INTO Funcionario (nome, fone, data_admissao, cargo, endereco, trabalha)`
`VALUES ('José da Silva' , ' 22332945' , ' 01/01/2000' , '4,2,1' , ' 1,1,1 |01/01/2000|', 'Rua 24 de maio,56 |01/01/2000|')`
 	
`INSERT INTO Funcionario (nome, fone, data_admissao, cargo, endereco, trabalha)`
`VALUES ('Marcos Lima' , ' 34343443', '25/03/1998 ', '4,2,1', '2,1,1|25/03/1998|' , Av. Silva Paes, 363|25/03/1998|')`
 	
`INSERT INTO Funcionario (nome, fone, data_admissao, cargo, endereco, trabalha)`
`VALUES ('Maria Cardoso', '22667788 ', '15/04/2002 ', '3,2,1', '1,1,1|15/04/2002|', 'Av. das Palmeiras,102 | 15/04/2002|')`
 	
`INSERT INTO Funcionario (nome, fone, data_admissao, cargo, endereco, trabalha)`
`VALUES ('Luiz Souza', '35436970', '15/01/2000', '3,2,1', '2,1,1|15/01/2000|', 'Av. das Araucárias,43|15/01/2000|' )`

Projeto

`INSERT INTO Projeto (nome, descricao, associado, documentacao)`
`VALUES ('PESSOAL', 'Sistema de Controle de Pessoal', '1,1,1', 'Pessoal01.doc|01/02/2002|')`
 	
`INSERT INTO Projeto (nome, descricao, associado, documentacao)`
`VALUES ('CHAMADOS', 'Controle de Chamados de Informática', '2,1,1', 'Chamados01.doc |20/08/2003|')`

Modulo
`INSERT INTO Modulo (nome, descricao, autor, fonte, pertence)` `VALUES ('cadastra_funcionario', 'Inclui os dados do funcionário' , '5,3,1' , 'cad_func.pas' , '9,4,1|01/05/2002|')`

Classe Comandos de Inserção
INSERT INTO Modulo (nome, descricao, autor, fonte, pertence) VALUES ('gera_pagamento', 'Calcula os salários ' , '5,3,1' , 'pagamento.pas' , '9,4,1|01/07/2002|')
INSERT INTO Modulo (nome, descricao, autor, fonte, pertence) VALUES ('cadastra_chamado', 'Cadastro de chamado via web' , '6,3,1' , 'cad_chamado.pas' , '10,4,1|01/08/2003|')

5.3.2 Alterações sobre os Dados

Nesta seção é exemplificado como o usuário pode modificar os dados que já estão no *TVM Extender*.

5.3.2.1 Atualização de Atributos/Relacionamentos Não Temporais

Os atributos e relacionamentos não temporais podem ser atualizados através da utilização de um comando `UPDATE` simples. Para este tipo de atributo/relacionamento o histórico não é armazenado. Um exemplo de atualização do atributo *tarefas* do objeto cujo `tvOID` é igual a '`3,2,1`', na classe `Funcao` pode ser observado na Figura 5.3.

```
UPDATE Funcao
SET tarefas = 'Desenvolver sistemas de software'
WHERE varchar(tvOID) = '3,2,1'
```

Figura 5.3: Exemplo de Atualização de Atributo Não Temporal

5.3.2.2 Atualização de Atributos/Relacionamentos Temporais

Para realizar a atualização dos atributos/relacionamentos temporais, é necessário utilizar um comando de `UPDATE`, passando o valor e/ou rótulos temporais, como especificado na seção 4.5.1. Os comandos apresentados na Tabela 5.5 exemplificam como estas atualizações são realizadas.

Tabela 5.5: Comandos de Atualização dos Dados

Classe Comandos de Atualização
Equipe UPDATE Equipe SET chefe='7,3,1|01/01/2000||' WHERE varchar(tvOID)= '2,1,1'
Funcionario UPDATE Funcionario SET endereco='Rua Sete de Setembro,555|30/11/2000||' WHERE varchar(tvOID)= '7,3,1'
Projeto UPDATE Projeto SET documentacao='Pessoal02.doc|01/06/2003||' WHERE varchar(tvOID)= '9,4,1'

5.3.2.3 Manipulação de Versões

A criação de uma nova versão para um determinado objeto pode ser realizada através da derivação da versão corrente. No estudo de caso, a equipe desenvolveu um módulo gera_pagamento para o projeto PESSOAL que gera a folha de pagamento da empresa, calculando os valores dos salários dos funcionários.

A empresa decidiu que daria um aumento aos funcionários, mas não estava bem certa de como gostaria que fosse aplicado. Ela tinha duas propostas e gostaria de ver o relatório para tomar uma decisão. Os desenvolvedores resolveram implementar duas versões do relatório. Na primeira, o aumento é de 25% sobre o salário, já na segunda é composto de uma gratificação de 20% sobre o salário mais um abono fixo de R$120,00.

Os comandos da Figura 5.4 representam estas derivações:

```
derive('Modulo', '12, 5, 1' , newTvOID) ◊ '12, 5, 2'

UPDATE Modulo SET descricao='Calcula os salários com uma
gratificação de 25%', autor =autor_T('7, 3, 1'),
fonte='pagamento2.pas', pertence='9,4,1|01/01/2003||'
WHERE varchar(tvOID) = '12,5,2'

derive('Modulo', '12, 5, 1' , newTvOID) ◊ '12, 5, 3'

UPDATE Modulo SET descricao='Calcula os salários com uma
gratificação de 20% + abono R$120,00', autor =autor_T('7, 3, 1'),
fonte='pagamento3.pas', pertence='9,4,1|01/01/2003||'
WHERE varchar(tvOID) = '12,5,3'
```

Figura 5.4: Exemplos de Comandos de Derivação

Com base nos relatórios gerados, a empresa optou pela versão que calcula a gratificação e o abono. A outra versão foi apagada, tendo seu *status* modificado para *Deactivated*. Os rótulos temporais de seus atributos/relacionamentos temporais foram encerrados. O comando para esta operação aparece na Figura 5.5.

```
delete ('Modulo','12,5,2', result)
```

Figura 5.5: Exemplo de Comando para Exclusão de uma Versão

Quando se aproximou o término do ano, os desenvolvedores decidiram gerar uma versão do relatório já com o 13º. Salário calculado, que será ativada em todos os meses de dezembro. Para criar a nova versão foi realizada uma nova derivação, como mostra a Figura 5.6.

```
derive('Modulo', '12, 5, 3' , newTvOID) ◊ '12, 5, 4'

UPDATE Modulo SET descricao=' Calcula os salários com uma gratificação de
20% + abono R$120,00 + 13o. Salário', autor =autor_T('7, 3, 1'),
fonte='pagamento4.pas', pertence='9,4,1||31/12/2003|'
WHERE varchar(tvOID) = '12,5,4'

UPDATE Modulo SET descricao=' Calcula os salários com uma gratificação de
20% + abono R$120,00 + 13o. Salário', autor =autor_T('7, 3, 1'),
fonte='pagamento4.pas', pertence='9,4,1|01/01/2004||'
WHERE varchar(tvOID) = '12,5,3'
```

Figura 5.6: Comandos de Derivação e Atualização

5.4 Armazenamento dos Dados

Após os comandos de instanciação e as atualizações terem sido executadas é possível observar como as tabelas das classes, tabelas auxiliares e tabelas dos metadados foram preenchidas para representar o cenário proposto. Nas tabelas apresentadas nesta seção, as linhas com fundo branco foram fruto das inserções iniciais e aquelas em destaque representam os resultados de atualizações posteriores.

