Utilizando Design Patterns GoF no apoio ao desenvolvimento de um Framework Java

Marcelo Torres de Albuquerque
IME/UERJ
torres.marcelo@gmail.com

Alexandre Rojas
PET/COPPE
rojas@pet.coppe.ufrj.br

Paulo Cezar M. Ribeiro
PET/COPPE
pribeiro@pet.coppe.ufrj.br

Resumo

Este trabalho consiste no desenvolvimento de um framework de acesso a banco de dados em Java baseado na análise de 6 dos 23 padrões de projeto GoF, catalogados na obra 'Design Patterns: Elements of Reusable Object Oriented Software' [1]. Cada padrão de projeto utilizado é analisado de forma em que é proposta uma hierarquia de classes típica, que pode ser aplicada a um problema específico e recorrente, observado em cenários de projetos orientados a objetos como o apresentado.

O objetivo do framework implementado é basicamente providenciar um meio simples e flexível de se mover um dado entre um objeto Java e um banco de dados relacional. Utilizando-se dos recursos da SQL descritos em arquivos XML, sem praticamente usar linhas de código JDBC, que, em grande parte das vezes é bastante prolixo. Em suma, visa-se a implementação de um framework que agilize e padronize a utilização de bancos de dados relacionais por aplicações Java.

Os Design Patterns explicados e exemplificados serão apenas os aplicados segundo as necessidades do desenvolvimento deste framework para que se evite a verborragia neste trabalho. Os exemplos apresentados estarão focados na tecnologia utilizada para o desenvolvimento do mesmo, no caso Java da Sun® (Mais especificamente a versão 6). Mesmo sabendo que estes conceitos podem ser aplicados a outras plataformas de desenvolvimento, esta possível aplicabilidade não fez parte do escopo deste trabalho.

Abstract

This work consists on the development in Java of a framework for accesing databases based on the analysis of 6 of the 23 GoF design patterns cataloged in the book 'Design Patterns: Elements of Reusable Object Oriented Software' [1]. Each design pattern used is analyzed so that it proposes a typical hierarchy, which can be applied to a specific problem, observed in scenarios of object-oriented projects as presented.

The objective of the implemented framework is primarily to provide a simple and flexible option to move data between an object given a Java and a relational database. Using the resources of SQL described in XML files, using almost none JDBC lines of code, that in most of the time is quite long winded. It seeks to implement a framework to streamline and standardize the use of relational databases in Java applications.

The Design Patterns explained and exemplified are only applied in accordance with the needs of the development of this framework in order to avoid the verbiage in this work. The examples presented will focus on technology used on this development, in this case, Sun's Java© (More specifically the version 6). Even though these concepts can be applied to other development platforms, this application is not part of the scope of this work.

1. Introdução

Padrões de Projeto (Design Patterns, em inglês) se tornaram conhecidos primordialmente pela grande aceitação do livro Design Patterns: Elements of Reusable Object-Oriented Software by Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides (frequentemente chamados de "Gangue dos Quatro" - Gang of Four ou apenas GoF). Hoje em dia, sua utilização é muito presente na arquitetura e design de softwares. Neste trabalho será utilizada também a expressão Design Patterns por se tratar do jargão usual empregado pelos especialistas na área.

Design Patterns e frameworks são ambos técnicas de reuso de software. A aplicação de Design Patterns constitui poderosa ferramenta para a criação de aplicações robustas que utilizam soluções já testadas, procurando minimizar o impacto de alterações durante o ciclo de vida do framework, explorando o baixo acoplamento entre os elementos de software do sistema. A aplicação de Design Patterns tem conseqüência direta na arquitetura da solução e promove o reuso de software.

O projeto foi decorrente da necessidade dos autores foi acessar base de dados já existentes relacionados a Sistemas Inteligentes de Transportes–ITS originados por Órgãos de Transito diversos, já em produção. Assim, utilizar JDBC de forma direta foi a opção que mais se adequava à necessidade, no entanto sua utilização é pouco produtiva, devido à complexidade para a execução de cada comando, e de difícil manutenção pois torna o código desorganizado, de forma que a lógica de negócio não fica separada do acesso aos dados.

Este trabalho baseia-se na aplicação dos padrões GoF direcionados à implementação de um framework em Java de acesso à banco de dados relacionais, batizado de dinSQL que soluciona todos os problemas citados acima, de modo que dinamiza e saneia a utilização de código SQL por aplicações Java.

Os principais objetivos deste framework são:

- **Padronização**: Todos os comandos de aceso a dados são definidos em arquivos XML (descritores);
- **Reaproveitamento** de comandos SQL;
- **Facilitação** do uso de SQL via Java, visto que torna transparente a utilização dos comandos JDBC;

2. Design Patterns

Trazendo os padrões ao mundo do software, é possível defini-los como: "Um padrão é uma idéia que tem sido útil em um contexto prático e que provavelmente será útil em outros." [2]

Os padrões de projeto tornam mais fácil reutilizar projetos e arquiteturas bem sucedidas. Expressar técnicas testadas e aprovadas as torna mais acessíveis para os desenvolvedores de novos sistemas. Os padrões de projeto ajudam a escolher alternativas de projeto que tornam um sistema reutilizável, evitando alternativas que comprometam esta reutilização. E podem melhorar a documentação e a manutenção de sistema ao fornecer uma especificação explícita de interações de classes e objetos e o seu objetivo subjacente. Em suma, ajudam um projetista a obter mais rapidamente um projeto adequado [1].

De um modo mais simples, um bom padrão é um par problema/solução denominado e bem conhecido que pode ser aplicado em novos contextos, com conselhos sobre como aplicá-lo em situações novas e discussão sobre seus compromissos, implementações, variações, etc. [3]. Dentro de um padrão sempre existe a solução para um problema num determinado contexto.

Cada pessoa tem a sua interpretação do que é um padrão. O que pode ser considerado um padrão para uma pessoa pode ser uma algo totalmente diferente para outra. Neste trabalho focamos os padrões em um certo nível de abstração, definindo padrões de projeto como sendo "descrições de objetos e classes comunicantes que precisam ser personalizadas para resolver um problema geral de projeto num contexto particular." [1]

2.1. Os padrões GoF

Considerado pela maioria dos especialistas como a "Bíblia" dos livros de padrões de projeto, o livro supracitado cataloga 23 padrões, chamados de padrões **GoF** (Gang of Four), ou padrões da Gangue dos Quatro. A partir desta publicação, o conceito de padrões tem sido estendido e aplicado a diversas áreas do desenvolvimento de sistemas orientados a objetos. "Cada padrão de projeto sistematicamente nomeia, explica e avalia um aspecto de projeto importante e recorrente em sistemas OO." [1]

Os padrões GoF são organizados em três grandes famílias, apresentadas na tabela 2.1: **padrões de criação**, relacionados à criação de objetos; **padrões estruturais**, que tratam das associações entre classes e objetos; e **padrões comportamentais**, responsáveis pelas interações e divisões de responsabilidades entre as classes ou objetos.

Tabela1. **Classificação dos padrões GoF**

		Propósito (família)		
		De criação	**Estrutural**	**Comportamental**
Escopo	**Classe**	**Factory Method**	Adapter (class)	Interpreter **Template Method**
	Objeto	Abstract Factory **Builder** Prototype **Singleton**	Adapter (object) Bridge Composite Decorator Facade Flyweight **Proxy**	Chain of Responsibility Command **Iterator** Mediator Memento Observer State Strategy Visitor

Os padrões GoF também podem ser classificados segundo o seu escopo. Os padrões com **escopo de classes** lidam com os relacionamentos entre classes e suas subclasses, através de herança e em tempo de compilação, sendo assim mais estáticos. Nos padrões com **escopo de objeto**, o relacionamento entre objetos pode ser modificado em tempo de execução, sendo mais dinâmicos. Praticamente as duas classificações utilizam herança em certa medida. A maioria dos padrões classificados na tabela 2.1 está no escopo de objeto.

2.1.1. Factory Method

Muitas aplicações desenvolvidas com linguagens orientadas a objetos fazem uso deste padrão, também conhecido como *virtual constructor*. Basicamente, o padrão *factory method* consiste na definição de uma interface comum para a criação de objetos, deixando que suas subclasses decidam que classe instanciar. Desta forma, somente as subclasses serão responsáveis pelas instanciações solicitadas por um cliente, através da implementação de um método abstrato de criação especificado na interface comum.

