Instituto **S**uperior de **E**ngenharia de **L**isboa

Licenciatura em Engenharia Informática e de Computadores

Semestre de Verão 2013/2014

Trabalho Final Modelação e Padrões de Desenho

Eng.ª Fernando Miguel Carvalho

Data de entrega: 07.07.2014

Trabalho elaborado por:

Flávio Cadete nº 35383

Tiago Formiga nº 35416

Índice

Introdução

O objectivo deste trabalho é desenvolver uma *framework* `SqlMapper` capaz de criar uma instância de uma implementação de `DataMapper` para uma determinada *entidade de domínio* (ED) tirando partido do serviço de reflexão da Java VM.

Para além do saber utilizar o serviço de reflexão, teremos de pôr em prática os conceitos e implementação dos Padrões de Desenho aprendidos no decorrer do semestre, sem tentar forçar um padrão em si, mas reconhecendo um quando é encontrada uma solução.

Estrutura da Solução do Trabalho

O projecto do trabalho está dividido em 2 principais packages:

- **Source Packages**: contém tudo o que corresponde á definição e implementação da framework *SqlMapper*:
  - **Binder**: conjunto de classes com a função de fazer set e get sobre os campos da entidade de dominio;
  - **Annotations**: conjunto das interfaces de annotação possíveis a usar na framework;
  - **CoreFw**: conjunto de classes em que se encontra a definição do DataMapper e como ele será construído.
  - **SqlExecutor**: conjunto de classes com o objectivo de controlar as ligações á base de dados.
  - **Utils**: utilitários usados na framework.

- **Test Packages**: contém todos os testes unitários do trabalho, além das entidades de domínio usadas nesses testes. Este Package exemplifica o que o cliente poderá fazer com a nossa framework *SqlMapper*.

Entidades de Domínio

De forma a dar suporte á informação acedida na base de dados é necessário criar uma entidade de domínio associada a cada tabela a que se pretende manipular através da framework. Uma entidade de domínio é representada através de uma classe, que pode ter qualquer nome, uma vez que todas as entidades de domínio estão anotadas com o nome correspondente à tabela na base de dados. Esta anotação é garantida através da anotação *DatabaseTable*, que tem um campo name para esse prepósito.

Os campos contidos nestas classes têm nomes, que por omissão, correspondem ao nome da coluna da respectiva tabela, no entanto, o utilizador da framework pode atribuir outro nome a essa variável, deste que utilize a anotação DatabaseField e indique o nome correcto a que o campo corresponde na base de dados.

Foram ainda criadas outras duas anotações, a primeira de nome *PrimaryKey*, que tem como objectivo indicar qual é o campo dentro de uma entidade de domínio que representa a chave primária da tabela correspondente; a segunda de nome ForeignKey que é utilizado para representar que o campo assinalado representa um objecto da entidade de domínio. No entanto, este campo apenas é obrigatório nas associações 1-N. Esta anotação apenas é necessária nas associações 1-N pois o mecanismo de introspeção em java não permite saber em run-time qual o tipo de um objecto genérico.

No caso da associação 1-1 é verificada se o tipo do campo é uma classe anotada com *DatabaseTable*, pois se isto acontece significa que é esse tipo é referente a uma ED.

É possível verificar, através da Figura 1, a definição de uma entidade de domínio existente no trabalho, além do uso das anotações.

```
@DatabaseTable(name = "Customers")

public class Customer {

    @PrimaryKey
    public String CustomerID;
   @ForeignKey(Order.class)
    public Iterable<Order> OrderID;
    private  String CompanyName;
    public String ContactName;
    private String ContactTitle;
    public String Address;
    public String City;
    private String Region;
    public String PostalCode;
    public  String Country;
    public String Phone;
    public String Fax;

    public Customer() {
    }
```

Figura 1: Classe *Customer* que tem uma associada 1-N com tabela Order

Classe Builder

É através desta classe Builder que é possível obter um data mapper para qualquer uma das entidades de domínio criada pela framework client e que respeite as convenções dadas pelas nossa framework.