5.4.1 Tabelas das Classes e Tabelas Auxiliares

Cada linha da tabela de uma classe contém os valores dos atributos e relacionamentos não temporais de uma instância e seu `tvOID` (calculado no momento da inserção). Os atributos e relacionamentos temporais têm seus valores armazenados em tabelas auxiliares, juntamente com seus rótulos temporais. Os campos referentes a este tipo de atributo/relacionamento existem na tabela principal da classe, com valor nulo, para acionar gatilhos que inserem/alteram os dados na tabela auxiliar. Os tempos de validade e transação são do tipo *timestamp*, que é representado por data e hora. Entretanto, nos históricos apresentados a seguir, são mostradas apenas datas para simplificar.

A Tabela 5.6 representa as instâncias da classe `Equipe`, com as equipes de desenvolvimentos de sistemas. A Tabela 5.7, Tabela 5.8 e a Tabela 5.9 apresentam o histórico dos atributos temporal *alive, status* e *chefe*, sendo que os dois primeiros foram herdados da classe `TemporalVersion`.

Tabela 5.6: Tabela referente à Classe *Equipe*

TvOID	RefVOC	nome	Descrição	Linguagens	chefe
1, 1, 1	—	Equipe 1	Desenvolvimento de programas cliente-servidor	Delphi, C++	—
2, 1 ,1	—	Equipe 2	Desenvolvimento de aplicações Web	Java, PHP, javascript	—

Tabela 5.7: Histórico do Atributo *Alive* da Classe *Equipe*

tvOID	value	VTimeI	vTimeF	tTimeI	tTimeF
1, 1, 1	T	25/03/1998	—	25/03/1998	—
2, 1 ,1	T	25/03/1998	—	25/03/1998	—

Tabela 5.8: Histórico do Atributo *Status* da Classe *Equipe*

TvOID	value	VTimeI	vTimeF	tTimeI	tTimeF
1, 1, 1	W	25/03/1998	—	25/03/1998	—
2, 1 ,1	W	25/03/1998	—	25/03/1998	—

Tabela 5.9: Histórico do Atributo *Chefe* da Classe *Equipe*

TvOID	value	VTimeI	VtimeF	tTimeI	TTimeF
1, 1, 1	5, 3, 1	25/03/1998	—	25/03/1998	—
2, 1, 1	6, 3, 1	25/03/1998	31/12/1999	25/03/1998	15/12/1999
2, 1, 1	7, 3, 1	01/01/2000	—	01/01/2000	—

A Tabela 5.10 representa as instâncias da classe `Funcao`, que não é temporal versionada e por isso não possui tabelas auxiliares. Nela estão descritas os cargos, com suas atribuições e salários.

Tabela 5.10: Tabela referente à Classe *Funcao*

TvOID	RefVOC	Nome	Tarefas	Salario
3, 2, 1	—	programador	Desenvolver sistemas de software	R$ 1000,00
4, 2. 1	—	Analista	Analisar, projetar e desenvolver aplicações.	R$ 2500,00

A Tabela 5.11 representa as instâncias da classe `Funcionario`. A Tabela 5.12 e Tabela 5.13 apresentam o histórico dos atributos `alive` e `status`, herdados de `TemporalVersion`. Já a Tabela 5.14 mostra o histórico do relacionamento `trabalha`, com a classe `Equipe`. E a Tabela 5.15 apresenta o histórico do atributo temporal `Endereco`, onde é possível observar que a funcionária *"Maria Cardoso"* se mudou para a *"Rua Sete de Setembro, 555"* no dia *"28/11/2000"*.

Tabela 5.11: Tabela referente à Classe *Funcionario*

tvOID	refVOC	Nome	Fone	Data_admissao	cargo	endereco	trabalha	alive	status
5, 3, 1	—	José da Silva	22332945	01/01/2000	4, 2, 1	—	—	—	—
6, 3, 1	—	Marcos Lima	34343443	25/03/1998	4, 2, 1	—	—	—	—
7, 3, 1	—	Maria Cardoso	22667788	15/04/2002	3, 2, 1	—	—	—	—
8, 3, 1	—	Luiz Souza	35436970	15/01/2000	3, 2, 1	—	—	—	—

Tabela 5.12: Histórico do Atributo *Alive* da Classe *Funcionario*

tvOID	value	vTimeI	vTimeF	tTimeI	tTimeF
5, 3, 1	T	01/01/2000	—	01/01/2000	—
6, 3, 1	T	25/03/1998	—	25/03/1998	—
7, 3, 1	T	15/04/2002	—	15/04/2002	—
8, 3, 1	T	15/01/2000	—	15/01/2000	—

Tabela 5.13: Histórico do Atributo *Status* da Classe *Funcionario*

tvOID	value	vTimeI	vTimeF	tTimeI	tTimeF
5, 3, 1	W	01/01/2000	—	01/01/2000	—
6, 3, 1	W	25/03/1998	—	25/03/1998	—
7, 3, 1	W	15/04/2002	—	15/04/2002	—
8, 3, 1	W	15/01/2000	—	15/01/2000	—

Tabela 5.14: Histórico do Relacionamento *Trabalha* da Classe *Funcionario*

tvOID	value	vTimeI	vTimeF	tTimeI	tTimeF
5, 3, 1	1, 1, 1	01/01/2000	—	01/01/2000	—
6, 3, 1	2, 1, 1	25/03/1998	—	25/03/1998	—
7, 3, 1	1, 1, 1	15/04/2002	—	15/04/2002	—
8, 3, 1	2, 1, 1	15/01/2000	—	15/01/2000	—

Tabela 5.15: Histórico do Atributo *Endereco* da Classe *Funcionario*

TvOID	value	VTimeI	vTimeF	tTimeI	tTimeF
5, 3, 1	Rua 24 de maio,56	01/01/2000	—	01/01/2000	—
6, 3, 1	Av. Silva Paes, 363	25/03/1998	—	25/03/1998	—
7, 3, 1	Av. das Palmeiras,102	15/04/2002	29/11/2000	15/04/2002	28/11/2000
8, 3, 1	Av. das Araucárias,43	15/01/2000	—	15/01/2000	—
7, 3, 1	Rua Sete de Setembro,555	30/11/2000	—	28/11/2000	—

A Tabela 5.16 representa as instâncias da classe `Projeto`, com os projetos de sistemas que são realizados pelas equipes. A Tabela 5.17 e a Tabela 5.18 apresentam o

histórico dos atributos temporal `alive` *e* `status`, herdados da classe `TemporalVersion`. A Tabela 5.19 mostra o histórico do atributo `documentacao`, que mantém o detalhamento do projeto. O projeto denominado *"PESSOAL"* teve a sua documentação atualizada em *"30/05/2003"*.

Tabela 5.16: Tabela referente à Classe *Projeto*

tvOID	refVOC	Nome	Descricao	associado	documentation	alive	status
9, 4, 1	—	PESSOAL	Sistema de Controle de Pessoal	1, 1, 1	—	—	—
10, 4, 1	—	CHAMADOS	Controle de Chamados de Informática	2, 1, 1	—	—	—

Tabela 5.17: Histórico do Atributo *Alive* da Classe *Projeto*

tvOID	value	vTimeI	vTimeF	tTimeI	tTimeF
9, 4, 1	T	12/01/2002	—	12/01/2002	—
10, 4, 1	T	15/06/2003	—	15/06/2003	—

Tabela 5.18: Histórico do Atributo *Status* da Classe *Projeto*

TvOID	value	vTimeI	vTimeF	tTimeI	tTimeF
9, 4, 1	W	12/01/2002	—	12/01/2002	—
10, 4, 1	W	15/06/2003	—	15/06/2003	—

Tabela 5.19: Histórico do Atributo *Documentacao* da Classe *Projeto*

tvOID	Value	VtimeI	vTimeF	tTimeI	tTimeF
9, 4, 1	Pessoal01.doc	01/02/2002	30/05/2003	01/02/2002	30/05/2003
10, 4, 1	Chamados01.doc	20/08/2003	—	20/08/2003	—
9, 4, 1	Pessoal02.doc	01/06/2003	—	30/05/2003	—

A Tabela 5.20 representa as instâncias da classe `Modulo`, que gerencia os módulos dos projetos desenvolvidos. Nesta tabela é possível observar que foram criados inicialmente dois módulos para o projeto *"PESSOAL"* e um módulo para o projeto *"CHAMADOS"*. Um dos módulos do projeto *"PESSOAL"* denomina-se *"gera_pagamento"*, e é responsável por gerar a folha de pagamento dos funcionários. Este módulo foi derivado, gerando duas novas versões: uma que calcula os salários com uma gratificação de 25% sobre o salário e outra que calcula uma gratificação de 20% sobre o salário mais um abono fixo de R$ 120,00. A empresa comparou os dois resultados e optou por manter a versão com o abono fixo. Posteriormente ainda, foi criada uma versão deste módulo para o mês de dezembro, que considera ainda o 13º. salário.

