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 título de Bacharel em Ciência da Computação da Faculdade de Engenharia da Universidade de Itaúna.

Orientador:
Angélica Aparecida 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:

Angélica Aparecida Moreira

Paulo de Tarso Gomide Castro Silva

Itaúna, 05 de Junho de 2014.

Resumo

As Linguagens Orientadas a Objetos em conjunto com os Sistemas Gerenciadores de Bancos de Dados Relacionais (SGBDs) são considerados padrões no desenvolvimento de Sistemas de Informação Empresarial. Embora o uso destas duas tecnologias em conjunto mostre resultados satisfatórios, encontra-se dificuldade em efetuar a transição de informações entre os seus contextos de representação de dados.

Para efetuar a transição de informações é necessário desenvolver componentes de *software* específicos para cada classe que represente dados que necessitam ser persistidos no banco de dados, o que se mostra uma tarefa repetitiva e propensa a erros. Com o objetivo de amenizar este problema, foram criadas as Bibliotecas de Mapeamento Objeto Relacional (ORMs), que são componentes de *software* capazes de automatizar o processo de transição de dados entre os dois contextos.

Devido a limitações nos recursos de reflexão e inserção de metadados oferecidos pela linguagem C++, o desenvolvimento de soluções ORM para esta linguagem se mostra uma tarefa bastante complexa. A maioria das soluções implementadas apresenta interfaces de configuração complexas, além de utilizar recursos da linguagem que promovem o aumento do nível de acoplamento do código.

Este trabalho propõe o desenvolvimento de uma biblioteca ORM para a linguagem C++, onde são implementados mecanismos próprios de reflexão e inserção de metadados a partir da utilização dos novos recursos propostos pela especificação *C++11*. A biblioteca utiliza vários componentes oferecidos pelo *framework* Qt para auxiliar em tarefas como comunicação com banco de dados e extensão de tipos nativos.

Palavras-chave: Orientação a Objetos, SGBD, C++, ORM

Abstract

The Object Oriented Languages combined with the Relational Database Management Systems (RDBMs) are considered a pattern in Enterprise Information Systems (EIS) development. Although their use show good results, is difficult to perform the information transition between their contexts of data representation.

To perform the information transition it is necessary to develop specific software components for each class that represents data persisted in the database, task that shows itself repetitive and error prone. With the goal of ease this problem, we created the Object Relational Mapping (ORM) libraries, software components that automates the process of data transition between the two contexts.

Due to limitations in the C++ reflection and metadata insertion features, the development of ORM solutions for this language is a very complex task. The existent solutions present complex configuration interfaces, besides using language resources that increase the coupling level of code.

This paper proposes the development of a ORM library for the C++ language, using own mechanisms of reflection and insertion of metadata based on the features of the C++11 specification. The library uses many components offered by the Qt framework to help in tasks like communication with the database and extension of native types.

Keywords: Object Oriented, RDBMS, C++, ORM

Glossário

API	:	*Application Programming Interface*;
ORM	:	*Object Relational Mapping*;
SGBD	:	Sistema Gerenciador de Banco de Dados;
SQL	:	*Structured Query Language*;
EIS	:	*Enterprise Information Systems*;

Sumário

1 Introdução **1**

2 Fundamentação Teórica **3**

2.1 Orientação a Objetos 3

2.2 A Linguagem de Programação C++ 6

2.2.1 Reflexão ou Introspecção 7

2.2.2 Processo Evolutivo da Linguagem 8

2.2.2.1 Listas de Inicialização 9

2.2.2.2 Navegação em Grupos de Elementos 10

2.2.2.3 Novo Identificador para Ponteiros Nulos 12

2.2.2.4 Inferência de Tipos 14

2.2.2.5 Programação Funcional 16

2.2.2.6 Ponteiros Inteligentes (*Smart Pointers*) 20

2.3 O *Framework* Qt 22

2.4 Sistemas Gerenciadores de Bancos de Dados Relacionais 25

2.5 O PostgreSQL 26

2.6 A combinação de SGBDs com Linguagens Orientadas a Objetos 26

2.7 Bibliotecas de Mapeamento Objeto Relacional 28

3 Trabalhos relacionados **31**

3.1 Um *Framework* de Mapeamento Objeto Relacional com um Exemplo em C++ 31

3.2 QxORM 32
3.3 ODB 33
3.4 Hibernate 34

4 A biblioteca ORM4Qt 35

4.1 Arquitetura em Camadas 36
4.2 Camada Objeto 37
4.2.1 Quebrando o Encapsulamento das Classes 38
4.2.2 Inserindo Metadados Através de Anotações 40
4.3 Camada de Armazenamento 42

5 Cronograma de Atividades 44

6 Conclusão 45

Referências Bibliográficas 46

Lista de Figuras

2.1 Linha do tempo: evolução da especificação da linguagem C++ 9
2.2 Sintaxe básica das expressões lambda 18
2.3 Janela no GNU/Linux 22
2.4 Janela no Microsoft Windows XP 23
2.5 Janela no Mac Os 24
2.6 Exemplo de definição de tabelas e relações 26
4.1 Interação entre componentes do software e desenvolvedor 37
4.2 Diagrama simplificado das classes de reflexão 41
4.3 Diagrama de classes simplificado da camada de armazenamento 43
5.1 Cronograma de atividades 44

Lista de Algoritmos

1 Definição de uma classe simples em C++ 5

2 Inicialização de vetores e estruturas *container* 10

3 Acesso sequencial via índice 11

4 Acesso sequencial via iteradores 11

5 Acesso sequencial com nova sintaxe (C++11) 12

6 Novo identificador para ponteiros nulos 13

7 Trecho de declaração longo 14

8 Trecho de declaração curto usando **"auto"** 14

9 Particularidade do uso da palavra-chave **"auto"** 15

10 Dedução de tipo com o uso da palavra-chave **"decltype"** 15

11 Sintaxe para utilização de ponteiros para funções e métodos 17

12 Unificação da representação de ponteiros para métodos e funções 18

13 Exemplo de definição de uma expressão lambda 20

14 Utilizando *smart pointers* 21

15 Exemplo de comunicação com SGBD utilizando o módulo QtSql 27

16 Obtendo o nome da classe de um objeto em tempo de execução 38

17 Exemplo de variações na declaração de métodos acessadores para atributos tipo inteiro 39

18 Retornando uma expressão lambda para acesso de atributo privado 40

19 Esboço da utilização de macros para registro de metainformação 42

Capítulo 1

Introdução

Vivemos em uma era onde os sistemas informatizados deixaram de ser vistos como meras ferramentas de automatização de tarefas. Onde o *software* desempenha 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 Objetos** 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** (**SGBDs**) são o meio consolidado para armazenamento de dados estruturados. Eles permitem o armazenamento de informações de tal forma que estas podem ser manipuladas através de comandos em linguagem **SQL** ou ***Structured Query Language***. 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 objetos 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 objetos em conjunto com os SGBDs surgiu o conceito de **Biblioteca de Mapeamento Objeto Relacional** ou **ORM** (*Object Relational Mapping*). Este tipo de 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 [Barnes, 2007]. 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 são apresentados os 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**, que são linguagens de programação que utilizam um paradigma ou padrão de estruturação de código que permite que o desenvolvedor crie 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 é conhecida como **objeto**. Um objeto é criado a partir de um modelo preestabelecido 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].

[1]Linguagem interpretada com tipagem dinâmica baseada em *scripts* muito utilizada em programação

O estado de um objeto é definido a partir da adição de variáveis em sua estrutura interna para armazenamento de valores. Estes valores podem ser unitários (onde são armazenados dados do tipo texto ou numérico, por exemplo), ou podem ser compostos (onde são utilizados outros objetos para compor a estrutura interna). As variáveis internas de um objeto são conhecidas como **atributos** ou **propriedades** e a utilização de valores compostos na definição da estrutura de uma classe é um mecanismo conhecido como **composição** [Nierstrasz, 1989].

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, ou para efetuar processamento de dados baseados no estado ou objetivo do objeto ao qual ela pertence. Estas funções são conhecidas como **métodos** [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 funções de comparação, como 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 1, 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.

Mecanismos de encapsulamento (métodos, atributos e restrição de interface), defini-

para a WEB e como complemento para interfaces com linguagens e/ou programas complexos.

```
1 class Pessoa {
2   private:
3     string m_nome;
4     string m_sobrenome;
5   public:
6     string nomeCompleto() { return m_nome + m_sobrenome; }
7 }
```

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

ção de classes e instanciação de objetos são as características mais comumente encontradas nas linguagens consideradas como orientadas a objetos. Porém, existem mecanismos mais avançados que permitem uma melhoria na definição do comportamento das classes e promovem reaproveitamento de código. Entre os principais estão a **herança** e o **polimorfismo** [Nierstrasz, 1989].