Na instanciação de builder é necessário o utilizador especificar qual o tipo de funcionamento que pretende que a framework tenha, definindo 3 argumentos:

- ➔ Política de gestão de ligações: se é reutilizada a mesma ligação em diferentes execuções dos métodos do data mapper, ou se é criada uma nova ligação em cada execução de cada método, ou outra estratégia qualquer de gestão de ligações que seja implementada à posteriori pela framework user e que extenda *AbstractSqlExecutor* de *SqlMapper Framework*;

- ➔ SQLServerDataSource: contém os dados necessários para a correcta conexão ao servidor da base de dados, se for necessário autenticação, este objecto deve vir já com os respectivos dados.

- ➔ Tipo de Binder pretendido entre a entidade de domínio e as colunas da tabela, se baseado em campos, se por métodos da entidade de domínio, ou ambos os tipos; Este ponto não foi implementado da forma pedida no enunciado interpretamos mal o que era pedido.

Construído o objecto da classe builder, é necessário invocar o método build de forma a gerar o *DataMapper* que contém todos os dados necessários para a sua utilização; ou então é reutilizado um mapper previamente criada e que o builder irá guardou num HashMap. Este HashMap contém todas as implementações de *DataMapper* para ED diferentes.

Nesta chamada ao método *build*, é necessário indicar por parâmetro a entidade de domínio que vai estar associado ao data mapper, para a framework poder fazer o que foi descrito no parágrafo anterior.

```
SQLServerDataSource ds = ConnectionManager.getDS();
Builder b = new Builder(SqlSingleExecutor.class, ds,BindFields.class);
DataMapper<Product> productDM = b.build(Product.class);
productDM.getAll().where("ProductID = 3");
```