Tabela 5.20: Tabela referente à Classe *Modulo*

TvOID	refVOC	Nome	Descricao	Autor	fonte	pertence	alive	status
11, 5, 1	—	cadastra_funcionario	Inclui os dados do funcionário	5, 3, 1	cad_func.pas	—	—	—
12, 5, 1	12, 5, 0	gera_pagamento	Calcula os salários	5, 3, 1	pagamento.pas	—	—	—
13, 5, 1	—	cadastra_chamado	Cadastro de chamado via web	6, 3, 1	cad_chamado.pas	—	—	—
12, 5, 2	12, 5, 0	gera_pagamento	Calcula os salários com uma gratificação de 25%	7, 3, 1	pagamento2.pas	—	—	—
12, 5, 3	12, 5, 0	gera_pagamento	Calcula os salários com uma gratificação de 20% + abono R$120,00	7, 3, 1	pagamento3.pas	—	—	—
12, 5, 4	12, 5, 0	gera_pagamento	Calcula os salários com uma gratificação de 20% + abono R$120,00 + 13o. Salário	7, 3, 1	pagamento4.pas	—	—	—

A Tabela 5.21 mostra o comportamento do atributo `alive` na classe `Modulo`. É possível observar as versões do módulo *"gera_pagamento"* (entidade 12), e mais especificamente a versão `(12, 5, 2)` que foi desativada em "15/01/2003".

Tabela 5.21: Histórico do Atributo *Alive* da Classe *Modulo*

tvOID	Value	vTimeI	vTimeF	tTimeI	TtimeF
11, 5, 1	T	15/04/2002	—	15/04/2002	—
12, 5, 1	T	20/06/2002	—	20/06/2002	—
13, 5, 1	T	01/08/2003	—	01/08/2003	—
12, 5, 2	T	01/01/2003	15/01/2003	01/01/2003	15/01/2003
12, 5, 2	F	16/01/2003	—	16/01/2003	—
12, 5, 3	T	01/01/2003	—	01/01/2003	—
12, 5, 4	T	01/12/2003	—	29/11/2003	—

A Tabela 5.22 mostra o comportamento do atributo *Status* na Classe *Modulo.* Quando o módulo *"gera_pagamento"* (12, 5, 1) é derivado em *01/01/2003*, passa do estado em trabalho (W) para o estado estável (S). O mesmo ocorre com a versão `(12, 5, 3)` em *"29/11/2003"* . É possível observar ainda que a versão `(12, 5, 2)` foi desativada em *"15/01/2003"* .

Tabela 5.22: Histórico do Atributo *Status* da Classe *Modulo*

tvOID	value	VTimeI	vTimeF	TtimeI	TtimeF
11, 5, 1	W	15/04/2002	—	15/04/2002	—
12, 5, 1	W	20/06/2002	31/12/2002	20/06/2002	30/12/2002
13, 5, 1	W	01/08/2003	—	01/08/2003	—
12, 5, 1	S	01/01/2003	—	01/01/2003	—
12, 5, 2	W	01/01/2003	15/01/2003	01/01/2003	15/01/2003
12, 5, 2	D	16/01/2003	—	16/01/2003	—
12, 5, 3	W	01/01/2003	30/11/2003	01/01/2003	29/11/2003
12, 5, 3	S	01/12/2003	—	01/12/2003	—
12, 5, 4	W	01/12/2003	31/12/2003	01/12/2003	30/12/2003
12, 5, 4	W	01/01/2004	—	01/01/2004	—

A Tabela 5.23 mostra o comportamento do relacionamento temporal `Pertence` na classe `Modulo`, que relaciona o módulo com o projeto. Assim, observa-se que a versão `(12, 5, 1)` do módulo *"gera_pagamento"* deixou de pertencer ao módulo "*PESSOAL"* no dia *"31/12/2 002"*, quando passou a ser utilizada a versão que calcula os salários com uma gratificação e um abono `(12, 5, 3)`. Mas no mês de dezembro de 2003 passou a ser válida a versão que calcula adicionalmente o 13º. salário `(12, 5, 4)`. Já em janeiro de 2004, voltou a ser válida a versão `(12, 5, 3)`.

Tabela 5.23: Histórico do Relacionamento *Pertence* da Classe *Modulo*

tvOID	value	vTimeI	VTimeF	TtimeI	TtimeF
11, 5, 1	9, 4,1	01/05/2002	—	01/05/2002	—
13, 5, 1	10, 4, 1	01/08/2003	—	01/08/2003	—
12, 5, 1	9, 4, 1	01/07/2002	31/12/2002	30/06/2002	30/12/2002
12, 5, 2	9, 4, 1	01/01/2003	15/01/2003	01/01/2003	1501/2003
12, 5, 3	9, 4, 1	01/01/2003	30/11/2003	30/12/2002	29/11/2003
12, 5, 4	9, 4, 1	01/12/2003	31/12/2003	29/11/2003	30/12/2003
12, 5, 3	9, 4, 1	01/01/2004	—	30/12/2003	—

5.4.2 Metadados

Considerando as instâncias definidas nas seções anteriores, as tabelas dos metadados são preenchidas com os valores apresentados na Tabela 5.24 eTabela 5.25, representando, respectivamente, as entidades e predecessores/sucessores. A tabela que gerencia os ascendentes/descendentes não está sendo apresentada pois este exemplo não está utilizando o conceito de herança por extensão.

Tabela 5.24: Metadados, Tabela Entity

id	Nome
1	equipe1
2	equipe2
3	
4	
5	
6	
7	
8	
9	
10	
11	Cad_func
12	Pagamento
13	Cad_chamado

Tabela 5.25: Metadados, Tabela *PredSuc*

predecessor	sucessor	vTimeI	vTimeF	tTimeI	tTimeF
12, 5, 1	12, 5 ,2	01/01/2002	—	01/01/2002	—
12, 5, 1	12, 5, 3	01/01/2002	—	01/01/2002	—
12, 5, 3	12, 5, 4	01/12/2003	—	29/11/2003	—

Já a Tabela 5.27 apresenta a tabela VOC que controla os objetos versionados. No exemplo só foi criado um objeto versionado `(12, 5, 0)`. Por simplificação só é apresentada uma das tabelas auxiliares de VOC que representa o histórico do valor da versão corrente na Tabela 5.26.

Tabela 5.26: Metadados, Tabela *VOC*

TvOID	configCount	currentVersion	firstVersion	lastVersion	nextVNumber	userCurrentFlag	versionCount
12, 5, 0	—	—	1	—	5	F	—

Tabela 5.27: Metadados, Tabela *VOC_currentVersion*

value	VTimeI	vTimeF	tTimeI	tTimeF
12, 5, 1	01/01/2002	31/12/2002	01/01/2001	30/12/2002
12, 5, 3	01/01/2003	30/11/2003	30/12/2002	29/11/2003
12, 5, 4	01/12/2003	—	29/11//2003	—

5.5 Exemplos de Consultas

A partir das informações inseridas no *TVM Extender* é possível realizar algumas consultas envolvento aspectos temporais e versões.

