

UNIVERSIDADE DE ITAÚNA
FACULDADE DE ENGENHARIA
CIÊNCIA DA COMPUTAÇÃO

MICHAEL DOUGRAS DA SILVA

**ORM4Qt: Biblioteca de Mapeamento Objeto
Relacional em C++ para o Framework Qt**

Itaúna

2014

UNIVERSIDADE DE ITAÚNA
FACULDADE DE ENGENHARIA
CIÊNCIA DA COMPUTAÇÃO

MICHAEL DOUGRAS DA SILVA

ORM4Qt: Biblioteca de Mapeamento Objeto Relacional em C++ para o Framework Qt

Trabalho de Conclusão de Curso apresentado
como requisito parcial para obtenção do tí-
tulo de Bacharel em Ciência da Computação
da Faculdade de Engenharia da Universidade
de Itaúna.

Orientador:
Angélica Moreira

Itaúna

2014

ORM4QT: Biblioteca de Mapeamento Objeto Relacional em C++ para o Framework Qt

Michael Dougras da Silva

Este trabalho foi julgado adequado para obtenção da aprovação na disciplina Trabalho de Conclusão do Curso de Ciência da Computação da Faculdade de Engenharia da Universidade de Itaúna.

Aprovado por:

Beltrano de Tal, Titulação/Instituição
(Orientador)

Sicrano de Tal, Titulação/Instituição
(Co-orientador)

Membro da banca, Titulação/Instituição

Itaúna, 11 de Maio de 2014.

Glossário

<i>k</i> -NN	: <i>k</i> -Nearest Neighbor;
MEME	: <i>Multiple Expectation Maximization for Motif Elicitation</i> ;
MS SQL	: <i>Microsoft Structured Query Language</i> ;
NB	: <i>NaiveBayes</i> ;
RNA	: Redes Neural Artificial;
SVM	: <i>Support Vector Machines</i> ;
UIT	: <i>Universidade de Itaúna, Minas Gerais</i> ;

Sumário

1	Introdução	1
2	Fundamentação Teórica	3
2.1	Orientação a Objetos	3
2.2	A Linguagem de Programação C++	6
2.3	O Framework Qt	7
2.4	Sistemas Gerenciadores de Bancos de Dados Relacionais	9
2.5	O PostgreSQL	11
2.6	A combinação SGBDs com Linguagens Orientadas a Objetos	12
2.7	Bibliotecas de Mapeamento Objeto Relacional (ORM)	13
3	Trabalhos relacionados	16
3.1	Um framework de mapeamento objeto-relacional com um exemplo em C++	16
3.2	QxORM	17
4	A biblioteca ORM4Qt	19
4.1	Arquitetura em camadas	20
4.2	Camada de Objeto	21
4.2.1	Quebrando o encapsulamento das classes	22
4.2.2	Inserindo metadados através de anotações	24
4.3	Camada de Armazenamento	26
	Referências Bibliográficas	27

Lista de Figuras

2.1	Janela no GNU/Linux	8
2.2	Janela no Microsoft Windows XP	9
2.3	Janela no Mac Os	10
2.4	Exemplo de definição de tabelas e relações	11
4.1	Interação entre componentes do software e desenvolvedor	21
4.2	Diagrama simplificado das classes de reflexão	25

Lista de Algoritmos

1	Definição de uma classe simples em C++	5
2	Exemplo de comunicação com SGBD utilizando o módulo QtSql	13
3	Obtendo o nome da classe de um objeto em tempo de execução	22
4	Exemplo de variações na declaração de métodos acessadores para atributos tipo inteiro	23
5	Retornando uma expressão lambda para acesso de atributo privado	24
6	Esboço da utilização de macros para registro de metainformação	26

Capítulo 1

Introdução

Vivemos em uma era onde os sistemas informatizados deixaram de ser vistos como meras ferramentas de automatização de tarefas. Nos últimos tempos, os softwares tem desempenhado papel fundamental no planejamento estratégico e desempenho de atividades nas grandes empresas. A proliferação de **Sistemas de Informação Empresarial** (*Enterprise Information Systems* - EIS) se mostra cada vez mais notória. Estes sistemas são caracterizados por sua alta complexidade e por trabalharem com gerenciamento constante de dados [Lhotka, 2009].

As **Linguagens Orientadas a Objeto** se tornaram um padrão consolidado no desenvolvimento destes sistemas. Devido à sua capacidade de abstrair representações de entidades do mundo real como componentes de software, estas linguagens permitem aos arquitetos e engenheiros de software terem uma visão de alto nível durante o planejamento e especificação do sistema[Nierstrasz, 1989].

Os **Sistemas Gerenciadores de Bancos de Dados Relacionais** ou **SGBDs** são o meio consolidado para armazenamento de dados estruturados. Eles permitem o armazenamento de informações de tal forma que estas possam ser manipuladas através de comandos em linguagem **SQL**. Esta linguagem permite executar operações de inserção, remoção, edição e geração de consultas de alta complexidade nos dados armazenados. [Barnes, 2007].

Apesar de as linguagens orientadas a objeto serem utilizadas em conjunto com os SGBDs no desenvolvimento de sistemas, a forma de representação dos dados nos dois contextos é diferente. No contexto orientado a objetos, as informações são vistas sobre uma perspectiva comportamental, onde os componentes do software são importantes não somente pelos dados que eles contém, mas pela habilidade de se comunicar com outros

componentes para compartilhar estas informações e executar ações sobre elas. Já no contexto dos SGBDs as informações são vistas sobre uma perspectiva estrutural, onde o objetivo principal das entidades é armazenar os dados e representar suas relações da forma mais otimizada possível. Não existe a preocupação em se definir o comportamento das informações.

Devido a essa diferença de representação de dados entre os dois contextos existe a necessidade de conversão ou mapeamento durante a transição entre eles. A geração de código para mapeamento se mostra uma tarefa repetitiva e propensa a erros.

Com o objetivo de otimizar a utilização das linguagens orientadas a objeto em conjunto com os SGBDs surgiu o conceito de **Biblioteca de Mapeamento Objeto Relacional** ou **ORM** (*Object Relational Mapping*). Esta biblioteca tem como objetivo automatizar as tarefas de mapeamento dos dados entre os dois contextos. Sua utilização promove maior produtividade e redução da complexidade envolvida em tarefas de manutenção e modificação do código. Neste trabalho é proposto o desenvolvimento de uma biblioteca ORM para a linguagem C++. O **Framework Qt** é utilizado como auxílio para comunicação com os SGBDs, e extensão dos tipos nativos da linguagem C++.

Capítulo 2

Fundamentação Teórica

Neste capítulo iremos inserir alguns conceitos básicos utilizados ao longo do trabalho, situando-os dentro do problema a ser resolvido.

2.1 Orientação a Objetos

Um conceito bastante difundido tanto no meio acadêmico quanto no mercado são as **Linguagens Orientadas a Objetos**. São linguagens de programação que utilizam um paradigma ou padrão de estruturação de código que permite ao desenvolvedor criar abstrações no contexto do software para modelar objetos ou entidades do mundo real[[Nierstrasz, 1989]].

Não existe uma definição universal que diga quais são as características ou funcionalidades que uma linguagem de programação deve apresentar para ser considerada orientada a objetos [Nierstrasz, 1989]. Porém de maneira geral, elas possuem mecanismos que permitem agrupar dentro de uma unidade de software, estruturas para prover informações de **estado, comportamento e identidade** [Barnes, 2007]. Este agrupamento de estruturas dentro de uma unidade é conhecido como **encapsulamento**.

Esta unidade de software que representa o estado, comportamento e identidade de uma entidade do software é conhecida como **objeto**. Um objeto é criado a partir de um modelo pré-estabelecido que define todas as estruturas internas que o compõe. Este modelo é conhecido como **classe** [Nierstrasz, 1989].

