Requisitos tecnológicos:

- Como tecnologia de ligação à base é usado ADO.NET e com classes *connected*.
- Para criação de proxy dinâmico é usada a biblioteca Linfu.

Esta *framework* gera automaticamente *data access objects* (DAO) para a especificação fornecida por uma interface.

- A *framework* cria uma implementação da interface e dos seus métodos, com base na informação fornecida nas anotações;
- A anotação `SqlCmd` especifica o comando SQL associado à execução do respectivo método;
- Os argumentos de um método serão ligados (*bind*) aos respectivos parâmetros do comando SQL especificado.

A listagem seguinte ilustra um exemplo de especificação de um DAO para a entidade `Product` e a sua instanciação através da classe `Builder` da *framework*.

```
public interface ProductDao {

  [SqlCmd("SELECT ProductID, ProductName, UnitPrice, UnitsInStock FROM Products")]
  IEnumerable<Product> getAll();

  [SqlCmd("SELECT ProductID, ProductName, UnitPrice, UnitsInStock FROM Products WHERE ProductID = @id")]
  Product getById(int id);

  [SqlCmd("SELECT ProductID, ProductName, UnitPrice, UnitsInStock FROM Products WHERE UnitPrice < @price ")]
  IEnumerable<Product> getWithMaxPrice(double price);


  [SqlCmd("UPDATE Products SET ProductName = @name, UnitPrice = @price, UnitsInStock = @stock WHERE ProductID = @id")]
  void update(String name, double price, int stock, int id);

  [SqlCmd("DELETE FROM Products WHERE ProductID = @id")]
  void delete(int id);

  [SqlCmd("INSERT INTO Products (ProductName, UnitPrice, UnitsInStock) VALUES (@name, @price, @stock)")]
  Product insert(String name, double price, int stock);

}
```

```
String connStr = ...;
ProductDao dao = DaoBuilder.build<ProductDao>(connStr);
IEnumerable<Product> res = dao.getAll();
```

A criação de um DAO é feita chamando um dos métodos **Build** do **DaoBuiler** passando como parâmetro de tipo a interface com os métodos a implementar.

Os métodos desta interface que queiram ser implementados devem estar marcados com o atributo **SqlCmdAtribute**, especificando neste atributo o comando **SQL** a ser executado na chamada do método marcado. Qualquer método que não tenha este atributo será ignorado.

Sobre os parâmetros dos métodos marcados são adquiridos os seus nomes, pois de acordo com a especificação estes nomes são coincidentes com os nomes dos parâmetros dos comandos, e os seus tipos para a formação da chave para os identificar.

A implementação destes métodos é gravada numa instância da classe **DaoMethodCollection** usando os vários tipos de métodos **AddMethod** para os adicionar, passando como parâmetro o tipo de retorno, os parâmetros e o corpo do método tal como as informações relacionadas com o Sql: O comando e a *string* de conexão. Ao adicionar um novo método é também criado um *delegate* do tipo **AccessorCall** que serve fazer a chamada ao método e tratar do seu tipo de retorno.

Esta classe tem internamente um dicionário de estruturas do tipo **Method** indexado por estruturas do tipo **MethodKey**. O tipo Method é apenas um contentor das coisas relacionadas com o método a implementar, implementando métodos *Equal* e *GetHashCode* pois a comparação de structs usa reflexão. A sua chave, MethodKey, representa o que é usado para identificar um método (nome, parâmetros e tipo de retorno) fazendo as comparações necessárias para tal.

Para os métodos que retornam um enumerável nas soluções que usam reflexão é necessário criar um novo tipo da classe genérica **ObjectIEnumerable<T>** para ser feito um cast dos objectos retornados pelos métodos dos accessors (tipo **Object**) para o tipo correcto de retorno.
Por fim é criado um Proxy para o Dao a ser criado usando a classe **DaoInterceptor** como o interceptor dos métodos que irá chamar o AccessorCall do método encontrado.

Internamente o método Build recebe um **IAcessor** que define como é que o acesso a base de dados será feito. Como as três soluções não diferem no acesso a base de dados foi criado uma classe abstracta **AbstractAcessor** que contêm o código de acesso a base de dados.
Os vários métodos do AbstractAcessor adicionam os valores passados como parâmetros ao comando Sql e executa-o delegando a criação dos objectos, quando necessário, as classes que o estendem. Para os métodos que retornam um escalar (tipo primitivo, string ou datetime) ou um enumerável de um escalar a criação de objectos não é necessária e por isso não é delegada.

Assim, as três soluções diferem em como as criações dos objectos é feita.
A primeira, definida com o tipo **GenericAcessor,** é aquela que é chamada com o método **Build<TDao,T>** sendo TDao a interface do Dao a ser criado e T o tipo único não escalar de retorno que a interface tem. Este T tem definir um constructor sem parâmetros pois é assim que um novo objecto será criado e também deve implementar a interface **ISettable** que define o método **Set** que será chamado com os nomes e valores conseguidos da execução do comando.

A vantagem desta solução é que não usa reflexão da execução dos vários métodos, tendo um maior desempenho em relação as outras soluções. Permite também que os valores não sejam gravados num objecto com campos definidos para o que é retornado pelo o comando (ex: O ISettable pode ser apenas uma coleção de valores permitindo que seja usado para conseguir facilmente valores de várias tabelas diferentes no mesmo comando).

As desvantagens são o facto de o seu uso é menos intituivo que os outros métodos, só permitir o retorno de um tipo não escalar e não permitir o retorno de retorno de enumeráveis de escalares devido ao seu uso de reflexão (decisão para não implementar foi feita devido ao desejo de não usar reflexão nesta solução) e se o tipo de retorno for um objecto para ser mapeado então obrigado que o *backing field* das propriedades deste objecto estejam numa coleção, aumentando o código necessário por parte do programador.

A segundo solução, definida com o tipo **ObjectAccessorWithProxy,** sendo chamada com método **Build<TDao>(connStr, true)** é aquela que cria um proxy para os objectos de retorno do DAO não escalares usando reflexão mas não usa reflexão para afectar as suas propriedades. O proxy criado tem um dicionário de objectos que terá todas as propriedades do objecto criado.
A vantagem é o seu diminuido uso de reflexão em comparação com a terceira solução, tendo um maior desenpenho na execução do método mas a desvantagem é o facto de só poder gravar os valores se o estado forem propriedades virtuais.

A terceira e última solução, definida com o tipo **ObjectAcessor**, sendo chamada com o método **Build<TDao>(connStr[, false])** é a solução regular que usa reflexão tanto para a criação do objecto como a afectação do seu estado.
A vantagem é que as propriedades já não precisam de ser marcadas como virtuais e o estado pode também estar em campos. Tendo a desvantagem do maior uso de reflexão e menor desempenho na chamada do método.