5.5.1 Exemplo de Consulta Envolvendo Tempo

Um consulta que pode ser realizada é: *Quais os chefes das equipes de desenvolvimento em "20/12/2002"?* , que pode ser mapeada para:

```
SELECT E.nome, E.descricao, EC.value as chefe
FROM equipe E, equipeChefe EC
WHERE E.tvOID = EC.tvOID and
      EC.vTimeI < '20/12/2002' and
     (EC.vTimeF > '20/12/2002' or EC.vTimeF IS NULL)
```

5.5.2 Exemplo de Consulta Envolvendo Versões

Um exemplo de consulta utilizando o conceito de versões pode ser: *Qual a versão sucessora do módulo "(12, 5, 3)"?* Que pode ser convertido na *query*:

```
SELECT *
FROM modulo M
WHERE M.tvOID = getSuccessor(tvOID)
```

5.5.3 Exemplo de Consulta Envolvendo Versões e Tempo

Um exemplo de consulta abrangendo os dois conceitos seria: *Quais os dados da versão corrente do modulo "gera_pagamento" em 26/12/2003?* Esta pergunta pode ser respondida através do `SELECT`:

```
SELECT *
FROM modulo M, VOC V, VOC_CurrentVersion VC
WHERE M.name="gera_pagamento" and
      M.tvOID = VC.value and
      M.refVOC = V.tvOID and
      V.tvOID = VC.tvOID and
     (VC.vtimeI < '26/12/2003' and
     (VC.vtimeF > '26/12/2003' or VC.vTimeF is null))
```

6 CONCLUSÕES

Este trabalho propõe uma extensão para o DB2 que implementa o suporte aos conceitos de tempo e versões. Ela foi baseada no Modelo Temporal de Versões (TVM), o qual apresenta duas diferentes ordens de tempo, ramificado para o objeto e linear para cada versão, como também possibilita a presença de versões em diferentes níveis da hierarquia de tipos/classes

A extensão, denominada *TVM Extender*, engloba o mapeamento da hierarquia do TVM; a criação de tabelas administrativas; procedimentos para especificação das classes, atributos e relacionamentos; a definição de gatilhos e restrições para a manipulação dos dados diretamente nas tabelas criadas; a especificação de procedimentos e UDFs para controle de versões e valores temporais; e de outras UDFs que permitem consultas envolvendo os dois conceitos.

Assim, a especificação de um sistema utilizando o *TVM Extender* pode ser feita considerando as alternativas de projeto, como também a evolução histórica dos dados, e aproveitando ainda toda a infra-estrutura de um SGBD comercial como o DB2. Outra característica importante é a facilidade de integração com especificações existentes, pois não é obrigatório que todas as classes sejam temporais versionáveis.

Além da definição dos tipos complexos e tabelas administrativas, a base do trabalho é o gerenciamento das versões juntamente com a manipulação dos valores temporais. Um protótipo que permite o funcionamento básico do *TVM Extender* foi implementado para validar o trabalho. Os métodos que não foram implementados no protótipo podem ser facilmente codificados utilizando os mapeamentos definidos.

Como produção científica, este trabalho contribuiu para a publicação dos seguintes artigos:

1. COSTA, R. C.; SANTOS, C. S.; EDELWEISS, N. *Projeto de um DB2 Extender para Suporte aos Conceitos de Tempo e Versão.* In: Workshop de Teses e Dissertações em Bancos de Dados, WTDBD, 2., 2003, Manaus. **Anais...** Manaus: UFAM, 2003.
2. SILVA, F.; COSTA, R.C.; SANTOS, C. S.; EDELWEISS, N. *Using the Temporal Versions Model in a Software Configuration Management Environment*. In: Simpósio Brasileiro de Engenharia de Software, SBES, 17., 2003, Manaus. **Anais...** Manaus: UFAM, 2003.
3. COSTA, R. C.; SANTOS, C. S.; EDELWEISS, N. *Projeto de um DB2 Extender para Suporte aos Conceitos de Tempo e Versão.* In: Conferência Latino-americana de Informática, CLEI, 2003, La Paz, Bolívia. **Proceedings...** La Paz: [s.n], 2003.

E por fim, entre os aprimoramentos e sugestões de trabalhos futuros estão:

- Utilizar o trabalho como base para a manipulação de dados com tempo e versões em outros estudos em andamento, como a evolução de esquemas.
- Aperfeiçoar o protótipo para representar todas as funções, procedimentos, gatilhos e restrições definidas neste trabalho;
- Ampliar o estudo permitindo que o usuário consiga modificar as classes depois que forem geradas;
- Planejar e executar um estudo de análise do desempenho de aplicações que utilizam tempo e versões em conjunto;
- Desenvolver um plano de testes da solução desenvolvida buscando abordar de maneira tão exaustiva quanto possível as diferentes possibilidades.

REFERÊNCIAS

AGRAWAL, R.; BUROFF, S; GEHANI, N; SHASHA, D. Object versioning in Ode. In: IEEE INTERNATIONAL CONFERENCE ON DATA ENGINEERING, ICDE, 7., 1991, Kobe. **Proceedings**… Los Alamitos, IEEE Computer Society, 1991. p.446-455.

AL-KHUDAIR, A.; GRAY, W.A.; MILES, J.C. Object-Oriented Versioning in a Concurrent Engineering Design Environment. In: BRITISH NATIONAL CONFERENCE ON DATABASES, BNCOD, 18., 2001, Oxford, Inglaterra. **Proceedings**… [S.l.: s.n.], 2001. p. 105-125.

ANTUNES, D; HEUSER, C; EDELWEISS, N TempER: uma abordagem para modelagem temporal de banco de dados. **Revista de Informática Teórica e Aplicada,** Porto Alegre, v. 4, n.1, p. 49-85, 1997.

BEECH, D.; MAHBOD, B. Generalized Version Control in an Object-Oriented Database. In: IEEE INTERNATIONAL CONFERENCE ON DATA ENGINEERING, ICDE, 4., 1988, Los Angeles. **Proceedings**… Los Angeles: IEEE Computer Society, 1988. p.14-22.

BILIRIS, A. Modeling Design Object Relationship in PEGASUS. In: INTERNATIONAL CONFERENCE DATA ENGINEERING, ICDE, USA, 1990. **Proceedings**... [S.l.]: IEEE Computer Society, 1990. p. 228-236.

BISCHOFF, J**.** DB2 V.5 and Oracle 8 - What They' ve Got, What They' ve Not. **DB2 On-line Magazine**, Aug. 1997. Disponível em: <http://www.db2mag.com>. Acesso em: 12 dez. 2001.

BJÖRNERSTEDT, A.; HULTÉN, C. Version Control in an Object-Oriented Architecture. In: KIM, W.; LOCHOVSKY, F.H. (Ed.). **Object-Oriented Concepts, Databases, and Applications.** New York: ACM Press, 1989. p. 451-485.

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

CAREY, M. et al. O-O, What have they done to DB2? In: INTERNATIONAL CONFERENCE ON VERY LARGE DATA BASES, VLDB, 25., 1999, Edinburg (UK). **Proceedings**... Edinburg: [s.n.], 1999.

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 BANCO DE DADOS, 10.,1995, Recife. **Anais**... Recife: UFPE, 1995.

CELLARY, W.; JOMIER, G. Consistency of Versions in Object-Oriented Databases. In: CONFERENCE ON VERY LARGE DATA BASES, VLDB, 16., 1990, Brisbane. **Proceedings**…Brisbane : [s.n.], 1990. p. 432-441.

CHAMBERLIM, D. D. **Using the New DB2**: IBM´s Object-Relational Database System. San Francisco: Morgan Kaufmann, 1996.

CHENG, J.; XU, J. IBM DB2 XML Extender. In: INTERNATIONAL CONFERENCE ON DATA ENGINEERING, ICDE, 2000, San Diego. **Proceedings...** Disponível em: <http://www.ibm.com>. Acesso em: 13 abr. 2001.

