Rodolfo Schulz de Lima

CPPObjects: Biblioteca de Mapeamento Objeto-Relacional em C++

Orientador:
Sergio Barbosa Villas-Boas, Ph.D

UNIVERSIDADE FEDERAL DO RIO DE JANEIRO
ESCOLA POLITÉCNICA
DEPARTAMENTO DE ELETRÔNICA

Rio de Janeiro/RJ - Brasil

Novembro (2008)

Resumo

Este projeto descreve a implementação de uma biblioteca na linguagem de programação C++ chamada CPPObjects que visa facilitar o desenvolvimento de aplicações que lidam com informações armazenadas em bancos de dados relacionais. Isto é feito através de procedimentos que mapeiam o modelo relacional em um modelo orientado a objetos com ênfase na possibilidade de navegação entre os objetos relacionados através de referências, tal como em um modelo em rede.

Partindo de uma descrição lógica das entidades a serem armazenadas e seus relacionamentos entre si (diagrama ER), a biblioteca oferece ferramentas que possibilitam o mapeamento desta descrição para código-fonte em C++ de uma forma direta e eficiente utilizando o paradigma de modelagem orientada a objetos. Nestes modelo, cada objeto representa uma entidade do domínio e pode estar ligado a zero ou mais objetos relacionados, e outros objetos por sua vez podem estar ligados a ele, formando assim uma rede de objetos onde a partir de um objeto se chega a qualquer outro que esteja direta ou indiretamente relacionado a ele. De posse da descrição do sistema em C++, a biblioteca utiliza as informações passadas sobre este sistema (metainformações) para poder criar o esquema de banco de dados respeitando as relações entre os objetos, suas cardinalidades e os tipos e propriedades dos atributos de cada entidade.

Uma vez criado o banco de dados, o desenvolvedor o popula instanciando objetos e preenchendo seus atributos e os de seus objetos relacionados. Partindo então de um objeto da rede, o desenvolvedor por meio da biblioteca inclui o objeto no banco de dados. Todos os objetos relacionados direta ou indiretamente ao objeto inserido serão também inseridos no banco.

Com o banco de dados populado, o desenvolvedor pode realizar então consultas fazendo o uso ou não de filtros que restringem o conjunto de objetos retornados. Estes filtros são criados utilizando-se uma sintaxe própria da linguagem C++ que se assemelha à sintaxe das cláusulas *WHERE* da linguagem SQL. Cada objeto retornado pela consulta está ligado aos seus objetos relacionados e estes podem ser acessados diretamente. Alterações em objetos que já armazenados podem ser realizadas. Ao atualizar o banco de dados, a biblioteca somente irá atualizar os objetos que foram modificados, inserindo, atualizando ou removendo objetos de forma que a rede de objetos alterada pelo usuário seja refletida no banco, de forma eficiente.

Diversas técnicas de desenvolvimento foram utilizadas para tornar a linguagem C++ própria para a representação e manipulação das entidades e seus relacionamentos. Aproveitando-se que ela aceita vários paradigmas de desenvolvimento, técnicas como programação genérica, programação orientada a objetos, meta-programação entre outras, foram empregadas visando ganhos em otimização tanto temporal quanto espacial, checagem de erros já em tempo de compilação, diminuição de tempo de compilação de aplicações e a minimização das situações onde uma alteração na biblioteca, seja por mudança de funcionalidade ou correção de defeitos, não acarrete a necessidade de recompilação de aplicações que dependam dela.

Abstract

This project describes the implementation of a software library called CPPObjects written in C++ that eases the development of database-driven applications with relational data storage. This is achieved by creating an object-oriented interface that isolates the application from the relational backend, with emphasis on object navigation capabilities, making it easy to access related objects, as it is on the network database model.

Based on a logic description of the entities of the domain and its relationships (ER diagram), the library offers tools that enable the developer to map this logic description directly and efficiently into C++ code by using an object-oriented model paradigm. In this model, each object represents an entity in the domain that can be linked to zero or more related objects, and other objects can be linked back to it, hence creating an object network where beginning on one object, other entities can be reached that may be direct or indirectly related to it. Once having the description of the system in C++, the library uses this information (called metainformation) to create the database schema taking into account the relationships between entities, their cardinalities and each entity's attribute type and properties.

Once created, the database can be populated by instantiating objects, setting their attributes and linking more objects to them, creating a network of related objects. The developer then takes one object of this network and uses the library to save it into the database. All related objects will automatically be saved and the whole network will be reflected in the database.

The developer can make queries to a populated database by using filters that restrict the retrieved dataset. These filters are created using directly written C++ code that mimics SQL's *WHERE* clause. Each object retrieved will already be linked to its related objects that can be accessed directly. These objects can be modified by altering its attributes, adding, modifying and removing related objects. When updated into database, the object will be processed by the library and all modifications will be detected so that a minimal set of update statements will be issued to the database management system. This way the new network object structure will be reflected into the database efficiently.

Several development techniques were employed to make the C++ language represent directly the entities and its relationships, as well as their manipulation. Based on the fact that C++ is a multi paradigm language, techniques such as generic programming, object-oriented programming, meta programming among others were employed with the goal of minimizing temporal and spatial bottlenecks as well as detecting early in the compile stage errors in the representation of the ER diagram in code by using static type checking capabilities of C++. A mix of generic programming and object-oriented programming were used to minimize compiling time and avoiding whole application recompilation due to modifications in the library.

Palavras-Chave

- C++
- base de dados
- programação genérica
- programação orientada a objetos
- modelo relacional
- objetos de negócio
- mapeamento objeto-relacional
- RDBMS
- OODBMS

Agradecimentos

Agradeço primeiramente à minha esposa Tânia pela sua paciência durante as várias noites em que tive que ficar isolado trabalhando no projeto ao invés de ficar ao lado dela.

Minha família por indiretamente ter me influenciado na minha decisão de seguir a carreira de Engenheiro Eletrônico, sempre me incentivando a terminar o curso nas horas que o peso de ter que trabalhar e estudar ao mesmo tempo me parecia ser maior do que eu poderia suportar.

Meu pai, José, por ter me dado uma outra perspectiva do que se pode fazer com computadores quando eu tinha 10 anos de idade. Através de um livro de programação usando Turbo Basic, eu saí de um mundo onde eu só jogava joguinhos em um velho IBM-PC/XT para outro novo onde eu atuava mais ativamente através da programação, e isto acabou tornando-se meu "joguinho" preferido, situação que se mantém até os dias de hoje.

Meu atual empregador, Ruben Zonenschein, por ter me permitido ter um horário mais flexível de trabalho para que eu pudesse concluir as últimas disciplinas da graduação. Seu apoio e encorajamento foram muito importantes para tornar este projeto viável, dado o cronograma apertado nas suas etapas finais.

Finalmente, agradeço ao povo brasileiro por ter pago meu curso de Engenharia através de seus impostos.

Conteúdo

Lista de abreviaturas e siglas

ER	Entity-Relationship,	p. i
SQL	Structured Query Language,	p. i
DLL	Dynamically Linked Library,	p. 2
RDBMS	Relational Database Management System,	p. 3
OO	Orientação a Objetos,	p. 3
ACID	Atomicidade, Consistência, Isolamento e Durabilidade,	p. 7
API	Application Programming Interface,	p. 10
SGD	Sistema de Gerenciamento de Disciplinas,	p. 14
RTTI	Runtime Type Information,	p. 17
STL	Standard Template Library,	p. 19
URL	Uniform Resource Locator,	p. 27
DBMS	Database Management System,	p. 29
POSIX	Portable Operating System Interface,	p. 33
GUI	Graphical User Interface,	p. 52
ORM	Object-Relational Mapping,	p. 59
EOF	Enterprise Objects Framework,	p. 64

Lista de Figuras

Listagem de Códigos-Fonte

Lista de Tabelas

1 Introdução

Banco de dados tem sido parte da história da computação desde os anos 60 e sua importância tem aumentado consideravelmente nas duas últimas décadas, dado que computadores se popularizam e sistemas informatizados estão em primeiro plano, tendo que lidar com informações de vários indivíduos de forma rápida, organizada e eficiente.

Várias formas de organizar dados foram propostas ao longo deste período, porém uma em especial tornou-se o padrão *de facto* para armazenagem e consulta de dados. O modelo relacional, proposto por E. F. Codd em 1970 tem por base sólidos conceitos matemáticos (ELMASRI; NAVATHE, 1999, p. 163) e é adequado para lidar com um grande volume de dados de forma rápida e eficiente, criando uma infraestrutura que permite consultas precisas, além de garantir a integridade dos dados armazenados e seus relacionamentos.

Apesar de o modelo relacional simplificar enormemente a organização e recuperação de dados, muitos modernos paradigmas de projeto de software, especialmente o da orientação a objetos, não se adequam perfeitamente a ele, já que neste caso não há o conceito de objeto no modelo relacional.

A modelagem orientada a objetos tem sido solução a uma grande gama de problemas, facilitando a organização e desenvolvimento de softwares cada vez mais complexos e robustos. Vários destes sistemas podem ser modelados como objetos que são manipulados direta ou indiretamente pelo usuário ou por outros sistemas de forma a atingir um certo objetivo. A programação orientada a objetos minimiza a distância entre o modelo e o código por permitir que o programador lide diretamente com o primeiro em termos do segundo.

Temos então duas soluções para a mesma situação, a modelagem relacional e a modelagem orientada a objetos, sendo que um não se adequa bem à ao contexto do outro e vice-versa. Quando ambos os contextos se apresentam em um mesmo sistema, em algum momento a transição entre um modelo e outro deve ser realizada. Devido a natureza dos dois modelos, não existe uma solução ótima para este problema. Sempre haverá um compromisso a ser feito. Esta situação é chamada de *descasamento de impedância* entre os modelos relacional e orientado a

objetos. Ambos os modelos são utilizados para organizar dados, porém um é focado na manipulação eficiente dos dados, enquanto que o outro é focado em minimizar a distância entre a representação dos dados no modelo real e o código implementado no software.

Todo desenvolvedor que decide armazenar os dados de sua aplicação em um banco de dados relacional porém deseja manipulá-lo através do paradigma da orientação a objetos deverá em algum momento lidar com o descasamento de impedância. Muitos compromissos devem ser considerados de acordo com o domínio do problema a ser resolvido. Como dito anteriormente, não há ainda uma solução ótima, mas existem várias técnicas que podem ser empregadas para minimizar a problemática envolvida. Este conjunto de soluções é comumente chamado de *Mapeamento Objeto-Relacional* e é o tema principal deste projeto.

1.1 Objetivo

Este projeto consiste em uma biblioteca em C++ que agrega diversas soluções para a problemática do descasamento de impedância entre o modelo relacional e o orientado a objetos quando ambos são utilizados por um mesmo sistema.

A biblioteca, chamada CPPObjects, conceitualmente é uma ponte entre os dois modelos, e se apropria de modernas técnicas de desenvolvimento em C++ para tornar seu uso o mais intuitivo e simples possível, sem comprometer parâmetros de eficiência temporal e espacial [1], liberando o programador de se preocupar com tarefas mundanas normalmente presentes em tais tipos de aplicações e fazendo-o se concentrar no modelo que está sendo tratado.

Adotou-se o conceito de biblioteca de *software* à solução pelo fato de CPPObjects ser plenamente reutilizável por diversas aplicações que necessitem acessar um banco de dados relacional utilizando um paradigma de orientação a objetos. Desta forma o mapeamento objeto-relacional em si fica forçosamente modularizado, fazendo que os subsistemas responsáveis por ele fiquem elegantemente confinados em uma parte bem definida do sistema como um todo, com um grau de acoplamento mínimo com este. O acoplamento é restrito, idealmente, somente ao acesso aos métodos e declarações públicas da biblioteca.

Como conseqüência desta organização, problemas que surjam na biblioteca podem ser solucionados de forma mais eficaz por estarem restritos a uma parte do sistema. Caso seja utilizada linkedição dinâmica (através de DLL's ou similares), a aplicação que utiliza a CPPObjects não precisa ser recompilada para que o problema seja solucionado. Basta realizar uma atualização da biblioteca dinâmica contendo a correção do problema e todas as aplicações que dependem

[1] "Things should be made as simple as possible, but no simpler" - Albert Einstein

dela serão corrigidas.

CPPObjects consiste em duas partes: a parte relacional que se comunica com um RDBMS tal como o *PostgreSQL* e se encarrega da persistência e recuperação dos dados, e a parte orientada a objetos, que apresenta formas de manipular estes dados mais intuitiva e simples, as principais caratecterísticas de um sistema OO. O programador somente lida com a parte orientada a objetos e a CPPObjects se encarrega de fazer o mapeamento das ações realizadas pelo programador para as equivalentes na parte relacional. Desta forma pretende-se ter o melhor dos dois mundos na medida do possível, facilitando bastante o desenvolvimento de sistemas, dos mais simples aos mais complexos.

1.2 Organização

Este projeto está organizado em capítulos que devem ser lidos em sequência para seu melhor aproveitamento, já que cada um se baseia em conceitos abordados em capítulos anteriores.

O capítulo 1 oferece uma introdução da problema que a biblioteca pretende resolver, assim como apresenta a solução proposta pelo projeto.

O capítulo 2 apresenta uma análise do problema do descasamento de impedância entre o modelo relacional e o orientado a objetos, e apresenta algumas soluções para os problemas mais comuns.

O capítulo 3 apresenta a biblioteca CPPObjects de um ponto de vista do usuário. Um modelo simples será montado passo a passo explicando os conceitos associados e como implementá-lo na biblioteca.

O capítulo 4 discute algumas idéias futuras que podem ser implementadas usando a estrutura internal atual da CPPObjects de modo a torná-la mais poderosa.

O capítulo 5 conclui o projeto dando um parecer final das dificuldades apresentadas durante o desenvolvimento da biblioteca, as lições aprendidas neste período e como algumas decisões podem ser aplicadas em outros projetos deste porte.

2 *Análise*

Neste capítulo iremos apresentar o problema do descasamento de impedância entre o modelo relacional e o orientado a objetos quando os dois são utilizados em um mesmo sistema para representar os dados manipulados por este. Em seguida algumas soluções serão apresentadas, juntamente com suas vantagens e desvantagens

2.1 Descasamento de Impedância

O descasamento de impedância objeto-relacional representa um conjunto de dificuldades técnicas que são comumente encontrados quando em um mesmo sistema seus dados são manipulados de forma relacional em uma parte e de forma orientada a objetos em outra. Como normalmente os dados deve transicionar entre estes dois modelos, um mapeamento se faz necessário.

O substantivo *impedância* faz referência ao conhecido problema de descasamento de impedância em circuitos elétricos e foi utilizado para dar nome ao problema de mapeamento apresentado. Na realidade não há nenhum conceito de impedância propriamente dito em programação, mas a intenção de quem a criou deve ter sido de fazer uma analogia bem humorada.

2.2 Diferenças entre os modelos relacional e OO

Os diversos problemas que aparecem no mapeamento objeto-relacional advém das grande diferenças conceituais e de implementação entre eles. Estas diferenças podem ser divididas nos seguintes grupos:

2.2.1 Manipulação dos dados

As diferenças semânticas entre os dois modelos são especialmente significantes na manipulação que se faz dos dados. O modelo relacional faz uso de um pequeno conjunto de operadores

matematicamente bem definidos baseados no cálculo de predicados de primeira ordem.

Tanto as operações de consulta quanto de manipulação atuam em cima de um subconjunto dos dados armazenados. O banco de dados relacional provê uma linguagem declarativa onde o usuário define qual a propriedade (tecnicamente um predicado) que este subconjunto deve ter através de operadores relacionais e qual ação deve ser executada neste subconjunto.

Já o modelo orientado a objetos apresenta uma interface imperativa onde os dados são manipulados um a um, e as ações são aplicadas a cada um em separado. Pode-se pensar que o modelo OO seja um caso específico do modelo relacional, onde cada subconjunto contém apenas um elemento.

A problemática neste caso é que o modelo relacional é otimizado para trabalhar com vários dados de uma só vez, e quando o faz lidando com cada dado em separado há uma perda considerável de performance temporal. Já no modelo OO o inverso acontece, é bastante eficiente trabalhar com um dado de cada vez, ao passo que manipular conjuntos de dados e aplicar ações a todos de uma vez torna-se dispendioso.

2.2.2 Estrutura

Os modelos relacional e orientado a objetos diferem enormemente quanto à organização interna dos dados e como eles se apresentam ao usuário. Em modelos OO as estrutura de dados podem ser vistas como um grafo onde cada nó representa um determinado dado e cada aresta representa um determinado relacionamento entre dois dados, ou ainda em uma hierarquia de dados, também implicitamente estabelecendo um relacionamento entre os dados.

Estes fatores torna o mapeamento para um modelo relacional complicado, pois neste todos os dados são representados em um conjunto global e planar de relações (nome técnico para tabela). Estas são um conjunto de tuplas com os mesmos tipos de atributos (colunas). Esta organização não tem relação direta com o modelo OO, tornando a utilização de um mesmo dado nos dois modelos problemática.

2.2.3 Integridade

As restrições em modelos OO não são declaradas explicitamente como tais. Elas estão definidas de uma forma dispersa através de checagem de invariantes feitas no instante que algum dado interno do objeto é alterado. Usualmente quando uma condição invariante é violada uma exceção é lançada e deve existir toda uma lógica para o tratamento da situação que a causou.

O modelo relacional trabalha com restrições declarativas que atuam nos tipos, atributos, relações e o próprio banco de dados como um todo. Estas restrições levam em conta não somente um dado sendo alterado, como também a relação deste com os demais, como por exemplo quando não é permitido ter mais de uma pessoa com o mesmo CPF no banco de dados[1]. Este tipo de restrição é fácil de ser aplicada no RDBMS, porém é mais trabalhosa de ser definida e implementada em um modelo OO.