A herança consiste na capacidade de uma classe estender a estrutura de uma classe já definida. Isso significa que a nova classe mantém a interface da anterior e tem a capacidade de adicionar novos atributos e métodos a ela [Nierstrasz, 1989]. 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++.

Algumas notações são utilizadas para demonstrar a utilização do mecanismo de herança. Por exemplo, 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 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 objetos 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].

A linguagem 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, 2014].

Dentre as características da linguagem estão a capacidade de definição de classes, utilização de herança múltipla, utilização de polimorfismo, 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.

A seguir são descritas características da linguagem presentes em contextos bem específicos e que interferiram bastante no desenvolvimento da biblioteca ORM proposta neste trabalho. Também é descrito o modelo de projeção do processo evolutivo da linguagem, onde são ressaltados mecanismos introduzidos na especificação **C++11** e **C++14** que contribuíram para eliminar ou amenizar algumas limitações da linguagem.

2.2.1 Reflexão ou Introspecção

Algumas vezes nos deparamos com a tarefa de desenvolver rotinas que dependem do conhecimento da estrutura de um objeto qualquer. Problemas como "escreva uma função que mostre o nome dos atributos de um objeto" ou "escreva uma função que armazene em um arquivo o valor de todos os atributos de um objeto" são alguns exemplos de rotinas deste tipo. É nestes momentos que a utilização de mecanismos de **introspecção** ou **reflexão** se torna útil.

Estes mecanismos consistem na disponibilização por parte da linguagem de programação de estruturas que permitem analisar a especificação de um objeto e acessar seus recursos, tudo em tempo de execução. Desta forma podemos escrever rotinas genéricas que podem trabalhar com qualquer tipo de objeto para executar diversos tipos de tarefas, como listar os atributos de um objeto, acessar e alterar o valor destes atributos, listar os métodos de um objeto e até mesmo executá-los [Lhotka, 2009].

Geralmente as linguagens híbridas e as interpretadas oferecem métodos de reflexão completos, pois as informações da estrutura dos objetos são utilizadas pelas próprias máquinas virtuais e pelos interpretadores em tempo de execução. Neste caso, já que a informação já está presente e formatada durante a execução, basta a linguagem criar mecanismos de disponibilização dos mesmos [Lhotka, 2009].

Já no cenário das linguagens compiladas a situação é bem diferente. Não há a necessidade de geração de informações sobre estruturas de objetos para o código ser executado,

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

visto que o programa é gerado em código nativo [Gregoire, 2014]. A linguagem C++ é do tipo compilada, portanto, também sofre com a falta de mecanismos de reflexão nativos.

2.2.2 Processo Evolutivo da Linguagem

A linguagem C++ foi criada para ser uma linguagem de propósito geral, com o objetivo principal de apresentar mecanismos para programação com alto nível de abstração, porém, ao mesmo tempo sem privar o desenvolvedor da capacidade de acessar rotinas e recursos de baixo nível. Sendo assim é possível obter programas de alto desempenho em diversas plataformas diferentes [Brokken and Kubat, 2014].

Para a linguagem ser utilizada em múltiplas plataformas é necessário que exista um compilador que suporte a criação de programas a partir da análise de código fonte em C++ para cada plataforma. Para evitar variações na implementação destes compiladores existe uma especificação oficial de todo o conjunto de recursos oferecidos pela linguagem bem como sua sintaxe [Brokken and Kubat, 2014]. Esta especificação é mantida pela **ISO** (*International Organization for Standardization*).

As linguagens de programação evoluem com a passar do tempo. Em alguns momentos é preciso acrescentar novos recursos para alinhar a linguagem com os padrões atuais, outras vezes é necessário melhorar recursos existentes ou até mesmo corrigir erros na especificação [Brokken and Kubat, 2014].

A especificação da linguagem C++ passou por uma grande atualização em 1998 (conhecida como **C++98**), seguida de algumas correções em 2003 (especificação conhecida como **C++03**). Desde então a especificação não sofria alterações significativas, o que levou a linguagem a ficar um pouco desatualizada de acordo com os conceitos que foram surgindo ao longo dos anos. Até que em meados de 2011 um comitê de especialistas começou a agir com o intuito de trazer novos recursos para a linguagem. Isso levou à publicação da especificação **C++11** no final do ano de 2011 e à projeção de atualizações da linguagem para os anos de 2014 e 2017, especificações conhecidas como **C++14** e **C++17** [Brokken and Kubat, 2014].

A figura 2.1 mostra a linha do tempo da evolução da especificação da linguagem C++. Nela é possível perceber a grande concentração de tarefas previstas para os anos entre 2014 e 2017. Durante a finalização deste trabalho (novembro de 2014) a publicação da especificação C++14 está prevista para ser publicada em breve[3].

[3]Para mais informações acesse o site https://isocpp.org/std/status.

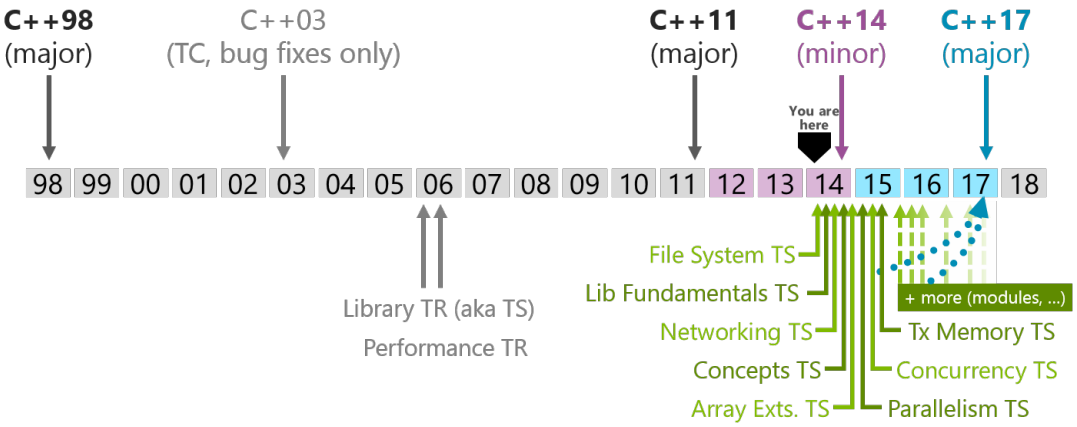

Figura 2.1: Linha do tempo: evolução da especificação da linguagem C++

Muitos recursos adicionados na linguagem pelas especificações C++11 e C++14 foram utilizados no desenvolvimento deste trabalho. Nas seções seguintes serão descritos alguns destes recursos, e como eles contribuem para melhorar algumas limitações que eram impostas pela linguagem.

2.2.2.1 Listas de Inicialização

Um recurso simples e ao mesmo tempo bastante útil, as **listas de inicialização** foram criadas com o objetivo de facilitar a instanciação de estruturas do tipo *container* (lista, pilha, fila, tabelas *hash*, etc) com um conjunto de elementos adicionados. É importante ressaltar que a linguagem na especificação C++98 já permitia a inicialização de vetores com uma lista de elementos [Gregoire, 2014].

Este recurso se baseia na utilização de uma nova classe chamada ***"initializer_list"*** definida dentro do *namespace*[4] "*std*". Esta classe implementa um *container* de dados simples, que suporta o acesso aos seus elementos através do uso de **iteradores**. Quando o compilador encontra algum trecho de código com o padrão "{x, y, z}" onde "x", "y" e "z" são valores literais ou variáveis que podem ser convertidas para um tipo de dado "T" comum de forma implícita, ele constrói um objeto do tipo "*initializer_list*<*T*>". Sendo assim, para que um objeto possa ser construído com uma lista de inicialização, basta que sua classe contenha um construtor que recebe como parâmetro um objeto do tipo "*initializer_list*<*T*>" [Brokken and Kubat, 2014].