Algumas linguagens não oferecem mecanismos explícitos para criação de classes, porém oferecem suporte à instanciação de objetos. Um exemplo de linguagem que apresenta esta característica é a linguagem **JavaScript**¹. Outras linguagens oferecem mecanismos

¹Linguagem de script dinâmica utilizada em programação para a WEB.

avancados para definição de classes, permitindo por exemplo que uma classe seja composta por objetos de outras classes. Um exemplo de linguagem com esta característica é a linguagem **C++**. O estado de um objeto é definido a partir da adição de variáveis em sua estrutura interna para armazenamento de valores unitários. Estas variáveis são conhecidas como **atributos** ou **propriedades** de uma classe [Nierstrasz, 1989].

Algumas linguagens permitem a adição de objetos como atributos de uma classe, como é o caso da linguagem C++. Este comportamento é conhecido como **composição**. O comportamento de um objeto é definido a partir da adição de funções em sua estrutura interna. Elas são utilizadas para efetuar operações sobre os atributos que o objeto contém. Estas funções são conhecidas como **métodos** de uma classe [Nierstrasz, 1989].

O conjunto de atributos e métodos de uma classe define o que chamamos de **interface**. Algumas linguagens permitem reduzir a interface de um objeto de forma que nem todas as estruturas internas sejam acessíveis externamente. Um exemplo é a linguagem C++, que nos permite utilizar os modificadores **"private"** (estruturas não acessíveis externamente) e **"public"** (estruturas acessíveis externamente). A identidade de um objeto se refere à capacidade de se referenciar instâncias de objetos de maneira unívoca, ou seja, se refere à capacidade de se distinguir diversas instâncias de objetos entre si através de algum mecanismo de comparação [Nierstrasz, 1989].

Em algumas linguagens este mecanismo se baseia no endereçamento de memória ocupado pela instância do objeto, como é o caso da linguagem C++. Já em outras linguagens existem mecanismos que permitem a criação de uma função de comparação pelo desenvolvedor, o que acontece por exemplo na linguagem **JAVA** onde podemos definir o método **"equals"** das classes criadas. Um exemplo de definição de uma classe utilizando a linguagem C++ é apresentado no trecho de código ??, para demonstrar os conceitos explicados anteriormente. No código é definida a estrutura de uma classe que representa uma pessoa. A classe foi definida com o identificador Pessoa (linha 1), possui dois atributos internos para armazenar o nome e sobrenome (linhas 3 e 4) e um método que retorna o nome completo de uma pessoa através da combinação do seu nome e sobrenome (linha 6). Os atributos não são acessíveis externamente devido ao modificador de acesso **"private"** utilizado na linha 2. Já o método é acessível devido ao modificador de acesso **"public"** utilizado na linha 5.

As características citadas anteriormente são as mais comumente encontradas nas linguagens consideradas como orientadas a objeto. Existem mecanismos mais avançados que permitem uma melhoria na definição do comportamento das classes e auxiliam também no

```
1 class Pessoa {  
2     private:  
3         string nome;  
4         string sobrenome;  
5     public:  
6         string nomeCompleto() { return this->nome + this->sobrenome; }  
7 }
```

Algoritmo 1: Definição de uma classe simples em C++

reaproveitamento de código. Algumas destas características são o suporte à **herança** e ao **polimorfismo**. A herança consiste na capacidade de uma classe estender a estrutura de uma classe já definida. Isso significa que a nova classe criada mantém a interface da anterior e tem a capacidade de adicionar novos atributos e métodos a ela. Algumas linguagens oferecem somente suporte para herança simples, onde uma classe pode herdar de somente uma outra classe, porém geralmente definem mecanismos alternativos de extensão. Um exemplo é a linguagem JAVA. Outras linguagens oferecem suporte para herança múltipla, onde uma classe pode estender as funcionalidades de uma ou mais classes já definidas. Um exemplo de linguagem com esta característica é a linguagem C++ [Nierstrasz, 1989].

Quando criamos uma classe "B" que herda de uma classe "A", dizemos que "A" é a classe **base** ou **pai**, enquanto a "B" é uma classe **especializada** ou **filha**. Uma característica importante a observar é que ao declararmos um objeto da classe "B", este objeto poderá ser utilizado em contextos onde é esperado um objeto da classe "A" [Nierstrasz, 1989].

Ao utilizar o mecanismo de herança, algumas linguagens permitem a modificação de um método existente na classe base. Este mecanismo é utilizado para especializar o comportamento herdado na nova classe, mantendo um mesmo padrão de interface. Quando isto acontece temos um impasse a ser resolvido. Quando um objeto de uma classe filha com métodos especializados for utilizado em um contexto como um objeto da classe pai, ao chamar este método, deverá ser executada a implementação da classe pai ou da classe filha? Quando desejamos que a implementação da classe filha seja utilizada, surge o conceito de polimorfismo, onde o comportamento de um objeto é mantido consistente em relação à sua definição ou declaração não importa em qual contexto ele seja utilizado [Nierstrasz, 1989].

Em algumas linguagens como JAVA o polimorfismo é implícito, ou seja, na pergunta anterior se desejássemos que a implementação da classe pai fosse executada, isso não seria possível. Já em outras linguagens como C++, o polimorfismo é definido explicitamente através do uso de palavras-chave da linguagem.

Ao analisar todas estas características citadas, percebemos que as linguagens orientadas a objeto promovem a criação de entidades de software bem descritivas. Isto facilita sua utilização por arquitetos e engenheiros de software, que podem visualizar os elementos envolvidos no desenvolvimento com uma visão de mais alto nível. Talvez esta seja a razão da vasta aceitação e utilização destas linguagens. Neste trabalho é utilizada a linguagem orientada a objetos C++, a qual é detalhada na seção 2.2.

2.2 A Linguagem de Programação C++

A linguagem C++ é uma linguagem orientada a objetos criada em 1980 por *Bjarne Stroustrup*. É uma linguagem compilada, ou seja, o código criado pelo desenvolvedor é convertido através de um programa denominado **compilador** para uma linguagem nativa de uma plataforma alvo, gerando um ou mais arquivos executáveis. Estes arquivos são passíveis de execução direta na plataforma sem a intervenção de ferramentas externas [Bueno, 2002].

As linguagens compiladas se caracterizam por gerar aplicativos com melhor desempenho em momento de execução. Isso se deve ao fato da geração de código nativo, que pode ser executado sem intervenção externa [Bueno, 2002]. Porém esta característica também implica em a linguagem oferecer poucos mecanismos de **reflexão** ou **introspecção**, que nos permitem conhecer em momento de execução as estruturas dos objetos sendo utilizados [Barnes, 2007].

C++ foi construída com base na linguagem C. Inicialmente o código escrito em C++ era traduzido para C e compilado com compiladores desta linguagem. Com o aumento da complexidade de implementação das características e funcionalidades que foram surgindo na linguagem ao longo de sua existência, surgiram compiladores específicos para C++ [Brokken and Kubat, 2008].

Dentre as características da linguagem estão a capacidade de definição de classes, utilização de herança múltipla, utilização de polimorfismo explicitamente, gerenciamento de memória controlado pelo desenvolvedor e compatibilidade com códigos escritos em C. Outra importante característica da linguagem é a sua não portabilidade, ou seja, o código gerado para uma plataforma (sistema operacional e/ou hardware específico) geralmente não é compatível com outras plataformas. Devido a isso, a linguagem possui uma biblioteca padrão bem reduzida, não oferecendo por exemplo bibliotecas de comunicação em rede e utilização de banco de dados [Bueno, 2002].