2.2.4 Encapsulamento

Programas orientados a objetos são projetados com métodos que resultam em objetos encapsulados cuja representação e detalhes de implementção permanecem escondidos do mundo externo. Mapear esta representação interna em tabelas em um banco de dados relacional faz com que estes se tornem de certo modo frágeis de acordo com os conceitos de OO, já que nestes as representações internas tem uma certa liberdade de serem modificadas ao longo do tempo, desde que a interface pública permaneça inalterada. Estas representações privadas são mapeadas em uma interface pública no RDBMS, que acabam tendo que ser alteradada quando a representação privada do dado muda. Isto acaba colidindo comum preceito básico de orientação a objetos, que é o encapsulamento

Em particular, a modelagem OO enfatiza o conceito de invariante, que requere o uso de encapsulamento antes que qualquer acesso a um dado de um objeto seja realizado. Estas invariantes não podem ser representadas em um banco de dados relacional. No jargão do modelo relacional, os conceitos de privado e público não são características absolutas do estado de um dado, e no modelo OO elas são.

Uma forma de implementar o conceito de encapsulamento em um RDBMS é criar visões que definem uma interface pública às tabelas referenciadas, estas contendo a "implementação" privada, ou seja, os dados propriamente ditos. Cabe ao administrador do banco de dados então configurá-lo corretamente para que as tabelas não sejam acessadas diretamente pelas aplicações, somente as visões.

O problema desta abordagem é que exige que as visões se comportem como tabelas propriamente ditas, possibilitando adições, atualizações e remoções de tuplas. No presente momento os poucos RDBMS que implementam visões só permitem que estas sejam consultadas. Somente RDBMS comerciais caros, como o Oracle, atendem os requisitos para que o conceito de encapsulamento seja implementado.

[1] Apesar de haver registros de um lote de CPF ter sido emitido com números duplicados, nos anos 50. Há então possibilitade de duas pessoas terem o mesmo número de CPF.

O acesso a objetos em programas orientados a objetos são realizados através de interfaces que juntas provêem a única forma de acesso à representação interna do objeto. O modelo relacional, por outro lado, utiliza variáveis de relação derivadas (as ditas visões) que provêem diferentes perspectivas e restrições a um mesmo conjunto de dados, permitindo acessá-los de forma diferente de acordo com a visão estabelecida.

Por fim, conceitos básicos de orientação a objetos tais como herança e polimorfismo não estão presentes no modelo relacional. Realizá-los em um RDBMS exige artifícios não ótimos que devem ser analisados caso a caso, dependendo das especificações de performance exigidas pelo projeto.

2.2.5 Transações

Os modelos relacionais diferem bastante com relação à forma que transações são tratadas. Em ambos os modelos a transação corresponde à menor unidade de trabalho realizada. Porém esta é muito mais abrangente no modelo relacional do que no OO. As transações no modelo relacional podem englobar diversas ações sobre os dados, quando que no modelo OO a granularidade é muito maior, cada transação tipicamente contém somente uma alteração a um atributo do objeto ou do objeto como um todo.

Sistemas orientados a objetos normalmente não tem um análogo das características ACID - atomicidade, consistência, isolamento e durabilidade - encontradas em sistemas relacionais. Estas só são aplicadas para as alterações em cada atributo, e só valem para estes.

2.3 Problemas a serem solucionados

Qualquer implementação de um mapeamento objeto-relacional deve lidar e solucionar diversos problemas que ocorrem durante seu desenvolvimento. Eles advém diretamente das diferenças entre os dois modelos explicitadas na seção anterior.

2.3.1 Mapeamento entre classes, objetos e tabelas

Um dos primeiros problemas que surgem durante a implementação de uma biblioteca de ORM é como mapear classes em tabelas relacionais. A princípio isto pode parecer simples, pois basta criar uma tabela por classe, e seus atributos viram colunas da tabela. O mapeamento dos tipos também não dá muito trabalho pois vários dos tipos comumente usados em modelos relacionais, tais como *VARCHAR*, *INTEGER*, *REAL* e *BOOLEAN* tem correspondentes diretos

em modelos orientados a objetos: strings, números inteiros, números em ponto flutuantes e valores booleanos, respectivamente.

Porém com o passar do tempo, é natural que características mais avançadas de OO sejam adicionadas em um sistema, e formas de mapeá-las para o modelo relacional devem ser escolhidas. Como dito anteriormente, vários dos conceitos OO não existem em modelos relacionais. Herança, por exemplo, é um conceito primordial de OO que não tem um análogo em um banco de dados relacional. O desenvolvedor então fica com três possíveis opções para implementar heranças em um RDBMS: uma tabela por classe, uma tabela por classe concreta ou uma tabela por hierarquia de classes. Cada uma destas tem vantagens e desvantagens.

Uma tabela por classe

A utilização de uma tabela por classe é a mais simples de ser entendida, já que procura minimizar a distância entre o modelo OO e o relacional. Cada classe na hierarquia de heranças é mapeada para uma tabela relacional, e objetos dos tipos derivados são agrupados através de operações de *JOIN* entre as várias tabelas envolvidas.

Por exemplo, digamos que uma classe base chamada Pessoa tenha como descendente a classe Aluno, e esta seja ascendente da classe Graduando. Cada uma destas classe são mapeadas em tabelas próprias. Estas conterão as colunas correspondentes aos atributos de cada classe excetuando-se os atributos das classes ascendentes.

Relacionar estas tabelas irá requerer que cada objeto tenha uma chave primária que permita recuperar a linha de cada tabela que corresponda a ele. Isto significa que para recuperar um objeto de um RDBMS um *join* de três tabelas é necessário para recuperar todo o estado do objeto.

Caso a hierarquia continue a crescer, por exemplo, incluindo Professor e Doutorando (herdando de Aluno) todos herdando direta ou indiretamente de Pessoa, uma simples consulta que deva retornar todas as pessoas cujo nome começa com “João”, por exemplo, deverá primeiramente pesquisar na tabela Pessoa, e depois fazer um *join* com as demais tabelas que mapeam classes concretas para recuperar o restante do estado do objeto. Como *joins* são operações custosas para serem feitas em RDBMS, esta solução deve ser analisada com muito cuidado para ver se as desvantagens (*joins* custosos) serão mais significativas do que suas vantagens (implementação simples).

Várias alternativas para este problema existem e podem ser agrupadas em duas categorias: uma tabela por classe concreta (a mais derivada) e uma tabela por família de classes. Ambas as

soluções incorrem em desnormalizações de dados e suas desvantagens resultantes.

Uma tabela por classe concreta

A opção por utilizar uma tabela por classe concreta faz com que, no exemplo acima, tenhamos tabelas para Graduandos, Doutorandos e Professores. Cada uma destas tabelas terá colunas correspondentes às classes ascendentes, ou seja, o atributo *nome* da entidade Pessoa será mapeado em uma coluna em cada tabela. Realizar consultas de pessoas dado um nome envolverá realizá-la três vezes, uma vez em cada tabela, claramente com um impacto negativo. Já realizar consultas de objetos de classes concretas é eficiente pois só uma tabela é acessada.

Uma tabela por hierarquia

A última solução envolve utilizar uma grande tabela para armazenar os dados de toda uma hierarquia de classes. Existe uma coluna a mais que indica a qual classe pertence uma determinada linha. As demais colunas são uma união de todas os atributos de todas as classes envolvidas. As colunas correspondentes a atributos que não pertencem a determinada classe devem ter seu valor nulificado. Esta solução resolve o problema de *joins* custosos e múltiplas consultas, porém o modelo relacional atinge o maior nível de desnormalização. Para grandes quantidades de dados esta desvantagem pode ser muito significativa para compensar as vantagens.

2.3.2 Identificação de objetos

Modelos orientados a objetos aplicam um sentido implícito à identificação de objetos. Um objeto é uma instância de uma classe, possivelmente localizada na memória. Ele é sua própria referência. Este objeto é referenciado dado sua localização na memória, e esta localização o diferencia de outros objetos. É possível ter mais de um objeto com o mesmo estado, mas como são duas instâncias, com posições de memória diferentes, são considerados objetos diferentes.

No modelo relacional a identificação de um objeto deve ser explícita. A única forma de referenciar objetos é designar um predicado que seja atendido pelo objeto, tal como: o objeto do tipo Pessoa cujo nome é “Pedro”. Nota-se que não é possível referenciar diretamente um objeto em um banco de dados relacional, mas sim um subconjunto destes que atenda ao predicado dado. Caso existam duas tuplas exatamente iguais no banco de dados, é impossível diferenciá-las, pois qualquer predicado que atenda a uma tupla atenderá a outra.

Para resolver este dilema criou-se o conceito de chaves primárias. Esta chave é encarada

como um identificador único para o objeto, ou seja, não pode haver mais de um objeto com o mesmo identificador. Desta forma o predicado que recupera somente um objeto fica sendo: me retorne o objeto cujo identificador é “132”, por exemplo. Este identificador, diferentemente da identificação no modelo relacional, é explícito e corresponde a uma coluna na tabela em questão.

A biblioteca de ORM deve mapear estas duas formas de identificação de objetos. A solução neste caso é explicitar a identificação do objeto no modelo OO. Esta não corresponde mais a uma instância de um objeto, mas sim às instâncias do objeto cujo identificador corresponda à chave primária no banco de dados.

2.3.3 Recuperação de objetos

Uma vez que os objetos estejam armazenados no banco de dados relacional, existe o problema de como recuperá-los eficientemente, fazendo desta forma o mapeamento inverso, do modelo relacional ao modelo OO.

Uma abordagem estritamente orientada a objetos faria uso de um construtor onde seria passado qual objeto deve ser recuperado. Esta solução infelizmente não é genérica o suficiente já que não proporciona a flexibilidade necessária para recuperar coleções de objetos, e normalmente consultas recuperam coleções, ao invés de um só objeto. As diversas consultas necessárias, uma para cada objeto da coleção a ser recuperada, são muito custosas em um RDBMS, que é otimizado para recuperar vários dados de uma só vez.

Para solucionar esta questão existem três alternativas: consulta através de API, através de exemplo e através de linguagem.

Consulta através exemplo

Neste tipo de consulta o usuário cria uma instância vazia de um objeto e preenche seus atributos com valores que os objetos retornados devem ter nos seus atributos correspondentes. A listagem 2.1 ilustra esta solução.

Listagem 2.1: Consulta através de exemplo

```
1 Professor p;
2 p.nome = "João da Silva";
3 std::list<Professor> professores = bdados.query(p);
```

O problema da solução por consulta através de exemplo é que ela só é comporta consultas simples. Consultas mais complexas tais como “retorne todos os professores que não se chamem João” não são possíveis. Embora seja possível alterar este tipo de consulta para se adequar a casos mais complexos, isto incorrerá em uma API mais complexa. Outro problema é que esta solução obriga que as classes dos objetos permitam que todos os seus atributos sejam nulificáveis, o que pode ser uma violação das regras do domínio em questão, já que no caso acima não faz sentido um professor não ter nome.

Consulta através de API

Esta solução envolve um conjunto de classes que juntas são utilizadas para especificar um critério para a recuperação dos dados. A listagem 2.2 exemplifica esta solução.

Listagem 2.2: Consulta através de API

```
1 Query q;
2 q.from<Professor>
3  .where(or_clause(equal_criteria(&Professor::nome, "João"),
4                   equal_criteria(&Professor::nome, "Roberto")));
5 std::list<Professor> professores = dbase.query(q);
```

Aqui a consulta não necessita mais de um objeto vazio, mas sim um conjunto de classes que ao serem utilizados juntas formam o predicado que será aplicado aos dados do banco, e os objetos que o satisfizerem serão retornados.

Repare que esta solução é bastante verbosa se comparada com a consulta correspodente expressa na linguagem *SQL*Structured Query Language que acaba sendo criada:

```
SELECT * FROM Professor WHERE nome = 'João'OR nome = 'Roberto'.
```

Consulta através de linguagem

A última solução é se utilizar de uma linguagem similar à SQL adaptada aos paradigmas da orientação a objetos. Os predicados criados podem fazer menção aos relacionamentos entre os objetos assim como hierarquia na qual eles se inserem. Esta solução é a mais difícil de ser implementada pois requer um processador de linguagem completo, incluindo checagem de sintaxe e semância, porém é o que oferece mais vantagens e flexibilidade no uso. A listagem 2.3 exemplifica esta forma de realizar consultas.

Listagem 2.3: Consulta através de linguagem

```
const char *query = "SELECT Professor p1, p1.alunos a"
                    "WHERE p1.nome LIKE 'João%'"
                    "    AND count(a) > 5"
                    "    AND a.disciplina.nome = 'Eletrônica'"
                    "LIMIT 10";
std::list<Professor> professores = dbase.query(query);
```

Esta consulta retorna no máximo 10 professores cujo nome começa por "João", que tenham mais de 5 alunos (exclusive) na disciplina "Eletrônica". Repare que estamos lidando com 3 tabelas diferentes (Disciplina, Professor e Aluno), porém não há a necessidade de especificar *joins* pois eles são criados pelo sistema de consulta a partir dos relacionamentos acessados por ela. A inclusão da cláusula *LIMIT* fornece um desafio a mais pois numa consulta SQL ela limita a quantidade de tuplas retornadas. Já no exemplo acima ela deve limitar o número de professores retornados. Como cada professor tem vários alunos e leciona várias disciplinas, o RDBMS retornará mais de uma tupla por professor, e a solução naïve de simplesmente agregar à consulta SQL gerada a cláusula *LIMIT 10* pode fazer com que menos de 10 professores sejam retornados mesmo que existam outros que atendam ao predicado.

2.3.4 Carregamento parcial de objetos

Uma consulta ao banco de dados usualmente incorre em diversos fatores que acabam por torná-la lenta. O fator mais importante ocorre quando o servidor de banco de dados encontra-se em um computador diferente da aplicação é a transferência de informações pela rede. Mesmo em servidores que rodam localmente existem atrasos devido ao acesso ao disco rígido que não podem ser desconsiderados.

Por este motivo, o banco de dados relacional permite especificar quais colunas de uma determinada tupla devem ser retornadas, de tal forma que somente as informações necessárias sejam transmitidas, minimizando assim o tempo gasto na transferência de dados.

Esta otimização não se adapta bem em modelos orientados a objetos. Cada objeto é de certa maneira a menor unidade indivisível de informação, não existe a opção de termos somente uma parte do objeto carregada. Isto torna-se problemático em situações onde só parte da informação é desejada. Se tivermos que mostrar em uma listagem o nome de todos os professores cadastrados, a consulta que retorna estes professores deve retornar todos os objetos do tipo Professor, incluindo todas as suas informações referenciadas tais como alunos, disciplinas, e assim

sucessivamente. Cada objeto deve estar completamente definido na memória.

Esta classe de problemas é chamada de problema do objeto parcial e a sua desconsideração em uma solução de ORM pode incorrer em severas penalidades em termos de desempenho e performance da aplicação.

Uma forma de mitigar este problema é implementar uma infraestrutura que carregue as informações do objeto na medida que sejam requisitadas. Em um primeiro momento o objeto retornado da consulta encontra-se vazio, ou seja, seus atributos não foram carregados. Mas à medida que eles são acessados o mapeador de dados realiza uma pequena consulta no banco de dados que retorna somente o atributo requerido. Esta solução é chamada de carregamento tardio e representa uma situação diametralmente oposta ao problema que está tentando solucionar. Dependendo da forma que o objeto for utilizado muitas consultas serão realizadas o que irá ocasionar em perdas de performance considerável, pois é preferível realizar uma consulta que retorne todas as informações necessárias de uma só vez do que realizar várias que retornam um dado por vez.

Como quase tudo em Engenharia, é necessário haver um compromisso entre ambas as soluções. Este paradoxo é chamado de paradoxo do tempo de carregamento e ilustra bem os diversos problemas que implementadores de soluções de ORM precisam solucionar.

Uma possível solução é o usuário indicar *a priori* no momento da consulta quais atributos ele precisará acessar para que estes sejam retornados de uma só vez. Isto requer uma implementação bastante complexa, além de requerer uma extensão na API de consulta onde ele pode especificar estes atributos, tal como numa cláusula *SELECT* em SQL. Um meio termo entre estas duas soluções é recuperar todos os atributos do objeto, mas não recuperar os objetos que se relacionam a ele. Estes só serão recuperados quando um deles for acessado, momento este que fará com que o mapeador de dados recupere todos os objetos que se relacionam com o objeto em questão.

3 A Biblioteca CPPObjects

Este capítulo é focado na utilização da CPPObjects baseado no ponto de vista do programador, sem levar em consideração detalhes de implementação. Será utilizado como exemplo modelo de um sistema de gerenciamento de disciplinas que será traduzido para C++ usando a CPPObjects. Uma solução de SGD modela entidades tais como disciplinas, suas turmas, alunos e outras entidades periféricas.

3.1 Fundamentos

A biblioteca CPPObjects baseia-se na noção de objetos e suas interrelações. Assim como no modelo orientado a objetos, o objeto manipulado através da CPPObjects é uma representação direta de uma entidade do domínio. Várias classes utilitárias estão disponíveis para auxiliar no mapeamento de termos e características de diagramas de classes e de entidade e relacionamento para código em C++. Internamente a biblioteca mapeia estes conceitos nos seus equivalentes relacionais utilizando várias soluções expostas no capítulo anterior, de uma forma transparente ao programador.

A principal conexão de cada objeto com o banco de dados é através de um *documento*. Ele é responsável por transferir objetos de e para o banco de dados, delegando algumas tarefas para outras partes da biblioteca caso seja necessário. Cada objeto é de alguma forma ligado à única instância da classe `orm::document` na aplicação.

Uma dos primeiros problemas a serem solucionados por uma biblioteca de ORM é de como fazer com que o programador informe à biblioteca a estrutura do modelo a ser mapeado para o banco de dados relacional. Como C++ não é uma linguagem reflexiva[1], o programador necessita passar as informações de quais classes serão persistidas no banco de dados, assim como quais são seus atributos e seus relacionamentos. Mais importante, é necessário que ele especifique o nome de cada atributo e de cada classe sob a forma de *string*. Estas em última instância

[1] Linguagens reflexivas são aquelas que permitem que o programa obtenha informações sobre a sua própria estrutura. A metaprogramação é um caso particular onde a reflexão é disponível somente em tempo de compilação.

corresponderão aos nomes de colunas e tabelas respectivamente no banco de dados.