O padrão *factory method* consiste na definição de uma interface comum para a criação de objetos, deixando que suas subclasses decidam que classe instanciar. Desta forma, somente as subclasses serão responsáveis pelas instanciações solicitadas por um cliente, através da implementação de um método abstrato de criação especificado na interface comum.

A estrutura básica do *factory method* é definida com apenas quatro participantes, como ilustrado na figura 2.2. As interfaces **Creator** e **Product** formam a base desta estrutura, também composta pelas respectivas implementações, citadas a seguir:

- **Product**: interface comum dos produtos a serem criados;
- **ConcreteProduct**: implementa a interface Product, definindo produtos concretos;
- **Creator**: classe abstrata; declara a operação (abstrata) factoryMethod, que retorna um objeto

Product. Pode definir uma implementação padrão de factoryMethod, que retorna um ConcreteProduct específico. Creator também pode definir alguns métodos que invocam factoryMethod;

- **ConcreteCreator**: estende a classe Creator, redefinindo factoryMethod para retornar uma instância de um ConcreteProduct criado.

Utilizou-se o padrão *factory method* para criar os objetos SQLDescriptor descritores dos XML do SQL em questão, cuja estrutura de classes é ilustrada na figura 2.3.

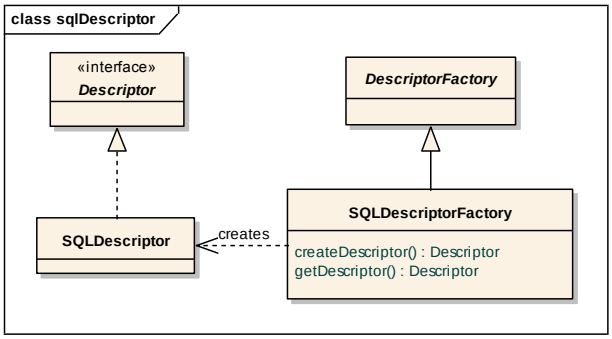

Figura 1. **Estrutura simplificada do exemplo da criação do SQLDescriptor**

O cliente solicita a criação de um determinado objeto informando apenas o nome do descritor XML, sem especificar a sua classe. A classe **SQLDescriptorFactory** implementa o método **getDescriptor**, que instancia o descritor XML específico. O cliente então utiliza o objeto criado, desconhecendo a sua classe específica.

2.1.2. Builder

Algumas aplicações possuem a necessidade da construção de objetos complexos de uma forma simples, invocando-se apenas uma única operação. O cliente apenas solicita a um "construtor" a instanciação de um determinado produto com base em alguns parâmetros. Os detalhes da construção de objetos complexos, que possivelmente inclui várias etapas na sua implementação, estarão ocultados do cliente.

O padrão *builder* possui como principal objetivo encapsular o algoritmo de construção de objetos complexos. O processo de criação não é visto pelo cliente, que desconhece as partes componentes do objeto e como é o processo de construção. Este padrão também permite a construção de diferentes representações para o mesmo objeto complexo.

A estrutura do padrão *builder* possui: uma interface ou classe abstrata, uma ou mais classes para a construção efetiva do produto, o produto a ser construído, e um cliente que solicita a criação do objeto. São presentes os seguintes participantes:

- **Builder:** interface ou classe abstrata que contempla todos os métodos necessários para criação de um objeto **Product**.
- **ConcreteBuilder:** implementa a interface de **Builder** para a criação de um objeto **Product** concreto. Oferece o método **getResult** para a recuperação do produto construído. Várias classes **ConcreteBuilder** podem ser definidas para diversos tipos de produtos diferentes.
- **Director:** constrói um objeto complexo, invocando sucessivamente as várias operações **buildPart** de um **ConcreteBuilder**, declaradas em **Builder**.
- **Product:** objeto criado com base nos métodos da classe **Builder**. É a classe **ConcreteBuilder** quem constrói este objeto, implementando métodos **buidPart** para cada etapa de sua criação.

A classe ConcreteBuilder conterá a implementação do algoritmo e do processo de construção de cada objeto complexo. Para cada tipo de produto é necessária uma classe ConcreteBuilder com os seus algoritmos e processos de construção. Como será herdada por todos os ConcreteBuilder, a interface de Builder deverá ser o mais abrangente possível, permitido a construção de todos os tipos de objetos complexos.

Caso necessário, alguns métodos podem ser omitidos ou implementados com um comportamento padrão. Ao herdar estes comportamentos, cada ConcreteBuilder deverá implementar apenas os métodos específicos para a construção de seus produtos.

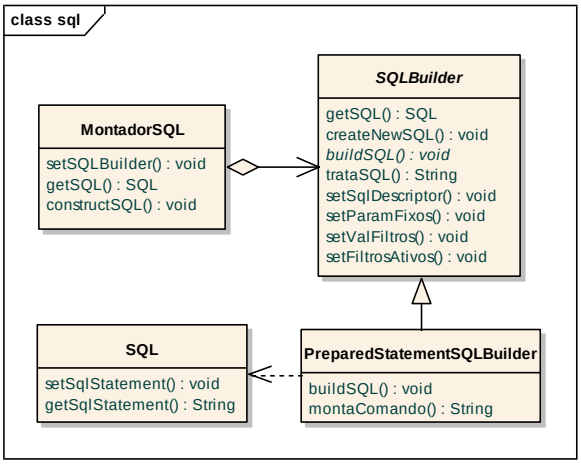

Figura 2. **Estrutura simplificada da implementação do padrão Builder**

No framework implementado, a classe PreparedStatementSQLBuilder, o ConcreteBuilder do exemplo, tem a função de manipular o objeto SQLDescriptor de forma a construir a string com o a instrução SQL final que será executada. Esta instrução é montada de forma que seja obrigatório se executar um Prepared Statement.

Se a diretriz do desenvolvimento for, por exemplo, utilizar Statement, e não Prepared Statement, apenas é necessário implementar um novo ConcreteBuilder, no caso StatementSQLBuilder.

A montagem de um comando SQL não varia de acordo com a implementação do statement. A classe SQLBuilder, ilustrada a seguir, representa o builder, contendo todos os métodos necessários para a montagem

de um comando SQL. Por ser uma classe abstrata, alguns métodos são definidos com uma implementação padrão: subclasses de SQLBuilder não precisarão implementar alguns destes algoritmos, como, por exemplo a adição da cláusula "WHERE" onde for necessário.

A implementação do montador do SQL é simples, como está sendo utilizado Prepared Statement apenas concatenar-se-ão as partes do comando, pois os valores são atribuídos aos "?" apenas em tempo de execução.

Ilustramos a chamada do diretor com o pequeno trecho de código abaixo, retirado da classe SQLExecutor explicada mais à frente neste trabalho:

```
MontadorSQL montador = new MontadorSQL();
SQLBuilder sqlBuilder =
                new
PreparedStatementSQLBuilder();

sqlBuilder.setSqlDescriptor(sqlDescriptor);
sqlBuilder.setFiltrosAtivos(filtrosAtivos);

montador.setSQLBuilder(sqlBuilder);

montador.constructSQL();
       //A variável sqlStatement recebe a
string com o comando
this.sqlStatement = montador

.getSQL().getSqlStatement();
```

2.1.3. Singleton

Algumas aplicações requerem classes que sejam instanciadas uma única vez. Como exemplo, um determinado ambiente computacional pode ter vários objetos da classe *printer*; porém apenas um único objeto *spooler* controla a fila de impressão do sistema.

O padrão singleton é uma estrutura que define classes que serão instanciadas uma única vez. Classes como *spooler*, *window manager*, *file system*, etc., são implementadas desta forma. Tais classes devem prover um mecanismo de controle para a criação de uma instância única, oferecendo acesso global ao único objeto criado, bem como permitir uma fácil especialização em subclasses.

A estrutura básica do padrão *singleton* é bem simples. O único participante é a própria classe **Singleton**:

- **Singleton**: classe com uma instância única, referenciada por **uniqueInstance**, atributo privado de classe (estático). A operação de classe **getInstance** provê o acesso global ao objeto, que deve estar disponível em diferentes pontos do sistema.

A implementação do *singleton* não deve permitir a instanciação direta de objetos através de declarações de variáveis. Em Java isto pode ser implementado declarando-se um construtor *default* (o único disponível) como inacessível, no caso, adicionando-se o modificador private. Isto acarreta erro de compilação em classes que tentem a instanciação direta dos objetos s*ingleton*.

