


Instituto Superior de Engenharia de Lisboa

Licenciatura em Engenharia Informática e de Computadores

Semestre de Verão 2013/2014



Trabalho Final Ambientes Virtuais de Execução

Eng.ª Cátia Vaz

Data de entrega: 07.07.2014

Trabalho elaborado por:

Pedro Lima nº 33684

Flávio Cadete nº 35383

LI41N – LEIC

Índice

Introdução.....	2
Estrutura da Solução do Trabalho	3
Entidades de Domínio	4
Classe Builder	5
Política de Gestão de Ligações	7
Command Builder	10
SqlEnumerable	12
Reflection	14
Lazy Load vs Semi-Lazy Load	14
MyMemberDictionary	15
FkMemberInfo	15
BindMember	16
Custom Attributes	17
Comparação de performances	18

Introdução

O objectivo deste trabalho é desenvolver uma framework capaz de criar instâncias de uma implementação de uma interface IMapper para uma determinada entidade de domínio tirando partido do serviço de reflexão do .NET e de um Dynamic Proxy proveniente do package LinFu que nos foi fornecido no início do trabalho.

Estrutura da Solução do Trabalho

A solução do trabalho está dividida em 3 projectos diferentes:

- ➔ **Projecto SqlMapperClient:** contém todas as entidades de domínio definidas;
- ➔ **Projecto SqlMapperFw:** contém tudo o que corresponde á implementação da framework SqlMapper;
- ➔ **Projecto SqlMapperTests:** contém todos os testes unitários do trabalho.

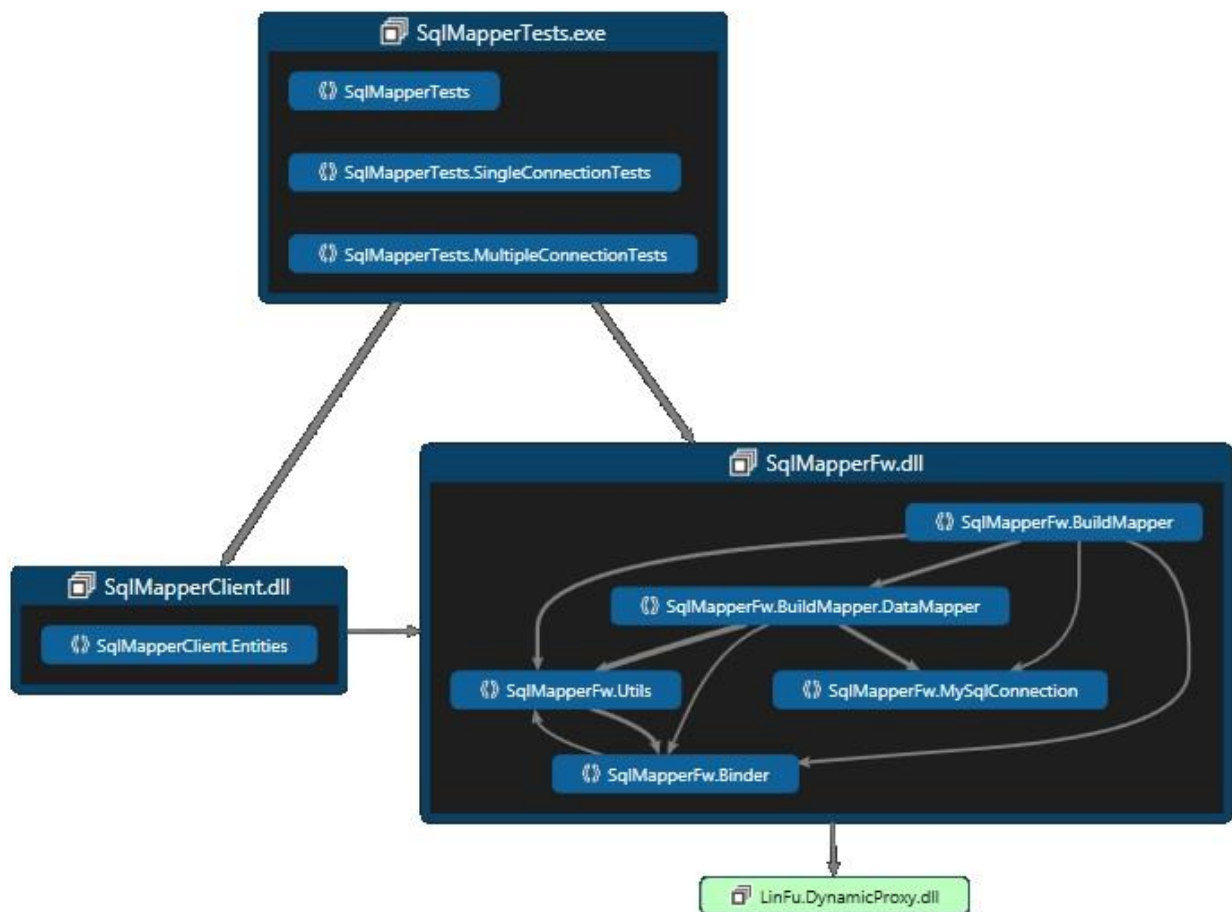


Figura 1: Dependências entre os 3 projectos e a biblioteca LinFu

Entidades de Domínio

De forma a dar suporte à 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 da tabela na base de dados correspondente. Esta anotação é garantida através do CustomAttribute DBTableNameAttribute, criado para este propósito.