Estas informações sobre o modelo são chamadas de metainformações e são a ferramenta base que dispõe a CPPObjects para realizar o correto mapeamento entre os dois modelos. Elas se assemelham aos metadados guardados por um DBMS por manter uma informação sobre o sistema sendo manipulado. As diferenças residem no fato de que os metadados guardam informações do modelo relacional manipulado, já as metainformações agregam a estes dados sobre o modelo orientado a objetos, como por exemplo os tipos de relacionamentos entre objetos, a cardinalidade destes, etc, informações inexistentes explicitamente em um RDBMS, portanto não passível de estarem armazenadas nos seus metadados.

Através das metainformações a biblioteca pode criar um esquema relacional no banco de dados, respeitando as cardinalidades dos relacionamentos entre entidades, propriedades de certas colunas tais como unicidade e nulicidade. Este processo é completamente automatizado e erros de definição do modelo são detectados em tempo de compilação devido à utilização de checagem estática de tipos, entre outras técnicas de metaprogramação.

O subsistema da CPPObjects que lida com consultas permite que o programador recupere objetos persistidos no banco de dados, possibilitando a utilização de filtros definidos utilizando operadores da própria linguagem C++. Isto é realizado através do uso extensivo de sobrecarga de operadores oferecida pela linguagem. Esta solução equivale à consulta por API (veja 2.3.3, p. 11), e foi escolhida por permitir que erros de definição do predicado sejam detectado já em tempo de compilação, evitando surpresas desagradáveis que sempre acabam acontecendo em tempo de execução. Alguns compromissos tiveram que ser levados em conta devido à complexidade de implementação. Características encontradas na solução baseada em consulta por linguagem (veja 2.3.3, p. 11) tais como definição implícita de *joins* não puderam ser implementadas no caso mais genérico por questões de tempo. Porém a solução apresentada é bastante útil e já foram desenvolvidas aplicações utilizando versões anteriores da CPPObjects onde ela mostrou-se bastante satisfatória.

Para garantir uma boa performance nos casos de uso mais comuns, um *cache* de objetos unidirecional foi implementado e é utilizado para minimizar consultas diretas ao banco de dados quando um objeto deve ser recuperado a partir do seu identificador. Esta situação é bastante comum quando a rede de objetos formada por um objeto e seus relacionamentos é navegada. Alguns conflitos de requisitos tiveram que ser considerados envolvendo a coerência do *cache* devido à sua natureza unidirecional. Estes problemas serão abordados nas seções subseqüentes.

Finalmente, uma grande parte da CPPObjects é composta por classes e funções utilitárias que ajudam a implementar o mapeamento objeto-relacional. Estas classes estão escondidas do

usuário comum e são encapsuladas de uma forma elegante fazendo uso de padrões conhecidos de desenvolvimento de software.

3.2 Lidando com um simples objeto

A maioria dos casos de uso administrados pela CPPObjects e outras bibliotecas de ORM se resume a algumas tarefas básicas a serem realizadas pelo programador: declaração de objetos, mapeamento de atributos, criação de objetos, sua recuperação, modificação e remoção. As diferenças entre as bibliotecas de ORM residem na qualidade da implementação destas tarefas e como o programador as utiliza para implementar o modelo.

As seguintes seções irão implementar uma entidade simples – *Aluno* – de forma que seja possível mostrar como a biblioteca é utilizada sem haver necessidade de entrar em detalhes para situações mais específicas.

O diagrama de classes 3.1 mostra a entidade Aluno que estamos definindo, assim como seus atributos.

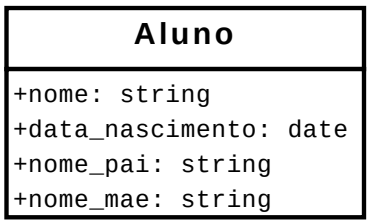

Figura 3.1: Diagrama de classes da entidade Aluno

3.2.1 Declaração

O primeiro procedimento a ser executado para implementar uma entidade é escrever sua declaração. A listagem 3.1 mostra como isto é realizado para a entidade *Aluno*.

Listagem 3.1: Declaração da entidade Aluno

```
1 struct Aluno : orm::object<Aluno>
2 {
3     static orm::metainfo<Aluno> meta;
4     // Atributos da entidade
5     std::string nome;
6     boost::gregorian::date data_nascimento;
7     boost::optional<std::string> nome_pai,
```

```
8                                   nome_mae;
9 };
```

O código acima declara a entidade `Aluno` que é composta por 4 atributos declarados explicitamente e um identificador declarado implicitamente chamado `id`. Este é declarado pela classe base `orm::object<Aluno>`. Ele corresponde à chave primária do objeto e tem papel fundamental para que outros objetos possam referenciar o objeto declarado, assim como este possa ser recuperado do banco de dados.

Como pode ser visto, todas as entidades devem herdar da classe `orm::object<T>`, que por sua vez herda da classe `orm::obj_t`. Desta forma todos os objetos podem ser manipulados através desta classe base polimorficamente, incluindo acessar seu identificador, que é declarado em `orm::obj_t`.

O tipo dos atributos `Aluno::nome_pai` e `Aluno::nome_mae` utilizam a biblioteca Boost para definir tipos nuláveis. A CPPObjects o utiliza para definir se a coluna correspondente no banco de dados deve permitir valores `NULL` ou não. Estes atributos são manipulados quase como se fossem ponteiros, ou seja, é necessário fazer uma dereferência para acessar seu conteúdo.

3.2.2 Especificação das metainformações

As classes de metainformação expõem informações referentes à entidade propriamente dita, ou ainda o tipo de cada objeto. Como a linguagem C++ não é reflexiva, meios artificiais são necessários para informar à CPPObjects qual é a diagramação das estruturas criadas, quais são seus atributos, etc. A CPPObjects utiliza-se de RTTI e polimorfismo dinâmico para chegar a resultados semelhantes.

Como as metainformações são definidas para cada estrutura, elas são acessadas através de um atributo estático desta. Por ser estático, este não está atrelado a qualquer instância da estrutura, e pode ser referenciado bastando utilizar o nome da estrutura, no caso acima, `Aluno::meta`.

A listagem 3.2 mostra a definição de `Aluno::meta` declarado na seção anterior, que deve estar em um arquivo de implementação (cpp).

Listagem 3.2: Definição da metainformação da entidade Aluno

```
1 orm::metainfo<Aluno> Aluno::meta =
2         orm::metainfo<Aluno>("aluno") // Usa a tabela 'aluno'
3             .ident("idaluno")
```

```
4             .member(&Aluno::nome,          "nome")
5             .member(&Aluno::data_nascimento, "data_nascimento")
6             .member(&Aluno::nome_pai,      "nome_pai")
7             .member(&Aluno::nome_mae,      "nome_mae");
```

O construtor de `orm::metainfo<Aluno>` recebe o nome da tabela que irá guardar os dados do aluno no banco de dados. Segue-se então a definição de cada atributo da entidade, informando qual é o membro correspondente na estrutura `Aluno` e qual é o nome da coluna no banco de dados. Isto é feito passando o ponteiro para o membro em questão e seu nome para o método `orm::metainfo<Aluno>::member`.

Foi escolhido usar a palavra *member* ao invés de *attribute* por ser entendido que a segunda é apropriada quando se está falando de diagramas ER, e está atrelada ao conceito de entidades (que possuem atributos). Já *member* é mais utilizada em modelagem OO, onde uma classe é formada de métodos (funções) e membros (correspondente aos atributos).

A coluna corresponde ao identificador do objeto (sua chave primária) é inicializada primeiro. Como seu membro correspondente – `Aluno::id` – é fixo, somente o nome da coluna correspondente deve ser especificado, o membro é associado implicitamente. Note que este passo é obrigatório e deve ser feito em primeiro lugar. Qualquer erro de definição será apontado em tempo de compilação.

Repare que em nenhum momento o tipo de cada atributo é passado. Isto é feito automaticamente através de recursos de sobrecarga de funções e especialização parcial de templates.

3.2.3 Instanciação

Uma vez declarados, objetos do tipo `Aluno` podem ser instanciados utilizando qualquer método de instanciação de objetos disponível em C++, seja instanciação na pilha ou no heap (utilizando o operador `new`). A listagem 3.3 mostra as diversas formas possíveis de se criar um objeto e preencher seus atributos.

Listagem 3.3: Instanciando objetos

```
1 // Cria a Alice na pilha
2 Aluno alice;
3 alice.nome = "Alice";
4 alice.data_nascimento = boost::gregorian::date(1980,9,26);
5 alice.nome_pai = "Vítor";
```

```
6  assert(!alice.nome_mae); // A sua mãe faleceu
7  assert(alice.id == 0); // Não está no banco

9  // Cria Roberto na memória heap gerenciada por um smart pointer.
10 std::shared_ptr<Aluno> roberto(new Aluno);
11 roberto->nome = "Roberto";
12 roberto->data_nascimento = boost::gregorian::date(1978,1,27);
13 roberto->nome_pai = "João";
14 roberto->nome_mae = "Maria";

16 // Cria vários alunos irmãos
17 std::vector<Aluno> alunos;
18 const char *nomes[] = {"Juliana","Margot","Pedro","Felipe"};

20 for(unsigned i=0; i<sizeof(nomes)/sizeof(nomes[0]); ++i)
21 {
22     Aluno st;
23     st.nome = nomes[i];
24     st.nome_pai = "Joaquim";
25     st.nome_mae = "Joaquina";
26     alunos.push_back(std::move(st));
27 }

29 // Mostra o nome de cada irmão
30 for(unsigned i=0; i<alunos.size(); ++i)
31     std::cout << alunos[i].nome << std::endl;
```

A linha 2 mostra um objeto `Aluno` instanciado na pilha. Na CPPObjects os objetos se comportam como estruturas em C, ou seja, cada objeto é uma mera coleção de atributos de forma que eles podem ser armazenados em diversos contêineres STL, arrays em C, serem movidos à vontade, criados na pilha ou na memória heap, etc. É por isso que a manipulação de objetos postrada na listagem 3.3 é tão familiar a programadores de C/C++.

As asserções nas linhas 6 e 7 são pós-condições garantidas pela biblioteca. A primeira asserção mostra que a mãe da Alice não foi atribuida. Esta condição será mapeada em um valor `NULL` na coluna correspondente à tupla da Alice no banco de dados. A segunda asserção mostra

que não há nenhum identificador associado à Alice. Identificadores somente são associados no momento que um objeto é persistido no banco de dados. Como isto não aconteceu ainda, o seu valor é 0.

De uma forma semelhante, as linhas 10 até 14 mostram um objeto sendo instanciando na memória heap e sendo gerenciado por um smart pointer.

A partir da linha 17 vários alunos são criados e inseridos em um `std::vector` tal como seria uma estrutura em C comum. Note que na linha 26 o aluno é passado para a função `std::vector<Aluno>::push_back` como uma referência a rvalue[2], não incorrendo em uma cópia de objetos custosa. O *move constructor* do aluno é chamado para transferir o conteúdo do objeto ao vetor, deixando a cópia original vazia.

3.2.4 Armazenagem, recuperação, modificação e remoção

Uma vez instanciados, os objetos podem ser armazenados no banco de dados, recuperados, modificados e removidos através da biblioteca. O programador realiza estas operações através da instância da classe `orm::document`, cujo construtor aceita uma string identificando as propriedades da conexão ao banco de dados. A listagem 3.4 mostra como estes diversos casos de uso são empregados na CPPObjects.

Listagem 3.4: Persistência de objetos no banco de dados

```
1  orm::document doc("psql://rodolfo:senha@localhost:5432/testdb",
2              Aluno::meta);
3  doc.create_schema(); // Cria o esquema do banco de dados

5  // Cria um aluno chamado João
6  Aluno joao;
7  joao.nome = "João";
8  // Como ainda não está no banco de dados, seu identificador é 0
9  assert(joao.id == 0);

11 // Adiciona o João no banco de dados
12 doc.add(joao);
13 // Como o João já foi persistido no banco, seu identificador é diferente
```

[2]Referencias à rvalue são uma novidade da próxima versão da linguagem C++, chamada temporariamente de C++0x. Elas representam valores que podem ser considerados temporários, passíveis de serem movidos ao invés de copiados.

```
14 // de 0.
15 assert(joao.id != 0);

17 // Recupera o João do banco de dados dado o seu identificador
18 std::shared_ptr<const Aluno>
19     objdb = doc.query<Aluno>(joao.id).lock();
20 assert(objdb && objdb->nome == "João"); // Realmente é o João

22 // Define um pai para o João
23 joao.nome_pai = "Luiz";
24 doc.edit(joao); // Atualiza o banco de dados/

26 // O objeto do cache também foi atualizado
27 assert(objdb->nome_pai_nome == "Luiz");

29 // Remove o João do banco de dados
30 doc.rem(joao);
31 assert(joao.id == 0); // Ele realmente não está mais no banco
```

Os detalhes de como a conexão ao banco de dados é feita, o sistema de consultas e *cache* mostrados na listagem serão abordados nas seções posteriores.

Isto conclui a exposição dos principais casos de uso abordados pela biblioteca. Nas seções subseqüentes cada subsistema da biblioteca será descrito na medida que o modelo utilizado até agora é acrescido de novas entidades e relacionamentos.

3.3 Identificador de Objeto

A classe `orm::obj_t` declara um membro chamado `id` cujo tipo é `orm::ident` que identifica unicamente o objeto, ou usando um jargão de banco de dados, se comporta como uma chave primária do tipo *surrogate*. Este tipo de chave primária é utilizada por não ter nenhuma correlação com os atributos do objeto. Desta forma estes podem ser alterados livremente sem a necessidade de atualizar outros objetos que referenciam o objeto alterado, já que estes comumente utilizam a chave primária do objeto em questão para referenciá-lo.

Quando um objeto é instanciado pelo usuário, seu identificador é inicializado em 0. Isto

serve para indicar que o objeto não está ainda persistido no banco de dados. Uma vez salvos no banco, o identificador ganha um número diferente de 0 único entre todos os identificadores dos objetos do mesmo tipo, indicando que de fato o objeto se encontra persistido. Este valor não deve ser mudando durante a vida do objeto e esta condição é garantida em tempo de compilação, já que o membro `orm::obj_t::id` é um atributo constante (`const`). Não é recomendável forçar uma alteração deste valor através do uso de um `const_cast` sob a pena de resultar em comportamentos não esperados da biblioteca.

Uma das desvantagens de se utilizar um identificador predefinido da forma que CPPObjects usa é que não é possível especificar chaves não-surrogate, ou ainda chaves primárias compostas. Estes problemas serão abordados em uma futura versão da biblioteca. Mesmo assim a solução empregada satisfaz os modelos de dados mais comuns.

3.4 Lista de Objetos

Devido às necessidades especiais da CPPObjects, um novo contêiner foi implementado que se comporta como um híbrido de um `std::unordered_set<std::shared_ptr<T>>`[3] e um `std::unordered_map<orm::ident, std::shared_ptr<T>>`. Objetos podem ser inseridos nele como se a lista fosse um `std::unordered_set<T>`, e ser recuperado dado seu identificador como se a lista fosse um `std::unordered_map<orm::ident, T>`.

Os objetos são gerenciados por *smart pointers*, porém o usuário tem controle da forma que é realizado este gerenciamento. Eles podem ser gerenciados por um `std::weak_ptr<T>` ou um `std::shared_ptr<T>`. Se ele escolher o primeiro, a lista irá rastrear o objeto. Se este for deletado externamente, a lista irá remover a referência ao objeto dela atualizando seu estado interno (número de objetos armazenados etc.). Estes recursos são bastante usados na implementação do relacionamento entre objetos para evitar vazamento de memória devido a referências cíclicas de *smart pointers*.

A `orm::list<T>` garante que no máximo um objeto de um dado identificador diferente de 0 seja mantido nela, com exceção de objetos que não foram inseridos no banco de dados ainda (seu identificador é 0). Isto permite que o usuário insira vários objetos vazios na lista e os popule depois com seus dados, possivelmente antes de adicioná-los no banco de dados, porém garante que duas instâncias do mesmo objeto persistido no banco não estejam na mesma lista. Quando o programador tenta inserir um objeto que viola estas invariantes, é dito que houve um

[3] O contêiner `std::unordered_set<T>` é novo no C++0x e representa um container onde os dados são buscados através do seu hash, com acesso médio em O(1), pior caso em O(n)

conflito e a ação não ocorrerá.

A lista implementada fornece duas interfaces para o usuário, dependendo se o objeto armazenado é `const` ou não, ou seja, se estamos lidando com um `orm::list<const T>` ou um `orm::list<T>`. Isto significa que se a lista estiver gerenciando objetos constantes ela fará de tudo para garantir que um objeto inserido nela não possa ser modificado em hipótese alguma. Esta garantia é importante pois os objetos gerenciados pelo *cache* não podem ser alterados externamente, somente através de operações no documento. A CPPObjects se aproveita de o fato da linguagem C++ ser fortemente tipada e delega o controle desta invariante à ela. Qualquer classe que lida com objetos constantes deve garantir esta invariante, sob o risco de haver algum efeito colateral não previsível ao longo da utilização de um sistema desenvolvido com a CPPObjects.

A figura 3.2 mostra o diagrama de classes da `orm::list<T>` e seus diversos métodos públicos.