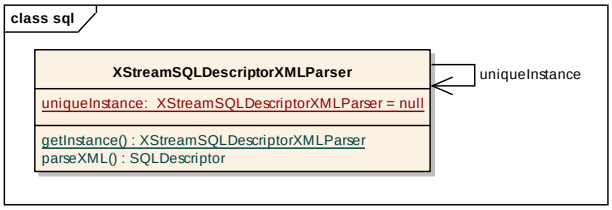

Figura 3. **Classe de implementação do *Singleton***

Todos os descritores XML do framework implementado seguem um padrão de formatação. Para efetuar o *parsing* deste tipo de arquivo XML, ou seja, para ler o descritor XML, utilizou-se o framework XStream, onde arquivos XML iguais sempre utilizam um mesmo objeto do tipo XStream, ou seja, não é necessário criar-se mais de uma instância desse mesmo objeto não só pois não é necessário logicamente, mas também para poupar recursos do computador onde esse framework funcionará.

2.1.4. Proxy

O padrão *proxy* também é conhecido como *surrogate* (substituto), pois fornece um substituto ou representante de outro objeto, seja ele um objeto remoto, um recurso oneroso, ou algo que exija segurança. Este padrão consiste em definir uma referência mais sofisticada do que um simples ponteiro para o objeto real.

Os participantes deste padrão e suas respectivas responsabilidades são:

- **Proxy**: mantém uma referência (atributo **realSubject**) ao objeto real, podendo encaminhar as solicitações a ele sempre que necessário. Se as interfaces de **Subject** e **RealSubject** forem idênticas, o atributo **realSubject** pode referenciar um objeto da classe **Subject**. A classe **Proxy** declara a mesma interface de **Subject**, permitindo que o *proxy* substitua um objeto **RealSubject**. Além de controlar o acesso ao objeto real, um *proxy* também pode ser responsabilizado pela sua criação e destruição.
- **Subject**: interface comum a ser implementada pelas classes **Proxy** e **RealSubject**, permitindo que qualquer cliente trate o *proxy* como se fosse um objeto real.
- **RealSubject**: o objeto real geralmente é o objeto que executa a maior parte do trabalho real; o *proxy* controla o acesso a ele.

Foi adaptado o padrão *proxy* com um mecanismo de gerência de tamanho do *cache* dos descritores já lidos, visando controlar a quantidade de descritores já "parseados", de acordo com o acordo com quantidade de objetos armazenados e a freqüência com cada um é utilizado. O objetivo é definir uma classe funcionalmente equivalente à **java.util.Hashtable**, adicionando o tratamento descrito. A figura 2.## ilustra a solução proposta.

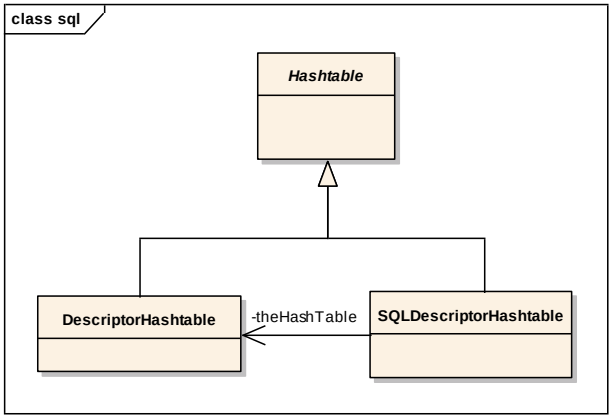

Figura 4. **Modelo do funcionamento da hashtable proposta**

Na solução proposta, a classe **Hashtable** será estendida pelas classes SQLDescriptorHashtable e DescriptorHashtable. SQLDescriptorHashtable atua como um proxy que controla o acesso a objetos da classe DescriptorHashtable(**RealSubject**), subclasse de java.util.Hashtable (**Subject**).

Quando os métodos **put** e **get** da Hashtable forem chamados, o tratamento de cache entra em ação. Verificando, de tempos em tempos, quando os descritores serão excluídos da memória, de acordo com o número de descritores persistidos depois de um certo tempo(timeout definido). Ou seja, seja **x** o número ótimo de descritores alocados em memória e **y** o número de descritores em memória, ao serem executados os métodos supracitados, serão excluídos os **y – x** descritores executados há mais tempo.

2.1.5. Template Method

Um *template method* (**método-template** ou **método-modelo**) implementa um algoritmo decompondo-o em operações abstratas, que devem ser implementadas por subclasses concretas de acordo com seu comportamento específico.

Apresentam-se abaixo os participantes deste padrão e suas respectivas responsabilidades:

- **AbstractClass** (Application): classe abstrata "modelo" que define uma interface com operações primitivas (abstratas), cujas implementações são de responsabilidade dos herdeiros (**ConcreteClass**). Na superclasse podemos implementar métodos comuns a todas as subclasses (métodos-modelo), que também podem ser sobrecarregados. Estes métodos podem invocar operações primitivas, bem como operações definidas em AbstractClass ou ainda em outros objetos.
- **ConcreteClass** (MyApplication): classe concreta que implementa as operações primitivas contendo passos específicos do algoritmo da subclasse. Depende de AbstractClass para variar partes de um algoritmo.

Os métodos-template constituem uma técnica fundamental para reuso de código, particularmente importante em bibliotecas de classes, porque são meios para decompor os comportamentos comuns, evitando duplicação de código. É necessário identificar as diferenças no código, para então separá-las em novas operações. A partir desta divisão, chama-se o método modelo na parte do código que necessita dessas novas operações.

Os métodos-template levam a uma estrutura de inversão de controle: uma superclasse chama as operações da subclasse, e não o contrário.

Foi utilizado o template method para efetuar a execução de instruções SQL (INSERT, DELETE, UPDATE, SELECT e execução de Stored Procedures) em uma aplicação, escrevendo uma solução genérica a qualquer comando para promover o reuso.

A classe mãe possui um esqueleto comum a qualquer tipo de comando SQL e ganchos para permitir que cada comando customize o seu processo. Podemos identificar as seguintes etapas:

1. Leitura do comando: Todo comando fica obrigatoriamente dentro em arquivos XML que seguem um mesmo padrão, e estes arquivos devem ser lidos de seus diretórios pré-configurados para posterior leitura, por parte do framework, dos mesmos;

2. Adição de parâmetros fixos: para o usuário adicionar parâmetros que serão introduzidos no comando incondicionalmente;

3. Adição de parâmetros dinâmicos: todos os comandos podem receber parâmetros que podem ser ou não aplicados ao comando de acordo com a necessidade do usuário;

4. Execução dos comandos: todos os comandos devem, necessariamente, ser executados;

No código de execução tanto dos comandos de INSERT, DELETE e UPDATE quanto dos comandos de select existiam diversar características congruentes, por isso, para se promover o reuso, ComandExecutor herda o comportamento de QueryExecutor, que, por sua vez, herda de SQLExecutor.

Estrutura de classes proposta:

Embora o processamento das etapas 1, 2 e 3 seja padrão, a lógica da etapa 4 pode variar muito entre diferentes comandos. E cada tipo de comando provém seu próprio código.

Neste exemplo, método **execute** é um método template.

2.1.6. Iterator

Um objeto que possua agregações deve permitir que seus elementos sejam acessados sem que sua estrutura interna seja exposta. Em geral, pode-se desejar que estes elementos sejam percorridos em várias ordens (de frente para trás, em ordem reversa, ou mesmo aleatória), sem modificar a interface da lista de acordo com o percurso.