Existe uma grande dificuldade em se desenvolver programas voltados para diversas plataformas utilizando a linguagem, pois quando começamos a utilizar funcionalidades um pouco mais avançadas temos que utilizar implementações específicas para cada plataforma. Como tentativa de diminuir esta dificuldade foram criadas bibliotecas e *frameworks*² multiplataformas para a linguagem [Thelin, 2007]. Neste trabalho é utilizado o **Framework Qt** que será descrito na seção 2.3.

2.3 O Framework Qt

Qt é um framework multiplataforma voltado para a criação de aplicativos com interface gráfica utilizando a linguagem C++. Sua primeira versão foi lançada em 1995 pelos seus criadores *Haavard Nord* e *Eirik Chambe-Eng* em sua empresa chamada *Trolltech*[Blanchette and Summerfield, 2006].

O objetivo do framework é basicamente prover uma interface de programação padrão para executar tarefas como criação de interface gráfica, programação paralela e acesso a banco de dados, para todas as plataformas suportadas. Esse comportamento é alcançado através do direcionamento da execução das tarefas para implementações específicas existentes na plataforma alvo [Blanchette and Summerfield, 2006]. Uma característica interessante do framework é a adaptação da interface gráfica criada ao seu ambiente de execução. Uma janela executada em um sistema operacional suportado assume a aparência nativa do mesmo, como é possível verificar nas figuras 2.1, 2.2 e 2.3, onde temos uma janela de um programa sendo exibida nos três sistemas suportados da plataforma desktop.

Inicialmente o framework era totalmente focado na padronização do processo de criação de interface gráfica entre diferentes plataformas, porém ao longo de sua existência foram sendo adicionadas novas funções voltadas para tarefas rotineiras no desenvolvimento de software, como por exemplo acesso a banco de dados, programação paralela e manipulação de arquivos multimídia (vídeos e imagens). Com a adição destes novos componentes o framework acabou crescendo muito, o que levou aos desenvolvedores o reestruturarem em forma de módulos. Dessa maneira o desenvolvedor pode adicionar em seu projeto somente os módulos que pretende utilizar, diminuindo o tamanho final do executável gerado para distribuição de seu aplicativo [Blanchette and Summerfield, 2006].

²Um conjunto de bibliotecas que implementam funcionalidades frequentemente utilizadas no desenvolvimento de softwares, como por exemplo, acesso a banco de dados e comunicação em redes.



Figura 2.1: Janela no GNU/Linux

Neste trabalho são utilizados basicamente três dos módulos existentes no framework, sendo eles o **QtSql**, o **QtTest** e o **QtCore**. O **QtSql** é um módulo voltado para utilização de **Sistemas Gerenciadores de Bancos de Dados Relacionais** (ver seção 2.4) como forma de persistência dos dados gerados pelo aplicativo. Ele provê suporte para utilização dos sistemas mais populares, como **MySQL**, **Oracle**, **PostgreSQL**, **Sybase**, **DB2**, **SQLite**, **Interbase**, e **ODBC** [Thelin, 2007]. Esse módulo será utilizado como apoio para utilização de bancos de dados relacionais em diferentes plataformas de sistema operacional.

O **QtTest** é um módulo voltado para a criação de testes unitários. Os testes são utilizados para validar o funcionamento do software desenvolvido ao longo de sua existência. Geralmente são executados quando é feita alguma modificação no código, como por exemplo adição de novos componentes. **QtCore** é o módulo principal do framework. Ele é responsável por prover rotinas para programação paralela, conversão de tipos, além de adicionar tipos de dados muito importantes na utilização de bancos de dados, como o **QDateTime**³ [Thelin, 2007].

³Tipo de dado que armazena um valor de data e hora. Não existe um tipo de dado nativo em C++ para esta finalidade.

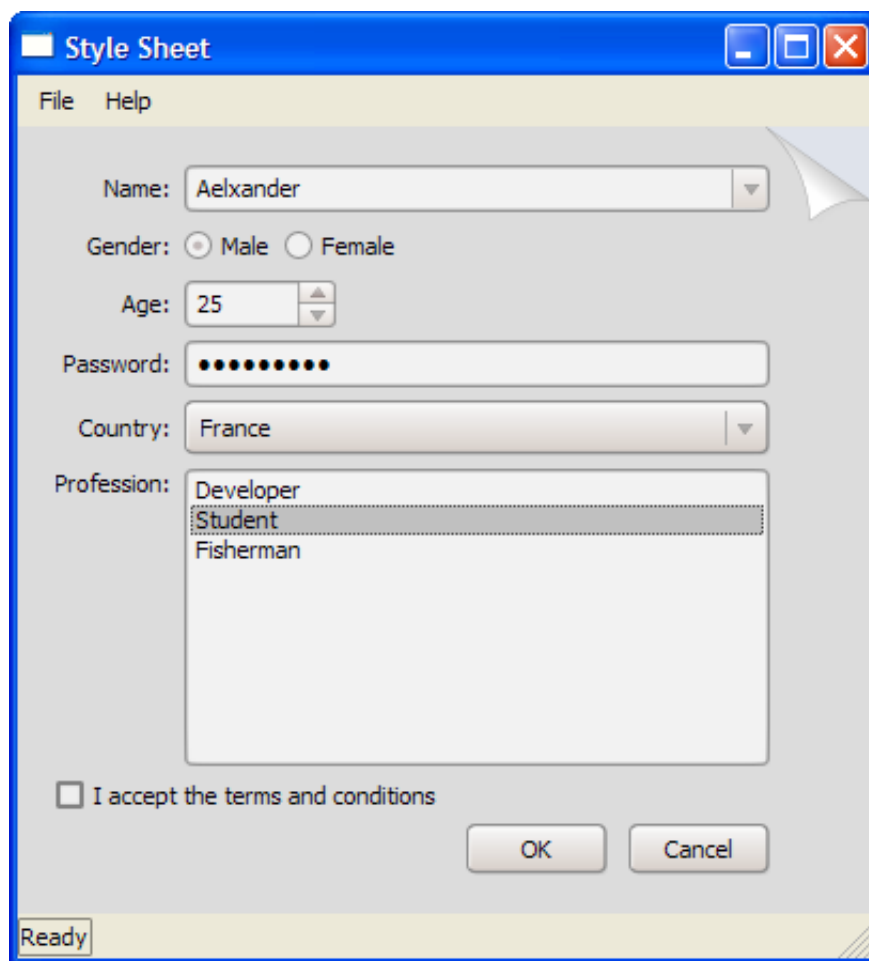


Figura 2.2: Janela no Microsoft Windows XP

No momento da criação deste documento, o framework está em sua versão 5.3.0, e é compatível com os sistemas operacionais *GNU/Linux*, *Microsoft Windows* e *Mac OS* na plataforma *desktop*. Ele oferece também suporte a dispositivos embarcados com sistema operacional *GNU/Linux* e para dispositivos móveis com os sistemas *Android*, *IOS* e *Windows Phone*.

2.4 Sistemas Gerenciadores de Bancos de Dados Relacionais

Assim como as linguagens orientadas a objeto (ver seção 2.1) são consideradas um padrão para desenvolvimento de software, os Sistemas Gerenciadores de Bancos de Dados Relacionais ou **SGBDs** são considerados um padrão para armazenamento estruturado de informações [Barnes, 2007].