CHIEN, S-Y.; TSOTRAS, V.J.; ZANIOLO, C. Version Management of XML Documents. In: WORLD WIDE WEB AND DATABASES, WEBDB, 3., 2000, Dallas, EUA. **Proceedings**... Berlin: Springer-Verlag, 200. p. 75-80.

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, Japan. **Proceedings**... Kyoto: [s.n.], 1986. p. 336-344.

CONRADI, R.; WESTFECHTEL, B. Version Models for Software Configuration Management. **ACM Computing Surveys**, New York, v.30, n.2, p.232-282, June 1998.

DADAM, P.; LUM, V.; WERNER, H. D. Integration of Time Versions into a Relational Database System. In: INTERNATIONAL CONFERENCE ON VERY LARGE DATA BASES, VLDB, 10., 1984, Singapore. **Proceedings…** San Mateo: Morgan Kauffman, 1984. p. 509-522.

DAVIS, J. R**. Creating An Extensible, Object-Relational Data Management Environment**: IBM's DB2 Universal Database. [S.l.]: DataBase Associates International, 1996.

EDELWEISS, N.; OLIVEIRA, J. P. M. **Modelagem de Aspectos Temporais de Sistemas de Informação**. Recife:UFPE – DI, 1994. Trabalho apresentado na Escola de Computação, 9., 1994.

EDELWEISS, N. Banco de Dados Temporais: teoria e prática. In: JORNADA DE ATUALIZAÇÃO EM INFORMÁTICA, JAI, 17.; CONGRESSO NACIONAL DA SOCIEDADE BRASILEIRA DE COMPUTAÇÃO, 18., 1998. **Anais**... Belo Horizonte: [s.n.], 1998. p 225-282.

EDELWEISS, N.; HÜBLER, P.; MORO, M.M.; DEMARTINI, G. A A Temporal Database Management System Implemented on Top of a Conventional Database. In: INTERNATIONAL CONFERENCE OF THE CHILEAN COMPUTER SOCIETY, SCCC, 20., 2000, Santiago do Chile. **Proceedings**…Santiago do Chile: [s.n.], 2000. p. 58-67

ELMASRI, R. et al. A temporal model and query language for EER databases. In: TANSEL, A. et al. (Ed.). **Temporal Databases:** Theory, Design, and Implementation. Redwood City: The Benjamin/Cummings, 1993. Chap. 9, p. 212-229.

GALANTE, R.et al. Dynamic Schema Evolution Management using Version in Temporal Object-Oriented Databases. In: INTERNATIONAL CONFERENCE ON DATABASE AND EXPERT SYSTEMS APPLICATIONS, 2002, Aix-en-Provence, France. **Database and Expert Systems Applications:** proceedings. Berlin:Springer-Verlag, 2002. p. 524-533.

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

HELD, G. D.; STONEBRAKER, M.; WONG, E. A relational data base management system. In: NATIONAL COMPUTER CONFERENCE, 1975, Anaheim, CA. **Proceedings**… Montvale:AFIPS, 1975. p. 409-416.

HUBLER, P. N. **Definição de um Gerenciador para o Modelo de Dados TF-ORM**. 2000. Dissertação (Mestrado em Ciência da Computação) – Instituto de Informática, UFRGS, Porto Alegre.

IBM CORPORATION. **Image, Audio, and Video Extenders Administration and Programming Version 7.** San Jose, U.S.A., 2000. Disponível em: <http://www.ibm.com/db2/extenders>. Acesso em: 10 dez. 2001.

IBM CORPORATION . **Application Development Guide Version 7.** San Jose, U.S.A., 2000. Disponível em: <http://www.ibm.com/ db2/extenders>. Acesso em: 15 out. 2001.

IBM CORPORATION. **XML Extender Administration and Programming Version 7.** IBM DB2 Universal Database. San Jose, U.S.A., 2000. Disponível em: <http://www.ibm.com/db2/extenders>. Acesso em: 23 nov. 2001.

IBM CORPORATION. **Text Extender Administration and Programming Version 7.** IBM DB2 Universal Database. San Jose, U.S.A, 2000. Disponível em: <http://www.ibm.com/db2/extenders>. Acesso em: 17 nov. 2001.

IBM CORPORATION. **DB2 Universal Database**. Disponível em: <http://www.ibm.com/db2> Acesso em: 16 nov. 2001.

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

KATZ, R. H. Toward a unified framework for version modeling in engineering databases. **ACM Computing Surveys**, New York, v. 22, p. 375-408, 1990.

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

MATTOS, N. **DB2 Object-Oriented Extensions and their Use to Build Relational Extenders.** San Jose, USA: IBM Database Technology Institute, 1994.

MATTOS, N. **DB2 Relational Extenders.** IBM BookManager Print Preview, 1995. Disponível em <http://booksrv2.raleigh.ibm.com>. Acesso em: 20 maio 2001.

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

MORO, M.M. et al. Adding Time to an Object-Oriented Versions Model. In: INTERNATIONAL CONFERENCE ON DATABASE AND EXPERT SYSTEM APPLICATIONS, DEXA, 2001, **Database and Expert Systems Applications:** proceedings. Berlin: Springer - Verlag, 2001. p. 805-814.

MORO, M. M.; GELATTI, P. C.; GOMES, C. H. P.; ROSSETTI, L. L. F.; ZAUPA, A.P.; EDELWEISS, N.; SANTOS, C.S. **Linguagem de Consultas para o Modelo Temporal de Versões**. Porto Alegre: PPGC da UFRGS, 2001. (RP-308).

NORTH, K. APIs for Universal Database Programming. **WebTechniques Magazine**, 1998. Disponível em: <http://www.webtechniques.com/archives/1998/08/data>. Acesso em 25 out. 2001.

PEIXOTO, C. E. L.; GASPARY, D. F.; MORO, M. M.; EDELWEISS, N. Ferramenta de Apoio à Especificação de Classes do Modelo Temporal de Versões. In: SALÃO DE INICIAÇÃO CIENTÍFICA, 14., 2002. **Livro de Resumos**. Porto Alegre, UFGRS, 2002.

RENNHACKKAMP M. **Extending Relational DBMS**. [S.l.]: Miller Freeman, 1997. Disponível em: <http://www.dbmsmag.com>. Acesso em: 27 jun. 2001.

RODRÍGUEZ, L; OGATA, H.; YANO, Y. TVOO: A Temporal Versioned Object-Oriented data model. **Information Sciences**, [S.l.], v.114, n.1-4, p. 281-300, Mar. 1999.

SIMONETTO, E. O. **Uma Proposta para a Incorporação de Aspectos Temporais, no Projeto Lógico de Bancos de Dados, em SGBDs Relacionais**. 1998. 73 p. Dissertação (Mestrado) – PUC-RS, Porto Alegre.

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

TANSEL, C. G. et Al. (Ed.). **Temporal Databases – Theory, Design and Implementation**. RedwoodCity: The Benjamin/Cummings, 1993.

WANG, H.; ZANIOLO, C. Using SQL to Build New Aggregates and Extenders for Object-Relational Systems, In: INTERNATIONAL CONFERENCE ON VERY LARGE DATA BASES, VLDB, 26., 2000, Cairo, Egypt. **Proceedings…** Cairo: [s.n.], 2000.

WUU, G. T. J.; DAYAL, U. A Uniform Model for Temporal and Versioned Object-Oriented Databases. In: TANSEL, A. et al. (Ed.). **Temporal Databases:** Theory,

Design, and Implementation. Redwood City: The Benjamin/Cummings, 1993. Chap. 10, p. 230-247.

YOUNG, C. It' s Not Just Data Anymore.**DB2 On-line Magazine,** Spring, 1997. Disponível em: <http://www.db2mag.com>. Acesso em: 27 nov. 2001.