Figura 2: modo de utilização de Builder

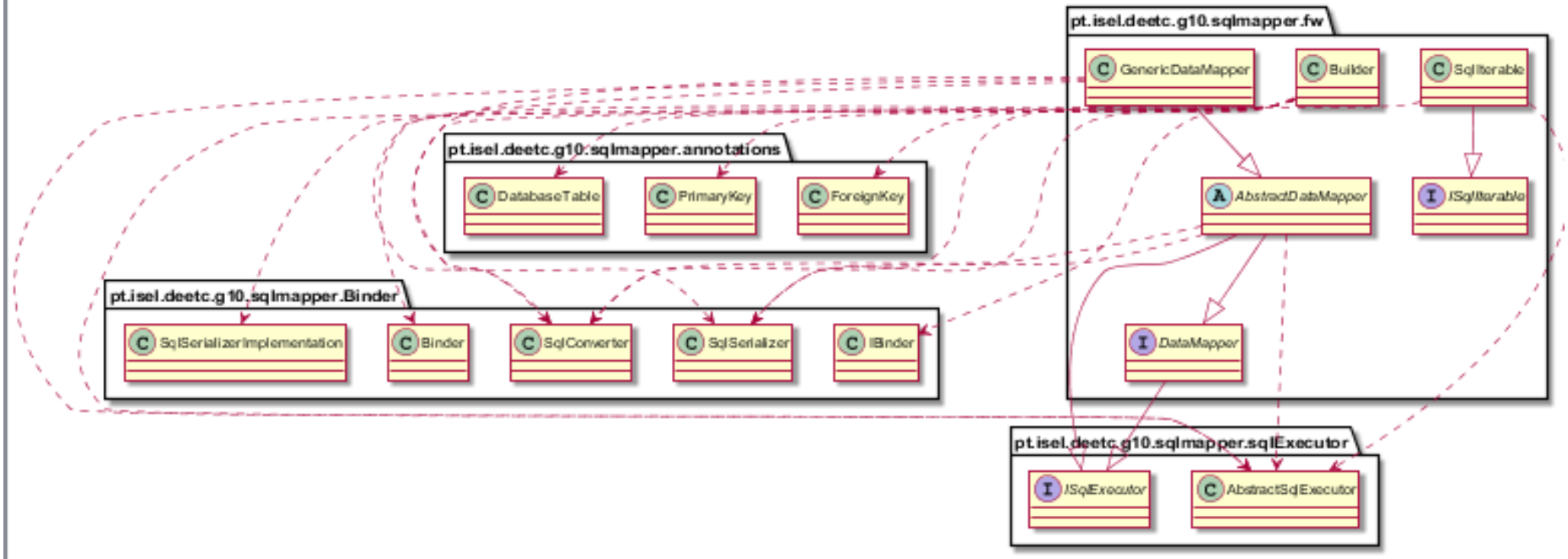


Figura 3: principais associações e dependências de Builder

Política de Gestão de Ligações

Para implementar este requisito foi necessário criar uma estrutura de classes, com base num protocolo definido numa interface.

Foi criada a interface *ISqlExecutor* que contém quais os métodos que têm que ser implementados pelas classes que definam *AbstractSqlExecutor* e *DataMapper* (para que o cliente possa fazer a chamada explícita a estas funcionalidades).

A Figura 4 contém a definição da interface *ISqlExecutor*:

```
public interface ISqlExecutor{
    void rollback();
    void commit();
    void closeConnection();
}
```

Figura 4: Interface *ISqlExecutor*

Devido ao facto de haver diversos troços de código iguais para diferentes tipos de ligações foi criada uma classe que implementa a interface *ISqlExecutor*, a classe abstracta *AbstractSqlExecutor*, sendo que as classes que correspondem aos tipos de ligações, extendem desta classe abstracta, partilhando o código comum na classe do qual extendem e sobrepondo os métodos com código específico do tipo de ligação na própria classe.

```
public abstract class AbstractSqlExecutor implements ISqlExecutor{
    protected final SQLServerDataSource ds;
    protected Connection c;
    protected boolean autocommit;
    protected Binder BinderED;
    protected int usingConnection;

    public AbstractSqlExecutor(SQLServerDataSource ds, Binder BinderED,
            boolean autocommit) throws SQLException {...6 lines }

    public Connection beginConnection() throws SQLException {...9 lines }

    @Override
    public void closeConnection(){...12 lines }
    public void closeAfterCommand() {...5 lines }

    public <T> SqlIterable<T> executeQuery(
            String sqlStmt,
            SqlConverter<T> conv,
            Object... args) throws SQLException {...5 lines }

    public int executeUpdate(String sqlStmt, Object... args)
            throws SQLException {...17 lines }

    public <T> int executeInsert(String sqlStmt, T val, List<String> primaryKeys
            , Object... args) throws SQLException {...28 lines }
}
```

Figura 5: Classe *AbstractSqlExecutor*

Conexão Única: A figura 6 demonstra como foi implementado este tipo de ligação.

```
public class SqlSingleExecutor extends AbstractSqlExecutor {
    public SqlSingleExecutor(DataSource ds, Binder BinderED) throws SQLException {
        super(ds,BinderED,false);
    }

    @Override
    public void rollback() {
        try {
            if(c==null|| c.isClosed()) return;
            c.rollback();
        } catch (SQLException ex) {
            Sneak.sneakyThrow(ex);
        }
    }

    @Override
    public void commit() {
        try {
            if(c==null|| c.isClosed()) return;
            c.commit();
        } catch (SQLException ex) {
            Sneak.sneakyThrow(ex);
        }
    }
}
```