Os SGBDs utilizam um modelo de persistência denominado **modelo relacional**,

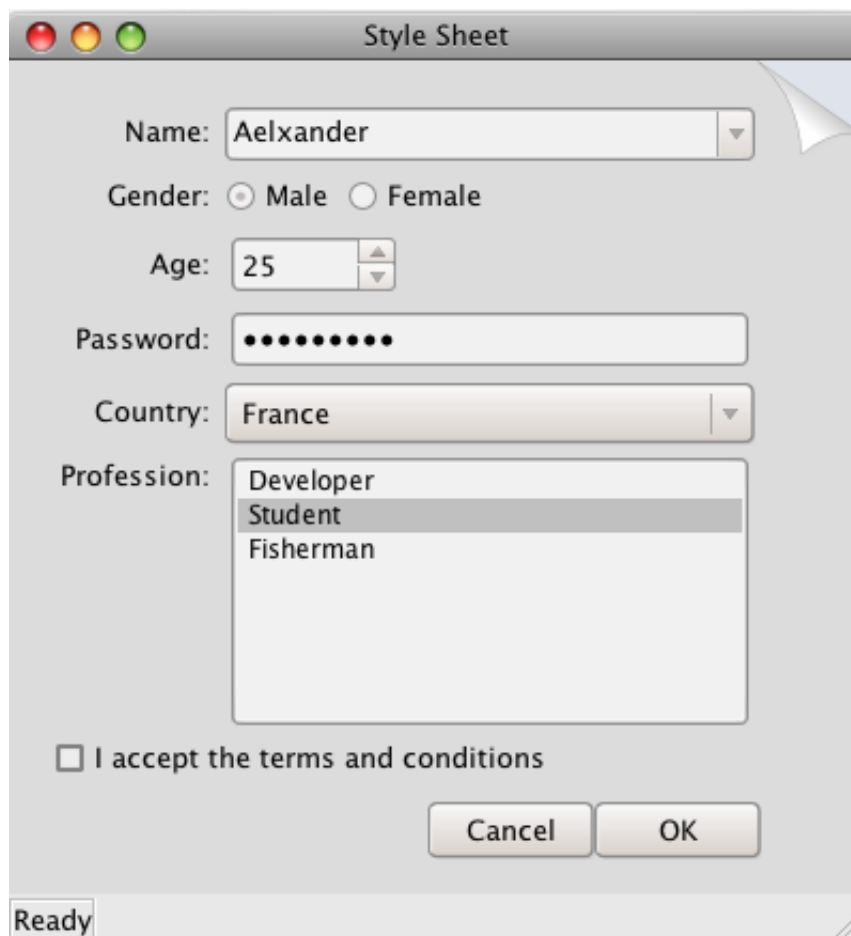


Figura 2.3: Janela no Mac Os

que se baseia no uso de **tabelas** e **colunas**. As tabelas representam entidades, ou um grupo de informação a ser armazenado, e podem se relacionar com outras tabelas para promover regras de armazenamento. As colunas representam as características da entidade armazenada. Neste modelo quando armazenamos um registro ele é acrescentado como uma linha em uma ou mais tabelas relacionadas. Nas tabelas é aplicado o conceito de unicidade dos registros armazenados, através da definição de um conjunto de colunas que deve representar uma combinação de valores única dentre os registros armazenados. Este conjunto de colunas define o que chamamos de **chave primária** da tabela ou "*primary key*" [Barnes, 2007].

A definição de chaves primárias nas tabelas criadas permite a criação de relações entre elas através do uso de referências entre suas chaves. A referência em uma tabela para uma chave primária de outra tabela é conhecida como **chave estrangeira** ou "*foreign key*" [Barnes, 2007]. Na figura 2.4 temos um exemplo de definição de duas tabelas, uma representando registros de pessoas e outra de telefones. Elas estão relacionadas, de modo que um registro de telefone pertence a uma pessoa. Este comportamento é obtido através

da definição da chave estrangeira "TelefoneId" na tabela "Pessoa", que se refere à chave "Id" na tabela "Telefone".

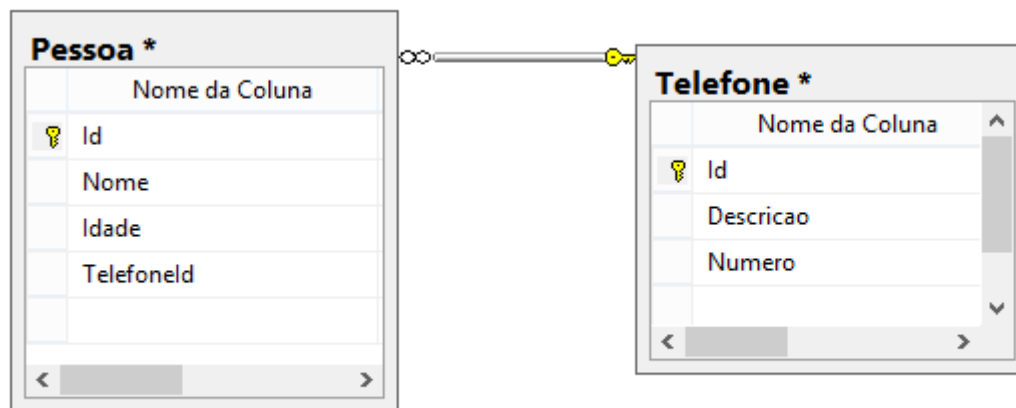


Figura 2.4: Exemplo de definição de tabelas e relações

Uma das funcionalidades mais importantes apresentada pelos SGBDs é a capacidade de aplicação de rotinas complexas de pesquisa usando a linguagem **SQL** ou **Structured Query Language**. Esta linguagem define um conjunto de operações que permite executar inserções, modificações, remoção e busca de registros armazenados em um SGBD [Barnes, 2007]. Talvez esta funcionalidade seja a causa de sua grande utilização e aceitação. Neste trabalho é utilizado o SGBD **PostgreSQL** (ver seção 2.5).

2.5 O PostgreSQL

O PostgreSQL é um SGBD de código aberto, e com licença de uso gratuita para qualquer pessoa sobre qualquer propósito. Ele pode ser utilizado, modificado e redistribuído livremente. Este SGBD é derivado do projeto **POSTGRES** desenvolvido na Universidade de Berkeley na Califórnia no ano de 1986. Por ser um software de código aberto desenvolvido em comunidade, ele é conhecido por ser pioneiro na adição de novas funcionalidades, e se mantém sempre atualizado em relação aos conceitos e especificações da linguagem SQL [PostgreSQL, 2014].

2.6 A combinação SGBDs com Linguagens Orientadas a Objetos

Os programas trabalham com dados em memória principal, que é volátil. Portanto ao desligarmos o equipamento todos os dados são perdidos. Devido a isso necessitamos de algum mecanismo de persistência, ou seja, que permita a gravação de dados importantes em memória secundária não volátil [Barnes, 2007].

A combinação mais comumente utilizada para alcançar este cenário é o uso de linguagens orientadas a objetos para desenvolver o software, e o uso de SGBDs para armazenar os dados gerados pelo software [Bauer and King, 2005]. Porém os dados no contexto orientado a objetos e no contexto relacional são estruturados de maneira diferente (ver seções 2.1 e 2.4), portanto precisamos realizar uma conversão ou mapeamento dos dados durante a transição de contextos.

Quando estamos trabalhando com classes simples (não compostas por membros de outras classes), o mapeamento segue uma lógica simples. Podemos criar uma correspondência entre a classe para uma tabela, onde os atributos da classe são mapeados para colunas de uma tabela. Porém quando começamos a utilizar mecanismos mais avançados de orientação a objetos como herança e composição, o mapeamento entre os contextos se torna mais complexo [Barnes, 2007].

Mesmo quando temos um cenário onde o mapeamento é simples, precisamos gerar código de conversão. As linguagens de programação fornecem bibliotecas ou *APIs*⁴ que permitem a comunicação com SGBDs através do envio de comandos em linguagem SQL [Barnes, 2007]. O código gerado para executar a transição de dados entre os dois contextos geralmente segue a seguinte sequência de passos:

1. Carregar um driver de comunicação com o SGBD utilizado e abrir a conexão;
2. Criar um objeto que permita a montagem de código SQL;
3. Enviar o comando para o SGBD para que seja executado;
4. Recuperar e processar os dados ou resposta gerados pelo SGBD;
5. Liberar os recursos alocados para execução da tarefa, como por exemplo fechar a conexão com SGBD.