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

ANEXO A METADADOS E HIERARQUIA DO TVM

Os comandos que o TVM Extender utiliza para criar os metadados são apresentados no script abaixo:

```
CONNECT TO SAMPLE USER TVM_USER USING TVM_USER;

CREATE TABLE Entity (id    integer NOT NULL, name varchar(30))
       CONSTRAINT EntityPK
       PRIMARY KEY (id);

CREATE TABLE Schema (id integer NOT NULL, name varchar(30))
       CONSTRAINT SchemaPK
       PRIMARY KEY (id);

CREATE TABLE Class (id     integer NT NULL,name varchar(30),
                    tempVers char(1), root char(1),
             generated char(1),defined char(1),schemaId integer)
       CONSTRAINT ClassPK
       PRIMARY KEY (id)
       FOREIGN KEY schemaId
       REFERENCES Schema;

CREATE TABLE Attribute (id  integer NOT NULL, name varchar(30),
                  temp char(1), type varchar(20), static char(1),
                        default  varchar(30), classId integer)
     CONSTRAINT AttributePK PRIMARY KEY (id) FOREIGN KEY classId
     REFERENCES Class;

CREATE TABLE Relationship (id integer NOT NULL,name varchar(30),
                           temp char(1), inverse char(1),
                           cardinality varchar(4), single char(1),
                           classId integer, relatedClass integer)
       CONSTRAINT RelationshipPK
       PRIMARY KEY (id)
       FOREIGN KEY classId
       REFERENCES Class;

CREATE TABLE AscDesc( seqAD : integer,
                      ascendant OIDt NOT NULL,
```

```
                    descendant OIDt NOT NULL,
                  tTimeI timestamp NOT NULL, tTimeF timestamp,
                  vTimeI timestamp NOT NULL,vTimeF timestamp)
       CONSTRAINT AscDescPK
       PRIMARY KEY (seqAD)
       CONSTRAINT AscDescUK
       UNIQUE KEY (ascendant, descendant, tTimeI, vTimeI);

CREATE TABLE PredSucc (seqPS: integer,
                    predecessor OIDt, successor OIDt,
                    tTimeI    timestamp, tTimeF timestamp,
                    vTimeI timestamp, vTimeF timestamp)
       CONSTRAINT PredSuccPK
       PRIMARY KEY (seqPS: integer)
       CONSTRAINT PredSuccUK
       UNIQUE KEY (predecessor, sucessor, tTimeI, vTimeI);

CREATE TABLE VOC  (tvOID OIDt, currentVersion OIDt, firstVersion
                  OIDt, lastVersion OIDt, nextVNumber integer,
                  userCurrentF char(1), versionCount integer);

CREATE TABLE VOClastVersion (tvOID OIDt, value OIDt,
                         TTimeI timestamp, tTimeF  timestamp,
                         vTimeI timestamp, vTimeF  timestamp);

CREATE TABLE VOCcurrentVersion (tvOID OIDt, value OIDt,
                        tTimeI  timestamp, tTimeF  timestamp,
                        vTimeI  timestamp, vTimeF  timestamp);

CREATE TABLE VOCuserCurrentF (tvOID OIDt, value char(1),
                        tTimeI  timestamp, tTimeF  timestamp,
                        vTimeI  timestamp, vTimeF  timestamp);

CREATE TABLE VOCfirstVersion (tvOID OIDt, value OIDt,
                        tTimeI  timestamp, tTimeF  timestamp,
                        vTimeI  timestamp, vTimeF  timestamp);

CREATE TABLE VOCversionCount (tvOID OIDt, value integer,
                        tTimeI  timestamp, tTimeF  timestamp,
                        vTimeI  timestamp, vTimeF  timestamp);
QUIT;
```

O script abaixo é utilizado para criar da hierarquia do TVM:

```
CONNECT TO SAMPLE user tvm_user using tvm_user;

CREATE DISTINCT TYPE OIDt AS VARCHAR(30)
```

```
WITH COMPARISONS;

CREATE TYPE TemporalLabel AS (tTimeI timestamp, tTimeF
timestamp, vTimeI  timestamp, vTimeF  timestamp)
MODE DB2SQL;

CREATE TYPE Object AS (refVOC OIDt)
NOT INSTANTIABLE MODE DB2SQL;
// O atributo tvOID só é adicionado no momento da criação da
// typed table como exige a sintaxe do DB2

CREATE TYPE TemporalObject UNDER Object as (alive  char(1)  )
NOT INSTANTIABLE
MODE DB2SQL;

CREATE TYPE TemporalVersion UNDER TemporalObject as (status
char(1), configuration char(1))
NOT INSTANTIABLE;

QUIT;
```

O script a seguir é responsável por criar algumas funções auxiliares que realizam a separação dos rótulos temporais:

```
CREATE FUNCTION get_value(s VARCHAR(200)) RETURNS VARCHAR(50)
LANGUAGE SQL
CONTAINS SQL
NO EXTERNAL ACTION
DETERMINISTIC
RETURN SUBSTR(s,1,POSSTR(s,'|')-1);

CREATE FUNCTION get_vTimeI(s VARCHAR(200)) RETURNS VARCHAR(50)
LANGUAGE SQL
CONTAINS SQL
NO EXTERNAL ACTION
DETERMINISTIC
RETURN SUBSTR(SUBSTR(s,LENGTH(get_value(s))+2),1,
             POSSTR(SUBSTR(s,LENGTH(get_value(s))+2),'|')-1);

CREATE FUNCTION get_vTimeF(s VARCHAR(200)) RETURNS VARCHAR(50)
LANGUAGE SQL
CONTAINS SQL
NO EXTERNAL ACTION
DETERMINISTIC
return
substr(substr(s,length(get_value(s))+length(get_vtimei(s))+3),1,
posstr(substr(s,length(get_value(s))+length(get_vtimei(s))+3),
'|')-1);
```