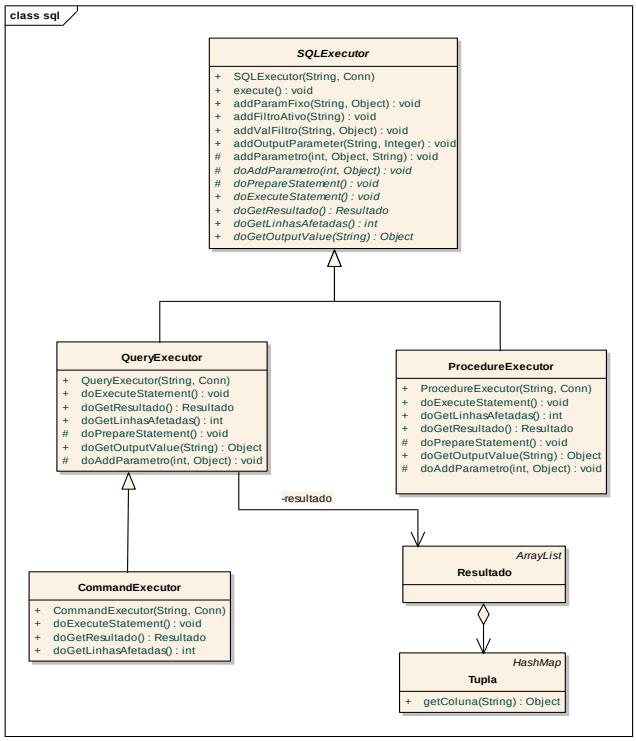

Figura 5. Classe de implementação do Template Method

Um exemplo de navegação tradicional em lista pode utilizar uma estrutura de repetição típica:

```
for i := 0 to (Class.Students.Count -1)
do
begin
        (...);
end;
```

A estrutura acima rompe o conceito de encapsulamento. A classe cliente sabe "muito" sobre o objeto **Class**, o que implica alto acoplamento, gerando uma dependência indesejada.

O padrão **iterator** (iterador) permite a navegação em uma coleção de diferentes maneiras através de funções (métodos), retirando do objeto **lista** a responsabilidade de acesso e percurso, transferindo-a para um objeto **iterator**. Esse objeto proverá o acesso à lista sem expor seu conteúdo, preservando o encapsulamento. A classe **Iterator** define uma interface uniforme para acesso aos diferentes elementos da coleção, ou seja, que suporte iteração polimórfica. Um iterator é responsável por manter a posição do elemento corrente, sabendo quantos elementos já foram percorridos.

A estrutura do padrão Iterator é composta pelos seguintes participantes:

- **Iterator**: define uma interface para o acesso e percurso entre elementos;
- **ConcreteIterator**: implementa a interface de Iterator. Mantém a informação sobre o elemento percorrido, podendo calcular qual o elemento seguinte;
- **Aggregate**: define uma interface para a criação de um objeto Iterator;
- **ConcreteAggregate**: implementa o método da interface que retorna uma instância de ConcreteIterator.

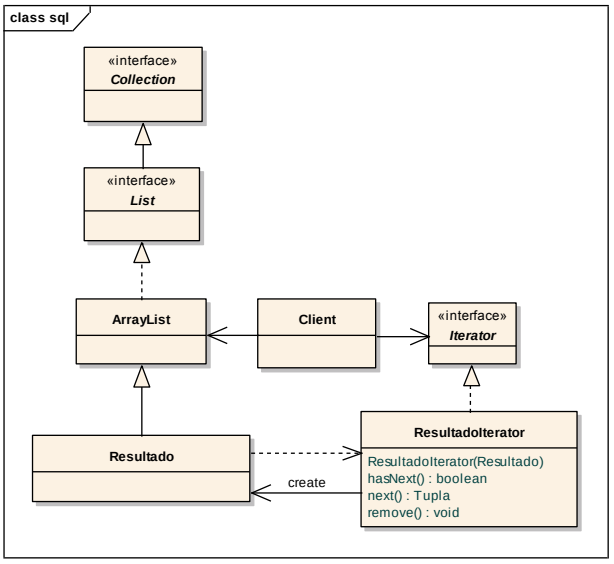

Figura 6. **Estrutura de classes do exemplo de *iterator* proposto**

Foi utilizado o padrão **Iterator** no framework para que fosse possível percorrer os resultados de comandos SELECT.

2.2. Framework

2.2.1 O que é um framework?

Um conceito intimamente relacionado com Design Patterns e Orientação a Objetos são os frameworks de desenvolvimento:

Um framework é uma mini-arquitetura reutilizável que fornece a estrutura de uma aplicação que pode ser customizada pelo programador e aplicada a um conjunto de aplicações de um mesmo domínio e características. Ao se utilizar um framework se procura reutilizar subsistemas de software, aumentando assim o grau de reuso e contribuindo para uma melhor qualidade do software.

Um framework não é geralmente uma aplicação completa: muitas vezes não possui todas as funcionalidades necessárias para uma aplicação específica. Até por isso, uma aplicação pode ser construída a partir de um ou mais frameworks, inserindo tais funcionalidades necessárias com as soluções *"plug-and-play"* fornecidas pelos frameworks. Ou seja, um framework fornece a infra-estrutura e os mecanismos que executam a interação entre os componentes abstratos e suas implementações.

Um framework é composto de *"frozen spots"* e *"hot spots"*:

- **Frozen Spots** são serviços já implementados pelo framework que normalmente realizam chamadas indiretas aos pontos host spots, ou seja, são as partes fixas de um framework;

- **Hot Spots** são funcionalidades, serviços, e que devem ser implementados pelos desenvolvedores que irão inserir os seus códigos inerentes ao domínio da aplicação, ou seja, são as partes flexíveis (No caso do framework dinSQL podemos citar os descritores XML como *hot spots*);

A utilização de frameworks apresenta os seguintes benefícios[4]:

- **Melhora a modularização**: encapsulamento dos detalhes voláteis de implementação através de interfaces estáveis;
- **Aumenta a reutilização**: definição de componentes genéricos que podem ser replicados para criar novos sistemas;
- **Extensibilidade**: favorecida pelo uso de métodos hooks que permitem que as aplicações estendam interfaces estáveis;
- **Inversão de controle (IoC)**: o código do desenvolvedor é chamado pelo código do framework. Dessa forma, o framework controla a estrutura e o fluxo de execução dos programas.

Frameworks podem também ser considerado um *wrapper* (embrulho) de funconalidades. Um *wrapper* é uma forma de reempacotamento de uma função ou conjunto de funções (relacionadas ou não) para alcançar um ou mais dos seguintes objetivos (provavelmente incompleta):

- A simplificação do uso (Simplifica uma interface para uma tecnologia);
- Redução / eliminação de tarefas repetitivas;
- Consistência na interface;
- Melhoria da funcionalidade do núcleo
- Coleta de processos discretos em uma associação lógica (um objeto)

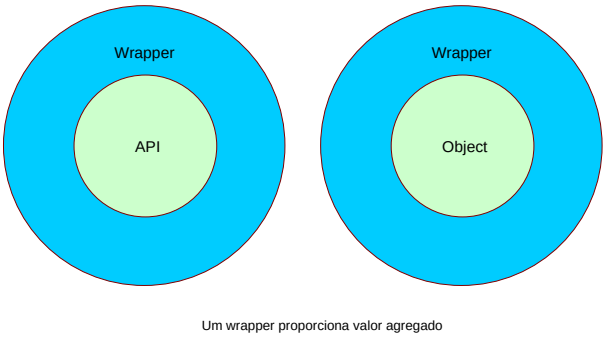

Figura 7. **Um Framework é um em pacote de funcionalidades**

2.2.2 Diferenciando frameworks e Design Patterns

Com o apresentado até agora é possível que haja confusão entre os conceitos de framework e Design Patterns, seus conceitos são realmente muito próximos, por isso é importante diferenciá-los. Suas principais diferenças são[1]:

- *Padrões de projeto são mais abstratos que frameworks*. Os *frameworks* podem ser materializados em código, mas somente exemplos de padrões podem ser materializados em código. Um ponto forte dos frameworks é que podem ser escritos em uma linguagem de programação, sendo não apenas estudados, mas executados e reutilizados diretamente. Em contraposição, os padrões de projeto deste livro têm que ser implementados cada vez que eles são usados. Os padrões de projeto também explicam as intenções, custos e benefícios (*trade-offs*) e consequências de um projeto.
- *Padrões de projeto são elementos de arquitetura menores* que frameworks. Um *framework* típico contém vários padrões de projeto, mas a recíproca nunca é verdadeira.
- *Padrões de projeto são menos especializados* que frameworks. Os frameworks sempre têm um particular domínio de aplicação. Um *framework* para um editor gráfico poderia ser usado na simulação de uma fábrica, mas ele não seria confundido com um *framework* para simulação. Em contraste, os padrões de projeto podem ser usados em quase qualquer tipo de aplicação. Embora padrões de projeto mais especializados sejam possíveis, mesmo estes não ditariam a arquitetura de uma aplicação da maneira como um *framework* o faz.