No trecho de código 2 é demonstrado a utilização das listas de inicialização. Primeiramente temos a inicialização de um vetor de números inteiros (linha 2) utilizando a sintaxe

[4]Mecanismo presente na linguagem C++ que permite agrupar estruturas dentro de um espaço de resolução de nomes comum. É utilizado para evitar conflitos de nomes entre estruturas.

já suportada pela linguagem na especificação C++98. Em seguida temos a inicialização de um objeto do tipo ***vector***[5] (linha 4) utilizando o novo recurso adicionado na especificação C++11. Por último, temos a definição de uma classe projetada para suportar o recurso de listas de inicialização (linhas 7 a 9) e em seguida a inicialização de um objeto desta classe (linha 13).

```
1 //Inicialização de vetores
2 int vetor[] = {1, 2, 3, 4};  //Válido em C++98
3 //Inicialização de containers
4 vector<int> lista1 = {1, 2, 3, 4}; //Válido em C++11
5
6 //Declaração da classe Container<T>
7 template<typename T> class Container {
8    Container(const initializer_list<T> &list);
9 };
10
11 //Válido em C++11 já que a classe Container<T> tem um construtor
12 //que recebe como parâmetro um objeto do tipo initializer_list<T>
13 Container<int> lista2 = {1, 2, 3, 4};
```

Algoritmo 2: Inicialização de vetores e estruturas *container*

2.2.2.2 Navegação em Grupos de Elementos

Quando precisamos acessar sequencialmente os elementos de um vetor ou de alguma estrutura do tipo *container* utilizando a linguagem C++, temos basicamente duas opções:

1) **Acesso sequencial via índice:** Utilizamos um laço de repetição para gerar valores numéricos sequenciais dentro do intervalo de posições ou índices dos elementos que desejamos acessar na estrutura. De posse dos índices, podemos acessar os elementos utilizando a sintaxe "estrutura[índice]". Esta técnica pode ser utilizada em vetores e em estruturas *container* que implementam o operador "[]" [Gregoire, 2014]. Um exemplo da utilização desta técnica é apresentado no trecho de código 3.

2) **Acesso sequencial via iteradores:** As estruturas *container* presentes na linguagem C++ disponibilizam o acesso a um ponteiro especial que permite navegar entre os elementos que ele contém. Estes ponteiros são chamados de **iteradores** e podem ser de dois tipos: constantes e não constantes. Os iteradores constantes permitem acessar os elementos em modo somente leitura. Enquanto os não constantes permitem a modificação dos valores destes elementos [Gregoire, 2014]. Um exemplo da utilização desta técnica é apresentado no trecho de código 4.

[5]Classe presente na biblioteca padrão de C++, e que implementa uma estrutura de lista de elementos utilizando áreas de memória contíguas.

```
1  //Declaração de um vetor
2  int vetor[] = {1, 2, 3, 4, 5};
3  //Acesso sequencial via índice
4  for (int i = 0; i < 5; ++i) {
5      cout << vetor[i] << endl;
6  }
7
8  //Declaração de um container
9  vector<int> container = {1, 2, 3, 4, 5};
10 //Acesso sequencial via índice
11 for (int i = 0; i < container.size(); ++i) {
12     cout << container[i] << endl;
13 }
```

Algoritmo 3: Acesso sequencial via índice

```
1  //Declaração de um container
2  vector<int> container = {1, 2, 3, 4, 5};
3  //Acesso sequencial via iteradores do tipo constantes
4  for (vector<int>::const_iterator i = container.cbegin();
5       i != container.cend(); ++i) {
6      cout << *i << endl;
7  }
8  //Acesso sequencial via iteradores do tipo não constantes
9  for (vector<int>::iterator i = container.begin();
10      i != container.end(); ++i) {
11     cout << *i << endl;
12 }
```

Algoritmo 4: Acesso sequencial via iteradores

Na especificação C++11 foi adicionada uma variação do laço de repetição ***"for"*** que é basicamente uma simplificação das técnicas 1 e 2 descritas anteriormente. Considerando que temos um grupo de elementos do tipo "T" armazenados em uma estrutura (vetor ou *container*) com uma instância chamada "elementos", podemos navegar sobre os elementos desta instância utilizando um trecho de código no formato "for (T e : elementos) {...}". Neste caso, o código contido entre as chaves será executado uma vez para cada elemento contido em "elementos", onde a variável "e" assume o valor de cada um destes elementos [Gregoire, 2014].

Esta nova variação pode ser utilizada tanto com vetores quanto com estruturas do tipo *container*, e também é possível navegar sobre os elementos em modo somente leitura. No trecho de código 5 é demonstrada a utilização desta nova sintaxe de diferentes formas.

```
1  //Declaração de um vetor
2  int vetor[] = {1, 2, 3, 4, 5};
3  //Acesso sequencial nova sintaxe, com cópia
4  for (int i: vetor) { //i copia o valor de cada elemento
5      cout << i << endl;
6  }
7  //Acesso sequencial nova sintaxe, com referência
8  for (int &i: vetor) { //Não há cópia de valores
9      cout << i << endl;
10 }
11
12 //Declaração de um container
13 vector<int> container = {1, 2, 3, 4, 5};
14 //Acesso sequencial nova sintaxe, com cópia
15 for (int i : container) { //i copia o valor de cada elemento
16     cout << i << endl;
17 }
18 //Acesso sequencial nova sintaxe, com referência
19 for (int &i: container) { //Não há cópia de valores
20     cout << i << endl;
21 }
22 //Acesso sequencial nova sintaxe, somente-leitura com referência
23 for (const int &i: container) {
24     cout << i << endl;
25 }
```

Algoritmo 5: Acesso sequencial com nova sintaxe (C++11)

2.2.2.3 Novo Identificador para Ponteiros Nulos

A linguagem C++ permite a manipulação direta de memória a partir da utilização de **ponteiros**, variáveis estas que armazenam o endereço inicial da porção de memória em que uma estrutura de dados está sendo armazenada durante a execução de um programa [Gregoire, 2014].

A manipulação de ponteiros é uma tarefa complexa, que deve ser feita com muito cuidado. Uma tentativa de acesso a uma região de memória inválida pode gerar problemas graves durante a execução do programa. Para evitar este tipo de problema, uma boa prática consiste em invalidar ou anular ponteiros durante a sua inicialização e quando o seu uso chegou ao fim (o que acontece por exemplo quando desalocamos a região de memória apontada por ele) [Gregoire, 2014].

Para anular um ponteiro simplesmente atribuímos a ele o valor 0. Porém, na maioria dos ambientes de desenvolvimento, temos a presença de uma macro chamada "NULL" que é utilizada no local do valor 0 durante a anulação de um ponteiro para fins de legibilidade.

Esta metodologia pode levar a comportamentos inesperados do código durante a execução, visto que a macro é expandida para o valor inteiro 0 durante a compilação. Se temos duas funções com o mesmo nome, onde uma recebe um ponteiro como parâmetro e a outra recebe um número inteiro, ao executar esta função passando a macro "NULL", a versão que recebe o número inteiro será executada, gerando assim um comportamento inesperado.

Com o objetivo de resolver este problema a especificação C++11 prevê a existência do identificador **"nullptr"** para ser utilizado com o significado de ponteiro nulo. Então se no mesmo problema citado executássemos a função passando "nullptr" como parâmetro, a versão que recebe o ponteiro será executada [Gregoire, 2014]. O exemplo citado pode ser observado no trecho de código 6.

```
1  void funcao(char* valor)
2  {
3      cout << "char*" << endl;
4  }
5
6  void funcao (int valor)
7  {
8      cout << "int" << endl;
9  }
10
11 void main()
12 {
13     funcao(NULL);   //Executa funcao (int valor)
14     funcao(nullptr); //Executa funcao(char* valor)
15 }
```

Algoritmo 6: Novo identificador para ponteiros nulos

2.2.2.4 Inferência de Tipos

A linguagem C++ é uma linguagem de tipagem estática, ou seja, todas as variáveis declaradas no código devem ter seu tipo de dado especificado em momento de compilação [Brokken and Kubat, 2014]. Esta característica acaba em certas situações deixando o código de declaração de variáveis muito grande. Veja por exemplo o trecho de código 7, onde são declarados um *container* para uma lista de inteiros e um iterador para esta lista.

```
1 //Declara uma lista de inteiros
2 vector<int>* vetor = new vector<int>();
3 //Obtém um iterador para a lista declarada
4 vector<int>::iterator iter = vetor->begin();
```

Algoritmo 7: Trecho de declaração longo

Com a finalidade de reduzir estes trechos de código, a especificação C++11 introduziu um mecanismo de inferência ou detecção automática de tipos de variáveis através do uso das palavras-chave **"auto"** e **"decltype"** [Gregoire, 2014].