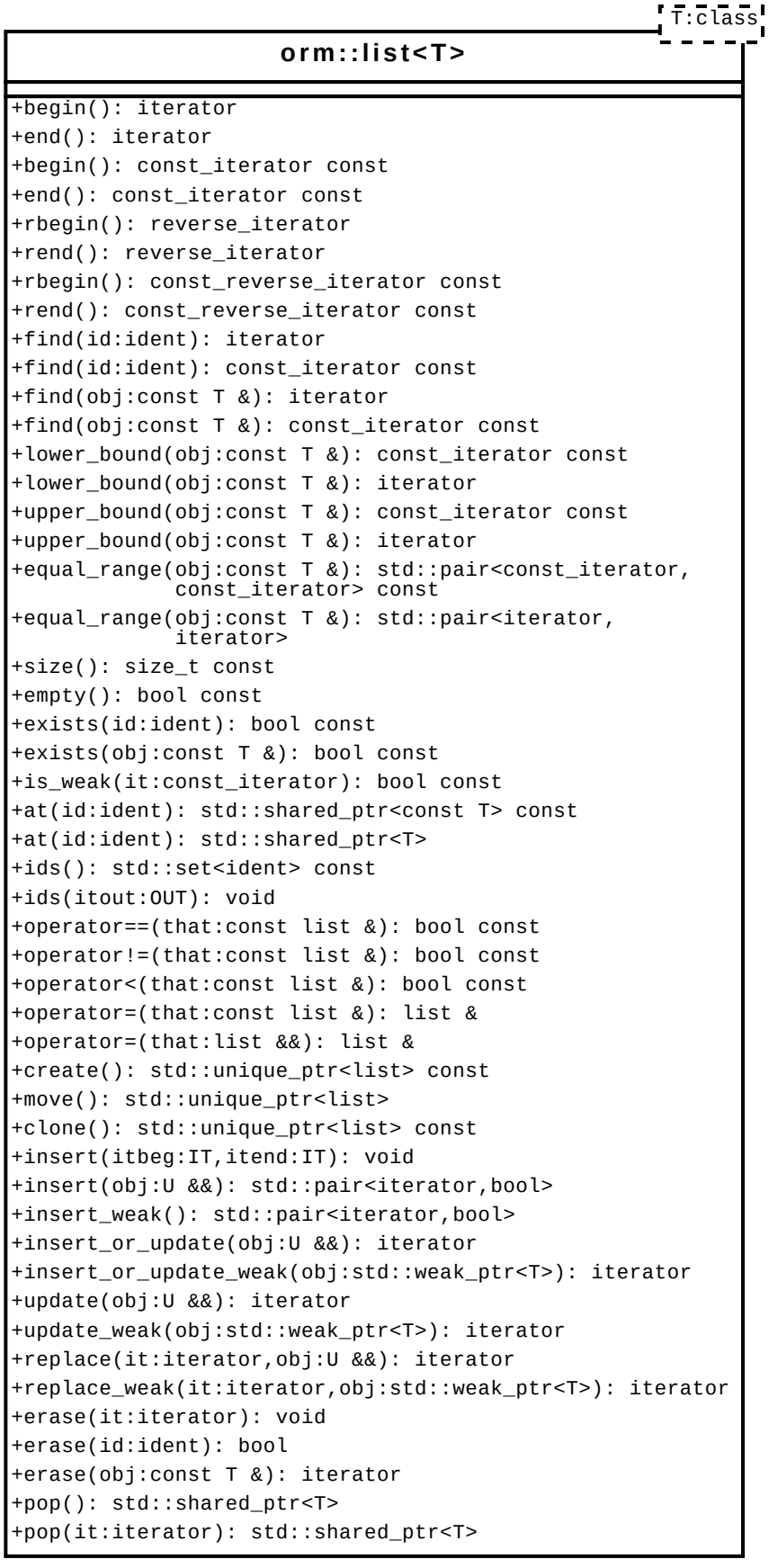

Figura 3.2: Diagrama de classes de `orm::list<T>`

A quantidade considerável de métodos é necessária pois a `orm::list<T>` deve garantir a manutenção da imutabilidade de objetos constantes ao longo de diversos casos de uso. Para facilitar o entendimento, estes métodos estão agrupados em 5 categorias, como mostrado na tabela 3.1.

Propósito	*Membros*
iteração	`begin, end, rbegin, rend`
estado	`size, empty, exists`
recuperação	`at, find, lower_bound, upper_bound, equal_range`
inserção	`insert, insert_weak, insert_or_update, insert_or_update_weak`
modificação	`update, update_weak, replace, replace_weak`
remoção	`erase, pop`

Tabela 3.1: Agrupamento dos métodos da classe `orm::list<T>` baseado nas suas funções

Muitos dos métodos se comportam exatamente como os seus semelhantes em uma `std::unordered_set<T>`. É o caso de `begin/end` e `rbegin/rend`, eles retornam iteradores diretos e reversos que permitem que cada objeto armazenado seja visitado. Note que a ordem dos objetos não é especificada, porém é a mesma se compararmos com a ordem dos mesmos objetos armazenados em outra lista.

Objetos podem ser recuperados e consultados de listas utilizando-se os métodos `list<T>::at` e `list<T>::find` respectivamente. Ambos são sobrecarregados em versões que aceitam um identificador ou um objeto. Quando o identificador é passado a uma destas funções, o objeto associado a ele é retornado caso exista. Nesta situação o identificador passado deve ser diferente de 0 já que podem existir mais de um objeto nesta situação. Uma exceção é lançada se esta precondição não for obedecida. Em casos onde é necessário recuperar objetos cujo identificador é 0, um outro objeto que seja igual ao objeto armazenado (os atributos são iguais, etc.) deve ser passado. O método `list<T>::at` retorna um `std::shared_ptr<T>` que aponta para o objeto encontrado, ou retorna um *smart pointer* nulo se o objeto não estiver na lista. Da mesma forma, o método `list<T>::find` retorna um iterador para o objeto achado, ou `list<T>::end()` caso ele não tenha sido encontrado.

Para inserções e modificações de objetos existem duas classes de métodos: uma que aceita um objeto (ambos referenciados via rvalue ou lvalue) ou um *smart pointer* que aponta para um objeto. A outra classe cujo nome das funções são sufixados por `_weak` aceitam um objeto que é apontado por um `std::weak_ptr<T>`. Desta forma o programador pode explicitamente informar à lista para armazenar o objeto em um smart pointer *shared* ou *weak*, dependendo na necessidade.

Os métodos `insert` e `insert_weak` simplesmente inserem um objeto na lista caso seu

identificador seja diferente de 0 e não haja nenhum outro objeto inserido com o mesmo identificador, ou se o identificador do objeto inserido seja 0, quando ele é sempre inserido. O valor retornado é um `std::pair` cujo primeiro membro é um iterador para o objeto inserido e o segundo membro é um valor booleano, valendo `true` em caso de sucesso na inserção. Se o objeto não for inserido por violar as invariantes da lista, o primeiro membro apontará para o objeto que previne a inserção, e o valor booleano retornado é `false`.

Os métodos `replace` e `replace_weak` permitem que o programador substitua um objeto por outro na lista dado um iterador que aponte para o primeiro. A função irá retornar um iterador para o objeto substituido em caso de sucesso. Se a substituição violar as invariantes da lista, a ação não ocorrerá e `orm::list<T>::end` será retornado. Esta situação ocorre quando é tentado substituir um objeto por outro que já se encontra na lista.

Um objeto pode ser atualizado chamando os métodos `update` e `update_weak`. A lista irá procurar por um objeto cujo identificador seja igual ao identificador do objeto a ser atualizado. Caso o objeto não seja encontrado ou seu identificador seja 0, `orm::list<T>::end` será retornado. Caso seja achado, `update` usa o método `replace` para realizar a atualização através da substituição do antigo objeto pelo novo.

Se o programador quiser colocar um objeto na lista não importando se ele já se encontra nela ou não ele pode usar o método `insert_or_update`. Este irá inserir o objeto na lista caso ele não exista. Caso contrário ele irá atualizar o objeto existente com o novo passado. Nestas situações nunca ocorrerão conflitos, e a função retornará um iterador apontado para o objeto inserido ou atualizado.

Quando é chegada a hora de remover objetos da lista, o programador pode fazer uso dos métodos `erase` ou `pop`. O primeiro é utilizado com o objeto removido não é mais necessário. Ele tem 3 sobrecargas que aceitam um iterador apontando para o objeto a ser removido, um identificador e um objeto. Quando um identificador for passado a função irá tentar remover o objeto cujo identificador é igual ao passado, retornando verdadeiro caso este seja encontrado ou falso caso contrário. Mais uma vez este identificador deve ser diferente de 0. Para remover objetos cujo identificador seja 0 um objeto que compare de forma igual ao objeto a ser removido deve ser passado, de forma semelhante ao que ocorre com o método `list<T>::find`.

Caso o programador queira utilizar ainda o objeto a ser removido, ele pode fazer uso do método `pop` para retirá-lo objeto da lista. Uma sobrecarga deste método aceita um identificador, e a outra não tem parâmetros, e simplementes retira da lista e retorna o objeto apontado por `list<T>::begin`.

3.5 Documento

O documento, representado pela classe `orm::document`, atua como uma ponte ligando os objetos do modelo orientado a objetos e seus correspondentes no modelo relacional. Ele é utilizado para armazenar e recuperar objetos de dispositivos de armanazenagem, gerenciado o trabalho necessário para realizar o mapeamento objeto-relacional. O documento faz uso das informações disponibilizadas pelo subsistema de metainformações criadas pelo programador para realizar estas tarefas

A figura 3.3 mostra o diagrama de classe de `orm::document` com seus métodos públicos. Estes podem ser agrupados em 3 conjuntos distintos de acordo com o subsistema sobre os quais trabalham.

- `query,add,edit,rem`: Membros que lidam com a transferência de objetos de/para o banco de dados.
- `create_schema, drop_scema`: Membros que lidam com a criação e destruição do esquema do banco de dados.
- `get_from_cache, clear_cache`: Membros que lidam com o *cache* de objetos.

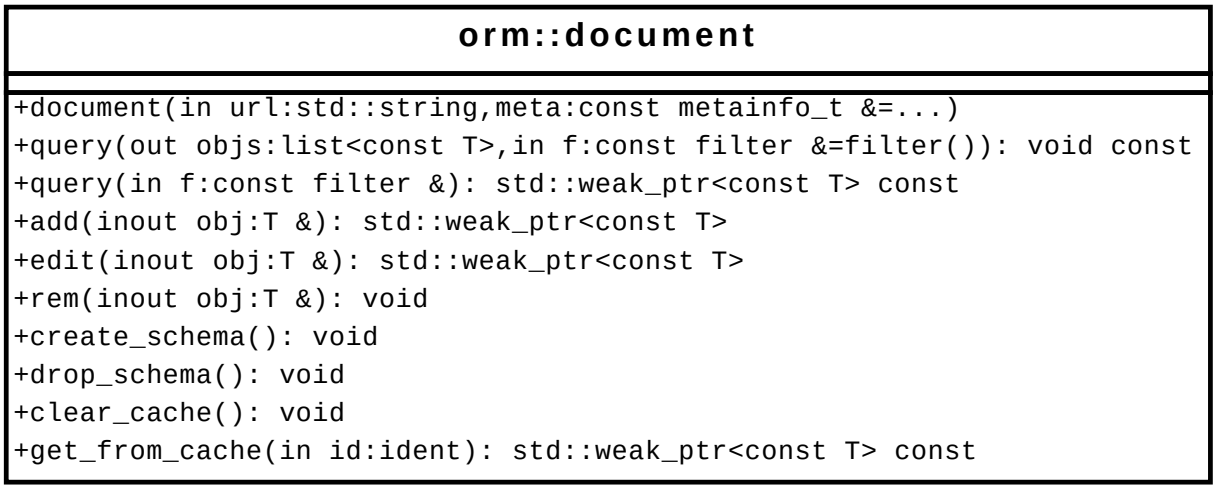

Figura 3.3: Diagrama de classes de `orm::document`

O construtor aceita ao menos 2 parâmetros. O primeiro é uma string com a URL utilizada para se conectar e autenticar com o sistema de banco de dados relacional. A tabela 3.2 mostra todos os parâmetros que podem ser passados à URL de conexão, assim como os valores padrão que são usados quando algum parâmetro não é passado.

Sintaxe
`dbms://[usuário[:senha]@]host[:porta]/banco`

Parâmetro	*Padrão*	*Significado*
dbms	–	O sistema de banco de dados utilizado. No momento somente o PostgreSQL (psql) foi implementado.
usuário	(nenhum)	O nome do usuário que tem acesso ao banco
senha	(nenhum)	A senha do usuário
host	localhost	O endereço do computador onde reside o servidor de banco de dados (IP ou nome da máquina resolvível pelo DNS)
porta	(varia)	A porta que o DBMS está escutando. Seu valor padrão depende do DBMS usado. Para o PostgreSQL o padrão é 5432.
banco	–	O banco de dados a ser conectado

Tabela 3.2: URL de conexão ao banco de dados

Os parâmetros restantes são as metainformações de entidades, utilizados pelo documento para recuperar informações que permitirão que ele crie o esquema do banco de dados. Para cada metainformação passada, o construtor irá seguir os relacionamentos de cada entidade de forma recursiva até que todas as entidades relacionadas direta ou indiretamente com a entidade cuja metainformação foi passada sejam adicionadas ao esquema. Portanto se o modelo tem dois conjuntos de entidades não relacionadas, a metainformação de uma entidade de cada conjunto deve ser passada para que todo o modelo seja gerado e suas tabelas correspondentes sejam criadas.

Os métodos `orm::document::create_schema` e `orm::document::drop_schema` são utilizados para criar e remover as tabelas do banco de dados, assim como as restrições de chave estrangeira de acordo com as metainformações recuperadas no construtor, como explicado no parágrafo anterior. A função `orm::document::drop_schema` deve ser utilizada com muito cuidado já que todo o conteúdo do banco de dados será removido.

O método `query` e suas sobrecargas são utilizados para carregar objetos do banco de dados. A sobrecarga que aceita um identificador é utilizada para carregar no máximo um objeto que tenha o identificador passado. A sobrecarga que aceita um `orm::list<T>` como primeiro parâmetro é utilizada quando mais de um objeto precisa ser retornado. Como segundo parâmetro deve ser passado um filtro que irá restringir os objetos retornados. Caso este filtro seja omitido, todos os objetos do tipo da lista serão retornados na lista.

É importante notar que todos os objetos retornados pelo documento são gerenciados por *smart pointers* que são controlados pelo *cache* de objetos do documento. Isto significa que outras ações realizadas no objeto através do documento irão se propagar aos objetos retornados, já que os *smart pointers* apontam para a mesma instância do objeto. Desta forma o sistema

garante a consistência do objeto por toda a aplicação, desde que ela previlegie manipular objetos que estejam sendo gerenciados pelo *cache*.

O método `orm::document::add` adiciona um novo objeto no banco de dados. O objeto deve ter seu identificador igual a 0, significando que ele ainda não se encontra no banco de dados. Se o objeto for corretamente adicionado, a função retornará um *smart pointer* para o objeto adicionado, gerenciado pelo *cache*. Ambos os objetos retornados e o objeto passado tem seu identificador setado para um valor designado pelo DBMS, este garante que ele seja único entre os objetos do tipo do objeto inserido.

O método `orm::document::edit` recebe um objeto já armazenado no banco – seu identificador deve ser diferente de 0 – e persiste as modificações feitas nele ou em seus objetos relacionados (caso houver). Assim como na adição, esta função retorna um *smart ponter* para o objeto já modificado, gerenciado pelo *cache*.

Finalmente o método `orm::document::rem` remove o objeto passado do banco. Seu identificador será setado para 0 e o objeto correspondente do *cache* será removido deste. Todos os `std::weak_ptr<const T>` que apontam para este objeto serão resetados.

As últimas três funções irão atualizar os objetos que referenciam o objeto modificado até que toda a rede de objetos compostas por objetos que se interrelacionam seja espelhada no banco de dados. Estas operações são envoltas por uma transação, de tal forma que se algum erro surgir, uma exceção será jogada e o estado do banco e dos objetos será mantido.

Os métodos `get_from_cache` e `clear_cache`, que lidam exclusivamente com o *cache* de objetos, possibilitam um maior controle dos dados do *cache* conforme necessário. A função `get_from_cache` recupera um dado do *cache* dado seu identificador e seu tipo (passado como um parâmetro template). Se o objeto não estiver no *cache*, um *smart pointer* nulo será retornado. O método `clear_cache` por sua vez simplesmente esvazia o *cache*. Se um objeto for referenciado em alguma parte do programa através de um relacionamento, este será recuperado diretamente do banco de dados. Esta é uma forma de garantir que o estado dos objetos utilizados estejam sincronizados com o estado das informações no banco. Um exemplo de utilização do documento foi dado na listagem 3.4, apresentada anteriormente.

3.6 Consulta a objetos

Como explicado na seção 3.5, o documento é a ponte entre a aplicação e o banco de dados relacional, portanto deve ser utilizado para realizar consultas a objetos armazenados nele. Na

maioria das situações o programador deve recuperar somente um subconjunto dos objetos de um determinado tipo. Filtros são então utilizados para impor uma restrição baseada em propriedades que os objetos recuperados devem ter, ou seja, um predicado que eles devem atender.

A especificação de filtros na CPPObjects faz o uso da linguagem C++ através da utilização de sobrecarga de operadores e funções especificas, tornando-a natural para o programador sem que este precise aprender uma nova linguagem específica para representar os predicados, como acontece em diversas bibliotecas de ORM. Esta abordagem é inovadora exatamente por causa destes motivos, além de contar com a vantagem de que a maioria dos erros na definição do predicado são apontados em tempo de compilação, evitando surpresas desagradáveis quando a aplicação está em produção.

Os filtros usualmente agem nos atributos das entidades, especificados através do operador ponteiro-para-membro do C++, como por exemplo `&Disciplina::nome` ou `&Avaliacao::data`. Estes atributos podem ser comparados uns com os outros ou com valores definidos estaticamente, utilizando-se operadores booleanos para agrupar termos, possibilitando a criação de filtros complexos.

A listagem 3.5 mostra vários casos de uso de utilização de filtros. Alguns dos exemplos irão fazer menção a itens explicados logo em seguida à listagem.