Os campos contidos nestas classes têm nomes, que por omissão, correspondem ao nome da coluna da respectiva tabela, no entanto, se um campo estiver anotado com o CustomAttribute DBNameAttribute, esse campo corresponde à coluna da tabela que tiver um nome igual ao que está contido no CustomAttribute em vez do nome do campo.

Foi ainda criado um outro CustomAttribute, de nome PKAttribute, que tem como objectivo indicar qual é o campo, dentro de uma entidade de domínio, que representa a chave primária da tabela correspondente.

É possível verificar, através da Figura 2, a definição de uma entidade de domínio existente no trabalho.

```
[DBTableName("Products")]
public class Product
{
    [PK("ProductId")]
    public Int32 ID
    { set; get; } //PK
    public String ProductName { set; get; }
    [FK]
    public Supplier Supplier { get; set; }
    public String QuantityPerUnit { set; get; }
    public Decimal UnitPrice { set; get; }
    public short UnitsInStock { set; get; }
    public short UnitsOnOrder { set; get; }
    public short ReorderLevel;
    public Boolean Discontinued;

    public override string ToString()
    {
        return "{Product: " + ID + ", " + ProductName + ", " + Supplier.ID + ", " + QuantityPerUnit + ", " +
            UnitPrice + ", " + UnitsInStock + ", " + UnitsOnOrder + ", " + ReorderLevel + ", " + Discontinued + "}";
    }
}
```

Figura 2: Classe Product que está associada à tabela Products e que têm uma associação para a entidade Supplier

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:

- ➔ Connection String Builder: String com os dados necessários para a correcta conexão à base de dados pretendida;
- ➔ 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 `AbstractSqlConnection` de `SqlMapperFw`;
- ➔ Tipo de mapeamento pretendido entre a entidade de domínio e as colunas da tabela: se baseado no nome dos campos ou das propriedades da entidade de domínio, ou ainda outro mapeamento qualquer que seja implementado à posteriori e que extenda `AbstractBindMember` de `SqlMapperFw`;

Construído o objecto da classe builder, é necessário invocar o método `build` de forma a obter o data mapper pretendido. Nesta chamada ao método `build`, que é genérico, apenas é necessário indicar a qual entidade de domínio vai estar o data mapper associado. Tendo o data mapper é possível invocar todos os métodos nele contidos, sabendo que todos iram manipular a entidade de domínio passada no método `build`.

A Figura 3 representa um exemplo do que foi descrito para a entidade de domínio `Product`.

```
List<Type> bindMemberList = new List<Type> {typeof (BindFields), typeof (BindProperties)};
Builder builder = new Builder(_connectionStringBuilder, typeof(SingleSqlConnection), bindMemberList);
IDataMapper<Product> productDataMapper = builder.Build<Product>();
productDataMapper.GetAll();
```