Figura 6: Classe SqlSingleExecutor

Múltiplas Conexões: A figura 7 demonstra como foi implementado este tipo de ligação.

```
public class SqlMultiExecutor extends AbstractSqlExecutor{
    public SqlMultiExecutor(DataSource ds, Binder binder) throws SQLException {
        super(ds, binder, true);
    }

    @Override
    public void closeAfterCommand(){
        super.closeConnection();
    }
    @Override
    public Connection beginConnection() throws SQLException {
        c=ds.getConnection();
        return c;
    }
    @Override
    public void rollback() {
        throw new UnsupportedOperationException
        ("Rollback cannot be called on a multi executor");
    }
    @Override
    public void commit() {
        throw new UnsupportedOperationException
        ("Commit cannot be called on a multi executor");
    }
}
```

Figura 7: Classe SqlMultiExecutor

As principais diferenças na implementação destes 2 tipos de ligação são:

1. ***CloseConnection:***
    a. *SqlMultiExecutor* é executado o método closeAfterComand (que por si chama o CloseConnection) no final da execução de cada comando;
    b. *SqlSingleExecutor* só será chamado com a chamada explícita do cliente, a partir do DataMapper;

2. ***BeginConnection***:
    a. *SqlMultiExecutor é retornado sempre uma nova ligação;*
    b. *SqlSingleExecutor apenas é retornado uma ligação se não se encontrar nenhuma activa.*

3. ***Rollback* e *Commit***,
    a. *SqlMultiExecutor* tem conexões **autocommited**, assim sendo este nunca irá conseguir chamar os métodos rollback ou commit explicitamente pois a ligação estará fechada, lançando uma excepção se o cliente o tentar fazer;
    b. *SqlSingleExecutor* tem conexões **não autocommited**, podemos assim o cliente poder fazer rollback ou commit quando pretender.

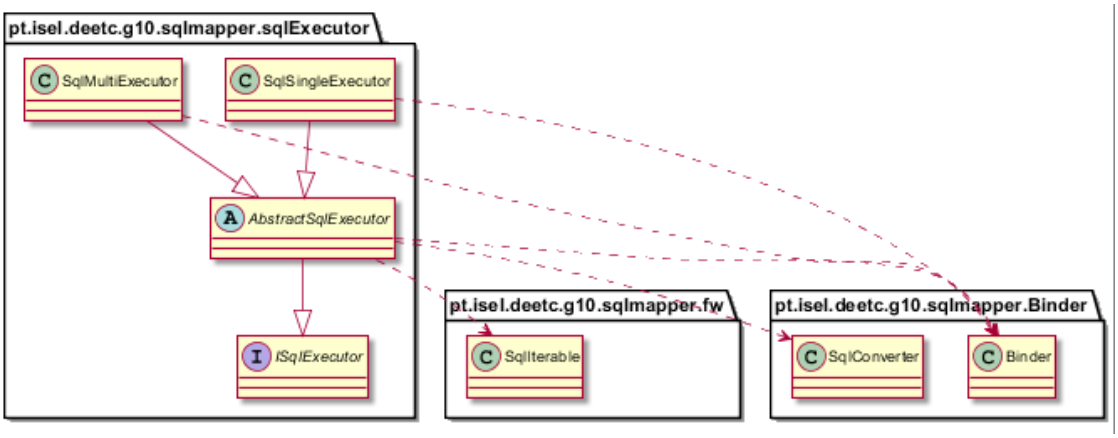


Figura 8: UML do gestor de ligações

DataMapper

```
public interface DataMapper<T> extends ISqlExecutor{
    SqlIterable<T> getAll();
    void update(T val);
    void delete(T val);
    void insert(T val);

}
```