Listagem 3.5: Filtragem de objetos

```
1  // Retorna todos os alunos nascidos após 27/jan/1978. Note a
2  // utilização de orm::member para especificar o atributo data_nascimento
3  Alunos alunos;
4  doc.query(alunos,
5      orm::member(&Aluno::data_nascimento)
6              >= boost::gregorian::date(1978,1,27));

8  // Retorna todas as disciplinas desabilitadas cujo nome
9  // começa com 'Calc'
10 Disciplinas disciplinas;
11 doc.query(disciplinas, !orm::member(&Disciplina::habilitada)
12                       && &Disciplina::nome == orm::like("Calc%"));

14 // Retorna todos os alunos nascidos após 5/out/2001 cujo nome não
15 // começa com 'Rod'
16 doc.query(alunos,
```

```
17      orm::member(&Aluno::bird_date) >= boost::gregorian::date(1978,1,27))
18      && &Aluno::nome != orm::like("Rod%"));

20  // Retorna todas as notas cujo valor, após ser multiplicado por 2 e
21  // adicionado a 5, seja maior do que 17
22  Notas grades;
23  doc.query(grades, orm::member(&Nota::valor)*2+5 > 17);

25  // Retorna todos os alunos nascidos entre 27/jan/1978 e 1/jan/1980
26  doc.query(alunos, &Aluno::data_nascimento
27              == orm::between(boost::gregorian::date(1978,1,27),
28                              boost::gregorian::date(1980,1,1)));

30  // Retorna todos os alunos cujo identificador é 5,8,10,17 ou 26. Não
31  // há necessidade de especificar o &Aluno::id usando orm::member pelo
32  // fato de o lado direito da expressão ser uma função CPPObjects .
33  doc.query(alunos, &Aluno::id == orm::in(5,8,10,17,26));

35  // Retorna todas as disciplinas cujo nome tem no máximo 7 caracteres
36  doc.query(disciplinas, orm::strlen(&Disciplina::nome) <= 7);

38  // Retorna todas as turmas cuja disciplina tenha um nome começado por
39  // 'Calc'. Note que esta consulta envolve um join de duas tabelas.
40  Turmas turmas;
41  doc.query(turmas, &Disciplina::nome == like("Calc%"));
```

As tabelas 3.3, 3.4, 3.5, 3.6 e 3.7, mostra os elementos que podem ser utilizados para compor uma especificação de filtro. Eles podem ser agrupados em 5 categorias:

- Operadores lógicos
- Operadores de comparação
- Funções
- Operadores aritméticos
- Operadores que manipulam bits.

A precedência destes segue a precedência usual dos operadores em C++ que podem, obviamente, ser aumentada através do uso de parênteses ao redor da expressão.

Operador	*Significado*
`expr1 && expr2`	E lógico
`expr1 || expr2`	OU lógico
`!expr`	NÃO lógico

Tabela 3.3: Operadores lógicos utilizados em filtros

Operadores	*Significado*
`expr1 + expr2`	adição
`expr1 - expr2`	subtração
`expr1 * expr2`	multiplicação
`expr1 / expr2`	subtração
`expr1 % expr2`	módulo (resto da divisão)
`-expr`	negação

Tabela 3.4: Operadores aritméticos utilizados em filtros

Operador	*Significado*
`expr1 & expr2`	E bit a bit
`expr1 | expr2`	OU bit a bit
`expr1 ^ expr2`	OU exclusivo bit a bit
`expr1 ~ expr2`	NÃO bit a bit
`expr1 << expr2`	deslocamento à esquerda bit a bit
`expr1 >> expr2`	descolamento à direita bit a bit

Tabela 3.5: Operadores bit a bit utilizados em filtros

Operador	*Significado*
`expr1 == expr2`	igual a
`expr1 != expr2`	diferente de
`expr1 > expr2`	maior que
`expr1 >= expr2`	maior ou igual a
`expr1 < expr2`	menor que
`expr1 <= expr2`	menor ou igual a
`expr1 == between(expr2,expr3)`	retorna verdadeiro se o valor de "expr1" está entre os valores "expr2" e "expr3", inclusive.
`expr1 != between(expr2,expr3)`	retorna verdadeiro se o valor de "expr2" não está entre os valore de "expr2" e "expr3", inclusive.
`expr1 == in(expr2,expr3,...)`	retorna verdadeiro se "expr1" se encontra na lista de expressões dada
`expr1 != in(expr2,expr3,...)`	retorna verdadeiro se "expr1" não se encontra na lista de expressões dada
`string == [i]like(pattern`[1]`)`	retorna verdadeiro se "string" casa com o padrão "pattern", diferenciando ou não letras maiúsculas de minúsculas.
`string != [i]like(pattern)`	retorna verdadeiro se "string" não casa com "pattern" diferenciando ou não letras maiúsculas de minúsculas.
`string == [i]regexp(pattern`[2]`)`	retorna verdadeiro se "string" casa com a expressão regular "pattern" diferenciando ou não letras maiúsculas de minúsculas.
`string != [i]regexp(pattern)`	retorna verdadeiro se "string" não casa com a expressão regular "pattern" diferenciando ou não letras maiúsculas de minúsculas.

Tabela 3.6: Operadores lógicos utilizados em filtros

[1] O padrão é o mesmo utilizado em comparações *LIKE*, tal como é especificado pelo padrão SQL: '%' casa com qualquer substring, '_' casa com qualquer caractere.

[2] O padrão de expressão regular segue o especificado pelas regras POSIX

Operador	*Significado*
`sin(expr)`	seno de “expr” (dado em radianos)
`cos(expr)`	cosseno of “expr” (dado em radianos)
`tan(expr)`	tangente of “expr” (dado em radianos)
`asin(expr)`	arco-seno of “expr” em radianos
`acos(expr)`	arco-cosseno of “expr” em radianos
`atan2(expr1,expr2)`	retorna o valor principal da arco-tangente de “expr1/expr2” em radianos utilizando os sinais dos dois argumentos para determinar o quadrante do resultado
`pow(expr1,expr2)`	retorna $(expr1)^{expr2}$
`abs(expr)`	retorna o valor absoluto de “expr”
`ceil(expr)`	arredonda “expr” para cima
`floor(expr)`	arredonda “expr” para baixo
`round(expr)`	arredonda “expr” para o inteiro mais próximo
`trunc(expr)`	trunca a parte fracional de “expr”
`sign(expr)`	retorna o sinal de “expr” (-1, 0 ou 1)
`degrees(expr)`	converte “expr” de radianos para graus
`radians(expr)`	converte “expr” de graus para radianos
`exp(expr)`	retorna $e^{(expr)}$
`log(expr)`	retorna $log_e(expr)$
`log10(expr)`	retorna $log_{10}(expr)$
`sqrt(expr)`	retorna $\sqrt{(expr)}$
`strlen(string)`	retorna o número de caracteres de “string”
`strcat(string1,string2)`	retorna “string1” e “string2” concatenados
`strlwr(string)`	retorna “string” convertida para caixa baixa
`strupr(string)`	retorna “string” convertida para caixa alta
`strtrim(string)`	retorna “string” com espaços removidos, tanto no início quanto no fim
`md5(string)`	retorna o hash MD5 de “string”

Tabela 3.7: Funções utilizadas em filtros

Os atributos de entidades podem ser utilizados livremente em expressões, porém devido a restrições da sintaxe da linguagem C++, em algumas situações a função `orm::member` deve ser

especificada para especificar um atributo. Desta forma é possivel adiar a execução da expressão até o momento que a expressão correspondente em SQL seja gerada pela CPPObjects. Como uma regra geral, utilize sempre esta função da seguinte forma: `orm::member(&Aluno::nome)`, por exemplo, quanto o membro é usado em um operador unário ou binário (exceto `like`, `regex` e seus semelhantes), de forma que pelo menos um dos membros do operador seja uma operador ou função de filtros definida pela CPPObjects. Atributos podem ser passados para funções sem utilizar `orm::member`.

Como estas regras são um tanto obscuras, é sempre válido utilizar a função `orm::member` para passar os atributos de uma entidade. Caso algo de errado seja feito, o programa não irá compilar.

No momento a utilização de atributos de objetos relacionados diretamente ou indiretamente ao objeto pesquisado só é possível nos casos onde a recuperação do objeto já incorra em um *join* entre tabelas. Se o objeto pesquisado tiver como atributo a chave estrangeira para um objeto relacionado, a recuperação do primeiro não necessita de um *join*, e portanto os atributos do segundo não podem ser utilizados em filtros. Esta restrição será tratada em versões futuras da CPPObjects.

3.7 Relacionamentos

As associações entre objetos são definidas na CPPObjects através da criação de um membro de relacionamento nas suas respectivas estruturas. Este membro é um atributo comum cujo tipo é `orm::rel<T,M,N>`, onde T é o tipo da entidade relacionada e M e N especificam a sua cardinalidade com respeito à entidade relacionada. Cada cardinalidade especifica o número de objetos que podem ser relacionados a cada objeto cujo tipo está sendo definido. Este número pode ser composto por um valor mínimo (incluindo 0) e um máximo, ou ainda o valor especial `orm::MANY`, que é utilizado quando não há um limite superior para o número de objetos relacionados.

Por exemplo, `rel<Endereco, 0, 1>` declara o relacionamento a uma entidade Endereço cuja cardinalidade é $(0:1)$, isto é, o relacionamento pode apontar para nenhum ou no máximo um objeto associado. `rel<Turma,0,MANY>` especifica que este relacionamento aponta para objetos do tipo `Turma` que podem existir um numero ilimitado de turmas, incluindo nenhuma turma.

Associações são caracterizadas pelo número de relacionamentos que as compõe e suas cardinalidades. Elas são representadas especificando-se a cardinalidade de ambas as suas extre-

midades, como em $(1:1)-(0:n)$ ou $(1:n)-(0-n)$, significando relacionamentos um-para-muitos e muitos-para-muitos respectivamente.

Há casos onde uma associação é formada por 3 ou mais relacionamentos quando uma relacionamento está ligado a outros 2 ou mais no outro lado da associação. Estas associações são formalmente consideradas duas associações distintas onde um lado das duas aponta para a mesma entidade, porém CPPObjects as trata como se fossem uma só associação. Estas associações são ditas "assimétricas" devido a este desbalanceamento do número de relacionamentos em cada extremidade. Relacionamentos assimétricos são representados por $(1:1)-[(0:n)(0:n)]$, por exemplo. Note que que os dois relacionamentos são especificados do lado esquerdo desta associação.

Todas as associações mencionadas até aqui são ditas bidirecionais pois um objeto pode alcançar seus objetos relacionados e vice-versa. Há um tipo de associação onde somente um relacionamento está envolvido. Elas são chamadas de associações unidirecionais e são representados por $-(1:1)$ ou $-(0:n)$, por exemplo. Este tipo de associação ainda liga duas entidades, porém somente uma pode apontar para a outra, e a recíproca não é verdadeira. Associações unidirecionais apresentam algumas vantagens sobre as bidirecionais em termos de performance durante a manutenção do estado interno das associações feitos pela CPPObjects.

A listagem 3.6 define as entidades `Turma` e `Disciplina` tal como é mostrado na figura 3.4.

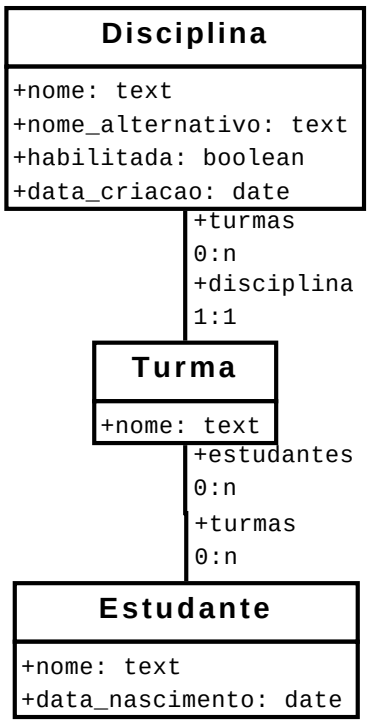

Figura 3.4: Diagrama de classes de Turmas e Disciplinas

Listagem 3.6: Implementação das entidades Turma e Disciplina

```
1  class Class; // declarações adiantadas

3  struct Disciplina : orm::object<Disciplina>
4  {
5      static orm::metainfo<Disciplina> meta;

7      Disciplina() : classes(*this) {}

9      // Atributos
10     std::string nome;
11     bool habilitado;
12     boost::gregorian::date data_criacao

14     // Relacionamentos
15     orm::rel<Turma,0,orm::MANY> turmas;
16 };

18 struct Turma : orm::object<Turma>
19 {
20     static orm::metainfo<Turma> meta;

22     Turma() : disciplina(*this) {}

24     // Atributos
25     std::string nome;

27     // Relacionamentos
28     orm::rel<Disciplina,1,1> disciplina;
29 };

31 // Definição das metainformações da Disciplina
32 orm::metainfo<Disciplina> Disciplina::meta =
33     orm::metainfo<Disciplina>("disciplina")
34         .ident("iddisciplina")
```

```
35          .member(&Disciplina::nome, "nome")
36          .member(&Disciplina::habilitado, "habilitado")
37          .member(&Disciplina::data_criacao, "data_criacao")
38          .member(&Disciplina::turmas, "idturma", &Turma::turma);

40  // Definição das metainformações da Turma
41  orm::metainfo<Turma> Turma::meta =
42      orm::metainfo<Turma>("turma")
43          .ident("idturma")
44          .member(&Turma::nome, "nome")
45          .member(&Turma::disciplina, "iddisciplina", &Disciplina::turmas)
46          .member(&Turma::alunos, "idaluno", &Aluno::turmas);
```

Estas duas definições de entidades se assemelham à definição da entidade `Aluno` feita anteriormente, com exceção das definições dos relacionamentos. A inclusão de relacionamentos na estrutura requer que o seu construtor passe uma referência para si próprio para o membro de associação para que este tenha acesso ao identificador do objeto que o contém, entre outros detalhes de implementação.

De acordo com o diagrama ER (veja 3.4, p. 36), cada disciplina está associada a zero ou mais turmas, e cada turma só pode ter uma disciplina. A primeira relação é representada pela declaração de `rel<Turma,0,MANY>` enquanto que a segunda é representada por `rel<Disciplina,1,1>`, mostrando que o mapeamento entre as associações descritas no diagrama ER e as declarações de relacionamento na CPPObjects é direta.

Logo após as metainformações associadas a cada uma destas relações devem ser criadas, como seria necessário para um atributo normal, exceto que agora o método `metainfo<T>::member` aceita um 3º parâmetro quando utilizado em relacionamentos bidirecionais, como mostrado nas linhas 38 e 45. O programador precisa especificar qual membro da entidade destino aponta de volta para a entidade que está sendo definida. Isto permite que a CPPObjects atualize, por exemplo, o relacionamento `Disciplina::turmas` quando uma turma é adicionada ou removida diretamente por algum processo na aplicação. Daí para frente os membros de relacionamentos podem ser tratados quase como se fossem um outro atributo qualquer.

3.7.1 Associações bidirecionais

Associações bidirecionais se referem à habilidade de um objeto referenciar um objeto que por sua vez referencia de volta o primeiro. Nesta situação há sempre dois relacionamentos envolvidos referenciando-se um ao outro e vice-versa. CPPObjects irá manter os dois relacionamentos atualizados. Por exemplo, se o programador remove um objeto do banco de dados, os objetos que diretamente dependem dele serão atualizados através remoção do objeto do relacionamento que continha uma referência ao objeto removido. Em outras palavras, associações entre objetos podem ser feitas a partir das duas extremidades da associação. No exemplo de Aluno e Turma, um aluno pode ser adicionado à turma, e também a turma pode ser atribuída ao aluno. O estado final de ambos os objetos é o mesmo nas duas situações.

A biblioteca pode inclusive remover objetos dependentes de acordo com a cardinalidade entre os objetos envolvidos quando a vida de um objeto é dependente da vida do objeto removido, como no caso quando o relacionamento tem cardinalidade $(1:1)$. A adição e modificação de objetos funciona da mesma forma com relação à atualização dos objetos relacionados.

3.7.2 Associações assimétricas

Algumas associações, tais como a entre uma pessoa e seus pais, são ditas assimétricas quando dois relacionamentos em uma mesma entidade (pai e mãe na entidade pessoa) se referem ao mesmo relacionamento em outra entidade (filhos na entidade pais).

Note que nestes casos não é possível adicionar um filho em um objeto da entidade *pais* pois o sistema não tem como atualizar a outra ponta da associação na entidade *pessoa* já que ele não sabe se o objeto da entidade *pais* na verdade é a mãe ou o pai do filho adicionado.

Segue então que as associações assimétricas só podem ser atualizadas a partir de uma extremidade. Em outras palavras, objetos só podem ser adicionados em um relacionamento da associação de forma que não ocorra alguma ambigüidade na operação. Esta ambigüidade só ocorre se o relacionamento estiver associado a mais de um relacionamento da outra entidade. Neste caso ele deverá ter ser do tipo `const_rel<T,M,N>`. Relacionamentos deste tipo são somente para leitura da mesma forma que um `const_iterator` só permite a leitura do objeto apontado. Objetos são adicionados neste relacionamento somente a partir da outra ponta da associação, que deve ser um relacionamento do tipo `rel<T,M,N>`, que permite adições e remoções. A CPPObjects irá atualizar o relacionamento somente para leitura automaticamente.

A listagem 3.7 mostra como associações assimétricas são implementadas usando CPPOb-

jects. Ela implementa o diagrama de classes mostrado na figura 3.5.

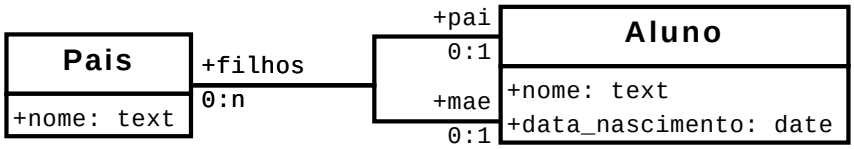

Figura 3.5: Diagrama de classes com associação assimétrica

Listagem 3.7: Definição da relação entre as entidades Aluno e Pais

```
1  struct Pais;
2  struct Aluno : orm::object<Aluno>
3  {
4      static orm::metainfo<Aluno> meta;

6      Aluno() : pai(*this), mae(*this) {}

8      // Atributos
9      std::string nome;
10     boost::gregorian::date data_nascimento;

12     // Relacionamentos
13     orm::rel<Pais,0,1> pai;
14     orm::rel<Pais,0,1> mae;
15 };

17 struct Pais : orm::object<Pais>
18 {
19     static orm::metainfo<Pais> meta;

21     Pais() : filhos(*this) {}

23     // Atributos
24     std::string nome;

26     // Relacionamentos
27     // Este aqui é a parte somente para leitura de uma associação
```

```
28      // assimétrica, daí o tipo ser const_rel.
29      orm::const_rel<Aluno,0,orm::MANY> filhos;
30  };

32  // Definição das metainformações da entidade Aluno
33  orm::metainfo<Aluno> Aluno::meta =
34      orm::metainfo<Aluno>("aluno")
35          .ident("idaluno")
36          .member(&Aluno::nome, "nome")
37          .member(&Aluno::data_nascimento, "data_nascimento")
38          .member(&Aluno::pai, "idpai", &Pais::filhos)
39          .member(&Aluno::mae, "idmae", &Pais::filhos);

41  // Definição das metainformações da entidade Pais
42  orm::metainfo<Pais> Pais::meta =
43      orm::metainfo<Pais>("pais")
44          .ident("idpai")
45          .member(&Pais::nome, "nome")
46          // Pais::filhos está associado com Aluno::mae e Aluno::pai,
47          // o que caracteriza como fazendo parte de uma associação
48          // assimétrica
49          .member(&Pais::filhos, "idfilho", &Aluno::mae, &Aluno::pai));
```

3.7.3 Associações unidirecionais

Em certas ocasiões o programador não deseja ou não precisa de associações bidirecionais pelo fato dela resultar em perdas de desempenho durante a recuperação de objetos e sua manipulação. Quando este é o caso, existe a possibilidade de se criar associações unidirecionais definindo-se somente um relacionamento em uma só entidade, deixando a outra entidade associada intacta. Isto normalmente é necessário quando um objeto referencia muitos outros segundo o diagrama ER.