Figura 3

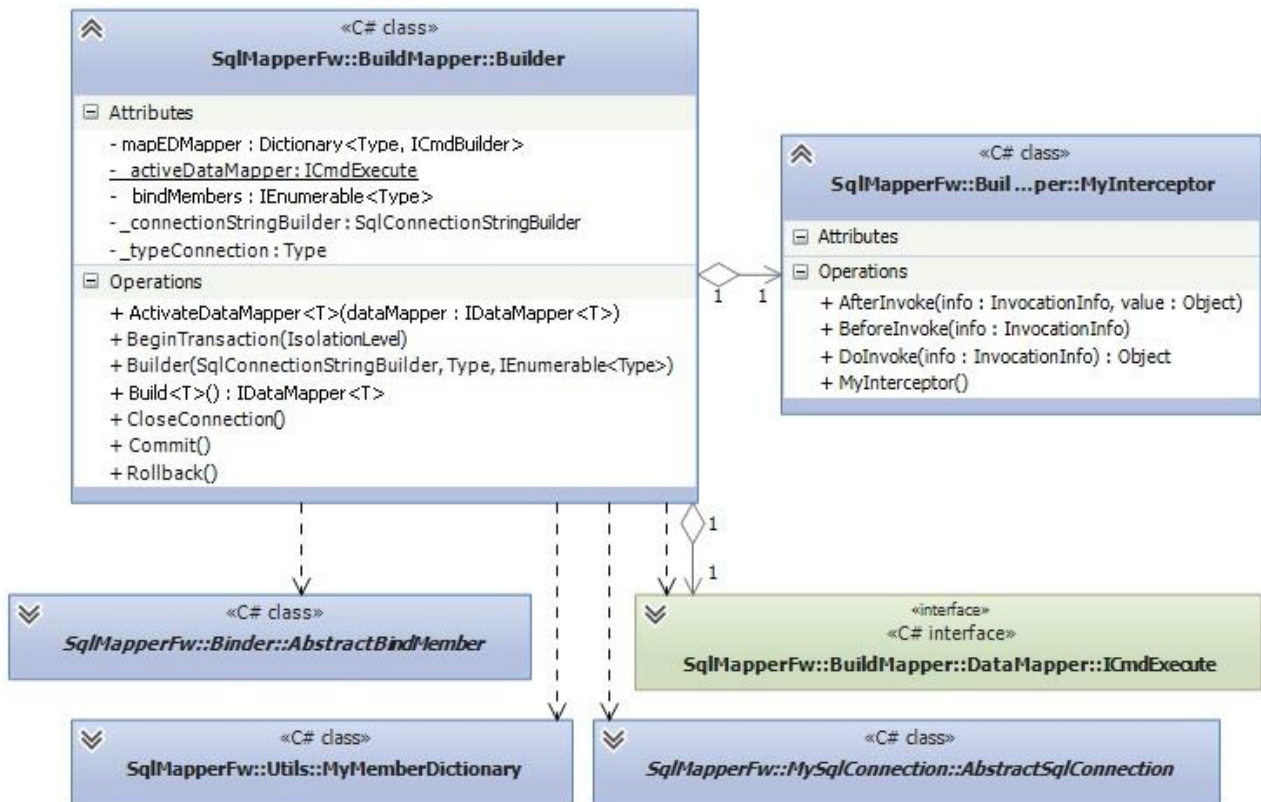


Figura 4: principais associações e dependências de Builder

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 `IMySqlConnection` que contém quais os métodos que têm que ser definidos pelas classes que a implementem.

A Figura 5 contém a definição da interface `IMySqlConnection`:

```
public interface IMySqlConnection
{
    void CloseConnection();
    void BeginTransaction(IsolationLevel isolationLevel);
    void Rollback();
    void Commit();
}
```

Figura 5: Interface `IMySqlConnection`

Devido ao facto de haver diversos troços de código iguais para diferentes tipos de ligações foi criada uma classe que implementa a interface `IMySqlConnection`, a classe abstracta `AbstractSqlConnection`, sendo que as classes que correspondem aos tipos de ligações, estendem desta classe abstracta, partilhando o código comum na classe do qual estendem e implementando o código específico do tipo de ligação na própria classe.

```
public abstract class AbstractSqlConnection : IMySqlConnection
{
    internal SqlConnection Connection { get; set; }
    internal SqlTransaction SqlTransaction;

    public abstract void Commit();
    public abstract void Rollback();
    internal abstract void AfterCommandExecuted();
    protected internal abstract void BeforeCommandExecuted();

    public void BeginTransaction(IsolationLevel isolationLevel) {...}

    public bool IsActiveConnection() {...}

    public void OpenConnection() {...}

    public void CloseConnection() {...}
}
```

Figura 6: Classe `AbstractSqlConnection`

Como é possível verificar na Figura 4, esta classe contém os campos associados ao tipo de ligação, implementa alguns métodos da interface `IMySqlConnection`, deixando alguns como abstract, pois a sua implementação só deve ser definida mais “abaixo”.

Conexão Única: A figura 7 demonstra como foi implementado este tipo de ligação.

```
public class SingleSqlConnection : AbstractSqlConnection
{
    public SingleSqlConnection(SqlConnectionStringBuilder connString)
    {
        Connection = new SqlConnection(connString.ConnectionString);
        OpenConnection();
    }
    public override void Rollback()
    {
        if (SqlTransaction == null || !IsActiveConnection())
        {
            Console.WriteLine("Cannot Rollback! Transaction doesn't have a active connection or is null!");
            return;
        }
        SqlTransaction.Dispose();
        SqlTransaction.Rollback();
        SqlTransaction = null;
    }
    public override void Commit()
    {
        if (SqlTransaction == null || !IsActiveConnection())
        {
            Console.WriteLine("Cannot Commit! Transaction doesn't have a active connection or is null!");
            return;
        }
        SqlTransaction.Commit();
        SqlTransaction = null;
    }
    protected internal override void BeforeCommandExecuted(){}
    internal override void AfterCommandExecuted(){}
}
```

Figura 7: Classe SingleSqlConnection

Múltiplas Conexões: A figura 6 demonstra como foi implementado este tipo de ligação.

```
public class MultiSqlConnection : AbstractSqlConnection
{
    public MultiSqlConnection(SqlConnectionStringBuilder connString)
    {
        Connection = new SqlConnection(connString.ConnectionString);
    }
    public override void Rollback()
    {
        if (SqlTransaction == null || !IsActiveConnection())
            throw new Exception("Cannot Rollback! Transaction doesn't have a active connection or is null!");

        SqlTransaction.Dispose();
        SqlTransaction.Rollback();
        SqlTransaction = null;
        CloseConnection();
    }
    public override void Commit()
    {
        if (SqlTransaction == null || !IsActiveConnection())
            throw new Exception("Cannot Commit! Transaction doesn't have a active connection or is null!");

        SqlTransaction.Commit();
        SqlTransaction = null;
    }
    protected internal override void BeforeCommandExecuted()
    {
        OpenConnection();
        BeginTransaction(IsolationLevel.ReadUncommitted);
    }
    internal override void AfterCommandExecuted()
    {
        Commit();
        CloseConnection();
    }
}
```

Figura 8: Classe MultipleSqlConnection

As diferenças na implementação destes 2 tipos de ligação são chamadas ao método `CloseConnection` dentro do método `Rollback` e a implementação dos métodos `BeforeCommandExecuted` e `AfterCommandExecuted` na classe `MultipleConnection`. Este método é chamado porque quando o tipo de ligação é `MultipleConnection` é estabelecida uma ligação á base de dados por cada execução de um método do data mapper, sendo assim necessário fechar a ligação quando o método termina. Na classe `SingleConnection` não é invocado o método `CloseConnection`, pois só é necessário estabelecer uma única conexão, ficando o utilizador com a responsabilidade de fechar a conexão quando entender que já fez tudo o que pretendia.