Figura 9: interface DataMapper

AbstractDataMapper

Esta é a classe responsável por grande parte do esqueleto do código, sendo que define o padrão para cada um dos comandos. As alterações nos comandos são na parte de fazer bind entre os argumentos da query e o objecto que se está a utilizar. Nessas situações é da responsabilidade das classes que estendem este método como é que esse bind é feito. No caso da nossa framework, é o GenericDataMapper que irá dizer como é feito o bind.

```
public abstract class AbstractDataMapper<T> implements DataMapper<T>, ISqlExecutor,Au

    protected final List<String> primaryKey;
    protected final AbstractSqlExecutor exec;
    protected abstract String sqlGetAll();
    protected abstract String sqlUpdate();
    protected abstract String sqlInsert();
    protected abstract String sqlDelete();
    protected abstract SqlConverter<T> conv();
    protected abstract SqlSerializer<T> insertserializer();
    protected abstract SqlSerializer<T> updateserializer();
    protected abstract SqlSerializer<T> deleteserializer();

    protected AbstractDataMapper(AbstractSqlExecutor exec, List<String> primaryKey) {
        this.exec = exec;
        this.primaryKey = primaryKey;
    }
    @Override
    public final SqlIterable<T> getAll(){...10 lines }

    @Override
    public final void insert(T val){...13 lines }

    @Override
    public final void update(T val){...11 lines }

    @Override
    public final void delete(T val) {...12 lines }
```

Figura 10: parte da implementação de AbstractDataMapper

GenericDataMapper

O GenericDataMapper é uma classe que extende de AbstractDataMapper e foi criado com o intuito de criar todos os componentes referentes ao mapper a ser criado. Para isso recebe no seu constructor os dados necessários. Para a criação das querys string, recebe no constructor as primary keys e os outros campos da ED. Esta classe tem também a função de converter os dados vindos da base de dados em objectos java (através do converter) e enviar os dados do objecto sobre o qual o comando está a ser executado da forma correcta (através do serializer).

Tanto o converter como os serializers são construídos pelo builder, que tendo acesso à metadata dos objectos, cria os objectos da forma correcta.

```
public class GenericDataMapper<T> extends AbstractDataMapper<T>{

    private String sqlGetAll;
    private String sqlUpdate;
    private String sqlInsert;
    private String sqlDelete;
    private SqlConverter<T> sqlconverter;
    private SqlSerializer<T> insertsqlserializer;
    private SqlSerializer<T> updatesqlserializer;
    private SqlSerializer<T> deletesqlserializer;

    public GenericDataMapper(AbstractSqlExecutor exec, SqlConverter<T> conv, SqlSerializer<T> insertSerial,
                             SqlSerializer<T> updateSerial, SqlSerializer<T> deleteSerial,
                             String table, List<String> primaryKey, List<String> fields) {
        super(exec, primaryKey);
        if(primaryKey==null || exec ==null) throw new InvalidParameterException("You must specify the primary key");

        this.sqlconverter=conv;
        this.insertsqlserializer = insertSerial;
        this.updatesqlserializer = updateSerial;
        this.deletesqlserializer = deleteSerial;


        StringBuilder sqlGetAllBuilder = new StringBuilder("SELECT ");
        StringBuilder sqlUpdateBuilder = new StringBuilder("UPDATE ");
        StringBuilder sqlInsertBuilder = new StringBuilder("INSERT INTO ");
        StringBuilder sqlDeleteBuilder = new StringBuilder("DELETE FROM ");

//          ...

        this.sqlGetAll = sqlGetAllBuilder.toString();
        this.sqlUpdate = sqlUpdateBuilder.toString();
        this.sqlInsert = sqlInsertBuilder.toString();
        this.sqlDelete = sqlDeleteBuilder.toString();
```

Figura 11: parte da construção dos comandos em GenericDataMapper

Binder

Esta classe tem como função associar aos objectos java o valor recebido da base de dados. Apenas é instanciada após chamada ao build, criando o Builder com todas as classes parametrizadas na instancialização do Builder. Cada classe define um método bind, que recebe o target da afectação, o nome pelo qual se está a procura para afectar o objecto e o valor.

O binder recebe no seu constructor uma vararg de IBinder, e define um método bindTo que recebe o target da afectação e um mapa com o campo e valor a associar ao objecto. Por cada campo, é verificado se existe um IBinder capaz de fazer a afectação, no final se o objecto não tiver sido binded é lançada uma excepção para informar o cliente.