3 Framework dinSQL

3.1 O que é o dinSQL?

O dinSQL é um framework de persistência totalmente baseado nos Design Patterns GoF para acesso a banco de dados relacionais com suporte a SQL e execução de stored procedures. dinSQL elimina praticamente todo o código JDBC e centraliza a configuração e utilização de conexões ao banco de dados. dinSQL utiliza XML's simples para descrever o acesso ao banco de dados.

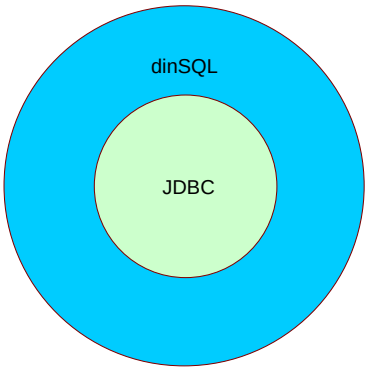

Figura 8. **dinSQL é um wrapper para a API JDBC**

3.2 Configurando o ambiente para utilizar o dinSQL

A configuração do framework é bastante simples, ela fica totalmente centralizada num arquivo de propriedades, que é definido na variável de ambiente "*sql-dinamico-config.file*". Apenas é necessário no script de inicialização da aplicação, definir uma variável como no exemplo: *-Dsql-dinamico-config.file=/_desenv/sql-config.properties*. Exemplo de um arquivo:

```
sql-
file.dir=/resources/sql/oracle,/resources/sql
sql-config.numMaxDescritores=5
sql-config.numOtimoDescritores=2
default.class=oracle.jdbc.driver.OracleDriver
default.url=jdbc:oracle:thin:@localhost:1521:x
e
default.userid=uerj2010
default.password=senha2010
default.auto-commit=true
default.jndi=
```

Segue a explicação das variáveis do arquivo:

- **sql-file.dir**: Lista de diretórios em que se encontrarão os arquivos descritores XML (Se houver a necessidade de se utilizar mais de um diretório apenas é necessário separar estes diretórios por vírgula). É necessário que estes diretórios estejam no classpath (*O classpath é um argumento - definido em linha de comendo, ou através de variáveis de ambiente - que dizem à JVM onde procurar por classes, sejam elas classes únicas ou pacotes jar, ao executar programas Java*) da aplicação cliente;
- **sql-config.numMaxDescritores:** Número máximo de descritores a serem mantidos em memória;
- **sql-config.numOtimoDescritores:** Número ótimo de descritores a serem mantidos em memória;
- **default.class:** Classe que implementa o driver JDBC referente ao SGBD utilizado pela conexão;
- **default.url:** URL JDBC da instância do SGBD;
- **default.userid:** O nome de usuário do SGBD;
- **default.password:** A senha do usuário do SGBD;
- **default.auto-commit:** Define se a conexão utilizará o commit automático de transações, se não for definido o padrão é false;
- **default.jndi:** A variável se refere à implementação do DataSource, a intenção é para o uso em aplicações que são executadas por containers, como Spring ou servidores de aplicação que configurem o DataSource de forma central ou externa, e coloca uma referência a este DataSource num contexto JNDI, cujo nome deve ser colocado nesta variável;

Pôde se reparar no termo default nas configurações das conexões listadas acima. Isto deve-se ao simples fato de que esta é a configuração da conexão padrão da aplicação. Esta conexão, ao ser utilizada, não necessita ter seu nome (default) explicitamente descrito na utilização pela classe. Caso se necessite efetuar conexões a mais de uma base de dados, apenas é necessário substituir o nome default pelo nome da conexão que se deseja utilizar. Imagine que a aplicação necessitasse se conectar na base da UERJ, apenas seria necessário escreverem as seguintes linhas:

```
uerj.class=oracle.jdbc.driver.OracleDriver
uerj.userid=uerj2010
uerj.password=uerj2010
uerj.url=jdbc:oracle:thin:@localhost:1521:xe
uerj.jndi=
```

Agora, imagine que a aplicação necessitasse se conectar na base do Aluno Online que se encontra num servidor de aplicação com um contexto JNDI configurado, apenas seria necessário se escrever a seguinte linha:

```
alunoOnline.jndi=java:uerjDS
```

Fica claro, desta forma, que quando uma conexão utiliza JNDI apenas a variável *.jndi da conexão é necessária.

3.3 Escrevendo o Descritor XML

No framework desenvolvido optou-se por definir todos os comandos SQL em arquivos XML. Chamaremos estes arquivos de descritores. Daí surge a seguinte pergunta:

Por que utilizar XML? Pois, por ser, o XML apenas um formato para a descrição de dados (é uma meta-linguagem e não uma linguagem de programação), que baseia seu formato em um arquivo texto, com estruturas organizadas hierarquicamente. E segundo a descrição do W3C (www.w3c.org), este formato permite troca de dados independente de plataforma e, utilizando o Document Object Model (DOM) nível 1, permite acesso segundo métodos padrões, facilitando a portabilidade dos dados.

Ou seja, XML, por ser apenas um formato a ser seguido, é independente de linguagem de programação e isto quer dizer que, caso algum dia, alguma aplicação, baseada neste framework, tiver a necessidade de ser escrita em outra linguagem que não seja Java, todos os acessos a banco de dados poderão ser reaproveitados por estarem em XML cuja leitura é suportada pela maioria absoluta das linguagens utilizadas no mercado.

Todos os XML's de comandos SQL possuem exatamente a mesma formação. Com as seguintes tags:

Tabela 2. **Descrição das tags**

Nome:	**Descrição:**
descricao	Descrição sucinta do objetivo do comando.
nome	Nome do comando, deve ter o mesmo nome que o arquivo sem a extensão.
tipo	Indica o tipo do comando. Pode ser Q (query), C (comando) ou P (procedure).
comandos:union	Indica se os comandos SQL serão encadeados em um (U)nion, union (A)ll, ou (N)ão possui unions.

comandos:footer	Trecho do comando a ser acrescentado no final de um comando, é útil para colocar Order By´s, Group By´s e Having´s.
comandos:comando	Possui todos as tags necessárias para a execução, pode ser repetida de acordo com a necessidade dos Union's.
comando:sql	Comando SQL que será executado.
paramFixos:paramFixo	Parâmetro que obrigatoriamente deve ser informado para a execução do comando caso haja algum ponto de interrogação no conteúdo da *tag* sql. Eles devem ser informados na ordem dos pontos de interrogação.
filtros:filtro	Filtros variáveis, ou seja, que podem ou não ser ativados para a execução do comando. Pode-se ativar mais de um filtro variável de cada vez. Identificados pelo atributo id.
filtro:where	Cláusula WHERE adicionada (com AND) caso o filtro seja ativado.
parametros:parametro	Parâmetros necessários para a utilização do filtro. Eles devem ser informados na ordem dos pontos de interrogação.

A seguir um exemplo de XML descritor de comando SQL:

```
<?xml version="1.0" encoding="UTF-8"?>
<descritor>
  <descricao>Consulta dos alunos de uma disciplina.</descricao>
  <nome>selalunomatricula</nome>
  <tipo>Q</tipo>
  <comandos>
    <union>N</union>
    <footer> ORDER BY turma.numero, aluno.nome</footer>
    <comando>
      <sql>
        SELECT aluno.nome, aluno.matricula, disciplina.nome, turma.numero
        FROM aluno, disciplina, turma, aluno_disciplina_turma
        WHERE aluno.matricula = aluno_disciplina_turma.matricula
        AND  aluno_disciplina_turma.numero_turma = turma.numero
        AND  aluno_disciplina_turma.codigo_disciplina = disciplina.codigo
        AND  turma.codigo_disciplina = dis<?xml version="1.0" encoding="UTF-8"?>
<descritor>
  <descricao>Consulta os cargos presentes no sistema.</descricao>
  <nome>selcargo</nome>
  <tipo>Q</tipo>
  <comandos>
    <union>N</union>
    <footer>ORDER BY DES_CARGO</footer>
    <comando>
      <sql>
        SELECT cod_cargo,
                  des_cargo,
                  fl_ativo
        FROM cargo
        WHERE fl_ativo = ?
      </sql>
      <paramFixos>
        <paramFixo>fl_ativo</paramFixo>
      </paramFixos>
      <filtros>
        <filtro id='codigo'>
          <where>
            cod_cargo = ? OR
            cod_cargo = ?
          </where>
          <parametros>
            <parametro>cod_cargo</parametro>
            <parametro>cod_cargo_2</parametro>
          </parametros>
        </filtro>
      </filtros>
    </comando>
  </comandos>
```

```
</descritor>ciplina.codigo
        AND  disciplina.codigo = ?
      </sql>
      <paramFixos>
        <paramFixo>cod_disciplina</paramfixo>
      </paramFixos>
      <filtros>
        <filtro id='turma'>
          <where>
            turma.numero = ?
          </where>
          <parametros>
            <parametro>num_turma</parametro>
          </parametros>
        </filtro>
        <filtro id='cpf_aluno'>
          <where>
            aluno.cpf = ?
          </where>
          <parametros>
            <parametro>cod_cpf</parametro>
          </parametros>
        </filtro>
      </filtros>
    </comando>
  </comandos>
</descritor>
```