Foi também implementado em ambos os tipos uma política de gestão de ligações com suporte para iniciar e finalizar uma transacção através de `rollback` e `commit` explícito, que faz parte dos requisitos pedidos.

Na Figura 9 é possível ver a estrutura resultante desta implementação que segue parcialmente o P.D. Strategy em que tenho o esqueleto (método `execute` em `CmdBuilder`) e 2 métodos ganchos (`AfterCommandExecuted` e `BeforeCommandExecute`) abstractos e a serem implementados pelas implementações de `AbstractSqlConnection` e que indicam o comportamento variável.

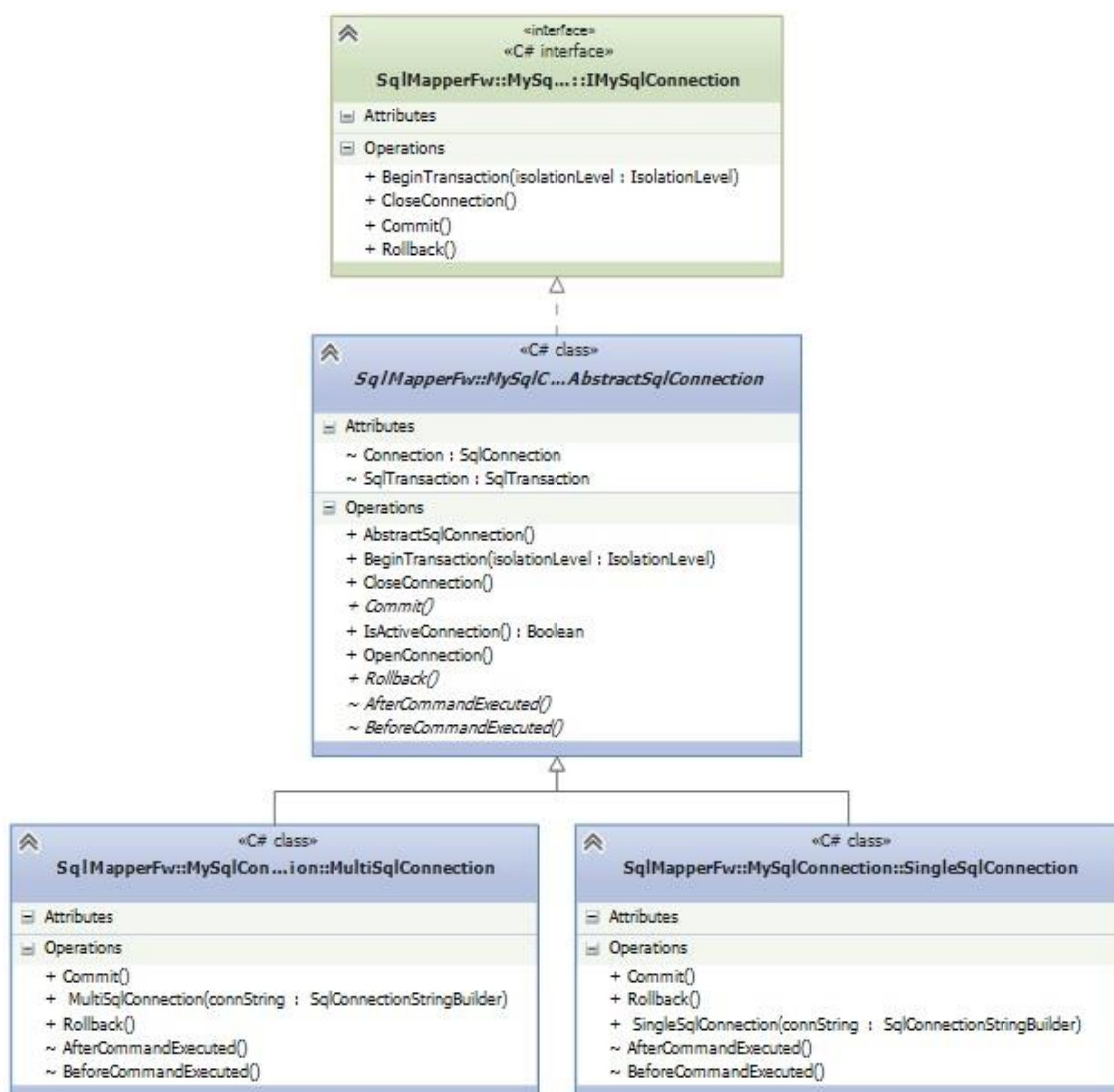


Figura 9: Estrutura das conexões

Command Builder

Para dar suporte à criação, implementação e execução dos comandos SQL foi implementada a estrutura da Figura 10.