A primeira é utilizada durante a atribuição de valores que já tem um tipo de dado definido por outro mecanismo, como por exemplo, pelo retorno de uma função ou pelo uso do operador de instanciação ***"new"***. O trecho de código 7 poderia ser simplificado para o código em 8.

```
1 //Declara uma lista de inteiros
2 auto vetor = new vector<int>();
3 //Obtém um iterador para a lista declarada
4 auto iter = vetor->begin();
```

Algoritmo 8: Trecho de declaração curto usando **"auto"**

O uso da palavra-chave "auto" tem uma particularidade importante que deve ser levada em consideração durante o seu uso. O tipo deduzido é o resultado do tipo do valor atribuído à variável após a remoção do modificador **"const"**[6] e do modificador **"&"**[7] [Gregoire, 2014]. Esta particularidade pode levar a resultados inesperados, como o que pode ser observado no trecho de código 9 na linha 10, onde a dedução de tipo irá levar a criação de uma variável do tipo inteiro com o valor copiado do retorno da função "constRef", enquanto que o valor atribuído originalmente é de uma referência para um valor inteiro constante.

[6] Utilizado para declarar variáveis constantes, ou seja, que não podem ser usadas para modificar valores.
[7] Neste contexto se refere ao modificador para declaração de variáveis do tipo referência na linguagem C++.

```
1 const int& constRef(int& valor)
2 {
3     return valor;
4 }
5
6 int main()
7 {
8     int valor = 10;
9     //O tipo deduzido é (int) e não (const int&)
10    auto numero = constRef(valor);
11 }
```

Algoritmo 9: Particularidade do uso da palavra-chave **"auto"**

Nestas situações podemos utilizar a palavra-chave "decltype" para alcançar o comportamento requerido. A função desta palavra-chave é avaliar o tipo de dado do resultado de uma expressão qualquer [Gregoire, 2014]. Ela pode ser utilizada em qualquer lugar do código onde se espera a especificação de um tipo de dado, como por exemplo em declarações de variáveis e como parâmetros para estruturas genéricas (que utilizam o mecanismo de **"templates"**[8]).

No trecho de código 10 podemos ver a utilização desta palavra-chave. Neste caso a variável "numero" (linha 10) irá ser deduzida como uma referência para um valor inteiro constante, que é exatamente o tipo retornado pela função "constRef". Porém o código gerado para a dedução não é tão simplificado, o que levou a proposta da adição da construção **"decltype(auto)"** (linha 12) pela especificação C++14, que tem o mesmo resultado neste caso [Gregoire, 2014].

```
1 const int& constRef(int& valor)
2 {
3     return valor;
4 }
5
6 int main()
7 {
8     int valor = 10;
9     //O tipo deduzido é (const int&)
10    decltype(constRef(valor)) numero = constRef(valor);
11    //Simplificação proposta na C++14
12    decltype(auto) num = constRef(valor);
13 }
```

Algoritmo 10: Dedução de tipo com o uso da palavra-chave **"decltype"**

[8]Mecanismo presente na linguagem C++ que permite o desenvolvimento de código que pode trabalhar com diferentes tipos de dados. É muito utilizado na construção de estruturas do tipo *container*.

2.2.2.5 Programação Funcional

O conceito de programação funcional propõe que funções e métodos sejam tratados de forma igualitária com variáveis e objetos, no aspecto de poderem ser transportados entre contextos de execução, ser armazenados em estruturas específicas, ser criados sob demanda e ser utilizados como parâmetros para outras funções e/ou métodos [Clugston, 2004].

A maneira tradicional utilizada para manipular funções e métodos na linguagem C++ consiste do uso de ponteiros. Porém esta metodologia utiliza uma sintaxe bem difícil e existem vários problemas ou limitadores em relação à conversão destes ponteiros. Um grande problema encontrado é que ponteiros para funções (declaradas no contexto global) e métodos (declarados dentro de classes, exceto os estáticos neste caso) possuem sintaxe de declaração diferentes e não são convertíveis entre si [Clugston, 2004].

No trecho de código 11 é demonstrada a sintaxe do uso de ponteiros para funções e métodos. Na linha 16 é declarado um ponteiro para a função "getTexto", e na linha 18 é declarado um para o método de mesmo nome presente na classe "Exemplo". Note que apesar da função e do método possuírem o mesmo protótipo (tipo de retorno e parâmetro) a sintaxe para declaração de seus ponteiros é diferente e eles não podem ser convertidos entre si.

A especificação C++11 propõe a unificação da representação destes ponteiros como instâncias da classe **"function"** que é definida dentro do *namespace* "std". Esta classe consegue representar ponteiros para funções, métodos e referências para *"function objects"*[9] [Gregoire, 2014].

O trecho de código 11 pode ser reescrito com o uso desta classe conforme o exemplo 12. Note que para atribuir um ponteiro para método é preciso utilizar a função **"bind"** para associar um objeto com o método a ser apontado (linha 22). Com esta nova sintaxe apontadores para métodos e funções com o mesmo protótipo podem ser convertidos entre si.

Outra funcionalidade adicionada pela especificação C++11 foi o suporte a definição de métodos anônimos, também conhecidos como **expressões lambda**. Esta funcionalidade consiste na capacidade de criar funções sob demanda durante blocos de execução do programa. Expressões lambda também podem ser armazenadas em instâncias da classe *"function"* [Gregoire, 2014].

[9] Instâncias de classes que reimplementam a função do operador parênteses.

```
1  class Exemplo {
2  public:
3      const string getTexto()
4      { return m_texto; }
5      //...
6  private:
7      string m_texto;
8  };
9
10 const string getTexto()
11 { return string("Exemplo de função."); }
12
13 int main()
14 {
15     //Ponteiro para função
16     const string (*funcao) () = &getTexto;
17     //Ponteiro para método
18     const string (Exemplo::*metodo)() = &Exemplo::getTexto;
19
20     //Utilizando o ponteiro de funçao
21     cout << funcao() << endl;
22     //Utilizando o ponteiro de método
23     Exemplo ex;
24     cout << (ex.*metodo)() << endl;
25
26     funcao = metodo; //Inválido
27     metodo = funcao; //Inválido
28 }
```

Algoritmo 11: Sintaxe para utilização de ponteiros para funções e métodos

```
class Exemplo {
public:
  const string getTexto()
  { return m_texto; }
  //...
private:
  string m_texto;
};

const string getTexto()
{ return string("Exemplo de função."); }

int main()
{
  //Declarando container unificado
  function<const string()> funcao = nullptr;
  //Atribuindo e utilizando uma função
  funcao = getTexto;
  cout << funcao() << endl;
  //Atribuindo e utilizando um método
  Exemplo e;
  funcao = bind(&Exemplo::getTexto, &e); //relaciona instância
  cout << funcao() << endl;
}
```

Algoritmo 12: Unificação da representação de ponteiros para métodos e funções

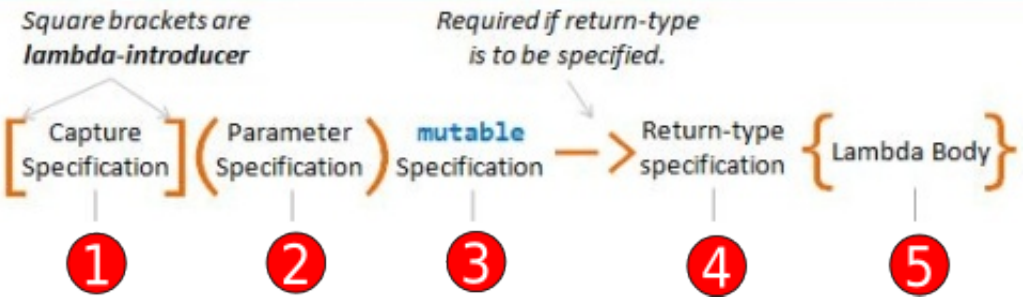

Figura 2.2: Sintaxe básica das expressões lambda

A sintaxe básica para utilização de expressões lambda é demonstrada na imagem 2.2. Os blocos presentes nesta construção são explicados em seguida:

1) Área de captura (*Capture specification*): As expressões lambda podem capturar as variáveis presentes no contexto atual de execução para utilizar seus valores no seu bloco de execução. Esta captura pode feita por valor, onde é efetuado uma cópia dos valores das variáveis do contexto alvo, ou pode ser feita por referência, onde as variáveis serão referenciadas no bloco de execução da expressão, ou seja, podemos trabalhar com modificação de valores das variáveis originais. Neste último caso é importante garantir que as variáveis capturadas existem no momento de execução da expressão, caso contrário teremos uma exceção de acesso a posição

inválida de memória. A captura pode ser feita especificando cada variável que queremos capturar, como por exemplo no trecho "[a, &b]" capturamos a variável "a" por valor e "b" por referência. Ou podemos capturar todas as variáveis do contexto utilizando "[=]" para capturar todas as variáveis por valor ou "[&]" para capturar por referência. Caso não seja necessário capturar variáveis especificamos através do de colchetes sem conteúdo, assim "[]" [Brokken and Kubat, 2014].