Durante a definição das metainformações da entidade que contém o único relacionamento da associação, o programador não irá ligá-la a outro relacionamento, de forma que a função `orm::metainfo<T>::member` não necessitará de um 3º parâmetro que normalmente indica a outra ponta da relação. Sua definição se tornará mais parecida com a definição de atributos

normais.

A listagem 3.8 define a entidade `Nota` e sua associação com a entidade `Aluno`. De acordo com o modelo a ser implementado, cada nota mantém uma referência a seu aluno correspondente, mas o contrário não é válido pois não é possível chegar no objeto `Nota` a partir do objeto `Aluno`. Isto é feito pois não há necessidade de a partir do aluno termos acesso à todas as suas notas de avaliações de todas as disciplinas que cursou. Faz mais sentido dada uma nota chegar no aluno que a obteve.

Listagem 3.8: Definição da entidade Nota

```
1  struct Aluno;
2  struct Nota : orm::object<Nota>
3  {
4      static orm::metainfo<Nota> meta;
5      Nota() : aluno(*this) {}

7      // Atributos
8      double valor;
9      // Relacionamentos
10     orm::rel<Aluno,1,1> aluno;
11 };

13 // Definição das metainformações da entidade Nota
14 orm::metainfo<Nota> Nota::meta =
15     orm::metainfo<Nota>("nota")
16         .ident("idnota")
17         .member(&Nota::valor, "valor")
18         // Não há a necessidade de especificar o relacionamento da
19         // entidade associada pois estamos configurando uma associação
20         // unidirecional. Note que a definição se assemelha à definição de
21         // um atributo normal
22         .member(&Nota::aluno, "idaluno");
```

3.7.4 Dominância de relacionamentos

Cada relacionamento de uma associação bidirecional pode assumir um dos seguintes papéis: mestre ou escravo. Esta característica intrínsica de um relacionamento é chamada de sua *dominância*. O relacionamento mestre normalmente é o que é responsável por gerenciar os identificadores dos objetos relacionados, ou seja, a tabela da entidade do relacionamento contém uma coluna com a chave estrangeira do objeto relacionado. A outra entidade da associação deve ser escrava por ser referenciada pela tabela mestre.

Na maioria dos casos a CPPObjects pode definir implicitamente a dominância dos relacionamentos de uma associação. Isto acontece quando estes tem diferentes cardinalidades, de forma que a biblioteca tem como definir sem ambigüidade suas dominâncias. Quando isto não pode ser feito, o programador deve definir explicitamente a dominância de um relacionamento como sendo mestre. O outro relacionamento fica sendo automaticamente o escravo.

Esta definição é feita simplesmenta passando `orm::MASTER` como o 4º parâmetro ao tipo `orm::rel` correspondente. Por exemplo, em associações muitos-para-muitos entre turmas e alunos, o sistema ao criar uma tabela auxiliar com duas colunas que guarda as associações pode escolher como primeira coluna tanto a chave estrangeira de alunos quanto de turmas, havendo aí então uma indefinição. A especificação de dominância mestre para o relacionamento que está na entidade aluno, por exemplo, faz com que a primeira coluna da tabela auxiliar guarde as chaves estrangeiras para alunos, a segunda fica com as chaves estrangeiras para turmas. Os relacionamentos são declarados como:

```
1 using namespace orm;
2 // Na estrutura Turma
3 rel<Alunos,0,MANY,MASTER> alunos;

5 // Na estrutura Aluno
6 rel<Turma,0,MANY> turmas; // Não há necessidade de definir a dominância,
7                           // ela é implicitamente escrava.
```

Associações assimétricas sempre têm a dominância de seus relacionamentos definida automaticamente pela biblioteca. Nestes casos, o relacionamento que se encontra associado a vários outros (com tipo `orm::const_rel` é declarado como escravo, enquanto que os outros relacionamentos da outra ponta da associação são todos mestres.

Já nas associações unidirecionais, como só há um relacionamento envolvido, esta é definida automaticamente como sendo mestre.

Associação	*dominância da 1º rel.*	*dominância do 2º rel.*
$(0:1)-(0:1)$	definido pelo usuário	definido pelo usuário
$(1:1)-(0:1)$	escravo	mestre
$(1:1)-(1:1)$	definido pelo usuário	definido pelo usuário
$(0:1)-(m:n)$	escravo	mestre
$(1:1)-(m:n)$	escravo	mestre
$(m:n)-(o:p)$	definido pelo usuário	definido pelo usuário
$-(0:1)$	não disponível	mestre
$-(0:n)$	não disponível	mestre
$(0:n)-[(1:1)(1:1)]$	escravo	mestre

Tabela 3.8: Dominâncias associadas a relacionamentos

A tabela 3.8 lista diversos tipos de associações, indicando as quais o programador deve definir explicitamente a dominância do relacionamento.

3.7.5 Relacionamentos monovalorados

Relacionamentos cuja cardinalidade é $(0:1)$ ou $(1:1)$ são considerados monovalorados pois eles podem apontar para no máximo um objeto por vez. A classe template `orm::rel<T, M,1>` é parcialmente especializada para prover uma interface apropriada para esta situação. O objeto relacionado é acessado como se o relacionamento fosse um *smart pointer* que aponta para o objeto.

A figura 3.6 mostra um diagrama de classes de um relacionamento monovalorado com seus métodos públicos. Este diagrama não mostra estritamente como o template `orm::rel<T,M,1>` é implementado, mas ajuda na compreensão dos seus princípios e operações básicas.

Existem dois tipos de relacionamentos monovalorados: um que sempre se refere a um objeto, representando a cardinalidade $(1:1)$, e o que pode apontar para zero ou mais objetos, equivalendo à cardinalidade $(0:1)$. No diagrama acima elas são representadas pelos templates `orm::rel<T,1,1>` e `orm::rel<T,0,1>` respectivamente.

Os relacionamentos monovalorados agem em ambos os casos como se fossem *smart pointers* para o objeto relacionado, implementando a semântica usual de ponteiros. O usuário pode designar um objeto como sendo o objeto apontado pelo relacionamento simplesmente realizando uma simples atribuição de valores, utilizando o operador de atribuição do relacionamento (`operator=(T)`). Se o relacionamento for bidirecional, a biblioteca irá atualizar o outro relacionamento da associação, garantindo assim que o objeto atribuido referencie o objeto cujo relacionamento ele está sendo atribuido.

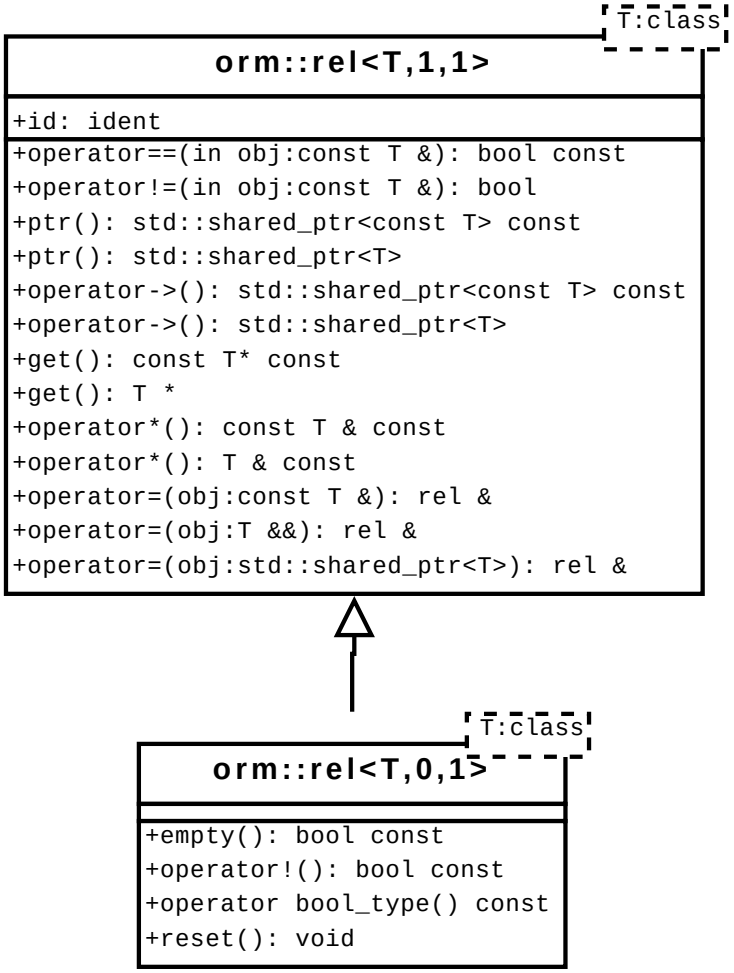

Figura 3.6: Diagrama de classes de relacionamentos monovalorados

Para os casos onde somente o identificador do objeto é necessário, o programador pode usar o membro `orm::rel<T,M,1>::id` que sempre está sincronizado com o identificador do objeto relacionado. Isto evita que o objeto seja recuperado do banco de dados se ele ainda não tiver sido. Isto é possível devido à forma que os relacionamentos mantêm as referências aos seus objetos-alvo. Quando um objeto é recuperado do banco de dados, para cada relacionamento do objeto a CPPObjects somente carrega o identificador do objeto relacionado. Desta forma evitam-se carregamentos espúrios quando somente o identificador do objeto é necessário. Caso o objeto apontado seja acessado através do operador de dereferência ou acesso a ponteiro (`rel::operator*()` ou `rel::operator->()`, ele será carregado do banco na sua totalidade.

Para relacionamentos de cardinalidade $(0:1)$ uma interface é disponibilizada que permite que o programador saiba se existe um objeto associado ou não – `orm::rel<T,0,1>::empty` – e ainda para resetar o relacionamento, ou seja, fazê-lo apontar para nenhum objeto – `orm::rel<T,0,1>::reset`. Esta interface foi inspirada na mesma utilizada pelo template `std::shared_ptr<T>` e deve ser familiar para programadores de C++, reforçando o fato que relações

monovaloradas devem ser encaradas como se fossem *smart pointers*.

Relacionamentos de cardinalidade $(1:1)$ garantem que sempre haja um objeto sendo apontado por ele, mesmo ele objeto seja um não inicializado. Quando um objeto que tenha relacionamentos $(1:1)$ novo é criado, estes são inicializados primeiramente com um identificador 0. Caso o objeto-alvo seja acessado, a biblioteca irá criar um novo objeto vazio que será retornado para possivelmente ser preenchido.

A listagem 3.9 implementa duas entidades que utilizam dois tipos de relacionamentos monovalorados: `AvaliacaoPlanejada` e `Avaliação`, conforme mostrados no diagrama 3.7.

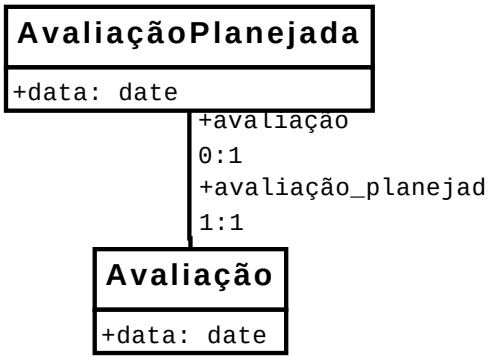

Figura 3.7: Diagrama de classes das entidades AvaliacaoPlanejada e Avaliacao

Listagem 3.9: Implementação de entidades com relacionamentos monovalorados

```
2  class Avaliacao
3  struct AvaliacaoPlanejada : orm::object<AvaliacaoPlanejada>
4  {
5      static orm::metainfo<AvaliacaoPlanejada> meta;

7      AvaliacaoPlanejada() : avaliacao(*this) {}

9      // Atributos
10     boost::gregorian::date data_avaliacao;

12     // Relacionamentos
13     orm::rel<Evaluation,0,1> avaliacao;
14 };

16 struct Avaliacao : orm::object<Avaliacao>
```

```
17 {
18     static orm::metainfo<Avaliacao> meta;

20     Avaliacao() : avaliacao_planejada(*this) {}

22     // Atributos
23     boost::gregorian::date data;

25     // Relacionamentos
26     orm::rel<AvaliacaoPlanejada,1,1> avaliacao_planejada;
27 };

29 // Definição das metainformações da entidade AvaliacaoPlanejada
30 orm::metainfo<AvaliacaoPlanejada> AvaliacaoPlanejada::meta =
31     orm::metainfo<AvaliacaoPlanejada>("avaliacao_planejada")
32         .ident("idavaliacao_planejada")
33         .member(&AvaliacaoPlanejada::data, "data")
34         .member(&AvaliacaoPlanejada::avaliacao, "idavaliacao",
35             ,&Avaliacao::avaliacao_planejada);

37 // Definição das metainformações da entidade Avaliacao
38 orm::metainfo<Avaliacao> Avaliacao::meta =
39     orm::metainfo<Avaliacao>("avaliacao")
40         .ident("idavaliacao")
41         .member(&Avaliacao::data, "data")
42         .member(&Avaliacao::avaliacao_planejada,"idavaliacao_planejada"
43             ,&AvaliacaoPlanejada::avaliacao)
```

Note que o tipo dos membros membros `AvaliacaoPlanejada::avaliacao` e `Avaliacao ::avaliacao_planejada` recebe como parâmetro template o tipo da entidade apontada, porém sem necessitar que esta esteja declarada no momento da especificação do tipo do relacionamento. Isto foi feito desta forma para mais uma vez para diminuir a dependência entre os cabeçalhos que definem cada entidade e tornar possível definir entidades que referenciam umas às outras e vice-versa. Caso a declaração de um relacionamento exigisse que a entidade relacionada estivesse declarada, esta não poderia referenciar de volta a primeira entidade, já que referências cíclicas não são possíveis nestas situações. A definição da entidade apontada

pelo relacionamento só é necessária quando o objeto apontado é acessado, ou no momento da definição das metainformações associadas ao relacionamento. Nos outros casos, somente uma declaração adiantada[4] é necessária.

A listagem 3.10 mostra várias operações que utilizam relacionamentos monovalorados, exemplificando assim vários conceitos abordados até agora.

Listagem 3.10: Utilização de relacionamentos monovalorados

```
1  orm::document doc("psql://localhost/testdb",Evaluation::meta);

3  // Cria uma avaliação planejada a ser aplicada dia 2/out/2008
4  AvaliacaoPlanejada avalplan;
5  avalplan.data = boost::gregorian::date(2008,10,2);
6  // Não há nenhuma avaliação realizada associada
7  assert(!avalplan.avaliacao);

9  // Cria uma avaliação que acabou acontecendo dia 5/out/2008
10 avalplan.avaliacao = Avaliacao(); // avaliacao vazia
11 avalplan.avaliacao->data = boost::gregorian::date(2008,10,5);

13 // A avaliação planejada e a avaliação planejada da avaliação são
14 // na verdade o mesmo objeto, já que há uma associação bidirecional
15 // entre as duas entidades.
16 assert(&avalplan == &*avalplan.avaliacao->avaliacao_planejada);

18 // Adiciona a avaliacao planejada no banco de dados
19 doc.add(avalplan);

21 // A biblioteca adiciona tanto a avaliação planejada quando a
22 // sua avaliação (objeto relacionado).
23 assert(avalplan.id != 0);
24 assert(avalplan.avaliacao.id != 0);
25 // O identificador do relacionamento da avaliação está em sincronia com
26 // o identificador do objeto apontado
27 assert(avalplan.avaliacao.id = avalplan.avaliacao->id);
```

[4] Em inglês: forward declaration

```
29 // Vamos mudar a data da avaliação para 6/out/2008
30 avalplan.avaliacao->data = boost::gregorian::date(2008,10,6);
31 // O documento irá percorrer todos os objetos relacionados ao objeto
32 // modificado e realizar as alterações realizadas, modificando então a
33 // data da avaliação. Note que estamos passando a avaliação planejada, e
34 // não a avaliação realizada, porém esta será atualizada no banco.
35 doc.edit(avalplan);

37 // Recupera a avaliação planejada adicionada anteriormente do
38 // banco de dados a partir do seu identificador
39 std::shared_ptr<const AvaliacaoPlanejada> planev_db
40     = doc.query<AvaliacaoPlanejada>(avalplan.id).lock();

42 // A avaliação planejada tem uma avaliação realizada (deve ter...)
43 if(planev_db->avaliacao)
44 {
45     // A avaliação alterada anteriormente realmente foi atualizada no
46     // banco de dados
47     assert(avalplan_bd->avaliacao->data == boost::gregorian::date
           (2008,10,6));
48 }

50 // Salva o identificador da avaliação realizada
51 orm::ident idavaliacao = avalplan.avaliacao.id;
52 // Desassocia a avaliação da sua avaliação planejada
53 avalplan.avaliacao.reset();
54 assert(!avalplan.avaliacao); // Realmente foi desassociada
55 doc.edit(avalplan); // Atualiza o banco de dados

57 // A avaliação realizada não está mais no banco de dados pois
58 // seu relacionamento com a entidade AvaliacaoPlanejada é (1:1), ou
59 // seja, uma avaliação realizada não pode existir sem estar associada
60 // a uma avaliação planejada
61 assert(doc.query<Avaliacao>(idavaliacao).expired());
```

A listagem acima mostra fatos interessantes sobre relacionamentos monovalorados em ge-

ral. O mais importante é que, como dito anteriormente, relacionamentos agem como um `std::shared_ptr<T>` como pode ser visto na linha 43 onde o relacionamento é testado para ver se ele aponta para um objeto ou não, ou ainda na linha 53 onde seu valor é resetado.