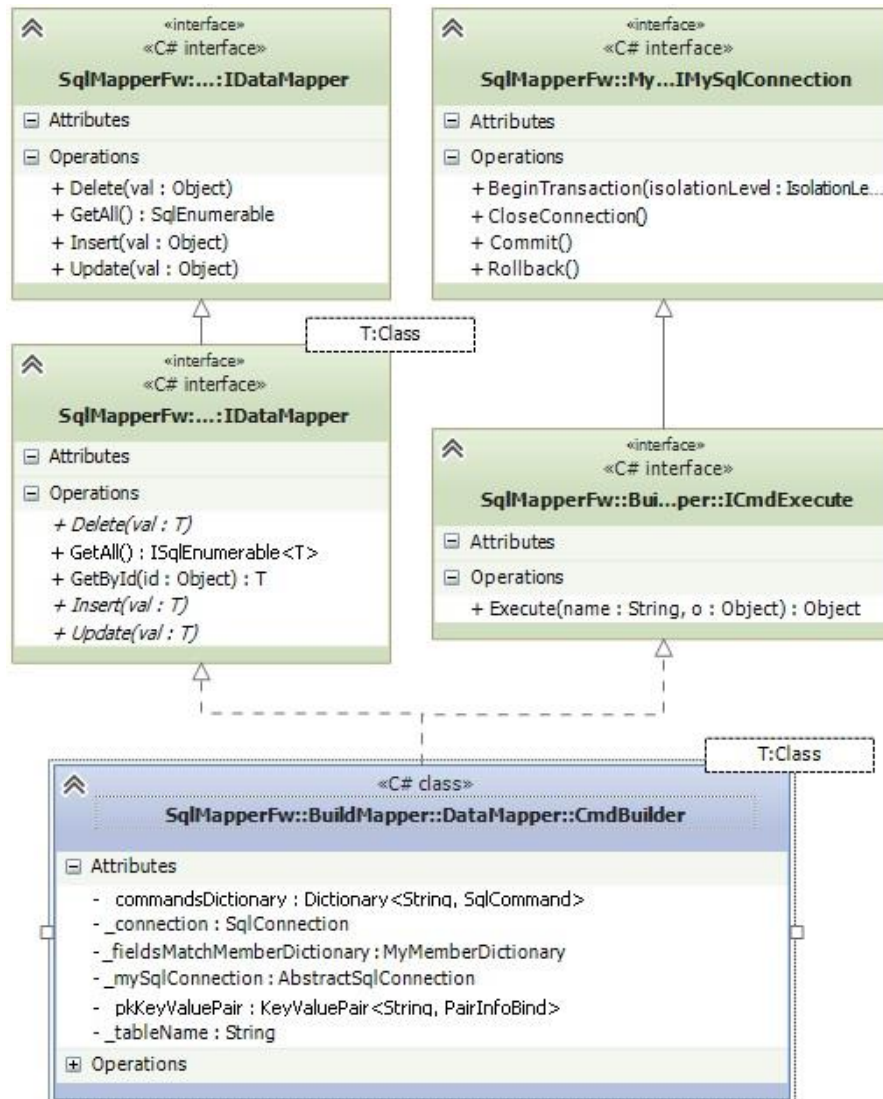


Figura 10: Implementações de CmdBuilder

A implementação do command builder é feita com base nas interfaces:

- ➔ **IDatamapper**: fornecida no enunciado, que indica quais os métodos a implementar para manipular a base de dados;
- ➔ **ICmdExecute**: contém apenas um método(`Execute`), além dos recebidos implicitamente de `IMySqlConnection`, que indica a assinatura necessária para a execução de um comando;
- ➔ **IMySqlConnection**: definições que o `CmdBuilder` tem de implementar para servir de ponte entre o Builder e a chamada dessas implementações na conexão.

Na instanciação de CmdBuilder é passada a conexão e informação relacionada com a entidade de domínio na qual os comandos vão incidir, permitindo a criação dos comandos SQL nesse momento. A utilização dos mesmos é apenas feita aquando fôr requisitado a execução desse comando, minimizando assim o custo de estar a criar um comando cada vez que é pedido para executar esse comando.

Na construção dos comandos SQL, cada comando é adicionado a um dicionário de comandos(commandsDictionary), onde cada comando está associado a uma string identificadora desse mesmo comando.

Ao longo da implementação do CmdBuilder há a preocupação de reduzir ao mínimo possível o uso da reflexão do .NET na execução dos comandos maximizando assim a eficiência da solução, pois a reflexão é maioritariamente utilizada na construção(build) do data mapper.

Apesar desta preocupação, existem casos onde é inevitável o uso desta funcionalidade do .NET, nomeadamente na iteração do resultado da execução do método GetAll, pois é criada uma instância da entidade de domínio em cada iteração. Em todos os métodos é necessário utilizar os binders(get/set) respectivos aos fields da entidade de domínio que também utiliza reflexão.