⁴Application Programming Interface. Define uma interface de um conjunto de bibliotecas de software.

No trecho de código 2 temos um esboço de como estas etapas são realizadas utilizando a linguagem C++ em conjunto com o módulo QSql do framework Qt (ver seção 2.3) e comunicando com o SGBD PostgreSQL (ver seção 2.5).

```
1 QSqlDatabase database = QSqlDatabase::addDatabase("QPSQL"); //Etapa (1)
2 database.open(); //Etapa (1)
3 QSqlQuery query("comando sql"); //Etapa (2)
4 query.exec(); //Etapa (3)
5 /** Processa o resultado retornado, etapa (4) **/
6 database.close(); //Etapa (5)
```

Algoritmo 2: Exemplo de comunicação com SGBD utilizando o módulo QSql

A utilização deste tipo de código se torna repetitiva, visto que temos que executar estes passos para todas as classes que armazenam dados utilizando SGBDs. Este tipo de comportamento é indesejado, pois diminui a produtividade do desenvolvimento, e tarefas repetitivas tem grande potencial de gerarem erros. Outro grande problema que encontramos é que os comandos em linguagem SQL variam entre os SGBDs, portanto ao mudar o SGBD utilizado pelo programa, temos de reescrever os comandos SQL utilizados. Este é outro fator com grande potencial de geração de erros, além de diminuir a flexibilidade do software. Em alguns casos a mudança de SGBD pode se tornar inviável [Barnes, 2007].

Para otimizar a combinação do uso de linguagens de programação orientadas a objeto em conjunto com SGBDs, surgiu o conceito de **Biblioteca de Mapeamento Objeto Relacional** ou **ORM** (*Object Relational Mapping*). Este assunto é abordado na seção 2.7.

2.7 Bibliotecas de Mapeamento Objeto Relacional (ORM)

As bibliotecas ORM têm como objetivo principal automatizar as tarefas relacionadas com a transição de informações entre os contextos orientado a objetos e o relacional. Não existe um conceito universal que defina quais são as funcionalidades que uma biblioteca ORM deve oferecer [Barnes, 2007], porém as três funcionalidades mais comumente encontradas são:

Definição de mapeamento As bibliotecas ORM permitem a definição explícita da correspondência entre entidades no contexto orientado a objetos (classes) e entidades no contexto relacional (tabelas). Os mecanismos de definição deste mapeamento variam entre as implementações;

Geração de banco de dados As bibliotecas ORM geralmente permitem a geração automática do banco de dados equivalente às estruturas do contexto orientado a objetos;

Definição de uma API Geralmente as bibliotecas ORM disponibilizam interfaces padronizadas para realização de tarefas que envolvam a transição de dados entre os dois contextos. Operações como salvar, deletar ou pesquisar registros são encapsuladas em classes de propósito geral, ou seja, que conseguem trabalhar sobre qualquer entidade mapeada.

Nem todas as bibliotecas existentes oferecem as três funcionalidades descritas, mas o mais comum é encontrar uma combinação das três. Existem ainda bibliotecas que permitem gerar o código de mapeamento e até mesmo gerar o código das classes a partir da análise de um banco de dados existente. Esta funcionalidade é conhecida como **Mapeamento Reverso** ou **Engenharia Reversa** [Barnes, 2007].

As bibliotecas ORM divergem bastante na forma de implementação e funcionalidades disponibilizadas. Isso se deve principalmente ao fato da divergência de recursos entre as próprias linguagens de programação em que são implementadas. Porém uma decisão comum a ser tomada no desenvolvimento de uma biblioteca ORM em qualquer linguagem, é a definição do ponto de partida para obtenção das informações de mapeamento [Barnes, 2007]. Os paradigmas mais utilizados são:

Orientado a metadados Neste paradigma, o desenvolvedor informa a estrutura das entidades envolvidas no software que devem ser mapeadas através de alguma fonte externa, como um arquivo XML. Neste arquivo são informados metadados sobre as entidades nos dois contextos, o que permite a biblioteca ORM criar o código de definição das classes e do código de mapeamento. Neste modelo podemos utilizar um banco de dados já existente, ou a biblioteca ORM pode construí-lo a partir da análise dos metadados. A vantagem deste paradigma é a abstração total da geração de código de manipulação do banco de dados, e sua grande desvantagem é a limitação de edição do código gerado pela biblioteca. Os códigos das classes mapeadas serão gerados automaticamente pela biblioteca e não poderão ser editados pelo desenvolvedor [Barnes, 2007].

Orientado a aplicação Neste paradigma o desenvolvedor deve escrever o código das classes de todo o programa normalmente, porém sem se preocupar com as operações de persistência. Após a definição das classes, a biblioteca ORM através da análise

do código e opcionalmente metadados, consegue criar o código de persistência. É possível utilizar um banco de dados existente ou a biblioteca pode construí-lo. A vantagem deste paradigma é a total abstração da geração de código de manipulação do banco de dados, e sua grande desvantagem é a alta complexidade de desenvolver o mecanismo de análise do código escrito pelo desenvolvedor para inferir informações de mapeamento. A definição deste mecanismo pode influenciar bastante no desempenho final da aplicação que utiliza a biblioteca ORM [Barnes, 2007].

Orientado ao banco de dados Neste paradigma o desenvolvedor deve primeiramente criar o banco de dados. Então a biblioteca ORM auxiliada por metadados consegue gerar o código das classes a serem mapeadas e das operações de persistência. A grande desvantagem deste paradigma é a não abstração do conhecimento de banco de dados, visto que o desenvolvedor deve primeiramente definir a estrutura da base de dados a ser utilizada. Outra desvantagem é o desenvolvedor não ter a permissão de modificar o código das classes mapeadas pela biblioteca ORM. Sua vantagem consiste no melhor controle da definição da estrutura da base de dados sendo utilizada [Barnes, 2007].

As bibliotecas de mapeamento mais robustas possuem a possibilidade de operar nos três paradigmas citados. Em alguns casos, elas ainda permitem que o desenvolvedor tenha controle dos três componentes principais citados pelos paradigmas, ou seja, o desenvolvedor pode definir o código das classes a serem mapeadas, os metadados e construir o banco de dados a ser utilizado, deixando para a biblioteca ORM somente a tarefa de adicionar o código de persistência [Barnes, 2007].

Existe ainda o conceito de **transparência**, que é aplicado a bibliotecas que utilizam o paradigma orientado a aplicação. Uma biblioteca ORM é transparente quando não requer que classes implementem interfaces, ou sigam um modelo específico de código para serem mapeadas [Barnes, 2007]. Estas bibliotecas seguem a filosofia de que as classes envolvidas no software não precisam saber como, porque ou quando elas serão persistidas, ou seja, elas devem se comportar como "classes ordinárias" ou comuns.

Analisando as principais funcionalidades que as bibliotecas ORM possuem, podemos perceber que elas promovem uma melhoria considerável de produtividade (economia de tempo de desenvolvimento) além de diminuir a complexidade do desenvolvimento de sistemas que lidam com gerenciamento constante de dados persistidos em bancos de dados relacionais [Barnes, 2007].

Capítulo 3

Trabalhos relacionados

Neste capítulo será descrito um trabalho semelhante a este, que também propôs o desenvolvimento de uma biblioteca ORM para C++ e que é utilizado como base para o desenvolvimento deste trabalho. Também será descrita uma biblioteca ORM presente no mercado, que é bastante utilizada em conjunto com o framework Qt no desenvolvimento de aplicativos comerciais. Esta biblioteca será utilizada na realização de testes comparativos ao final do trabalho.