As asserções que se iniciam na linha 23 mostram que uma vez que os objetos relacionados são criados, adicionar um deles no banco de dados acaba adicionando todos os outros recursivamente, já que o identificador do objeto relacionado é diferente de 0. Atualizações de objetos também são propagados para os objetos relacionados como é checado pela linha 47. A modificação de `avalplan.avaliacao` é realizada quando `avalplan` é atualizada.

Finalmente, a biblioteca sempre garante que as restrições de cardinalidade são respeitadas. Na linha 53 a avaliação realizada é desassociada da sua avaliação planejada. Já que a primeira deve estar associada sempre à segunda, a primeira deve ser removida do banco de dados já que sua associação foi desfeita. Se a cardinalidade fosse $(0:1)$, a avaliação realizada permaneceria no banco de dados, porém seu relacionamento `Avaliacao::avaliacao_planejada` não apontaria para nada.

3.7.6 Relacionamentos multivalorados

Quando um objeto pode ser associado a mais de um objeto relacionado, o relacionamento correspondente é chamado de multivalorado. Na CPPObjects este tipo de relacionamento é modelado como uma lista de objetos utilizando uma interface similar à `orm::list<T>` (veja 3.2, p. 24).

Relacionamentos multivalorados são declarados utilizando a especialização parcial do template `orm::rel<T,M,N>` quando $N \in [2,\infty)$, $M \in [0,N]$ e $M,N \in \mathbb{N}$. Na CPPObjects `orm::MANY` é utilizado no lugar de ∞ quando esta cardinalidade é requerida.

A listagem 3.6 na página 37 mostra como relacionamentos multivalorados são declarados. Estes são do tipo `orm::rel<Turma,0,orm::MANY>`. Como fazem parte de uma associação bidirecional, o relacionamento `Turma::disciplina` é o relacionamento correspondente na entidade `Turma`.

A listagem 3.11 mostra como relacionamentos multivalorados são utilizados na prática.

Listagem 3.11: Utilização de relacionamentos multivalorados

```
1 orm::document doc("psql://localhost/testdb",Discipline::meta);

3 // Cria uma disciplina
```

```
4  Disciplina calc1;
5  calc1.nome = "Cálculo I";
6  calc1.turmas.insert(Turma()); // associa a uma turma vazia
7  calc1.turmas.begin()->nome = "EL1"; // atribui um nome à turma criada

9  // Cria uma nova turma gerenciada por um smart pointer
10 std::shared_ptr<Turma> em1(new Turma);
11 em1->nome = "EM1";
12 calc1.turmas.insert(em1); // associa-a à disciplina Cálculo I

14 // Uma vez associada, a biblioteca seta automaticamente a disciplina
15 // da turma EM1 para Cálculo I
16 assert(em1->disciplina->nome == "Cálculo I");

18 // É possível modificar a disciplina a partir de suas turmas
19 em1->disciplina->nome = "Calculus 1";
20 assert(calc1.nome == "Cálculo 1"); // Realmente funciona!

22 doc.add(*em1); // Adiciona a turma no banco de dados
23 // Adiciona também os objetos relacionados (a disciplina Cálculo I).
24 // Note que ela está na pilha, mesmo assim é atualizada.
25 assert(calc1.id != 0);

27 // Recupera do banco de dados a turma cujo nome é EL1
28 std::shared_ptr<const Turma> el1_bd
29     = doc.query<Turma>(orm::member(&Turma::nome) == "EL1").lock();
30 assert(el1_bd && el1_bd->nome == "EL1"); // O objeto está correto
31 // A sua disciplina também está correta
32 assert(el1_bd->disciplina->nome == "Cálculo 1");

34 // Podemos iterar por todas as turmas da disciplina, já que
35 // o relacionamento multivalorado é uma lista.
36 for(Disciplina::const_iterator it = calc1.turmas.begin();
37     it != calc1.turmas.end(); ++it)
38 {
```

```
39      std::cout << "Turma: " << it->nome << std::endl;
40  }

42  doc.rem(calc1); // Remove a disciplina
43  // Já que a disciplina foi removida e não pode existir uma turma
44  // sem disciplina, todas as turmas também foram removidas do
45  // banco de dados
46  assert(!doc.query<Turma>(orm::member(&Turma::nome) == "EL1").lock());
47  assert(!doc.query<Turma>(orm::member(&Turma::nome) == "EM1").lock());
```

3.8 *Cache* de Objetos

Quando uma aplicação utiliza extensivamente vários objetos, é normal armazenar cópias duplicadas deles em diversos lugares como em controles GUI, contêineres locais, buffers temporários etc.

Uma desvantagem desta abordagem é que modificações de um objeto por uma parte do sistema não são facilmente propagáveis às outras cópias do objeto. O sistema fica então com várias cópias inconsistentes destes, cada uma refletindo seu estado em um determinado tempo. A única forma de se ter a versão mais recente do objeto é realizando uma consulta no banco de dados para retorná-lo no seu estado mais atual.

Um *cache* de objetos foi então criado para solucionar este problema. Ele serve como um repositório central de objetos que são garantidos pela biblioteca de terem seus estados refletindo os estados armazenados no banco de dados. As vantagens são duas: o número de acessos ao banco é minimizado e os subsistemas da aplicação podem manipular os objetos através de ponteiros para os objetos que são gerenciados pelo *cache* para que eles sempre os vejam no seu estado mais recente caso alguma outra parte da aplicação realize alguma alteração nele, minimizando os problemas de inconsistência mencionados anteriormente.

3.8.1 Características

A CPPObjects implementa um *cache* de objetos unidirecional, síncrono e do tipo *write-through* projetado para satisfazer as consultas mais comuns feitas principalmente pelo subsistema de gerenciamento de relacionamentos da biblioteca, onde os objetos são freqüentemente carregados a partir do seu identificador. As consultas do usuário são realizadas pelo próprio

cache quando elas devem retornar objetos baseado nos seus identificadores. Consultas mais complexas são realizadas diretamente pelo banco de dados, garantindo uma implementação mais simples do *cache*.

Os objetos adicionados, modificados ou removidos pela aplicação são sempre atualizados diretamente ao banco de dados. Somente se esta operação for bem sucedida é que os objetos do *cache* são atualizados de acordo. A esta característica dá-se o nome de *write-through*.

A natureza unidirecional do *cache* de objetos implica que modificações feitas diretamente no banco de dados possivelmente por outras aplicações não serão vistas pelo *cache*. Para minimizar problemas de coerência de *cache* que podem ocorrer, sempre que um objeto é carregado do banco de dados sua cópia do *cache* é atualizada. Embora esta solução não seja a ótima, análises feitas em aplicações existentes que utilizam versões anteriores da CPPObjects mostram que o usuário acaba por vezes fazendo consultas que são realizadas diretamente pelo banco de dados, sem passar pelo *cache*. Os objetos retornados representam o estado mais atual deles, e isto acaba sendo refletido no *cache*. As partes da aplicação que referenciam objetos do *cache* têm acesso então ao estado mais recente dos objetos. Ao passar do tempo mais consultas são feitas e o *cache* vai sendo mantido em sincronia com o banco de dados devido à atualização constante feita nele quando objetos são carregados do banco.

A solução ideal para resolver problemas de coerência de *cache* especialmente quando várias aplicações acessam o mesmo banco de dados seria criar uma arquitetura em múltiplas camadas onde os acessos ao banco de dados passam primeiro por um servidor que envia notificações às aplicações quando um determinado objeto é adicionado, alterado ou removido. Esta solução não foi implementada devido a restrições de tempo, mas versões anteriores da biblioteca já utilizaram esta técnica em um sistema de rastreamento de veículos onde a integridade e coerência dos objetos era de suma importância, obtendo resultados satisfatórios. De qualquer forma seria fácil estender a CPPObjects para trabalhar em uma arquitetura em múltiplas camadas, como será abordado ao capítulo 4 referente a idéias futuras.

Uma outra opção que o programador tem para lidar com o problema de coerência de dados é de tempos em tempos apagar o conteúdo do *cache*, ou ainda permitir que o usuário o faça. Isto garante que próximos acessos a objetos que não estejam carregados inteiramente sejam realizados através de consultas diretas ao banco de dados, que retornará o objeto com o estado mais recente.

O *cache* pode ser manipulado através do documento, que em última intância é o objeto que gerencia o *cache*, como foi explicado na seção 3.5, página 29. Sempre que o *cache* precisa ser esvaziado, basta chamar a função `orm::document::clear_cache()`. Cada referência a

objetos do *cache* utilizando um `std::weak_ptr<const T>` será resetada, como será explicado na próxima seção.

3.8.2 Objetos gerenciados

Os objetos são armazenados no *cache* em um contêiner de *smart pointers* do tipo `std::shared_ptr<const T>`. Eles não devem ser modificados diretamente pelo programador, daí sua qualificação `const`. As referências externas aos objetos do *cache* devem ser feitas preferencialmente através de *smart pointers* do tipo `std::weak_ptr<const T>`.

Esta configuração implementa o conhecido padrão de observador, só que aplicado aos objetos do *cache*. Um *smart pointer* do tipo `shared_ptr` é observado por outros *smart pointers* do tipo `weak_ptr`. Todos eles apontam para o mesmo objeto, porém quando o último `shared_ptr` é destruido, todos os *smart pointers* `weak_ptr` são resetados, já que eles não observam mais o objeto que foi removido. É importante ressaltar que o objeto apontado só será destruído quando o último `shared_ptr` que aponta pra ele for destruído, não importanto o número de `weak_ptr` que apontam para o objeto.

Baseado nesta explicação, é compreensível a necessidade de sempre que possível referenciar objetos do *cache* através de `weak_ptr`, já que isto possibilita que quando uma parte da aplicação remova um objeto, este será removido do *cache* e todos os `weak_ptr` que apontam para ele serão resetados, garantindo que todas as partes da aplicação enxerguem a remoção do objeto.

A listagem 3.12 mostra vários usos do *cache* e como os objetos gerenciados e referenciados se comportam quando algum objeto é atualizado ou removido do *cache*.

Listagem 3.12: Exemplo de utilização do *cache*

```
1 orm::document doc("psql://localhost/testdb",Disciplina::meta);
2 // Criação da disciplina Eletrônica IV com uma turma EL1
3 Disciplina disc;
4 {
5     disc.nome = "Eletrônica IV";
6     Turma el1;
7     el1.nome = "EL1";
8     disc.turmas.insert(std::move(el1));
9 }
```

```
11 // Adiciona a disciplina no banco de dados
12 std::weak_ptr<const Disciplina> disc_cache = doc.add(disc);

14 // Os objetos do cache devem ser transferidos para um shared_ptr para
15 // serem acessados
16 if(std::shared_ptr<const Disciplina> d = disc_cache.lock())
17 {
18     // A disciplina está sendo gerenciada pelo cache
19     assert(d->in_cache());
20     // Ela tem uma turma
21     assert(d->turmas.size() == 1);
22     // E é a que a gente adicionou, a EL1
23     assert(d->turmas.begin()->nome == "EL1")
24     // E esta turma também é gerenciada pelo cache.
25     assert(d->turmas.begin()->in_cache());

27     // Vamos modificar o nome da turma da disciplina para EL2;
28     disc.turmas.begin()->nome = "EL2";
29     doc.edit(disc);

31     // O objeto correspondente no cache foi atualizado também, e a
32     // turma que estamos referenciando enxerga esta modificação
33     assert(d->turmas.begin()->nome == "EL2");

35     // Vamos remover todas as turmas da disciplina.
36     disc.turmas.clear();
37     doc.edit(disc);

39     // A disciplina do cache foi atualizada, ela também não está
40     // associada a nenhuma turma.
41     assert(d->turmas.empty());
42 }

44 // Vamos pôr a disciplina do cache em uma lista. Esta disciplina será
45 // guardada na lista em um weak_ptr
```

```
46 orm::list<const Disciplina> disciplinas;
47 disciplinas.insert_weak(disc_cache);

49 // Ao remover a disciplina,
50 doc.rem(disc);
51 // Ela é removida do cache e todos os weak_ptr's que a referenciam são
52 // resetados, ficando expirados.
53 assert(disc_cache.expired());

55 // A disciplina removida foi removida da lista automaticamente já que
56 // ela era gerenciada por um weak_ptr. Legal!
57 assert(disciplinas.empty());
```

3.9 Rede de objetos

Quando um objeto tem relacionamentos que referenciam outros objetos, que por sua vez fazem referências a outros objetos e assim por diante, toda a coleção de objetos referenciados direta ou indiretamente forma uma rede fechada onde cada objeto pode ser alcançado a partir de outros através de suas associações, como se cada objeto fosse um nó de um grafo e suas associações fossem arestas ligando estes. Esta semântica de navegação de objetos é de grande valia para o programador em contraste com uma manipulação de entidades utilizando somente o modelo relacional, e é a maior caraterística da CPPObjects.

A figura 3.8 mostra uma rede de objetos típica que pode ser criada usando a biblioteca envolvendo disciplinas, alunos, turmas e pais.

Neste diagrama existem alguns laços que, presumindo uma implementação naïve da biblioteca, causariam vazamentos de memória e recursões infinitas durante seu processamento por causa das diversas referências circulares. Estes problemas foram eliminados utilizando-se `weak_ptr` para quebrar as referências circulares e proteções contra recursão infinita em lugares apropriados.

Redes de objetos podem ser classificadas em três grandes grupos de acordo com o fato de elas serem compostas de objetos gerenciados pelo *cache*, não gerenciados por ele ou uma mistura dos dois.

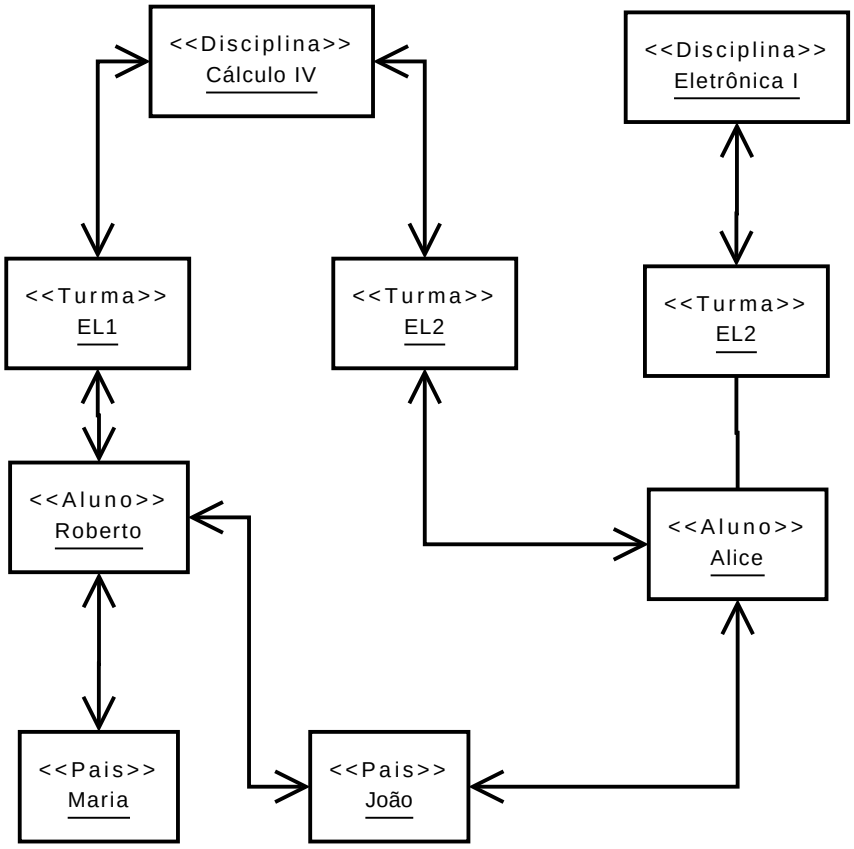

Figura 3.8: Uma rede de objetos típica

3.9.1 Rede gerenciada pelo *cache*

Em uma rede gerenciada pelo *cache* todos os seus objetos também o são. Isto significa que a rede não pode ser modificada indiretamente caso um procedimento externo aja em uma cópia do objeto. Ela só será modificada quando o programador persistir a alteração no banco de dados, momento este que fará com que o *cache* seja atualizado, atualizando também a rede.

Cada objeto da rede forma uma rede totalmente conectada e explorável onde cada objeto pode ser alcançado a partir de qualquer outro, desde que ambos sejam membros da rede.

Quando um objeto da rede é clonado ou copiado, somente seus atributos são copiados. Seus relacionamentos continuam apontado para objetos gerenciados pelo *cache*, fazendo com que o objeto copiado, que não é mais gerenciado pelo *cache*, faça parte de uma rede mista.

3.9.2 Rede não gerenciada pelo *cache*

Seus objetos não são gerenciados pelo *cache*. A rede toda serve como se fosse uma fotografia do estado de todos os seus objetos em um determinado momento, já que ela só pode

ser modificada se um objeto for alterado diretamente. Suas características de conectibilidade e explorabilidade são as mesmas de uma rede gerenciada pelo *cache*.

Quando um objeto de uma rede não gerenciada pelo *cache* é copiado ou clonado, toda a rede é copiada e a rede resultante não possui objetos em comum com a original. Isto pode parecer ineficiente, mas é o único jeito de criar uma "fotografia" dela, muito utilizado em entradas de dados em caixas de diálogo.

3.9.3 Rede mista

A rede mista contém objetos gerenciados e não gerenciados pelo *cache*. A vantagem desta configuração reside nas situações onde somente uma parte da rede deve ser modificada (os objetos não gerenciados pelo *cache*), enquanto que o resto deve refletir o que está armazenado no banco de dados.