2) Lista de parâmetros (*Parameter specification*): É onde é especificada a lista de parâmetros que a expressão lambda recebe. A sintaxe é a mesma utilizada na definição de protótipos de funções e métodos. Por exemplo no trecho "(int a, char b)" dizemos que a expressão recebe um número inteiro que será referenciado no corpo da expressão como "a" e também recebe um caractere que será referenciado como "b" [Brokken and Kubat, 2014].

3) *Mutable specification*: Por padrão, variáveis capturadas por valor são tratadas dentro do bloco de execução da expressão como do tipo constante, ou seja, não podemos modificar seus valores. Caso utilizemos o modificador **"mutable"** elas não serão mais tratadas como do tipo constante, então poderemos manipular os valores copiados no bloco de execução da expressão [Brokken and Kubat, 2014].

4) Especificação do tipo de retorno (*Return type specification*): Neste ponto devemos especificar o tipo de dado que será retornado pela expressão lambda. Caso a expressão não retorne valores ou o tipo do valor retornado pode ser deduzido pelo compilador, podemos omitir esta parte [Brokken and Kubat, 2014].

5) Bloco de execução (*Lambda body*): Consiste do bloco de código que será executado quando a expressão for utilizada. Assim como a lista de parâmetros, este trecho de definição segue a mesma sintaxe da definição de funções e métodos comuns. Podem ser utilizadas quaisquer construções válidas para blocos de execução na linguagem C++. É importante ressaltar também que não há um limite de tamanho para o bloco, porém expressões lambda geralmente são bem concisas e simplificadas, ocupando poucas linhas. Caso a expressão comece a ficar muito grande e complexa, talvez seja melhor definir uma função comum para que possa ser reutilizada por outros componentes do programa [Brokken and Kubat, 2014].

No trecho de código 13 temos um exemplo de definição de uma expressão lambda com o mesmo protótipo utilizado nos exemplos anteriores. Note que a expressão pode ser

armazenada em uma instância da classe "*function*" e a sua execução é feita da mesma maneira que os ponteiros para funções e métodos armazenados nestas instâncias.

```
1 //Exemplo de expressão lamda
2 function<const string()> lambda = []() { return string("Exemplo."); };
3 //Executando a expressão
4 cout << lambda() << endl;
```

Algoritmo 13: Exemplo de definição de uma expressão lambda

2.2.2.6 Ponteiros Inteligentes (*Smart Pointers*)

Um dos maiores desafios enfrentados por desenvolvedores que utilizam a linguagem C++ consiste na garantia da liberação de blocos de memória alocados dinamicamente a partir do uso de ponteiros. Sempre que alocamos uma região de memória a partir do uso do operador ***"new"***, temos que garantir que em algum ponto do código aquela região será liberada, tarefa que deve ser realizada explicitamente com o uso do operador ***"delete"*** [Brokken and Kubat, 2014].

Regiões de memória alocada que não são liberadas, irão persistir marcadas como utilizadas durante todo o fluxo de execução do programa, causando assim um efeito de utilização extra de memória indevida também conhecido como ***"memory leak"*** [Gregoire, 2014].

Com o objetivo de evitar este tipo de problema, a especificação C++11 prevê a criação de ponteiros inteligentes ou *"smart pointers"*, que são basicamente estruturas que simulam o funcionamento de um ponteiro comum, porém adicionando mecanismos de gerenciamento de memória. Do ponto de vista de padrões de projeto, um *smart pointer* seria um *proxy* para a interface de ponteiros comuns [Brokken and Kubat, 2014].

A especificação C++11 define a presença de três *smart pointers*, sendo eles:

1) ***"unique_ptr"*:** Este ponteiro têm basicamente as mesmas características de um ponteiro comum. A única diferença é que ele automaticamente libera a região de memória apontada por ele quando ele é deletado ou quando o escopo de sua utilização chega ao fim [Gregoire, 2014].

2) ***"shared_ptr"*:** Este ponteiro possui uma funcionalidade um pouco mais avançada. Ele mantém um contador interno que armazena a quantidade de referências que existem para a região de memória apontada por ele, ou seja, ele armazena quantas instâncias do tipo "shared_ptr" se encontram ativas no momento para aquela

região de memória. O contador é incrementado quando o ponteiro é copiado e decrementado quando instâncias de ponteiros são deletadas ou perdem escopo. Somente quando o contador chegar a zero, ou seja, quando não existir mais nenhuma cópia do ponteiro original ativa, a região de memória será automaticamente liberada [Gregoire, 2014].

3) ***"weak_ptr"*****:** Este ponteiro é utilizado para testar se uma instância de "shared_ptr" ainda é válida (ainda não foi desalocada), e caso o for, permite efetuar uma cópia desta instância para acessar a região de memória apontada [Gregoire, 2014].

O trecho de código 14 demonstra a utilização dos três tipos de *smart pointers* descritos anteriormente. Nele são alocados dois espaços de memória para armazenar dados do tipo inteiro dentro de um escopo de execução delimitado por chaves. Assim que o escopo é finalizado, ambos os ponteiros desalocam a região de memória apontada por eles automaticamente. No código também é mostrado como utilizar um "weak_ptr" para verificar se um "shared_ptr" ainda é válido a partir da utilização do método **"lock"** que retorna uma cópia deste segundo ponteiro no caso de validação ou retorna "nullptr" caso contrário.

```
1  //Declara um weak_ptr sem inicilização
2  weak_ptr<int> wptr;
3
4  //Abertura de escopo
5  {
6      //Declara um shared_ptr
7      shared_ptr<int> sptr = shared_ptr<int>(new int(10));
8      //Declara um unique_ptr
9      unique_ptr<int> uptr = unique_ptr<int>(new int(10));
10     //Inicializa o weak_ptr
11     wptr = sptr;
12     //Testa se memória ainda não foi desalocada
13     if(auto ptr = wptr.lock())
14     {
15         //Código será executado, pois sptr ainda não perdeu escopo
16         cout << *ptr << endl;
17     }
18 } //Fim do escopo, sptr e uptr são desalocados
19
20 //Testa se memória ainda não foi desalocada
21 if(auto ptr = wptr.lock())
22 {
23     //Não será executado, pois sptr perdeu escopo
24     cout << *ptr << endl;
25 }
```

Algoritmo 14: Utilizando *smart pointers*

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 2.3, 2.4 e 2.5, onde temos uma janela de um programa sendo exibida nos três sistemas suportados da plataforma *desktop*.

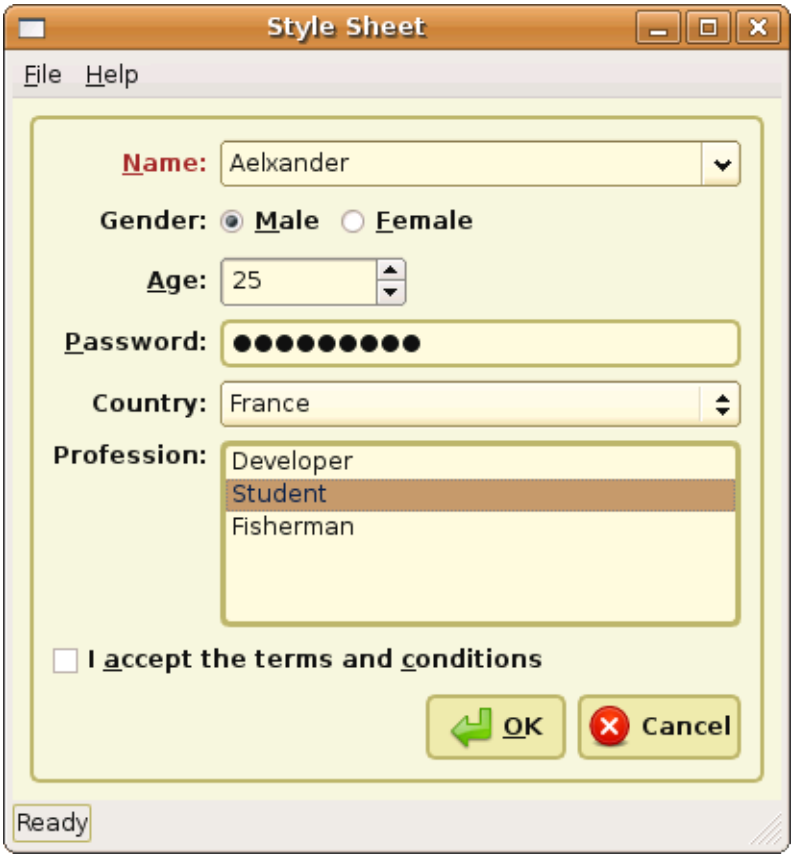

Figura 2.3: Janela no GNU/Linux

Inicialmente o *framework* era totalmente focado na padronização do processo de criação de interface gráfica entre diferentes plataformas, porém ao longo de sua existência