3.4 Conectando-se ao banco de dados

Após serem efetuadas as configurações citadas no item 3.2, o desenvolver da aplicação cliente apenas necessita, para utilizar a conexão desejada ao banco de dados, utilizar a classe **br.com.mtorres.sql.conn.ConnManager**. A classe *ConnManager* é , como sugere o nome, a classe que gerencia as conexões e a partir dela é possível obter uma instância de **br.com.mtorres.sql.conn.Conn**. A classe *Conn* possui todos os métodos necessários para se executar um comando SQL numa base de dados, sua utilização é feita como no exemplo:

```
...
import br.com.mtorres.sql.conn.Conn;
import br.com.mtorres.sql.
conn.ConnManager;
...

try {

 Conn con =
  ConnManager.get().openConnection("uerj");

 // Início do código que executa o comando SQL
  ...
 // Fim do código que executa o comando SQL

 ConnManager.get().closeAll(ConnManager.COMMIT);

} catch (DinSQLException e) {
 try {
  ConnManager.get().closeAll();
 } catch (DinSQLException ee) {
  ee.printStackTrace();
 }
}
```

No caso anterior, o desenvolvedor não está utilizando a conexão padrão (*default*) , e sim está se conectando à base **uerj**, por isso, ele necessariamente precisa passar a string **"uerj"** como parâmetro para o método *getConnection*. Caso fosse utilizada a conexão padrão, não seria necessário passar nenhum parâmetro para o método *getConnection*, como no exemplo:

```
Connection con =
            ConnManager.get().openConnection();
```

Ao implementar a classe, foi utilizada a classe *ThreadLocal* presente na própria *Java Standard Edition*, ou seja, é padrão das distribuições atuais da linguagem Java. A classe *ThreadLocal* é utilizada para armazenar valores que estão "amarrados" ao escopo da Thread corrente (ou em inglês: *thread-local variables*). Ela possui uma funcionamento parecido com o de um *Singleton*, no entanto, a única instância do *Singleton* existe durante toda a linha de vida da aplicação cliente, no caso da *ThreadLocal* a única instância existe apenas durante a thread corrente.

Apenas uma conexão de cada nome é mantida aberta por *thread*, não importando quantos *getConnection* para o mesma conexão são chamados numa mesma *thread*. A classe *ConnManager* verifica se uma conexão já foi aberta para a *thread* corrente e a reutiliza. Esta implementação foi feita visando-se possuir um maio controla nas transações de banco de dados utilizadas.

Cada thread possui uma, e apenas uma, instância de cada conexão utilizada. Instâncias de *Conn* não devem ser compartilhadas. Portanto, o melhor escopo é o *request* ou escopo de método. Não é boa prática manter referências à instâncias de *ConnManager* em variáveis estáticas ou em variáveis de instância de uma classe.

Não é boa prática manter instâncias de *Conn* em nenhum tipo de escopo como *HttpSession* do framework de *Servlets*. Ao utilizar em qualquer tipo de framework web, considere que *Conn* seguirá um escopo similar à uma requisição HTTP. Ou seja, ao receber uma requisição HTTP, deve-se abrir uma instância de *Conn* e ao mandar a resposta (*response*), deve-se fechar a

conexão. Fechar a conexão é muito importante. Sempre se deve assegurar que esteja fechada dentro de um bloco *finally*. A seguir segue um padrão a utilizar para assegurar o fechamento:

```
try {
 ...
} finally {
 ...
 ConnManager.get().closeAll();
 ...
}
```

Utilizando o padrão acima de forma consistente no código da aplicação cliente assegura-se que todas as conexões ao banco de dados estejam fechadas de forma apropriada (se a conexão não for fechada de forma explícita, o framework se assegura de fechar as conexões no momento que o *Garbage Collector (O trabalho do Garbage Collector se resume a vasculhar a memória procurando pelo lixo que é deixado na memória, onde este lixo está ocupando espaço, o que se torna desnecessário e prejudicial ao sistema, visto que aquela parte suja da memória poderia estar sendo usada por outro aplicativo*) é executado pela JVM.

3.5 Controle transacional

Existem quatro métodos na classe *ConnManager* para controlar o escopo de uma transação de forma explícita. Obviamente, estes métodos não possuem efeito se você escolheu usar *auto-commit*. Os quatro métodos são:

- `public void commit();`
- `public void rollback();`
- `public void commit(String provider);`
- `public void rollback(String provider);`

Os métodos *commit* e *rollback* que não possuem parâmetros efetuam o "*commit/rollback*" na conexão padrão (*default*). Na maioria das vezes não é necessário executar o método *rollback()*, pois o *dinSQL* se ocupa de executá-lo. No entanto, se for necessário um maior controle em conexões onde *commits* e *rollbacks* múltiplos são possíveis, existe a opção de efetuar o *rollback* para tornar a execução múltipla desses comandos possível.

Também é possível efetuar o *commit* e o *rollback* ao se fechar a conexão, no final de uma execução de um comando. São oferecidos os seguintes métodos:

- `public void close(int acao);`
- `public void close(String provider, int acao);`
- `public void closeAll();`
- `public void closeAll(int acao);`

Os métodos close fecham apenas as conexões que são passadas por parâmetro, se a variável **provider** não for informada isto eu dizer que a conexão default é que será fechada. Os métodos *closeAll* fecham todas as conexões que foram abertas na thread de execução. O parâmetro **acao**, é preenchido com `ConnManager.COMMIT` e `ConnManager.ROLLBACK`, auto explicativos. O método *closeAll* sem parâmetros fecha todas as conexões abertas na thread e efetua o *rollback*.

3.6 Executando comandos SQL

Os comandos SQL são executados a partir da classe *SQLExecutor*, que é a classe mais importante do dinSQL. É onde são encontrados todos os métodos necessários para a execução de qualquer comando SQL. É importante lembrar que a execução de comandos SQL se baseia no padrão *Factory Method* e isto quer dizer que a classe que vai implementar a classe *SQLExecutor* apenas é conhecida em tempo de execução. O template *method* é o método *execute()*. As classes filhas, concretas, que implementam *SQLExecutor* são *CommandExecutor*, *ProcedureExecutor* e *QueryExecutor*:

- **CommandExecutor**: Classe que implementa a execução dos comandos INSERT, DELETE e UPDATE;
- **ProcedureExecutor**: Classe que implementa a execução de store procedures;
- **QueryExecutor**: Classe que implementa a execução do comando SELECT;

Existem aproximadamente dez métodos na classe *SQLExecutor* que são implementados nas classes filhas de acordo com sua especialização, abaixo segue a explicação dos métodos cuja visibilidade permite sua execução pelas classes cliente, ou seja, os métodos públicos:

- O construtor **SQLExecutor**: nesse caso o construtor recebe dois parâmetros, são eles o **descriptorName** que recebe o nome do arquivo XML com os comandos SQL e **con** que recebe a instância de **br.com.mtorres.sql.conn.Conn** criada a partir da classe **br.com.mtorres.sql.conn.ConnManager**;
- **execute()**: simplesmente o método mais importante de todo o framework, ele executa, de fato, o comando descrito no XML na base de dados;
- **addParamFixo(String nomeParametro, Object valParametro)**: adiciona todos os parâmetros descritos pelas tag's paramFixo do XML, deverá ser chamado um número de vezes igual ao número de tag's paramFixo presentes no arquivo. O parâmetro **nomeParametro** o nome do parâmetro presente na tag e **valParametro** remete ao valor a ser passado para o SQL;
- **addFiltroAtivo(String filtro)**: adiciona as cláusulas WHERE existentes nas tag's where que possuem às tag's filtro. O parâmetro filtro remete ao atributo id da tag filtro;