As classes que implementam o IBinder apenas podem ter dois tipos de constructores, ou um constructor vazio, ou um constructor que receba uma class<T>. Na instanciação da classe pelo build é dada preferência a construção através do constructor que recebe a classe.

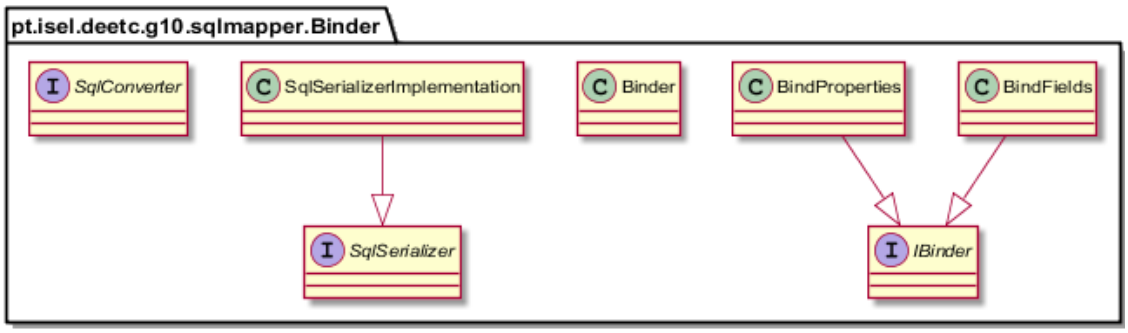


Figura 12: UML dos responsáveis que gravam os resultado vindos da BD numa instância de ED, tal como o processo inverso

SqlIterable

Esta classe tem o intuito de dar suporte ao encadeamenteo de cláusulas where (e bind sobre as cláusulas where) sobre o resultado do comando GetAll. Na figura 10 entra-se a interface que SqlIterable tem de implementar e que a mesma implementa a interface *autoclosable.*

```
public interface ISqlIterable<T> extends Iterable<T>,AutoCloseable{
    ISqlIterable<T> where(String clause);
    int count();
    SqlIterable<T> bind(Object ... bindArgs);
}
```

Figura 10: IsqlItarable que dá as assinaturas para o encadeamento de cláusulas

Where

Na classe SqlIterable existem 2 construtores, conforme indicado na figura 11. O primeiro construtor é sempre o primeiro a ser chamado, pois é ele que cria a instância da lista de Strings para que seja possível guardar todas as cláusulas where associadas ao comando. O segundo construtor é utilizado sempre que é adicionada uma nova cláusula Where, retornando uma nova instância de SqlEnumerable, já com a nova cláusula associada.

```
public class SqlIterable<T> implements ISqlIterable<T> {
    private final AbstractSqlExecutor exec;
    private final SqlConverter<T> converter;
    private final String query;
    private final Object[] args;
    private final List<String> whereClauses;
    private final List<IteratorImpl> iterators = new LinkedList<>();

    public SqlIterable(AbstractSqlExecutor executor, SqlConverter<T> converter,
            String query, Object... args) {
        this.query = query;
        this.args = args;
        this.exec = executor;
        this.converter = converter;
        this.whereClauses = new LinkedList<>();
    }

    private SqlIterable(AbstractSqlExecutor executor, SqlConverter<T> converter,
            String query, Object[] args, List<String> whereClause) {...7 lines }

    @Override
    public SqlIterable<T> where(String clause) {
        List<String> listWhere = new LinkedList<>(whereClauses);
        listWhere.add(clause);
        return new SqlIterable<>(exec, converter, query, args, listWhere);
    }
```

Figura 11: Contrutores de SqlItarable e implementação de Where

Binding

Na 3ª parte do trabalho foi-nos pedido para que o where possa incluir parâmetros que podem ser ligados (*bind*) a diferentes argumentos em tempo de execução. Para resolver este problema o nosso bind estamos a percorrer a nossa lista de cláusulas Where e a verificar se se encontra o character *'?'*. Se isso se verificar fazemos a substituição desse caracter pelo pelo próximo valor do array de objectos recebido por parâmetro.