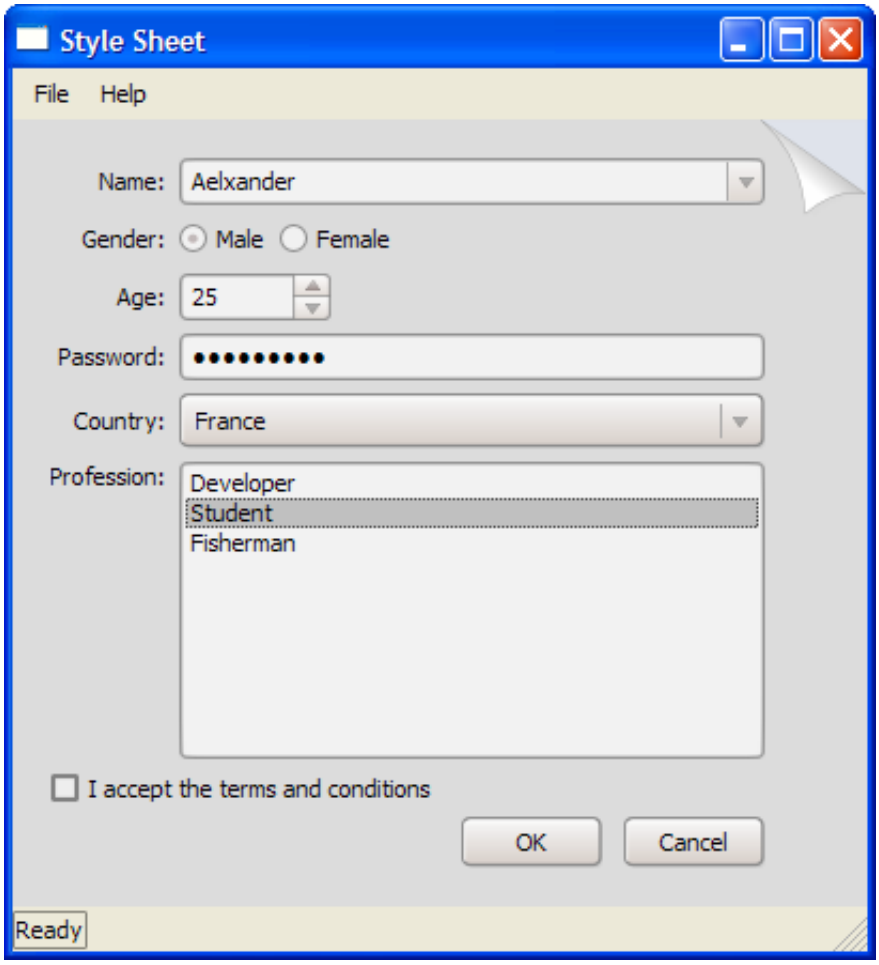

Figura 2.4: Janela no Microsoft Windows XP

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 de forma que o desenvolvedor possa 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].

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 nas plataformas **Microsoft Windows** e **GNU/Linux**.

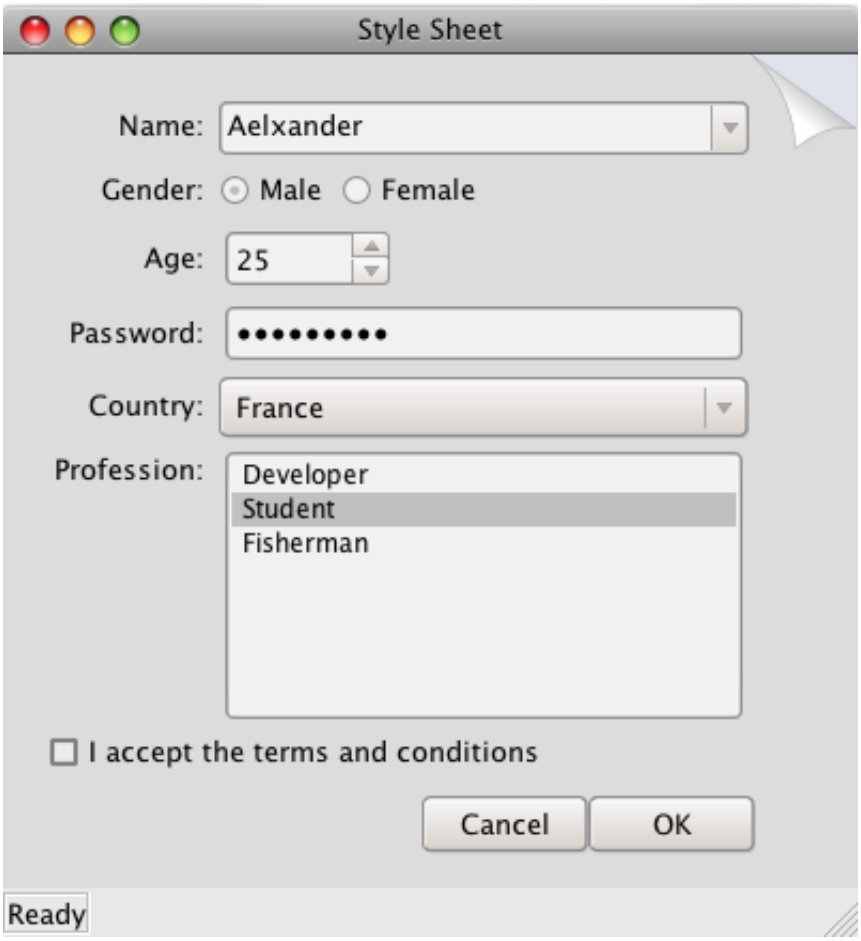

Figura 2.5: Janela no Mac Os

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. E, por fim, 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***[10] [Thelin, 2007].

No momento da criação deste documento, o framework está em sua versão 5.3 (versão que é utilizada no trabalho), 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*.

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

2.4 Sistemas Gerenciadores de Bancos de Dados Relacionais

Assim como as linguagens orientadas a objetos são consideradas um padrão para desenvolvimento de *software*, os Sistemas Gerenciadores de Bancos de Dados Relacionais ou **SGBD**s são considerados um padrão para armazenamento estruturado de informações [Barnes, 2007].

Os SGBDs utilizam um modelo de persistência denominado **modelo relacional**, 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, pode ser 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, em inglês, ***"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, em inglês, ***"foreign key"*** [Barnes, 2007]. Na figura 2.6 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".

Uma das funcionalidades mais importantes apresentada pelos SGBDs é a capacidade de aplicação de rotinas complexas de pesquisa usando a linguagem **SQL**. 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).

Figura 2.6: Exemplo de definição de tabelas e relações

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 de SGBDs com Linguagens Orientadas a Objetos

A princípio, os programas trabalham com dados em memória principal, que é volátil. Isto é, 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 maneiras diferentes, 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 e sua respectiva tabela, onde os atributos da classe são mapeados para colunas de uma tabela. Porém quando começamos a utilizar mecanismos mais avançados da 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***[11] 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 SGDB.

No trecho de código 15 temos um esboço de como estas etapas são realizadas utilizando a linguagem C++ em conjunto com o módulo QtSql 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 15: Exemplo de comunicação com SGBD utilizando o módulo QtSql

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, além

[11] *Application Programming Interface*. Define uma interface de um conjunto de bibliotecas de *software*.

do fato de que 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*, tornando, em alguns casos, inviável a mudança de SGBD [Barnes, 2007].

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

2.7 Bibliotecas de Mapeamento Objeto Relacional

As bibliotecas ORM têm como objetivo principal automatizar as tarefas relacionadas com a transição de informações entre os contextos orientado a objetos e 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:

1) 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;

2) 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;

3) 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 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:

1) 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[12] 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].

2) Orientado à 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].

3) Orientado ao banco de dados: Neste paradigma o desenvolvedor deve primeiramente criar o banco de dados. Então a biblioteca ORM auxiliada por metadados

[12]Informações complementares sobre outro conjunto de dados. Podem ser utilizados, por exemplo, para descrever a estrutura destes.

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 persitência [Barnes, 2007].