- **addValFiltro(String nomeFiltro, Object valorFiltro)**: tem um funcionamento semelhante ao addParamFixo, com a diferença que se refere às tag's parâmetro, dentro dos filtros;
- **addOutputParameter(String nomeParametro, Integer tpParametro)**: método apenas utilizado na execução de stored procedures. Stored Procedures possuem parâmetros que podem retornar valores de forma separada, ou seja parâmetros de OUTPUT, este método define quais são os parâmetros (tag's paramFixo) que são parâmetros de OUTPUT. Sendo variável **nomeParametro** o nome do parâmetro presente na tag paramFixo e o parâmetro **tpParametro**, o tipo de parâmetro(Parâmetro dependente de cada implementação do driver JDBC de cada SGBD);
- **getResultado()**: retorna uma coleção iterável (do tipo Resultado), com os resultados da execução de um comando SELECT. Apenas pode ser chamado na execução de uma query;
- **getLinhasAfetadas()**: retorna o número de linhas afetadas na execução de um comando INSERT ou UPDATE. Apenas pode ser chamado na execução destes comandos;
- **getOutputValue(String chave)**: apenas pode ser chamado na execução de stored procedures, retorna os valores dos parâmetros de output, o parâmetro chave remete ao nome do parâmetro na tag paramFixo;

3.7 Executando um comando SQL

Um exemplo da utilização do dinSQL para a execução de um comando é dado a seguir:

A execução dos comandos INSERT, DELETE e UPDATE, conforme descrito acima é feita por meio da classe CommandExecutor, a explicação de sua utilização será feita a partir do exemplo a seguir:

Imagine que se queira manter numa aplicação cliente uma tabela de cargos existentes. Para inserir dados nessa tabela, que chamaremos de CARGO, apenas são necessários o XML e o trecho de código abaixo:

3.7.1. Descritor XML utilizado

Na tag "tipo" do XML supracitado, o conteúdo "Q" define um descritor que apenas pode executar comandos SELECT, o código para execução deste comando é exemplificado abaixo, a explicação de cada linha do comando encontra-se em comentários no próprio trecho:

3.7.1. Trecho de código utilizado

3.7.2 Tabela de compatibilidade de tipos

Ao buscar um campo do banco de dados, é necessário saber para que tipo da linguagem Java vamos efetuar o *cast* das variáveis retornadas, isto é possível com a implementação desta tabela:

Tabela 3. **Tabela de compatibilidade de tipos**

Tipos Java	Tipos JDBC
Boolean,boolean	Qualquer tipo compatível com BOOLEAN
Byte,byte	Qualquer tipo compatível com NUMERIC ou BYTE
Short,short	Qualquer tipo compatível com NUMERIC ou SHORT INTEGER
Integer,int	Qualquer tipo compatível com NUMERIC ou INTEGER
Long,long	Qualquer tipo compatível com NUMERIC ou LONG INTEGER
Float,float	Qualquer tipo compatível com NUMERIC ou FLOAT
Double,double	Qualquer tipo compatível com NUMERIC ou DOUBLE
BigDecimal	Qualquer tipo compatível com NUMERIC ou DECIMAL
String	CHAR,VARCHAR
String	CLOB,LONGVARCHAR
String	NVARCHAR,NCHAR
String	NCLOB
byte[]	Qualquer tipo compatível com byte stream type
byte[]	BLOB,LONGVARBINARY
Date (java.util)	TIMESTAMP
Date (java.util)	DATE
Date (java.util)	TIME
Timestamp (java.sql)	TIMESTAMP
Date (java.sql)	DATE
Time (java.sql)	TIME
Object	OTHER,ou não especificado

3.8 SQL Dinâmico

Uma das mais poderosas funcionalidades do dinSQL é sua capacidade de montar SQL's de forma dinânica. Se o desenvolvedor possui alguma experiência com o JDBC ou qualquer outro framework similar, ele entenderá o quão trabalhoso é concatenar strings de SQL de forma condicional, se assegurando em não esquecer espaços em branco antes das aspas ou em omitir alguma vírgula no final de uma lista de colunas.

O dinSQL adiciona as cláusulas WHERE presentes nas tags filtro de forma simples, apenas é necessário utilizar no código de implementação o método *addFiltroAtivo* para cada tag filtro com o atributo "id" correspondente ao parâmetro passado ao se executar este método.

```
...
import br.com.mtorres.exception.DinSQLException;
import br.com.mtorres.sql.conn.Conn;
import br.com.mtorres.sql.conn.ConnManager;
import br.com.mtorres.sql.execution.QueryExecutor;
import br.com.mtorres.sql.execution.ResultadoIterator;
import br.com.mtorres.sql.execution.SQLExecutor;
import br.com.mtorres.sql.execution.Tupla;
...

    try {

      // Iniciando a conexão ao banco de dados, no caso a conexão default
      Conn con = ConnManager.get().openConnection();

      // Instanciando a implementação do SQLExecutor
      SQLExecutor query = new QueryExecutor("selcargo", con);

      // Insere os parâmetros referentes às tags paramFixo
      query.addParamFixo("fl_ativo", "1");

      // Aplicando a cláusula where presente na tag filtro com o atributo
      //id = codigo
      query.addFiltroAtivo("codigo");

      // Insere os parâmetros referentes às tags parametro
      query.addValFiltro("cod_cargo", "07");
      query.addValFiltro("cod_cargo_2", "15");

      // Executa o comando
      query.execute();

      //Recupera a coleção iterável (do tipo Resultado)
      //com os resultados da execução do comando SELECT
      ResultadoIterator rIt = new ResultadoIterator(query.getResultado());

      System.out.println("Código\t-\tDescrição do Cargo");

      //É possível percorrer a coleção de acordo com o
      //padrão Iterator, utilizando o método hasNext
      while (rIt.hasNext()) {
        //Cada linha de Resultado possui uma instância de Tupla
        //Com cada tupla do resultado da query
        Tupla tupla = rIt.next();

        //Ao buscar uma coluna de tupla, é necessário efetuar-se o
        //type cast, é possível saber qual type cast efetuar de acordo
        //com a tabela 3.1 de tipos
        System.out.print((String) tupla.getColuna("cod_cargo"));
        System.out.print("\t-\t");
        System.out.println((String) tupla.getColuna("des_cargo"));
      }

      // Fecha todas as conexões abertas
      ConnManager.get().closeAll();

    } catch (DinSQLException e) {
      try {
        // Caso algum erro ocorra, fecha, por segurança, todas as conexões
        // abertas
        ConnManager.get().closeAll();
      } catch (DinSQLException ee) {
        ee.printStackTrace();
      }
    } finally {
      try {
        //É boa prática garantir o fechamento das conexões no bloco
        //finally
        ConnManager.get().closeAll();
      } catch (Exception e) {
        e.printStackTrace();
      }
    }
```

3.9 Instalando o dinSQL numa aplicação

A instalação do dinSQL é muito simples. Apenas é necessário efetuarem-se os passos descritos no item 3.4 e colocar o arquivo din-sql.jar em um diretório que esteja no classpath da aplicação.

O dinSQL depende de outros projetos, por isso, é necessário colocarem-se no mesmo diretório onde estará o din-sql.jar, os seguintes arquivos: xpp3_min-1.1.4c.jar e xstream-1.3.1.jar. Além do jar que contenha o driver JDBC do banco de dados utilizado.

4. Validação da Arquitetura

A validação da arquitetura será feita na prática, foi criada uma aplicação de exemplo que apenas efetua um simples login no site a partir da base de dados da aplicação. A aplicação foi testada no servidor de aplicação Jboss, Versão 5.1.0.GA.