ANEXO B *SCRIPTS* DE ESPECIFICAÇÃO DE CLASSES

Neste anexo são apresentados os códigos-fonte dos procedimentos armazenados para especificação das classes do *TVM Extender*. São eles: defineSchema, defineClass, addAttribute, addRelationship

```
/*****************************************************************
/**
 * JDBC Procedimento Armazenado TVM_USER.createSchema
 */
package especificacao;

import java.sql.*;                    // Classes JDBC

public class CreateSchema
{
    public static void createSchema ( String[] SQLSTATE,
                                      String schemaName,
                                      String[] schemaId,
                                      ResultSet[] rs ) throws
Exception
    {
        // Obter conexão com o banco de dados
        Connection con =
             DriverManager.getConnection("jdbc:default:connection");
        PreparedStatement stmt = null;
        int updateCount = 0;
        String id = "0";
        String sql;

        try
        {
            sql = "SELECT id FROM schema WHERE name ='"+schemaId+"'";
            stmt = con.prepareStatement( sql );
            rs[0] = stmt.executeQuery();

            if (!rs[0].next())
            {
                sql = "select max(id)+1 id from schema";
                stmt = con.prepareStatement( sql );
                rs[0] = stmt.executeQuery();
                rs[0].next();
                id = rs[0].getString(1);

                sql = "insert into schema (id,name) values
                      ("+id+",'"+schemaName+"')";
                stmt = con.prepareStatement( sql );
                updateCount = stmt.executeUpdate();
                schemaId[0] = id;
```

```
            }
            else
            {
               schemaId[0] =  "-1";
            }
            if (con != null) con.close();

            // Definir parâmetros de retorno
            SQLSTATE[0] = "00000";  // Bom SQLSTATE
        }
        catch (SQLException e)
        {
            // Definir parâmetros de retorno
            SQLSTATE[0] = e.getSQLState();

            // Fechar recursos abertos
            try {
                if (rs[0] != null) rs[0].close();
                if (rs[1] != null) rs[1].close();
                if (stmt != null) stmt.close();
                if (con != null) con.close();
            } catch (SQLException e2) { /* ignorar */ };
        }
    }
}
/*********************************************************************
/**
 * JDBC Procedimento Armazenado TVM_USER.defineClass
 */
package especificacao;

import java.sql.*;                  // Classes JDBC

public class DefineClass
{
    public static void defineClass ( String[] SQLSTATE,
                                     String className,
                                     String schemaId,
                                     String tempVers,
                                     String root,
                                     String[] classId,
                                     ResultSet[] rs ) throws Exception
    {
        // Obter conexão com o banco de dados
        Connection con =
              DriverManager.getConnection("jdbc:default:connection");
        PreparedStatement stmt = null;
        int updateCount = 0;
        boolean bFlag;
        String sql;
        String id = "0";
        try
        {
           sql = "SELECT id FROM class where name='"+className+"'";
           stmt = con.prepareStatement( sql );
           rs[0] = stmt.executeQuery();

           if (!rs[0].next())
           {
               sql = "SELECT max(id)+1  FROM class";
              stmt = con.prepareStatement( sql );
```

```
            rs[0] = stmt.executeQuery();
            rs[0].next();
            id = rs[0].getString(1);

            sql = "INSERT INTO class(id, name, tempVers, root,
                   schemaId, defined, "+ "generated) VALUES (" + id
                   + ",'"+ className+"','"+ tempVers +"','"+ root
                   + "'," + schemaId + ",'F','F')";
            stmt = con.prepareStatement(sql);
            updateCount = stmt.executeUpdate();
            classId[0] = id;

            if (tempVers.equals("F"))
            {
                  sql = "CREATE TYPE "+ className + "_T UNDER object
                         MODE DB2SQL";
                  stmt = con.prepareStatement( sql );
                  bFlag = stmt.execute();
            }
            else
            {
                  sql = "CREATE TYPE "+ className + "_T UNDER
                         temporalVersion  MODE DB2SQL";
                  stmt = con.prepareStatement( sql );
                  bFlag = stmt.execute();
                  // Inserindo o atributo status nos metadados
                  sql_attr = " addAttribute(“+id+”,‘status’,”+
                             “char(1), ‘T’, ‘ ‘, ‘T’, ‘’)";
                  stmt_attr = com.prepareStatement(sql_attr);
                  bFlag = stmt_attr.execute();

                  // Inserindo o atributo alive nos metadados
                  sql_attr = "addAttribute(“+id+”,‘alive’,”+
                             “char(1), ‘T’, ‘ ‘, ‘T’, ‘’)";
                  stmt_attr = com.prepareStatement(sql_attr);
                  bFlag = stmt_attr.execute();
            }

          }
          else
          {
             classId[0] = "-1";
          }
          if (con != null) con.close();

          // Definir parâmetros de retorno
          SQLSTATE[0] = "00000";  // Bom SQLSTATE
      }
      catch (SQLException e)
      {
          // Definir parâmetros de retorno
          SQLSTATE[0] = e.getSQLState();

          // Fechar recursos abertos
          try {
              if (rs[0] != null) rs[0].close();
              if (stmt != null) stmt.close();
              if (con != null) con.close();
          } catch (SQLException e2) { /* ignorar */ };
      }
}
```

```
}

/*****************************************************************
 * JDBC Procedimento Armazenado TVM_USER.addAttribute
 */
package especificacao;

import java.sql.*;                    // Classes JDBC

public class AddAttribute
{
    public static void addAttribute ( String[] SQLSTATE,
                                      int classId,
                                      String name,
                                      String type,
                                      String temp,
                                      String visibility,
                                      String statical,
                                      String defaultValue,
                                      String[] id,
                                      ResultSet[] rs )
                                      throws Exception
    {
        // Obter conexão com o banco de dados
        Connection con =
              DriverManager.getConnection("jdbc:default:connection");
        PreparedStatement stmt = null;
        int updateCount = 0;
        boolean bFlag;
        String sql;
        String generated;

        try
        {
           sql = "SELECT generated, name FROM class WHERE id = "
                 +classId;
           stmt = con.prepareStatement( sql );
           rs[0] = stmt.executeQuery();

           if (rs[0].next())
           {
              generated = rs[0].getString(1);
              if (generated.equals("F"))
              {
                sql = "ALTER TYPE "+ rs[0].getString(2) +
                      "_T ADD ATTRIBUTE "+name+" "+type;
                stmt = con.prepareStatement( sql );
                bFlag = stmt.execute();

                sql = "SELECT max(id)+1 FROM attribute";
                stmt = con.prepareStatement( sql );
                rs[0] = stmt.executeQuery();
                rs[0].next();
                id[0] = rs[0].getString(1);


                sql = "INSERT INTO attribute (id,classId, name,type,
                       temp, static, default) " +
                      " VALUES ("+id[0]+","+classId+",'"+name+"','"
                      +type+"','"+temp +"','"+statical+"','"
                      +defaultValue+"') ";
```

```
                stmt = con.prepareStatement( sql );
                updateCount = stmt.executeUpdate();

             }
            }
            if (con != null) con.close();

            // Definir parâmetros de retorno
            SQLSTATE[0] = "00000";  // Bom SQLSTATE
        }
        catch (SQLException e)
        {
            // Definir parâmetros de retorno
            SQLSTATE[0] = e.getSQLState();

            // Fechar recursos abertos
            try {
                if (rs[0] != null) rs[0].close();
                if (stmt != null) stmt.close();
                if (con != null) con.close();
            } catch (SQLException e2) { /* ignorar */ };
        }
    }
}
/*****************************************************************
/**
 * JDBC Procedimento Armazenado TVM_USER.addRelationship
 */
package especificacao;

import java.sql.*;                  // Classes JDBC

public class AddRelationship
{
    public static void addRelationship ( String[] SQLSTATE,
                                         int sourceClass,
                                         int destClass,
                                         String name,
                                         String temp,
                                         String cardinality,
                                         String inverse,
                                         String single,
                                         String[] id,
                                         ResultSet[] rs ) throws
Exception
    {
        // Obter conexão com o banco de dados
        Connection con =
              DriverManager.getConnection("jdbc:default:connection");
        PreparedStatement stmt = null;
        int updateCount = 0;
        boolean bFlag;
        String sql;
        String generated;
//        String id;

        try
        {
           sql = "SELECT generated, name FROM class WHERE id =
"+sourceClass;
           stmt = con.prepareStatement( sql );
```

```
            rs[0] = stmt.executeQuery();

            if (rs[0].next())
            {
               generated = rs[0].getString(1);
               if (generated.equals("F"))