Existe ainda o conceito de **transparência**, que é aplicado a bibliotecas que utilizam o paradigma orientado à 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 são apresentadas algumas bibliotecas ORM presentes no mercado e um trabalho que propôs o desenvolvimento de uma biblioteca ORM para a linguagem C++. O modelo de uso e desenvolvimento destas bibliotecas serviram como inspiração para a construção deste trabalho. E duas das bibliotecas citadas são utilizadas em testes de comparação de usabilidade no capítulo 5.

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++", originalmente em inglês, *"A Framework for Object-Relational Mapping With An Example in C++"*, *Zhang Xiaobing* apresenta um modelo para desenvolvimento de uma biblioteca ORM que pode ser utilizado na maioria das linguagens orientadas a objetos. 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:

1) 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.

2) 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

QxORM é uma biblioteca ORM de código aberto desenvolvida para ser utilizada em conjunto com o *framework* Qt. Ela utiliza vários módulos deste *framework* como auxílio na execução das tarefas de mapeamento, como por exemplo, o módulo QtSql para realizar interações com os SGBDs. Ela foi criada em 2003 e é mantida pelo engenheiro de *software* Lionel Marty [Marty, 2014].

Esta biblioteca 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 se-

rem mapeados. Portanto, nesta biblioteca também não existe um sistema de anotações [Marty, 2014].

Esta biblioteca é utilizada durante testes de usabilidade, onde é comparada a sua abordagem de especificação de mapeamento com a interface de anotações criada neste trabalho. Além de testes de usabilidade, também será feita a comparação do processo de configuração do ambiente de desenvolvimento para utilização desta biblioteca com a desenvolvida neste trabalho.

3.3 ODB

ODB é uma biblioteca ORM desenvolvida para ser utilizada com a linguagem C++. Também é uma biblioteca de código aberto e é mantida pela empresa "Code Synthesis" [Synthesis, 2013].

Ao contrário da biblioteca QxOrm, a ODB não foi desenvolvida para ser utilizada especificamente em conjunto com um *framework* ou outra biblioteca, mas sim com a linguagem C++ pura. Para ser utilizada em conjunto com o *framework* Qt, por exemplo, a biblioteca disponibiliza um recurso chamado ***"profile"***, que atua como uma espécie de *"plugin"* adicional que pode ser acoplado a biblioteca para habilitar seu uso em conjunto [Synthesis, 2013].

Mesmo quando utilizada com *"profiles"*, a biblioteca utiliza mecanismos próprios para acesso a banco de dados e construção de sentenças em linguagem SQL. As informações de mapeamento são especificadas através do uso de diretivas de pré-processamento ***"pragma"***[1]. Estas informações são analisadas por um compilador disponibilizado em conjunto com a biblioteca, que gera então código em C++ com os comandos para realizar tarefas de persistência [Synthesis, 2013].

A biblioteca utiliza o paradigma orientado a aplicação com uma abordagem transparente, pois não obriga o desenvolvedor a implementar interfaces ou aplicar herança nas classes a serem mapeadas para o banco de dados. Seu sistema de inserção de metadados é bem parecido com o sistema de anotações, porém estes metadados somente são acessíveis pelo seu compilador externo [Synthesis, 2013].

Esta biblioteca também é utilizada durante testes de usabilidade e de configuração do ambiente de desenvolvimento no capítulo 5.

[1] Diretiva de pré-processamento que permite ao desenvolvedor passar informações durante a compilação que são específicas de um compilador utilizado.

3.4 Hibernate

Uma das bibliotecas ORM mais conhecidas atualmente, o Hibernate é uma biblioteca desenvolvida para ser utilizada com a linguagem JAVA. É um projeto de código aberto mantido pela empresa **Red Hat** [Bauer and King, 2005].

O Hibernate é uma das bibliotecas ORM mais completas, sendo pioneira na implementação de vários mecanismos de otimização. A biblioteca utiliza um paradigma orientado a aplicação e a inserção de metadados de mapeamento pode ser feita a partir da utilização de arquivos XML, ou a partir da utilização do mecanismo de anotações presente na linguagem JAVA [Bauer and King, 2005].

A biblioteca é capaz de mapear classes que utilizam mecanismos de herança e composição, além de ter a capacidade de a partir da análise das classes mapeadas e dos metadados, criar o banco de dados equivalente [Bauer and King, 2005].

Seu mecanismo de anotações é a inspiração principal do desenvolvimento deste trabalho, pois tal mecanismo traz um nível de facilidade enorme na configuração dos dados de mapeamento, e até então poucas bibliotecas voltadas para a linguagem C++ apresentam algo semelhante.

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 utiliza o paradigma orientado a aplicação e a abordagem transparente para definição das classes mapeadas. A interface de configuração do mapeamento é 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 desenvolvida é capaz

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

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 da orientação a objetos.

Das bibliotecas ORM existentes para o cenário abordado, a QxOrm e a ODB são as que mais se aproximam das características citadas, portanto elas são utilizada em testes ao final do desenvolvimento que comparam 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 objeto** ou ***"Object Layer"*** e a **camada de armazenamento** ou ***"Storage Layer"***, seguindo a nomenclatura utilizada no trabalho desenvolvido por Zhang Xiaobing. As duas camadas oferecem interfaces acessíveis diretamente pelo desenvolvedor e cooperam entre si através de troca de informações.

A camada objeto tem como objetivo apresentar uma interface transparente para o desenvolvedor que permita a configuração das classes a serem mapeadas e apresentar 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 apresentar 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 é implementada para o SGBD PostgreSQL, e utiliza 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 são detalhadas em conjunto com os

mecanismos específicos envolvidos no seu desenvolvimento.

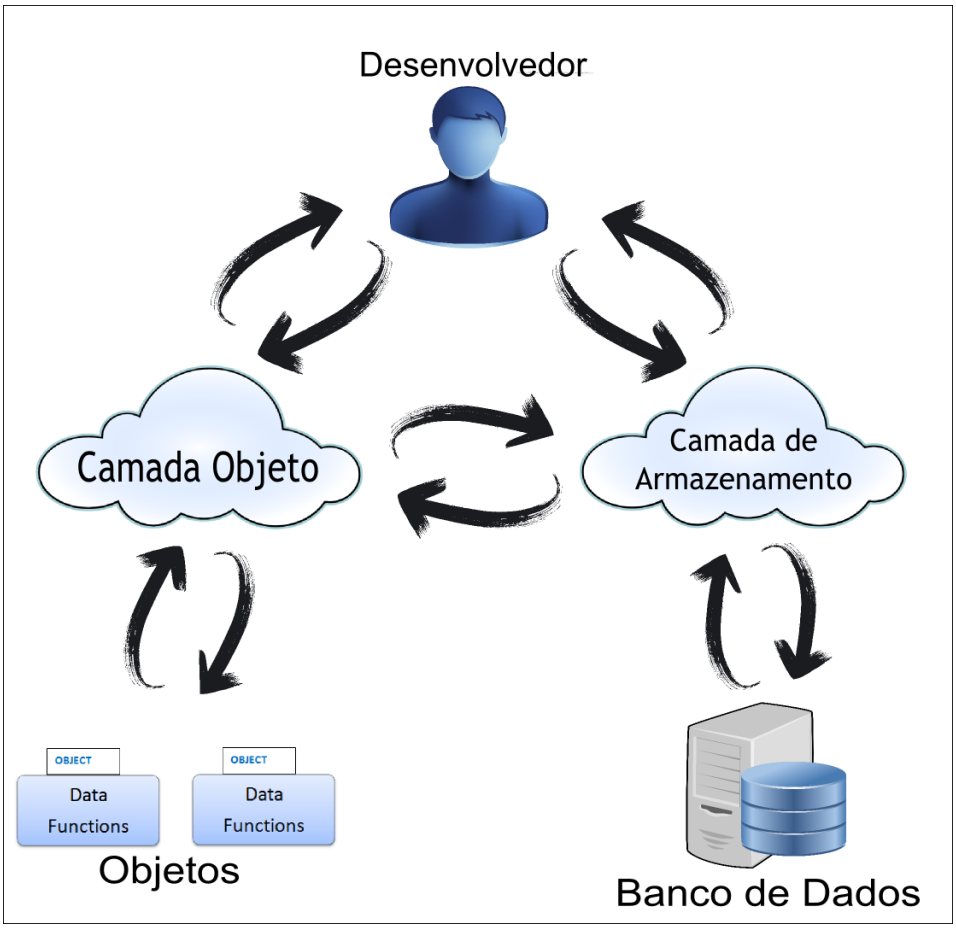

Figura 4.1: Interação entre componentes do software e desenvolvedor

4.2 Camada 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 16 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 16: Obtendo o nome da classe de um objeto em tempo de execução

A camada objeto utiliza dois mecanismos para contornar estes limitadores, os quais são 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"*** na declaração de métodos de leitura.