3.1 Um framework de mapeamento objeto-relacional com um exemplo em C++

Em seu trabalho intitulado "Um framework de mapeamento objeto-relacional com um exemplo em C++", *Zhang Xiaobing* apresenta um modelo para desenvolvimento de uma biblioteca ORM que pode ser utilizado na maioria das linguagens orientadas a objeto. Em seu modelo, ele define padrões a serem utilizados para resolver problemas como mapeamento de classes simples, herança, composição, associação, e ainda define mecanismos de otimização como criação de um cache de objetos resgatados do banco de dados [Zhang, 2004].

Seu modelo define uma biblioteca ORM que segue o paradigma orientado a aplicação, onde o desenvolvedor ficará responsável por definir toda a parte do código relacionado a criação dos objetos a serem mapeados. Todo objeto a ser mapeado deve herdar de uma classe específica, e o desenvolvedor deve reimplementar determinados métodos de modo a informar para a biblioteca metadados que definem o mapeamento. Devido a esse comportamento dizemos que a biblioteca não é transparente. O modelo ainda define uma arquitetura de desenvolvimento em duas camadas:

Camada de objetos (*Object Layer*) Camada responsável por prover uma interface simples para definição de classes a serem mapeadas e seus metadados, além de trafegar dados entre os objetos mapeados e a camada de persistência. Esta camada é a única acessível diretamente pelo desenvolvedor.

Camada de persistência (*Storage Layer*) Camada responsável por abstrair a comunicação com o SGBD, provendo rotinas de persistência, concorrência, recuperação de erros e execuções de pesquisas no banco de dados. Esta camada não é diretamente acessível pelo desenvolvedor.

O modelo definido pode ser aplicado em diversas linguagens pelo fato de não se aproveitar de recursos específicos de determinadas linguagens, porém devido a isso ele exige um trabalho adicional do desenvolvedor ao reimplementar métodos de diversas interfaces definidas. Em alguns momentos, o desenvolvedor deve explicitamente executar funções de consulta e ajuste de valores de atributos em suas classes trocando informações com o framework para permitir a execução de tarefas no banco de dados [Zhang, 2004]. Isso ocorre devido a biblioteca não definir um mecanismo de listagem e acesso às estruturas internas dos objetos mapeados.

Neste trabalho o modelo de Zhang é utilizado como base para a implementação de uma biblioteca ORM para C++, porém com alguns ajustes como a definição de uma interface de anotações para definições de mapeamento, e a capacidade de mapeamento de classes arbitrárias, sem a necessidade de herança de uma classe específica (biblioteca transparente).

3.2 QxORM

A QxORM é uma biblioteca ORM de código aberto desenvolvida para ser utilizada em conjunto com o framework Qt. Ela utiliza vários módulos do framework para auxiliar as tarefas de mapeamento, como por exemplo o módulo QSql para realizar interações com os SGBDs. É uma biblioteca que segue o paradigma orientado a aplicação, e é transparente, ou seja, permite o mapeamento de classes arbitrárias. Para mapear uma classe, o desenvolvedor deve inserir algumas marcações na definição da classe, e implementar um método utilizando funções específicas para registros de informações das classes e atributos a serem mapeados. Portanto, nesta biblioteca também não existe um sistema de anotações [Marty, 2014].

Esta biblioteca será utilizada durante testes de usabilidade, onde será comparado a abordagem de especificação de mapeamento dela com a interface de anotações criada neste trabalho. Além de testes de usabilidade, também serão feitos testes de desempenho em relação ao uso de memória e tempo de resposta.

Capítulo 4

A biblioteca ORM4Qt

Desenvolver aplicativos multiplataforma utilizando a linguagem C++ é uma tarefa muito difícil, devido à reduzida biblioteca padrão da linguagem, e aos diversos componentes específicos desenvolvidos para cada plataforma. Esta complexidade pode ser minimizada a partir do uso de *frameworks* multiplataforma, como o Qt, para compor nosso ambiente de desenvolvimento. Porém, em comparação com ambientes oferecidos por linguagens mais recentes, este apresenta poucos mecanismos de automatização de tarefas diversas, ou os que existem apresentam interfaces complexas para utilização. É o que acontece por exemplo com as bibliotecas de mapeamento objeto relacional ou ORM.

Analisando o ambiente de desenvolvimento oferecido a partir da combinação da linguagem C++ com o *framework* Qt, encontramos algumas implementações de bibliotecas ORM, destacando-se o QxOrm. Estas bibliotecas, devido ao pouco suporte a reflexão oferecido pela linguagem C++, em sua maioria apresentam interfaces de configuração complexas. Além de usarem mecanismos como herança e "**classes friend**"¹ para quebra de encapsulamento das classes a serem mapeadas, o que é indesejável por aumentar o nível de acoplamento do código.

Neste trabalho é proposto o desenvolvimento de uma biblioteca ORM intitulada ORM4Qt para ser utilizada neste ambiente de desenvolvimento. A biblioteca utilizará o paradigma orientado a aplicação e a abordagem transparente para definição das classes mapeadas. A interface de configuração do mapeamento será feita através de um mecanismo de anotações desenvolvido especificamente para a biblioteca. Para a quebra de encapsulamento das classes será utilizada a manipulação de ponteiros de funções através do uso de estruturas de alto nível oferecidos pela linguagem C++. A biblioteca desenvol-

¹Este recurso permite que uma classe ou método global externo acesse os componentes privados de uma classe.

vida será capaz de mapear somente classes simples, ou seja, que contém somente atributos escalares e não utilize herança. Posteriormente, ela poderá ser estendida para suportar o mapeamento de classes que utilizem mecanismos mais avançados de orientação a objetos.

Das bibliotecas ORM existentes para o cenário abordado, a QxOrm é a que mais se aproxima das características citadas, portanto ela será utilizada em testes ao final do desenvolvimento. Os testes deverão comparar a configuração do ambiente de desenvolvimento para utilização das bibliotecas, a facilidade em utilização dos mecanismos de configuração de mapeamento e a facilidade em migração de código legado.

Nas próximas seções serão detalhados os mecanismos utilizados para o desenvolvimento, bem como a arquitetura utilizada.

4.1 Arquitetura em camadas

O desenvolvimento da biblioteca é estruturado em duas camadas, a camada de Objeto ou "*Object Layer*" e a camada de Armazenamento ou "*Storage Layer*", seguindo a nomenclatura utilizada no trabalho desenvolvido por *Zhang Xiaobing* [Zhang, 2004]. As duas camadas oferecem interfaces acessíveis diretamente pelo desenvolvedor e cooperam entre si através de troca de informações.

A camada de Objeto tem como objetivo prover uma interface transparente para o desenvolvedor que permita a configuração das classes a serem mapeadas e prover uma interface para a camada de Armazenamento que permita o acesso aos metadados bem como à estrutura interna das classes sendo mapeadas. Esta camada é a mais complexa de ser desenvolvida devido ao uso intenso de estruturas de baixo nível da linguagem para quebra de encapsulamento e criação do mecanismo de anotações.

A camada de Armazenamento tem como objetivo prover uma interface para o desenvolvedor que permita executar tarefas relacionadas com a persistência de objetos no banco de dados, além de definir uma interface comum de geração de código SQL que possa ser implementada para diferentes SGBDs. Inicialmente esta interface será implementada para o SGBD PostgreSQL, e utilizará os mecanismos oferecidos pelo módulo QtSql para se comunicar com ele.

Na imagem 4.1 temos uma representação de alto nível da interação entre os módulos, o desenvolvedor, o banco de dados e as classes a serem mapeadas durante o funcionamento da biblioteca. Nas próximas seções as duas camadas serão detalhadas juntamente com os

mecanismos específicos envolvidos no desenvolvimento de cada uma.