               {
                 sql = "ALTER TYPE "+ rs[0].getString(2) +
                       "_T ADD ATTRIBUTE "+name+ " varchar(30)" ;
                 stmt = con.prepareStatement( sql );
                 bFlag = stmt.execute();

                 sql = "SELECT max(id)+1 FROM relationship";
                 stmt = con.prepareStatement( sql );
                 rs[0] = stmt.executeQuery();
                 rs[0].next();
                 id[0] = rs[0].getString(1);


                 sql = "INSERT INTO relationship
                       (id,classId,relatedClass, name, temp, inverse,
                        cardinality, single) "+
                       " VALUES ("+id[0]+","+sourceClass+","+destClass
                       +",'"+name+"','"+temp+"','"+inverse+"','"+
                        cardinality+"','"+single+"') ";
                 id[0] = sql;
                 stmt = con.prepareStatement( sql );
                 updateCount = stmt.executeUpdate();

              }
             }
             if (con != null) con.close();

             // Definir parâmetros de retorno
             SQLSTATE[0] = "00000";  // Bom SQLSTATE
         }
         catch (SQLException e)
         {
             // Definir parâmetros de retorno
             SQLSTATE[0] = e.getSQLState();

             // Fechar recursos abertos
             try {
                 if (rs[0] != null) rs[0].close();
                 if (stmt != null) stmt.close();
                 if (con != null) con.close();
             } catch (SQLException e2) { /* ignorar */ };
         }
     }
}



/*********************************************************************
/**
 * JDBC Procedimento Armazenado TVM_USER.generateClasses
 */
package especificacao;

import java.sql.*;                   // Classes JDBC
```

```
public class GenerateClasses
{
    public static void generateClasses ( String[] SQLSTATE,
                                         String[] id,
                                         ResultSet[] rs )
                                                   throws Exception
    {
        // Obter conexão com o banco de dados
        Connection con =
               DriverManager.getConnection("jdbc:default:connection");
        PreparedStatement stmt = null, stmt_attr = null, stmt_rel =
                           null, stmt_temp = null, stmt_trigger = null
;
        int updateCount = 0;
        boolean bFlag;
        String sql,sql_attr,sql_rel,sql_tempsql_trigger;
        String generated;
        ResultSet rs_attr,rs_rel,rs_temp;

        try
    {
           sql = "SELECT id,name from Class where generated = 'F'";
           stmt = con.prepareStatement( sql );
           rs[0] = stmt.executeQuery();

           while (rs[0].next())
           {
              id[0] = rs[0].getString(1);

              sql_attr = "SELECT id,name,type,temp from attribute
                          where classId = "+id[0];
              stmt_attr = con.prepareStatement( sql_attr );
              rs_attr = stmt_attr.executeQuery();

              sql_rel = "SELECT id,name,temp from relationship where
                         classId = "+id[0];
              stmt_rel = con.prepareStatement( sql_rel );
              rs_rel = stmt_rel.executeQuery();


              sql = "CREATE TABLE "+rs[0].getString(2)+" OF "+
                   rs[0].getString(2)+"_T (REF IS tvOID USER
                   GENERATED)";
              stmt = con.prepareStatement( sql );
              bFlag = stmt.execute();

              // Atributos
              while (rs_attr.next())
              {
             // Cria as tabelas auxiliares para os atributos temporais
                 if (rs_attr.getString(4).equals("T"))
                 {
                   sql_temp = "CREATE TABLE "+rs[0].getString(2)
                            +"_"+rs_attr.getString(2) +" "
                            +"(tvOID OIDt,value " +rs_attr.getString(3)
                            +",vTimeI timestamp,vTimeF timestamp,tTimeI
                             timestamp,tTimeF timestamp)" ;

             // Inserir as constraints e verificar o OIDt

                   stmt_temp = con.prepareStatement( sql_temp );
```

```
                    bFlag = stmt_temp.execute();
// Trigger que insere o valor do atributo temporal na tabela auxiliar
                    sql_trigger = "CREATE TRIGGER " +rs[0].getString(2)
                                  +"_"+ rs_attr[0].getString(1)+"_AI1 "
                                 +"AFTER INSERT ON "+rs[0].getString(2)+
                                 + " REFERENCING NEW AS N "
                                 + "FOR EACH ROW MODE DB2SQL "
                                 + "BEGIN ATOMIC "
                                 + "INSERT INTO "+rs[0].getString(2)+"_"
                                 + rs_attr.getString(2) +" (tvOID,value,"+
                                 +"vTimei,vtimef,tTimei,ttimef) VALUES "
                                 +"(N.tvOID,get_value(N."+
                           rs_attr.getString(2)+"),get_vtimei(N."+
                           rs_attr.getString(2) +"),get_vtimef(N."+
                           rs_attr.getString(2)+"),CURRENT TIMESTAMP)" +
                           ";END;";
                    stmt_trigger = con.prepareStatement( sql_trigger );
                    bFlag = stmt_trigger.execute();

//Trigger que encerra os tempos finais do atributo antigo para o tempo
// inicial do novo valor - 1 dia
                   sql_trigger = "CREATE TRIGGER " +rs[0].getString(2)
                                  +"_"+ rs_attr[0].getString(1)+" +"_AI2"
                                  +" AFTER INSERT ON "+rs[0].getString(2)+
                                 "_"+ rs_attr[0].getString(2)+
                                 " REFERENCING NEW AS N "+
                                 " FOR EACH ROW MODE DB2SQL "+
                                 " BEGIN ATOMIC "+
                                 " UPDATE "+rs[0].getString(2)+"_"+
                                 rs_attr[0].getString(2)+" SET VTIMEF="+
                                 "N.vtimei- 1 day, TTIMEF=N.vtimei-1 day"+
                                 "WHERE (VTIMEF = '1900-01-01-00.00.00'"
                                 " and (value <> N.value) and "+
                                 "(tvOID=N.tvOID));END;";
                    stmt_trigger = con.prepareStatement( sql_trigger );
                    bFlag = stmt_trigger.execute();
                   }
                 }
                 // Relacionamentos
                 while (rs_rel.next())
                 {
                   // Cria as tabelas auxiliares para os relacionamentos
                   // temporais
                   if (rs_rel.getString(3).equals("T"))
                   {
                      // Incluir tratamento para os diferentes
                      // tipos de relacionamentos
                      sql_trigger = "CREATE TABLE "+rs[0].getString(2)+
                                    "_"+rs_rel.getString(2) +" "+
                                    "(tvOID OIDt,value OIDt,vTimeI
                                    timestamp,vTimeF timestamp,tTimeI
                                    timestamp,tTimeF timestamp)" ;
```

```
                    stmt_trigger = con.prepareStatement( sql_temp );
                    bFlag = stmt_temp.execute();
// Trigger que insere o valor do relacionamento temporal na tabela
// auxiliar
                sql_trigger = "CREATE TRIGGER " +rs[0].getString(2)
                             +"_"+ rs_rel[0].getString(1)+"_AI1 "
                            +"AFTER INSERT ON "+rs[0].getString(2)+
                           + " REFERENCING NEW AS N "
                           + "FOR EACH ROW MODE DB2SQL "
                           + "BEGIN ATOMIC "
                           + "INSERT INTO "+rs[0].getString(2)+"_"
                           + rs_rel.getString(2) +" (tvOID,value,"+
                           +"vTimei,vtimef,tTimei,ttimef) VALUES "
                           +"(N.tvOID,get_value(N."+
                      rs_rel.getString(2)+"),get_vtimei(N."+
                      rs_rel.getString(2) +"),get_vtimef(N."+
                      rs_rel.getString(2)+"),CURRENT TIMESTAMP)" +
                      ";END;";
                stmt_trigger = con.prepareStatement( sql_trigger );
                bFlag = stmt_trigger.execute();

//Trigger que encerra os tempos finais do relacionamento antigo para o
// tempo inicial do novo valor - 1 dia
               sql_trigger = "CREATE TRIGGER " +rs[0].getString(2)
                            +"_"+ rs_rel[0].getString(1)+" +"_AI2"
                            +" AFTER INSERT ON "+rs[0].getString(2)+
                           "_"+ rs_rel[0].getString(2)+
                           " REFERENCING NEW AS N "+
                           " FOR EACH ROW MODE DB2SQL "+
                           " BEGIN ATOMIC "+
                           " UPDATE "+rs[0].getString(2)+"_"+
                           rs_rel[0].getString(2)+" SET VTIMEF="+
                           "N.vtimei- 1 day, TTIMEF=N.vtimei-1 day"+
                           "WHERE (VTIMEF = '1900-01-01-00.00.00'"
                           " and (value <> N.value) and "+
                           "(tvOID=N.tvOID));END;";
                stmt_trigger = con.prepareStatement( sql_trigger );
                bFlag = stmt_trigger.execute();
                }
              }
              sql_temp = "UPDATE CLASS SET GENERATED='T' WHERE
                          ID="+id[0];
              stmt_temp = con.prepareStatement(sql_temp);
              updateCount = stmt_temp.executeUpdate();
          }

          if (con != null) con.close();

          // Definir parâmetros de retorno
          SQLSTATE[0] = "00000";  // Bom SQLSTATE
      }
      catch (SQLException e)
```

```
        {
            // Definir parâmetros de retorno
            SQLSTATE[0] = e.getSQLState();

            // Fechar recursos abertos
            try {
                if (rs[0] != null) rs[0].close();
                if (stmt != null) stmt.close();
                if (con != null) con.close();
            } catch (SQLException e2) { /* ignorar */ };
        }

    }
}
```