4.1 Configurando a aplicação

Nossa aplicação de exemplo apenas terá a conexão default, e utilizará JNDI, portanto seu arquivo de configuração ficará da seguinte forma:

```
sql-file.dir=/resources
default.jndi=java:uerjDS
sql-config.numMaxDescritores=10
sql-config.numOtimoDescritores=5
```

Conforme descrito no item 3.8, os arquivos xpp3_min-1.1.4c.jar e xstream-1.3.1.jar são necessários no classpath da aplicação:

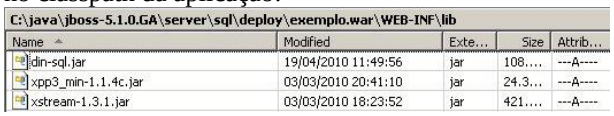

Figura 9. **Arquivos jar da aplicação**

A pasta com os arquivos XML descritores foi definida na pasta resources (Numa aplicação web em Java, o classpath é definido a partir da pasta WEB-INF/classes), portanto, os descritores ficarão dispostos da seguinte forma:

Figura 10. **Disposição dos descritores no diretório definido**

4.2 Aplicação

Por ser uma aplicação que apenas efetua o login, teremos apenas três telas no sistema:

A tela inicial de login será a seguinte:

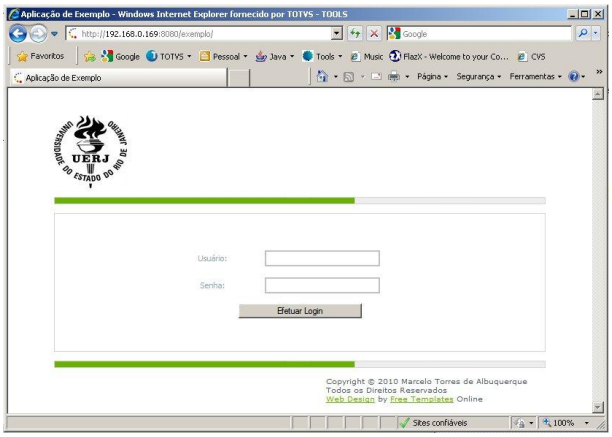

Figura 11. **Tela de Login**

Caso o login seja efetuado com sucesso, a seguinte tela será exibida:

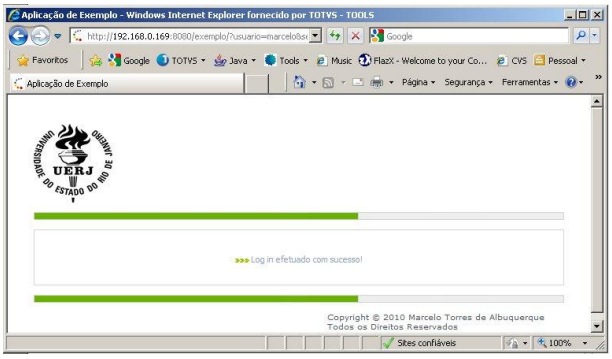

Figura 12. **Login efetuado com sucesso**

Caso ocorra erro ao efetuar o login a seguinte tela aparece:

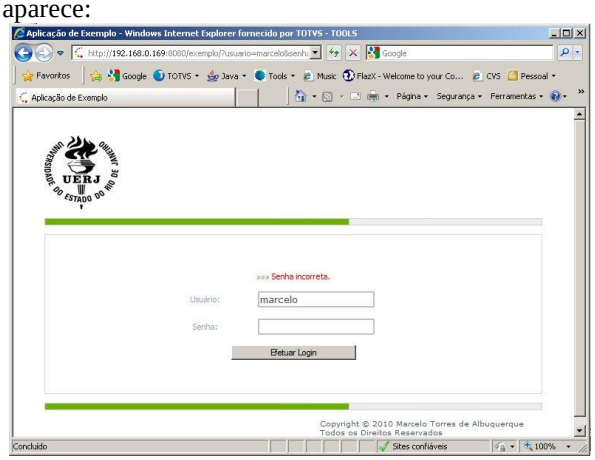

Figura 13. **Erro ao efetuar o login**

4.3 Teste de Carga

Com o intuito de efetuar as medições necessárias, foi efetuado um teste de carga utilizando-se do programa JMeter: JMeter é um ferramenta utilizada para testes de carga em serviços oferecidos por sistemas computacionais. Esta ferramenta é parte do projeto Jakarta da Apache Software Foundation.

O JMeter disponibiliza também um controle de threads, chamado Thread Group, no qual é possível configurar o número de threads, no caso 100 (Como se

fossem 100 usuários tentando efetuar o login no site ao mesmo tempo), a quantidade de vezes que cada thread será executada, no caso configuramos para 10 vezes, e o intervalo entre cada execução, no caso 5 segundos, que ajuda a realizar os testes de stress.

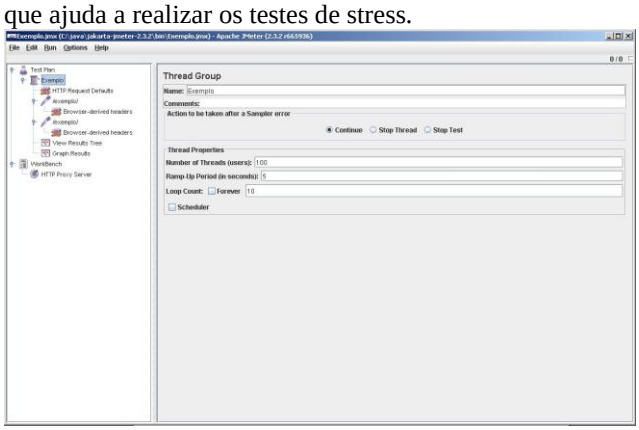

Figura 14. **Configuração do JMeter, de acordo com o descrito**

4.3.1 Comportamento do Pool de conexões

Como dito acima, a aplicação utilizará apenas a conexão default que utiliza um pool de conexões JNDI. Este pool de conexões está configurado com um número máximo de conexões igual a dez. O servidor de aplicação JBoss fornece uma área de gerência de recursos, onde podemos ver o comportamento do pools de conexões, podemos visualizar neste caso que o framework utiliza o pool de conexões JNDI de forma saudável, ou seja, o número de conexões em uso sempre volta para zero:

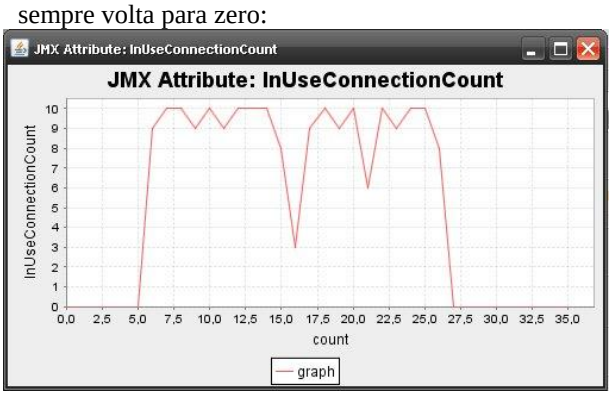

Figura 15. **Comportamento do Pool de Conexões**

5. Conclusões

A aplicação de Design Patterns constitui poderosa ferramenta para a criação de uma aplicação robusta que utiliza soluções já testadas, procurando minimizar o impacto de alterações durante o ciclo de vida do framework, explorando o baixo acoplamento entre os elementos de software do sistema. A aplicação de Design Patterns tem conseqüência direta na arquitetura da solução e promove o reuso de software.

Este trabalho apresentou todo o cliclo de vida do desenvolvimento de um framework de acesso a banco de dados em Java. É possível concluir, ao fim deste, que a utilização de Design Patterns traz grandes benefícios, pois reduz a duplicidade de operações semelhantes (reutilização), e isto traz uma vantagem muito grande na manutenção, pois um bug descoberto na superclasse implica em uma alteração somente na superclasse e todos os pontos do sistema estarão corrigídos.

Ao final do capítulo 4, onde uma aplicação de teste foi utilizada e testes de carga foram feitos para a validação da arquitetura, pode-se concluir que a utilização de Design Patterns foi bastante simples porém igualmente poderosa no desenvolvimento do framework e trouxe benefícios claros para o desenvolvimento no que tange a confiabilidade do produto final obtido.

6. Referências

[1] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns – Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1995.

[2] Paulo Roberto Quicole, *Padrões de projeto, modelando flexibilidade no software orientado a objetos. Trabalho de conclusão de curso (Tecnólogo) – Curso Processamento de Dados. Taquaritinga: Faculdade de Tecnologia de Taquaritinga*, 2004.

[3] Graig Larman, *Utilizando UML e Padrões: uma introdução à analise e ao projeto orientados a objetos e ao desenvolvimento iterativo*, Bookman, 2007

[4] M. Fayad, D. Schmidt and R. Johnson - *Building Applications Frameworks*, John Willey, 1999.