SqlEnumerable

Esta classe foi criada para dar suporte ao comando GetAll e que é utilizada no IMapper como é possível verificar na figura 10 e implementa a interface ISqlEnumerable.

```
public interface ISqlEnumerable<out T> : IEnumerable<T>
{
    ISqlEnumerable<T> Where(string clause);
    int Count();
}
```

Figura 11: Interface ISqlEnumerable

Na classe SqlEnumerable existem 2 construtores, conforme indicado na figura x. 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 cláusula nova, pois é retornado uma nova instância de SqlEnumerable, já com a nova cláusula associada.

```
public sealed class SqlEnumerable<T> : ISqlEnumerable<T>
{
    internal readonly SqlCommand MySqlCommand;
    private readonly string _tableName;
    internal readonly Dictionary<string, PairInfoBind>.ValueCollection MembersInfoBind;
    internal readonly PairInfoBind PkMemberInfoBind;
    internal readonly CloseConnection CloseSqlConnection; //depends on type of connection
    internal List<string> WhereClauses;

    public delegate void CloseConnection();

    public SqlEnumerable(SqlCommand cmd,
        String tableName,
        Dictionary<string, PairInfoBind>.ValueCollection membersInfoBind,
        PairInfoBind pkMemberInfoBind,
        CloseConnection closeSqlConnection) //depends on type of connection [...]

    public SqlEnumerable(SqlCommand cmd,
        String tableName,
        Dictionary<string, PairInfoBind>.ValueCollection membersInfoBind,
        PairInfoBind pkMemberInfoBind,
        CloseConnection closeSqlConnection, //depends on type of connection
        List<string> whereClauses) [...]

    public ISqlEnumerable<T> Where(string clause) [...]

    public int Count() [...]

    public IEnumerator<T> GetEnumerator() [...]

    IEnumerator IEnumerable.GetEnumerator() [...]
}
```

Figura 12: Implementação classe SqlEnumerable

No construtor da classe `SqlEnumerator` cada uma das cláusulas contidas na lista `whereClauses` são incluídas na query do comando aquando da sua execução. No fim, para que a query não fique com as cláusulas do `where`, é reposta novamente a query inicial do comando.

Ainda na classe `SqlEnumerable`, os métodos `GetEnumerator` e `Where` retornam sempre uma nova instância de `SqlEnumerable` para evitar que o iterador retornado seja partilhado por vários `IEnumerables`.

```
public sealed class SqlEnumerator<T> : IEnumerator<T>
{
    readonly SqlEnumerable<T> _mySqlEnumerable;
    readonly SqlDataReader _sqlDataReader;

    private bool gotCurrent;
    private T current;

    public SqlEnumerator(SqlEnumerable<T> mySqlEnumerable) ...

    public void Dispose() ...

    public bool MoveNext() ...

    public void Reset() ...

    public T Current ...

    object IEnumerator.Current ...
}
```

Figura 13: Implementação classe `SqlEnumerator`

Na execução do comando `GetAll` é passado o método `AfterCommandExecute` da conexão ao `SqlEnumerable<T>` e recebido como um delegate que tem responsabilidade à assinatura de `AfterCommandExecute`, e que é executado, com o nome do delegate `CloseConnection`, no momento em que é feito `Dispose` do `SqlDataReader`.

Reflection

Os métodos criados na implementação da framework que fazem uso da reflexão do .NET são os métodos contidos no ficheiro ReflectionMethods.cs e os binders.

```
public static class ReflectionMethods
{
    public static bool ImplementsInterface(this Type t, Type tIntf)...
    public static bool IsPrimaryKey(this MemberInfo type)...
    public static bool IsForeignKey(this MemberInfo type)...
    public static string GetTableName(this Type type)...
    public static string GetDBFieldName(this MemberInfo type)...
    public static MemberInfo GetPkMemberInfo(this Type type)...
    public static Object GetEDFieldValue(this Object instance, MemberInfo mi, AbstractBindMember bm)...
    private static object ValidDBValue(object Value)...
    public static Object BindEDFieldValue(this Object instance, MemberInfo mi, AbstractBindMember bm, Object dbvalue)...
    // Convert System.Type to SqlDbType
    public static SqlDbType GetSqlDbType(this MemberInfo mi, Object instance, AbstractBindMember bm)...
    public static SqlDbType GetSqlDbType(object value)...
```

Figura 14: Classe ReflectionMethods

Lazy Load vs Semi-Lazy Load

Durante a implementação da framework cruzámo-nos com esta questão em relação á informação obtida através do comando GetAll. Optando pela versão Semi-Lazy Load existia a possibilidade de haver uma excepção por falta de memória pois era necessário guardar todas as instâncias de T na execução do método build. Para além deste problema, seria ainda preciso aumentar a capacidade do array utilizado para guardar as instâncias de T sempre que o limite deste fosse igualado. Por outro lado a versão Lazy Load não provoca este tipo de problemas mas implica a utilização da reflexão pois é necessário criar uma instância de T á medida que se for avançando no iterador.

Após saudável discussão optámos por implementar a versão Lazy Load porque é melhor suportar o custo da reflexão durante a execução do que enfrentar todos os outros problemas da solução Semi-Lazy Load.