Ainda na figura 12 é possível verificar que é só na altura em que o iterador é chamado que iremos adicionar as cláusulas where ao comando pedido. O iterador tem ainda funcão de retornar um novo IteratorImpl que contém a nossa implementação Lazy do iterador.

```
@Override
public SqlIterable<T> bind(Object... bindArgs) {
    List<String> where = new LinkedList<>();
    SqlIterable<T> iterable = this;
    if (!whereClauses.isEmpty()) {
        int bindIdx = 0;
        for (String clause : whereClauses) {
            if (clause.contains("?"))
                where.add(clause.replace("?", getBindArg(bindArgs[bindIdx++])));
        }
        if (bindIdx != bindArgs.length)
            Sneak.sneakyThrow(new InvalidParameterException("You must bind
        iterable = new SqlIterable(exec, converter, query, args, where);
    }
    return iterable;
}

@Override
public Iterator<T> iterator() {
    StringBuilder sqlQueryBuilder = new StringBuilder(query);
    if (!whereClauses.isEmpty()) {
        String delimiter = " ";
        sqlQueryBuilder.append(delimiter).append("WHERE");
        for (String clause : whereClauses) {
            sqlQueryBuilder.append(delimiter).append(clause);
            delimiter = " AND ";
        }
    }
    String sqlQuery = sqlQueryBuilder.toString();
    IteratorImpl iter = new IteratorImpl(exec, sqlQuery);
    iterators.add(iter);
    return iter;
}
```

Figura 12: implementação do bind e do iterator que retorna um novo IteratorImpl

IteratorImpl

O construtor desta implementação é que tem responsabilidade de abrir, executar o comando e fechá-lo quando não o precisa mais. O resultado da execução do comando Query é guardado num ResultSet e por cada pedido feito é extraído um linha desse ResultSet e feita a conversão necessária para obter e retornar uma instância de T.

```
private final class IteratorImpl implements Iterator<T>, AutoCloseable {
    private ResultSet rs;
    boolean calledHasNext = false;
    boolean hasNext = false;
    private final AbstractSqlExecutor exec;
    private PreparedStatement cmd;

    IteratorImpl(AbstractSqlExecutor exec, String query) {
        this.exec = exec;
        try {
            cmd = exec.beginConnection().prepareStatement(query);
            cmd.setFetchSize(1);
            int idx = 1;
            for (Object arg : args) {
                cmd.setObject(idx, arg);
                idx++;
            }
            rs = cmd.executeQuery();
        } catch (SQLException ex) {
            close();
            Sneak.sneakyThrow(ex);
        }
    }

    @Override
    public boolean hasNext() {...13 lines }

    @Override
    public T next() {...14 lines }

    @Override
    public void close() {...16 lines }
}
```