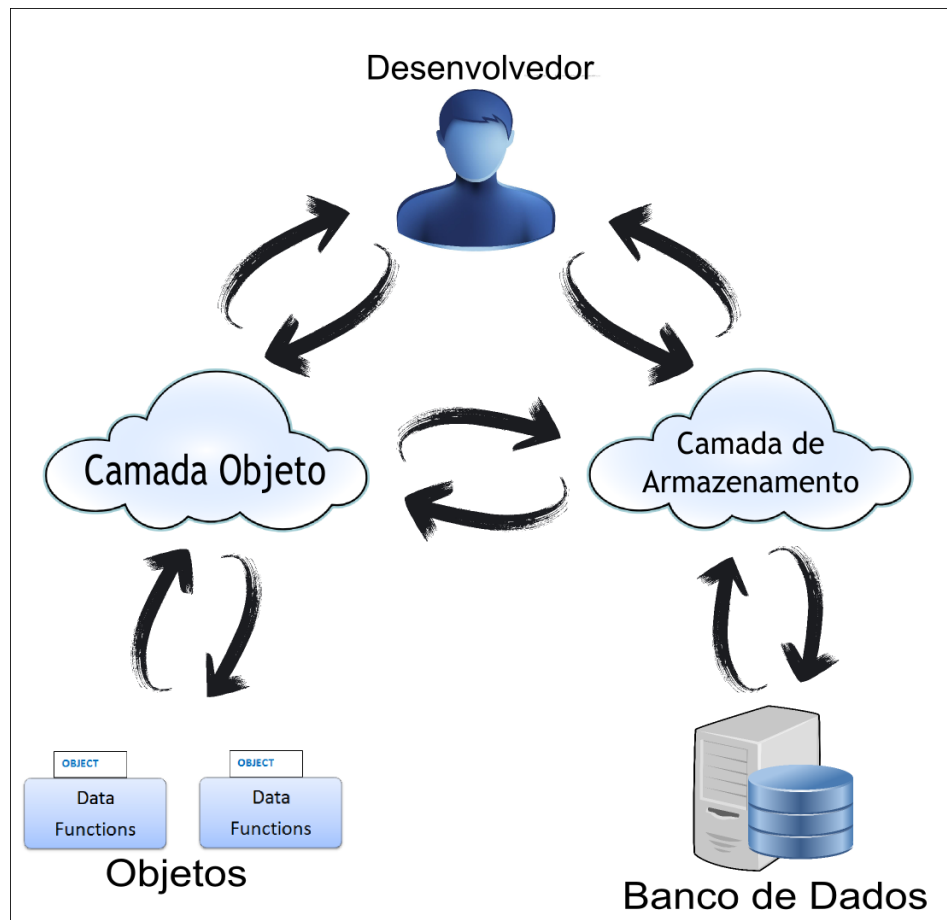


Figura 4.1: Interação entre componentes do software e desenvolvedor

4.2 Camada de Objeto

Para que seja possível realizar o mapeamento, a biblioteca ORM deve ser capaz de conhecer e acessar a estrutura interna das classes a serem mapeadas. Existem basicamente dois limitadores que dificultam alcançar este cenário na linguagem C++. O primeiro deles consiste na possibilidade de o desenvolvedor limitar o acesso à estrutura interna através das diretivas de proteção oferecidas pela linguagem durante a definição das classes. A biblioteca ORM precisa ter acesso de leitura e escrita nos atributos das classes sendo mapeadas, porém em geral eles são mantidos com permissão de acesso privado, onde somente podem ser acessados diretamente de dentro da classe.

O segundo deles é o baixo suporte da linguagem a mecanismos de reflexão. Antes de acessar os atributos das classes a biblioteca deve saber quais são os atributos que a classe contém, entretanto em seu atual estado, a linguagem oferece somente informações básicas

sobre os objetos, como por exemplo o nome de sua classe. No trecho de código 3 temos um exemplo da obtenção do nome da classe de um objeto em tempo de execução. Na linha 1 é criado um objeto da classe Pessoa, e na linha 2 é capturado o nome da classe deste objeto e exibido no console. A saída gerada por este programa quando compilado no compilador que acompanha o Visual Studio 2013 é o texto *"class Pessoa"*.

```
1 Pessoa *obj = new Pessoa();  
2 std::cout << typeid(*obj).name() << std::endl;
```

Algoritmo 3: Obtendo o nome da classe de um objeto em tempo de execução

A camada Objeto utiliza dois mecanismos para contornar estes limitadores, os quais serão descritos nas seções a seguir.

4.2.1 Quebrando o encapsulamento das classes

Quando os atributos das classes são declarados com acesso público, a biblioteca ORM pode acessá-los diretamente, porém este cenário não é normalmente utilizado pelos desenvolvedores e a utilização da biblioteca impor tal cenário é uma característica indesejável e que poderia diminuir sua aceitação no mercado. Para resolver este problema então partimos do pressuposto de que todos os atributos a serem acessados nas classes a serem mapeadas estarão com acesso privado, ou seja, só podem ser acessados de dentro das classes. Com isso em mente podemos também definir que somente poderemos acessar os atributos das classes através do uso de um intermediador que componha a estrutura da classe, ou mais precisamente um método que compõe a interface da classe.

A primeira ideia que vem em mente é utilizar métodos acessadores (popularmente conhecidos como métodos *"get"* e *"set"*) criados pelos desenvolvedores, porém temos alguns limitadores que dificultam a sua utilização. Um deles é que não podemos assumir que para todo atributo existem métodos acessadores, pois podem existir atributos somente leitura ou cujo valor é controlado internamente na classe. Outro limitador é a não padronização do protótipo dos métodos acessadores. Estes métodos podem ser definidos com uma quantidade variável de parâmetros, e ainda a linguagem C++ permite a criação de variações através da modificação do tipo de parâmetro e/ou retorno (ponteiro, referência ou por valor), além do uso do modificador *"const"*² na declaração de métodos de leitura.

²Métodos declarados com este modificador não podem modificar o valor dos atributos da classe durante sua execução. Esta especificação permite ao compilador efetuar otimizações no código gerado.

Devido a estas características, o uso de ponteiros genéricos para métodos, por exemplo, não poderia ser utilizado, pois teríamos que variar a definição dos ponteiros de acordo com o protótipo dos métodos utilizados. No trecho de código 4 temos por exemplo as possíveis variações de declarações de métodos acessadores para um atributo do tipo inteiro.

```
1 //Métodos get
2 int get();
3 int *get();
4 int &get();
5 int get() const;
6 int *get() const;
7 int &get() const;
8 //Métodos set
9 void set(int valor);
10 void set(int &valor);
11 void set(int *valor);
12 void set(const int valor);
13 void set(const int &valor);
14 void set(const int *valor);
```

Algoritmo 4: Exemplo de variações na declaração de métodos acessadores para atributos tipo inteiro

Já que não podemos utilizar os métodos acessadores, a solução proposta é a inserção de métodos intermediadores na definição das classes a serem mapeadas. Dessa maneira podemos criar os métodos seguindo protótipos pré-definidos, o que nos permite manipulá-los mais facilmente através de ponteiros. Porém esta solução ainda tem um problema. Se formos criar um método para cada atributo da classe, a quantidade destes pode se tornar muito grande, o que causaria uma modificação extrema da interface original da classe mapeada, o que é indesejável. Para diminuir os efeitos deste problema é proposto a utilização de **expressões lambda**.