MyMemberDictionary

A razão para o que nos levou a criar esta classe e não utilizar um dicionário com `keyValuePair` foi devido a aumentar a simplicidade, clareza e expressividade do nosso código.

```
public struct PairInfoBind
{
    public MemberInfo MemberInfo;
    public AbstractBindMember BindMember;

    public PairInfoBind(MemberInfo memberInfo, AbstractBindMember bindMember)
    {
        MemberInfo = memberInfo;
        BindMember = bindMember;
    }
}

public class MyMemberDictionary : Dictionary<String, PairInfoBind>
{
    public void Add(String key, MemberInfo memberInfo, AbstractBindMember bindMember)
    {
        Add(key, new PairInfoBind(memberInfo, bindMember));
    }
}
```

Figura 15: Implementação da classe `MyMemberDictionary` e da estrutura auxiliar `PairInfoBind`

FkMemberInfo

Esta classe foi criada com o intuito de encapsular as chaves estrangeiras como um `MemberInfo` e poderem ser guardadas no `MyMemberDictionary` e utilizadas igualmente aos `MemberInfos` dos restantes campos da entidade de domínio.

Esta implementação segue o P.D. Adaptor (objectos).

```
public class FkMemberInfo : MemberInfo
{
    public Type InstanceType { get; private set; }
    public MemberInfo PkInfo { get; private set; }
    private String PkName { get; set; }
    public Object MyInstance { get; private set; }
    public MemberInfo ToBindInfo { get; private set; }

    public FkMemberInfo(MemberInfo memberInfo, Type instanceFrom, AbstractBindMember bindMember) {...}

    public override object[] GetCustomAttributes(bool inherit){...}

    public override object[] GetCustomAttributes(Type attributeType, bool inherit){...}

    public override bool IsDefined(Type attributeType, bool inherit){...}

    public override MemberTypes MemberType{...}

    public override string Name{...}

    public override Type DeclaringType{...}

    public override Type ReflectedType{...}
}
```

Figura 16: Implementação da classe `FKMemberInfo`

BindMember

É-nos pedido no enunciado para que possa haver diversos tipos de mapeamento entre a entidade de domínio e a tabela presente na base de dados.

Para dar essa funcionalidade á framework cliente, criá-mos uma classe abstracta com um método *bind* que irá fazer as verificações necessárias e chama o método *SetValue* que tem como função afectar o campo da MemberInfo fornecida.

Todos os derivados de AbstractBindMemeber, além do método abstracto SetValue, também terão de implementar os restantes método abstractos e que são necessários para a correcto mapeamento e uso do seu tipo.

Esta implementação segue o P.D. Template Method, com o bind a ser o esqueleto e o SetValue o método gancho abstracto, com todos os derivado a terem que implementar este método.

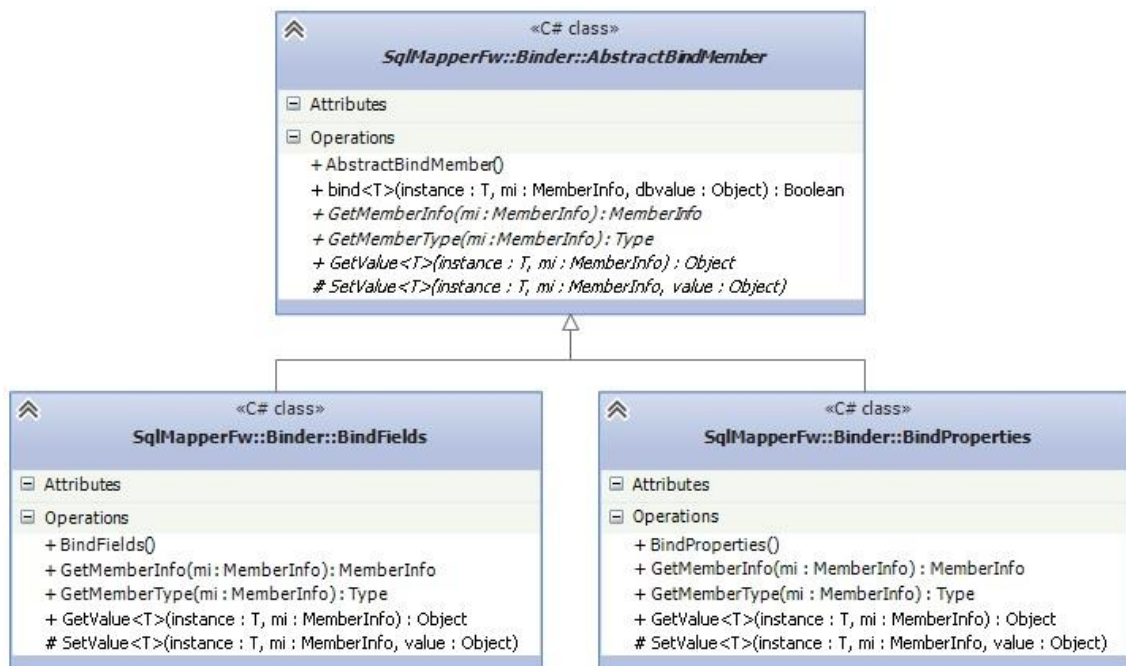


Figura 17: Diagrama UML dos Binders

```
public class BindProperties : AbstractBindMember
{
    public override MemberInfo GetMemberInfo(MemberInfo mi)
    {
        return (mi.MemberType == MemberTypes.Property) ? mi : null;
    }

    public override Type GetMemberType(MemberInfo mi)
    {
        return ((PropertyInfo)mi).PropertyType;
    }

    protected override void SetValue<T>(T instance, MemberInfo mi, object value)
    {
        if(mi.MemberType == MemberTypes.Property)
            ((PropertyInfo)mi).SetValue(instance, value);
    }

    public override object GetValue<T>(T instance, MemberInfo mi)
    {
        return (mi.MemberType == MemberTypes.Property)?((PropertyInfo)mi).GetValue(instance, null): null;
    }
}
```