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 17 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 17: 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 18 temos um exemplo de uma classe com um 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

```
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 18: Retornando uma expressão lambda para acesso de atributo privado

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 foi 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 citado 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.

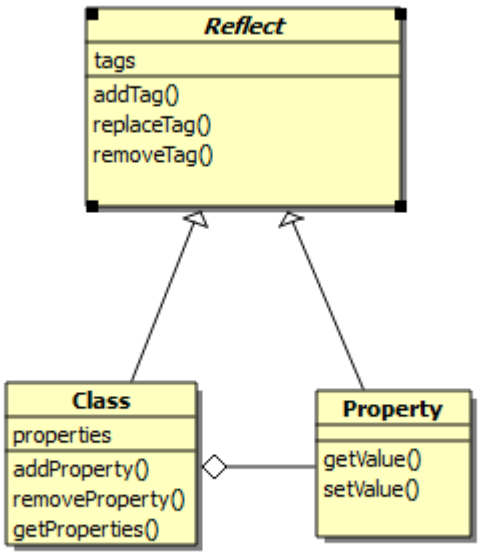

Figura 4.2: Diagrama simplificado das classes de reflexão

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 19 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 19: Esboço da utilização de macros para registro de metainformação

Com este mecanismo definido foi possível implementar as responsabilidades da camada objeto. Nas próximas seções são detalhados os mecanismos utilizados para implementação da camada de armazenamento.

4.3 Camada de Armazenamento

O objetivo principal de uma biblioteca ORM é abstrair do desenvolvedor a criação dos comandos SQL para executar as tarefas de persistência, bem como a comunicação com o banco de dados. A camada de armazenamento alcança este cenário a partir da utilização de duas classes. A primeira é a ***"Repository"***, que disponibiliza em sua interface métodos para salvar, atualizar, deletar e carregar registros de objetos no banco de dados. Esta classe oferece a possibilidade do uso de transações para garantir que um grupo de operações seja executado de forma atômica. Ela também tem a responsabilidade de gerenciar a comunicação com o banco de dados, o que é feito através da utilização da API disponibilizada pelo módulo QtSQL oferecido pelo *framework* Qt.

A segunda classe é a ***"SQLProvider"*** que define uma interface para geração de co-

mandos em linguagem SQL para a execução das tarefas de persistência de objetos. Ela utiliza os objetos de reflexão disponibilizados pela camada objeto para construir sentenças de acordo com a instância de objeto a ser persistida. Esta interface deve ser implementada para cada tipo de SGBD que se deseja utilizar, desta forma a adição de suporte da biblioteca para diversos SGBDs se torna uma tarefa mais simples. A classe *"Repository"* utiliza uma implementação desta interface para gerar os comandos necessários para execução de suas tarefas.

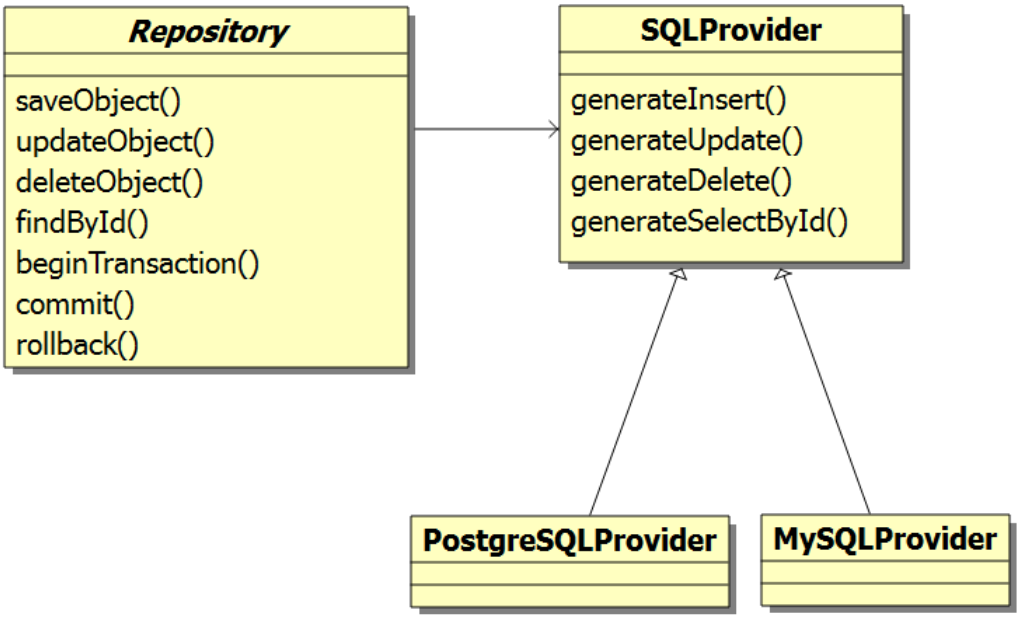

Figura 4.3: Diagrama de classes simplificado da camada de armazenamento

Na figura 4.3 temos um esboço do diagrama de classes da camada de Armazenamento, onde temos a representação de duas possíveis implementações da interface *"SQLProvider"*, uma que daria suporte ao SGBD PostgreSQL e outra ao MySQL. A camada de armazenamento é menos complexa pelo fato de utilizar a própria camada objeto e o módulo QtSql para facilitar sua implementação, porém ela também é responsável por tratar os erros que podem ocorrer durante a comunicação com o banco de dados ou execução de comandos SQL, caso em que ela deve retornar mensagens bem formatadas descrevendo os erros e oferecer mecanismos para geração de logs.

Capítulo 5

Cronograma de Atividades

Segue tabela com a representação do cronograma de atividades relacionadas com o desenvolvimento do trabalho.

Etapas	Fev	Mar	Abr	Mai	Jun	Jul	Ago	Set	Out	Nov
1) Estudo do problema	[x]	[x]								
2) Estudo de trabalhos relacionados		[x]	[x]	[x]						
3) Planejamento e Implementação						[x]	[x]	[x]		
4) Testes								[x]	[x]	[x]
5) Análise de testes								[x]	[x]	[x]
6) Redação da monografia		[x]	[x]	[x]	[x]	[x]	[x]	[x]	[x]	[x]

Figura 5.1: Cronograma de atividades

Capítulo 6

Conclusão

Nesta primeira etapa do trabalho, o objetivo foi conhecer melhor os conceitos envolvidos no problema sendo estudado. Conceitos como linguagens orientadas a objetos e sistemas gerenciadores de bancos de dados relacionais foram estudados a partir da análise de trabalhos específicos sobre os mesmos. A partir do conhecimento adquirido, foi possível definir uma metodologia ou abordagem a utilizar para o desenvolvimento da biblioteca ORM.

Na próxima etapa, a biblioteca será implementada seguindo as premissas aqui definidas. Serão utilizados dois ambientes de desenvolvimento para garantir o funcionamento em múltiplas plataformas e compiladores. O primeiro ambiente é baseado no sistema operacional *"Microsoft Windows 8.1"*, onde será utilizado o compilador de C++ que acompanha o Visual Studio 2013. O segundo ambiente é baseado no sistema operacional *Ubuntu 14.04 GNU/Linux*, onde será utilizado o compilador G++ versão 4.8. Em ambos os ambientes será utilizada a versão 5.3.0 do framework Qt, que no momento de criação deste documento é a versão mais recente.

O SGBD que será suportado pela primeira implementação da biblioteca é o PostgreSQL, onde será utilizado a versão 9.3.4 instalada em ambos os ambientes citados. Ao fim do desenvolvimento serão feitos testes para comparar o nível de usabilidade da biblioteca por desenvolvedores com pouca experiência em C++ e a facilidade em migrar códigos que não utilizem bibliotecas ORM. A biblioteca QxORM será utilizada como padrão de comparação, por se tratar de um componente bastante estável e utilizado atualmente em soluções desenvolvidas utilizando o framework Qt.

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, 2014] Brokken, F. B. and Kubat, K. (2014). *C++ annotations*. Citeseer.

[Bueno, 2002] Bueno, A. (2002). Apostila de programação orientada a objeto em c++.

[Clugston, 2004] Clugston, D. (2004). Member function pointers and the fastest possible c++ delegates. *Online Article*.

[Gregoire, 2014] Gregoire, M. (2014). *Professional C++*, volume 3. WROX.

[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*.

[Synthesis, 2013] Synthesis, C. (2013). *C++ Object Persistence with ODB*.

[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.