As expressões lambda são estruturas que permitem a criação de métodos anônimos, ou seja, que não têm um nome ou marcador de referência, o que implica em eles não fazerem parte de interfaces de classes ou até mesmo do escopo global. Estas estruturas são manipuladas de maneira semelhante aos ponteiros de funções, porém possuem um tipo de dado padrão para seu armazenamento, o *"std::function"*. Desta maneira podemos inserir somente um método na classe a ser mapeada e dentro deste criar expressões lambda para manipular os atributos. As expressões criadas podem ser então agrupadas em uma estrutura de lista e retornadas. Como as expressões foram criadas dentro da classe sendo manipulada, elas têm acesso aos atributos privados normalmente, além de poderem ser transportadas como variáveis comuns.

No trecho de código 5 temos um exemplo de uma classe com um atributo privado

```
1 class Class
2 {
3     public:
4         std::function<int ()> getLambda()
5         {
6             return [&]() -> int { return this->valor; };
7         }
8     private:
9         int valor;
10 };
```

Algoritmo 5: Retornando uma expressão lambda para acesso de atributo privado

do tipo inteiro, e uma função que retorna uma expressão lambda capaz de acessar este atributo. Na linha 4 temos a definição do método que retorna a expressão lambda e na linha 6 a sua criação e retorno. A sintaxe de criação de expressões lambda pode parecer estranha inicialmente, porém com o decorrer do seu uso ela se torna prática e simples. O método criado pela biblioteca não retorna uma simples lista de expressões lambda, mas um objeto que além de armazenar as expressões, armazena metadados, como veremos no próximo tópico.

4.2.2 Inserindo metadados através de anotações

Como não temos um mecanismo nativo para obter conhecimento sobre as estruturas das classes sendo mapeadas em momento de execução, temos que criar algum mecanismo que permita a criação destas informações. Uma maneira de fazer isto seria criar um analisador de código, que a partir da leitura dos arquivos de definição das classes geraria estas informações automaticamente. Esta solução tem a grande vantagem de gerar as informações em momento de compilação e agir de forma transparente. Entretanto, a implementação de tal solução é uma tarefa bastante complexa, além de seu uso promover uma quebra no fluxo padrão de compilação de programas, pois o desenvolvedor terá que inserir a execução deste analisador no fluxo de compilação antes da execução do próprio compilador. Outro problema, é que somente a informação das estruturas das classes não é suficiente para realizar o mapeamento, precisamos de informações a mais, como o nome das colunas equivalentes aos atributos.

A solução proposta neste trabalho consiste em inserir um método nas classes mapeadas que retorne uma estrutura com todos os metadados necessários para o mapeamento, ampliando a ideia exposta na seção 4.2.1. Para organizar as informações a serem retornadas será criada uma hierarquia de classes de armazenamento de metainformações,

baseada em uma classe chamada **"Reflect"**. Esta classe permite o registro de tuplas do tipo chave e valor, chamadas de **"tags"**, que podem ser recuperadas através de funções de sua interface. A partir desta classe são definidas as classes **"Property"** e **"Class"**. A primeira é responsável por descrever as informações relativas a um atributo de uma classe, e provê métodos para acesso a este atributo em uma instância de classe utilizando o mecanismo de expressões lambda citados anteriormente. A segunda classe é responsável por descrever as informações relativas a uma classe. Ela contém uma lista de objetos de descrição de atributos, além de permitir a definição de informações adicionais através da inserção de tags. Na imagem 4.2 temos um diagrama de classe simplificado que demonstra a hierarquia criada.

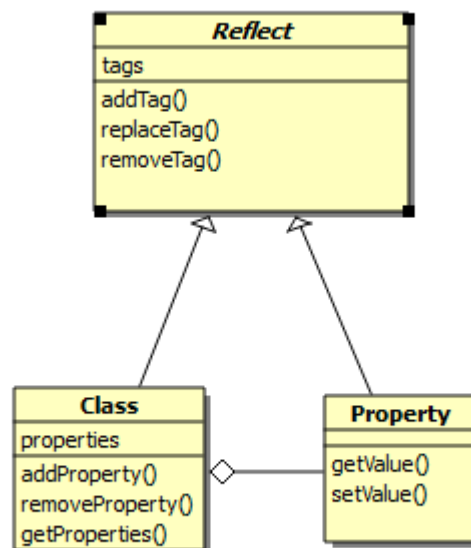


Figura 4.2: Diagrama simplificado das classes de reflexão

Com o uso desta técnica conseguimos contornar os dois limitadores impostos pela linguagem C++ para conhecimento e acesso a estrutura de objetos em tempo de execução, porém, ainda temos um problema relacionado com a inserção do método que retorna o objeto de reflexão. Ao impormos ao desenvolvedor a necessidade de criar tal método, a biblioteca deixa de ser transparente, pois estamos impondo a implementação de uma interface nas classes a serem mapeadas. Para contornar este problema é proposto a criação de uma camada de abstração através do uso de macros de registro, simulando o recurso de **anotações** existentes na linguagem JAVA. Estas macros em tempo de pré-processamento do código serão expandidas, construindo o método de retorno do objeto de reflexão. Desta maneira a criação do método será feita de forma transparente para o desenvolvedor.

No trecho de código 6 temos um esboço da utilização deste mecanismo. Nas linhas 7 e 13 temos as macros **ORM4QT_BEGIN** e **ORM4QT_END**, que delimitam o início e final da área de especificação de mapeamento. A primeira macro será expandida gerando a declaração do método de retorno do objeto de reflexão e a segunda expandirá o encerramento do método. Todas as macros compreendidas entre elas irão expandir o corpo do método. As outras duas macros utilizadas são a **CLASS** que recebe como parâmetros uma lista variável de tags, e a macro **PROPERTY** que recebe o atributo a ser mapeado, seguido de uma série de tags. Estas macros servem para registrar metadados sobre classes e atributos respectivamente.

```
1 class Classe
2 {
3 private:
4     int inteiro;
5     std::string texto;
6
7 ORM4QT_BEGIN
8
9 CLASS(name="Classe", table="tabela")
10 PROPERTY(inteiro, column="numero")
11 PROPERTY(texto, column="frase")
12
13 ORM4QT_END
14 };
```

Algoritmo 6: Esboço da utilização de macros para registro de metainformação

Com este mecanismo definido, as responsabilidades da camada Objeto já podem ser implementadas. Nas próximas seções serão detalhados os mecanismos a serem utilizados para implementação da camada de Armazenamento.

4.3 Camada de Armazenamento

Referências Bibliográficas

- [Barnes, 2007] Barnes, J. M. (2007). Object-relational mapping as a persistence mechanism for object-oriented applications.
- [Bauer and King, 2005] Bauer, C. and King, G. (2005). *Hibernate in Action*, volume 1. Manning Publications Co.
- [Blanchette and Summerfield, 2006] Blanchette, J. and Summerfield, M. (2006). *C++ GUI programming with Qt 4*. O'Reilly Japan.
- [Brokken and Kubat, 2008] Brokken, F. B. and Kubat, K. (2008). *C++ annotations*. Citeseer.
- [Bueno, 2002] Bueno, A. (2002). Apostila de programação orientada a objeto em c++.
- [Lhotka, 2009] Lhotka, R. (2009). *Expert C# 2008 Business Objects*, volume 1. Apress.
- [Marty, 2014] Marty, L. (2014). Qxorm - c++ object relational mapping library. <http://www.qxorm.com/doxygen/html/index.html>.
- [Nierstrasz, 1989] Nierstrasz, O. (1989). A survey of object-oriented concepts.
- [PostgreSQL, 2014] PostgreSQL, T. G. D. G. (2014). *PostgreSQL 9.3.4 Documentation*.
- [Thelin, 2007] Thelin, J. (2007). *Foundations of Qt development*, volume 7. Springer.
- [Zhang, 2004] Zhang, X. (2004). *A framework for object-relational mapping with an example in C++*. PhD thesis, Concordia University.