Figure 18: Exemplo de uma implementação de um determinado tipo de mapeamento, neste caso, de propriedades

Custom Attributes

Para a correcta implementação da framework foram criados diversos Custom Attributes, como já foi possível verificar. A implementação destes Custom Attributes foi feita num único ficheiro(MyAttributes.cs), de forma a ficarem todos os atributos juntos.

```
[AttributeUsage(AttributeTargets.All, AllowMultiple = true)]
public class DBNameAttribute : Attribute
{
    public string Name;

    public DBNameAttribute(string name)
    {
        Name = name;
    }
}

[AttributeUsage(AttributeTargets.Class, AllowMultiple = false)]
public class DBTableNameAttribute : DBNameAttribute
{
    public DBTableNameAttribute(string name) : base(name){}
}

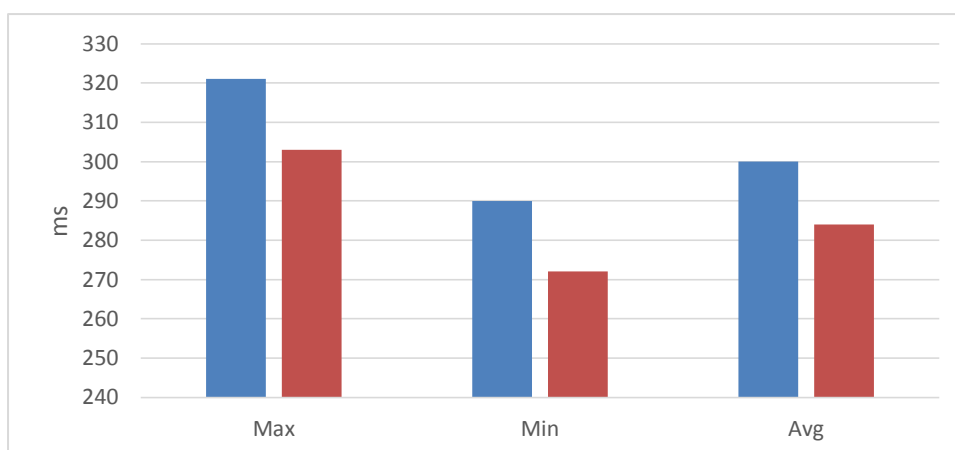
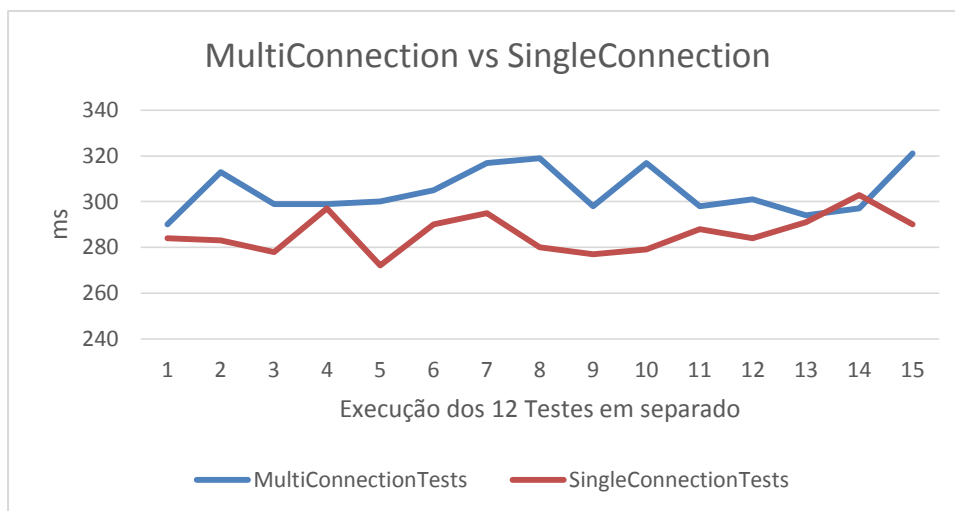
[AttributeUsage(AttributeTargets.All, AllowMultiple = false)]
public class PKAttribute : DBNameAttribute
{
    public PKAttribute(string name) : base(name){}
}

[AttributeUsage(AttributeTargets.All, AllowMultiple = true)]
public class FKAttribute : Attribute
{
}
```

Figura 19: Custom Attributes

Os Custom Attributes DBNameAttribute, PKAttribute e FKAttribute têm como target qualquer tipo de forma a que o mapeamento entre a entidade de domínio e as colunas da tabela não fique restrito a apenas propriedades ou campos mas sim a qualquer tipo de mapeamento que o utilizador possa implementar no futuro.

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 política usar múltiplas conexões caso exista muita concorrência no acesso á base de dados.