Figura 13: implementação Lazy de IteratorImpl

Lazy Load

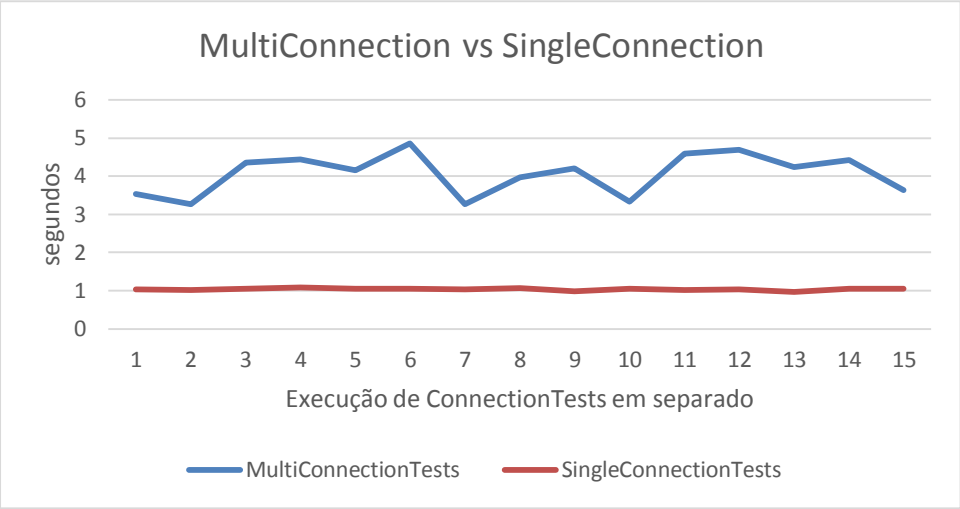
A implementação da leitura dos resultados da BD passou por duas fases distintas. Na primeira, utilizando semi-lazy load, adiavamos o pedido à base de dados ao máximo, apenas fazendo-o quando o cliente chamasse o iterator do Iterable implementado por nós.

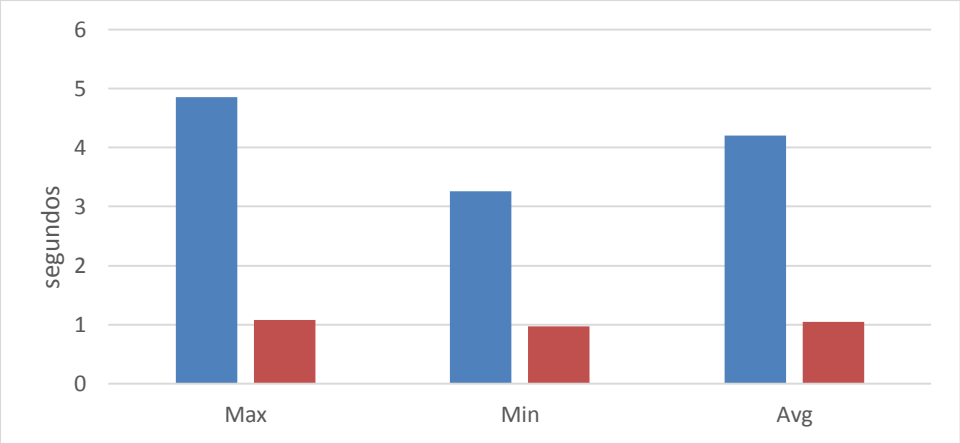
Nesta solução, apesar da leitura ser lazy, existe um inconveniente grande no uso de memória, pois o resultado da query é trazido todo para memória, mesmo que o cliente apenas pretenda iterar sobre um ou dois objectos. Para resolver esta situação, passamos a adiar ainda mais o carregamento dos dados, e passamos a implementar o lazy load de forma total, isto é, apenas traz para memória o objecto quando é pedido o next sobre o iterador.

Esta solução trouxe dificuldades acrescidas na gestão da ligação, pois esta não pode ser fechada, caso contrário o iterator não consegue retornar mais resultados após o fecho da ligação. De forma a que este inconveniente seja resolvido, criamos uma lista de iteradores, que contém todos os iteradores devolvidos pelo iterável, e quando implementamos a interface autocloseable tanto no iterador como no iterable.

Quando existe alguma excepção ou o iterador termina a sua execução ele fecha os seus recursos criticos e remove-se da lista. Quando é chamado o close sobre o iterable, este percorre a lista de todos os seus iteradores e fecha-os. Neste caso, os iteradores também deixarão de poder ser utilizados, no entanto nesta situação essa impossibilidade é uma consequência directa do cliente chamar o close, portanto o controlo está no cliente.

Comparação de performances





Podemos conferir que tendo um ligação aberta é mais eficiente do que abrir e fechar a ligação em cada comando executado. Porém é boa polícia usar múltiplas conexões caso exista muita concorrência no acesso á base de dados.