As redes mistas, contrariamente aos outros dois tipos apresentados, não são completamente exploráveis. À medida que a rede é percorrida, o subsistema de gerenciamento de relacionamentos checa se o objeto visitado é gerenciado pelo *cache* ou não. Caso este objeto tenha sido alcançado a partir de um objeto não-`const` (significando que o objeto visitado pode ser modificado), o subsistema automaticamente cria uma cópia do objeto caso ele seja gerenciado pelo *cache* de forma que está cópia possa ser modificada. Esta é a forma que a biblioteca tem de garantir que objetos gerenciados pelo *cache* não sejam modificados diretamente pelo programador, somente através do documento, persistindo-o no banco de dados. Se o objeto visitado for alcançado a partir de um objeto `const`, a linguagem C++ garante que ele não pode ser modificado e assim a biblioteca não necessita realizar cópias espúrias durante o percurso, tornando a navegação eficiente.

Nota-se que a qualificação `const` dos objetos é o fator chave para uma exploração eficiente de objetos, com exceção nas situações onde eles precisam ser modificados ao longo do percurso.

Diferentemente dos outros dois tipos de redes apresentados, a rede mista não é completamente explorável, pelo menos não de forma bidirecional. Isto reside no fato de que os objetos gerenciados pelo *cache* sempre estão associados a outros objetos do *cache*. Esta fato pode vir a criar uma situação onde alguns objetos não gerenciados pelo *cache* referem a objetos do *cache*, e estes por sua vez ao invés de referenciarem os primeiros, referem-se às suas cópias gerenciadas do *cache*, de forma que alguma alteração realizada não é vista por este. Esta situação não pode ser contornada e deve ficar sempre em mente durante o desenvolvimento usando a biblioteca.

4 *Idéias Futuras*

Diversas características encontradas em outras bibliotecas de ORM não foram incluidas na CPPObjects devido às restrições de tempo impostas para o projeto. A seguir listaremos algumas das idéias para versões futuras da biblioteca.

4.1 Outros tipos de DBMS

Atualmente a CPPObjects somente suporta o banco de dados PostgreSQL. O ideal é utilizar uma biblioteca que abstraia o conceito de banco de dados, permitindo que este seja trocado sem que a aplicação necessite de modificações.

Existem várias alternativas de bibliotecas que resolvem este problema. A mais interessante chama-se SOCI, encontrada no endereço *http://soci.sourceforge.net/*. Ela é baseada nos mesmos fundamentos de bibliotecas como a Boost, a STL e a própria CPPObjects, Tanto é que ela é uma forte candidata para virar uma subbiblioteca Boost. Sua versão atual, 3.0.0 (em 2008), suporta os seguintes DBMS: Oracle, PostgreSQL, MySQL, SQLite3, Firebird e ODBC. Sua licença é a mesma licença da biblioteca Boost, o permite que ela seja utilizada na CPPObjects sem ônus algum.

4.2 Herança

Uma grave limitação da CPPObjects é que ela não suporta hierarquia de entidades. Como analisado na seção 2.3.1, não existe uma solução ótima e algum comprometimento acaba sempre tendo que ser feito. Um protótipo de tratamento de herança foi realizado no passado, mas ele não era flexível e robusto o suficiente. Outras tentativas serão feitas no futuro e espera-se que em breve a biblioteca poderá contar com este poderoso recurso do modelo orientado a objetos

4.3 Utilizar entidades relacionadas em consultas

Na versão atual da biblioteca não é possível fazer uso de atributos de entidades relacionadas em consultas. Isto só funciona em determinadas situações que dependem de detalhes de implementação. A melhor solução seria utilizar o sistema de consulta por linguagem (veja 2.3.3, p. 11), porém fazer isso irá fazer com que erros de definição de consultas sejam postergados para o tempo de execução, ao invés de serem detectados já em tempo de compilação, como ocorre com a implementação atual da CPPObjects.

4.4 Arquitetura em três camadas (*three-tier*)

Uma solução robusta para o problema da coerência de *cache* em relação ao objeto armazenado no banco de dados é a utilização de uma arquitetura em três camadas, onde existe um servidor de dados entre cada aplicação e o servidor de banco de dados. Este servidor de dados notifica cada aplicação quando algum dado é inserido, alterado ou removido do banco. Ele ainda pode garantir que algumas invariantes dos dados sejam mantidas, invariantes estas muito complexas para serem especificadas direto no banco de dados relacional. Ainda há a possibilidade de fazer com que o servidor administre uma coleção de banco de dados que ofereça um certo nível de redundância, garantido a alta disponibilidade dos dados.

Alguns sistemas de banco de dados oferecem estas características, como o PostgreSQL e seu sistema de notificações, assim como algumas soluções de gerenciamento de clusters de banco de dados. Nestes casos o servidor intermediário não é necessário e o sistema como um todo fica mais simples.

Quando não é possível utilizar um DBMS com estas funcionalidades, é necessário implementar um sistema de serialização de objetos eficiente e robusto. Esta tarefa pode ser delegada à biblioteca boost::serialization, escrita por Robert Ramey. Parte das rotinas de serialização foram escritas para a CPPObjects, mas somente com o intuito de obter uma representação textual dos objetos com fins de depuração e testes. A serialização é unidirecional, e não é possível ainda construir um objeto a partir de sua representação textual.

5 *Conclusão*

A biblioteca CPPObjects representou um novo passo nas soluções de sistemas de mapeamento objeto-relacional, aproveitando-se de modernas técnicas de desenvolvimento tais como programação orientada a objetos e programação genérica, visando tornar a criação de aplicações que lidam com objetos de negócios menos laborosa.

A sua abordagem baseada em redes de objeto baseados em *cache* soluciona vários problemas que ocorrem na prática e exigem do programador maior atenção a detalhes, sob o risco de incorrer em problemas de incoerência de informações caso este quesito não seja tratado corretamente.

A CPPObjects mostrou-se bastante eficaz em atingir estes objetivoss quando utilizada, nas suas versões anteriores, para desenvolver uma grande aplicação de controle de vendas e faturas que inclui mais de 25 entidades relacionadas entre si das mais diversas maneiras possíveis. Construir caixas de diálogo que manipulam estas informações, normalmente lidando com várias entidades simultaneamente, tornou-se de fácil implementação, com um resultado bastante intuitivo para o usuário devido a utilização de "fotografias" da rede de objetos sendo alterada por ele. Desta forma o usuário realiza as alterações nos objetos da rede e só as aplica ao final, quando toda a rede é persistida no banco de dados, podendo ainda assim desistir das alterações sem que a aplicação precise criar transações no banco de dados.

A biblioteca foi desenvolvida utilizando ferramentas e bibliotecas de código aberto, e por este motivo também foi tornada livre, utilizando a licença BSD que permite que ela seja utilizada tando em aplicações de código aberto quanto em aplicações comerciais. A intenção com isso é divulgá-la para que ela auxilie o maior número de pessoas possível, resultando em uma comunidade altamente técnica e com capacidade de criar aplicações cada vez mais complexas e poderosas.

Bibliografia

CODD, E. F. “A relational model of data for large shared data banks”. *Commun. ACM*, ACM, New York, NY, USA, v. 13, n. 6, p. 377–387, 1970. ISSN 0001-0782. Disponível em: <http://doi.acm.org/10.1145/362384.362685>.

CODD, E. F. “*The Relational Model for Database Management*”. 2nd. ed. Boston, MA, USA: Addison-Wesley Publishing Company, Inc., 1990. ISBN 0201141922.

COPLIEN, J. “Curiously recurring template patterns”. *C++ Report*, SIGS Publications, p. 24–27, February 1995.

ELMASRI, R. A.; NAVATHE, S. B. “*Fundamentals of Database Systems*”. 3rd. ed. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1999. ISBN 0805317554.

FOWLER, M. “*Patterns of Enterprise Application Architecture*”. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2002. ISBN 0321127420.

GREGOR, D. et al. “Foundational concepts for c++0x standard library (revision 4)”. ISO/IEC JTC 1, Subcommittee 21, Working Group 21 - The C++ Standards Committee, n. N2737, August 2008. Disponível em: <http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2008/n2737.pdf>. Acesso em: 2008/09/09.

INTERNATIONAL STANDARDS ORGANIZATION. “*ISO 14882:2003*: Programming languages — c++”. [S.l.], 2003.

MEYERS, S. “*Effective C++: 55 specific ways to improve your programs and designs*”. 3rd. ed. [S.l.]: Addison-Wesley Professional, 1992. ISBN 0321334876.

MEYERS, S. “*More Effective C++: 35 new ways to improve your programs and designs*”. 1st. ed. [S.l.]: Addison-Wesley Professional, 1996. ISBN 020163371X.

STROUSTRUP, B. “*The C++ Programming Language*”. special ed. [S.l.]: Addison-Wesley Professional, 2000. ISBN 0201700735.

SUMATHI, S.; ESAKKIRAJAN, S. “*Fundamentals of Relational Database Management Systems*”. 1st. ed. New York: Springer Berlin Heidelberg, 2007. ISBN 3540483977.

SUTTER, H. “*Exceptional C++: 47 Engineering Puzzles, Programming Problems, and Solutions (C++ in Depth Series)*”. 1st. ed. [S.l.]: Addison-Wesley Professional, 1999. ISBN 0201615622.

SUTTER, H. “*Exceptional C++ Style: 40 new engineering puzzles, programming problems, and solutions*”. 1st. ed. [S.l.]: Additon-Wesley Professional, 2005. ISBN 0201760428.

SUTTER, H.; ALEXANDRESCU, A. "*C++ Coding Standards: 101 Rules, Guidelines, and Best Practices (C++ in Depth Series)*". 1st. ed. [S.l.]: Addison-Wesley Professional, 2004. ISBN 0321113586.

YODER, J. W. et al. "Connecting business objects to relational databases". In: PROCEEDINGS OF CONFERENCE ON THE PATTERN LANGUAGES OF PROGRAMS, 5th., 1998, Monticello, IL, USA. 1998. Disponível em: <http://citeseer.ist.psu.edu/yoder98connecting.html>. Acesso em: 2008/07/20.

APÊNDICE A – Soluções já existentes

A.1 NeXT's Enterprise Objects Framework

Being the first object-relational mapping library to achieve some success and recognition, NeXT's EOF came to be as a solution to the need of applying the advantages of object-oriented programming to applications that used relational-based data storage. It is used to map Java and Objective-C [1] objects to relational tables stored in a RDBMS and create medium to large single or multi-tiered applications.

After the merger of NeXT with Apple in 1996, the latter turned EOF into a Java web application framework available for free, as part of its Xcode Developer Tools normally included with Apple's Mac OS X.

EOF was based on an earlier attempt to create an ORM library, called Database Kit (DBKit), in 1992, that wrapped an OO framework around any database backend. Due to serious design flaws, the project was set back and two years later in a second attempt, EOF was finally created.

Among its many features, EOF is able to communicate with several data sources simultaneously. This is useful when some objects should be persisted in one RDBMS, while others should go to another RDBMS, maybe for performance reasons, as commonly used objects might be stored in a faster server with less storage capacity. It is even possible to use this capability to create a high availability solution with mirrored data sources. Due to its layered design, developers don't have to think in terms of data sources or data source vendors with different API. Instead, they manipulate objects (in the OO sense) that can be read from or written to a virtual data source that isolates them from the real DBMS (or many of them).

In EOF, the object-to-relational mapping is done by using a visual tool called EOModeller, where database tables are mapped to classes, columns mapped to class attributes and rows

[1] Objective-C is a programming language derived from C with OOP extensions, being primarily used in Mac OS X and GNUstep environments

to objects (class instances). It is even possible to implement inheritance amongst classes, or associating multiple tables to one class, as when two classes are related in some way.

Objects in EOF are called *Enterprise Objects*, which is another name for *Business Objects*, more commonly known. Instead of being simply objects that can be persisted on a relational database, EOF implements a complete framework around them, involving message passing between objects (through *notifications*), clear separation of business logic from application logic.

EOF implements relationships between objects using the *Lazy Load* pattern. An relationship class manages the loading of related objects semi-automatically, as the user shall execute its member function `willRead` before acessing the objects, or creating `get` methods that call `willRead` before returning the delay loaded object. The same is valid for updating objets. The `willChange` function must be called before changing an attribute.

Relationships between objects, such as $(0:1)$, $(1:1)$, $(0:n)$ and $(1:n)$ are correctly handled internally. But the user must differentiate explicitly between relationships that need an inner join from the ones that need an outer join (left, right or full) between the involved tables. This stems from the fact that there are only two basic relationships cardinalities: to one and to many. Each one may serve as $(0:1)$, $(1:1)$ and $(0:n)$, $(1,n)$ respectively. The difference between then is the choice of joins used. This may be seen as if the underlying relational model somehow has to be taken into account while designing the object model. Furthermore, $(n:n)$ relationships aren't implicitly handled, as well as bidirectional relationships. To create a $(n:n)$ relationship, one must explicitly create an auxiliary table and two $(1:n)$ relationships to the involved tables to realise an $(n:n)$ relationship. It should also be noted that EOF supports reflexive relationships and multiple key relationships, something that cannot be taken for granted in existing ORM libraries.

Unfortunately, the syntax adopted by EOF seems a little bit awkward to beginners. Simple query filters are cumbersome to write, as listing A.1 shows. Notice how simple boolean operations are declared and how parameters are passed to the primitive qualifiers.

Listagem A.1: NeXT's EOF query sample

```
1 // Creates the predicate (qualifier):
2 // agent.lastName in ('Basset','Travers')
3 //   and agent.lastName like 'B%'
4 EOQualifier compoundQualifier = new EOAndQualifier(
5     new NSArray(new Object[]
6     {
```

```
7         EOQualifier.qualifierWithQualifierFormat("agent.lastName=%s",
8             new NSArray(new Object[] {"Basset","Travers"})),
9         EOQualifier.qualifierWithQualifierFormat("agent.lastName
10            caseInsensitiveLike %s" new NSArray(new Object[]{"B*"})
11    }));
```

All in all, EOF is a powerful and highly scalable ORM framework, implementing a complete business objects solution. However it seems to require from the developer non-trivial knowledge of how some use cases must be implemented, often in a verbose manner. Its requirement that the application must be written in Java or Objective-C can be impractical for some applications. The dependence on EOModeller to create the enterprise objects might be seen as an disadvantage because in some contexts a graphical interface might not be available, and usually the generated source files cannot be modified to a great extent without making it unrecognizable to EOModeller when the object's structure should be modified. Those issues might come from the fact that EOF pioneered the niche of ORM solutions, and for this same reason cannot cope with an overhaul in its implementation due to code legacy concerns. As it'll be clear from the following sections, recent libraries picked up where EOF left off and approached the ORM problem from different perspectives, resulting in an easier library to work with.

A.2 Apache Cayenne

Apache Cayenne is an open source solution to be used with Java applications, providing object-relational mapping, object persistence and remoting services that gives web applications access to remotely stored data, completely isolating them of its relational origin.

One of its main features includes the possibility of reverse engineer an relational database schema into a OO model, which comes useful when it isn't practical to migrate an existing database to a new OO-inspired schema. To achieve this, the developer uses a graphical application, the CayenneModeler, to map the relational schema into its OO model. Naturely it is possible to generate a relational schema from an existing OO model, and in this aspect the solution resembles EOF.

Being a recent project, released in July 2002, Cayenne benefits from mature knowledge of ORM issues and tries to be as complete as possible, with some features not easily found in other solutions. According to its documentation, Cayenne implements a complete object query syntax, relationship pre-fetching or on-demand object and relationship fetching (lazy load),

object inheritance, etc. It also implements object caching, thus minimizing data source access and overall performance. The cache is configurable, letting the developer choose among 3 levels of caching. Each one differs on the level of object consistency achieved when several clients are accessing the database.

Central to Cayenne is the `DataContext` entity. To be persisted, objects must be associated with a DataContext that bridges them to the underlying database. Every modification made to the associated objects gets committed when DataContext's member `commitChanges` is invoked. This way a transaction like behaviour is achieved, although one cannot control the granularity of such transaction, as either all objects gets committed or none gets. This only could be achieved by a series of object modification and `commitChanges` invocation.

The code listing A.2 exemplifies the basic usage of Cayenne. It assumes that the object-to-relational mapping is already done with CayenneModeler and the database is created. It can be noted in lines 28 and 32 that it uses a similar approach to EOF's query's filter expression creation, namely the Query Objects pattern, although less verbose.

Listagem A.2: Apache Cayenne

```
1  import org.apache.cayenne.access.DataContext;
2  public class Main
3  {
4      public static void main(String[] args)
5      {
6          // Create a data context (connection to data source)
7          DataContext ctx = DataContext.createDataContext();

9          // Create some objects
10         Artist picasso = (Artist)ctx.newObject(Artist.class);
11         picasso.setName("Pablo Picasso");
12         Gallery metropolitan = (Gallery)ctx.newObject(Gallery.class);
13         metropolitan.setName("Metropolitan");
14         Painting girl = (Painting)ctx.newObject(Painting.class);
15         girl.setName("Girl Reading at a Table");
16         Painting stein = (Painting)ctx.newObject(Painting.class);
17         girl.setName("Gertrude Stein");

19         // Assign paintings to their artists (or vice-versa)
```

```
20          girl.setArtist(picasso); // or picasso.addToPaintings(girl)
21          picasso.addToPaintings(stein); // or stein.setArtist(picasso)

23          ctx.commitChanges(); // save changes to database

25          // Return all paintings
26          List paintings1 = ctx.performQuery(
27              new SelectQuery(Paintings.class));

29          // Return all paintings that begin with 'gi'
30          List paintings2 = ctx.performQuery(
31              new SelectQuery(Painting.class,
32                  ExpressionFactory.likeIgnoreCaseExp(
33                      Painting.NAME_PROPERTY, "gi%")));

35          // Remove the first painting returned from database
36          if(paintings2.size() > 0) {
37              ctx.deleteObject(paintings2.get(0));
38              ctx.commitChanges();
39          }
40      }
41  }
```

One more thing worth mentioning is Cayenne's bidirectional relationships semantics that can be seen in lines 22 and 23. Unlike EOF, here we're able to change either side of the relationship, the other will be updated accordingly. This is important as in the domain level assigning an artist to a painting is conceptually equivalent to assigning a painting to an artist.