AVE

Verão 2015/2016

Série de Exercícios 3

Grupo 1

João Pereira nº 37628

Henrique Calhó nº 38245

Edgar Demétrio nº 39393

Docente

Miguel Gamboa Carvalho

Objetivo

Implementar uma biblioteca, *AutoMapper*, que realize o mapeamento entre propriedades de objetos de tipos diferentes, valor ou referência. Esta biblioteca também deve permitir o mapeamento entre sequências de elementos.

Parte 1

Nesta fase do trabalho o *AutoMapper* só teria que mapear as propriedades que tivessem o mesmo nome e tipo compatível garantindo o mapeamento entre sequências de elementos para um tipo pretendido. E.g. *Map<List<Person>>(…)* esta chamada, com este parâmetro de tipo, iria produzir uma lista de *Person* a partir dos *Student* recebido como enumerável.

Para garantir a forma de execução indicada no enunciado, *AutoMapper.Build<Student, Person>().CreateMapper(),* foi necessário criar uma classe auxiliar de modo a garantir que os parâmetros de tipo só fossem passados ao *Build* e não ao *CreateMapper*, sendo apelidada de *PropertyBuilder*. O *PropertyBuilder* tem como objetivo inicial encontrar as propriedades que validem os critérios de seleção, nesta primeira fase apenas o mesmo nome e tipo compatível, e retornar uma instância de *PropertyMapper*, classe que implementa a interface *Mapper*, em que lhe é passado a lista das propriedades que cumprem os requisitos.

PropertyMapper, sendo a classe que implementa *Mapper*¸ contém as implementações de *Map* e *Map<TColDest>*. A logica por de trás de *Map* é instanciar o tipo de retorno, *TDest*¸ seja ele tipo valor ou referência tendo construtor com ou sem parâmetros, percorrer a lista de propriedades validas, passada por parâmetro na criação do *PropertyMapper,* obter o valor da propriedade do tipo fonte e afetar a propriedade equivalente no tipo destino, através do uso de reflexão. Em *Map<TColDest>* é criada uma instância do tipo da *ICollection* pretendida e é percorrido o *IEnumerable* passado como parâmetro ao método utilizando a função *Map* com cada elemento deste enumerável e preenchendo a coleção com o objeto retornado.

Foi tido em atenção só transformar, fazer o *cast*, do objeto retornado pela chamada a *Activator.CreateInstance* no final do método *Map*. Uma vez que ocorreu o problema, no caso do mapeamento para tipos valor, de como estarmos a fazer logo a transformação assim que obtínhamos o objeto não era observável a modificação das propriedades do tipo destino pois estaríamos a alterar os valores do resultado do *unbox* do objeto e não os valores do objeto retornado pelo *CreateInstance*.

Parte 2

Nesta fase foi-nos pedido para adicionar a possibilidade de mapear sequências de elementos para *arrays* ou *IEnumerable<T>.* Em *MapToArray* foi seguida a mesma logica implementada em *Map<TColDest>* só que aqui já sabíamos qual o tipo a criar, um *array* do tipo *TDest*. Em *MapLazy*, em que era necessário garantir que os elementos mapeados só "existissem" no momento da sua iteração (processamento *lazy*) foi utilizado a instrução *yield,* de modo a "atrasar" existência do objeto retornado pelo *Map*.

Parte 3

Nesta parte do trabalho é-nos requisitado a possibilidade de indicar quais as propriedades que deverão ser ignoradas pelo *AutoMapper,* ou indicando o nome e/ou indicando o *CusttomAttribute* anotado nas propriedades.

Para garantir esta seleção foi criada uma lista parametrizada com Func<PropertyInfo, bool> de validadores, em que é adicionado o critério. E.g pi => !pi.Name.Equals(name) para verificar se a propriedade não tem o nome indicado a ignorar. Estes validadores são aplicados às propriedades que passem no critério de seleção automático, mesmo nome e tipo compatível, se passarem em todos os testes são adicionadas à lista de propriedades válidas.

Parte 4

Como ultima fase do trabalho foi-nos pedido para permitir a adaptação de mapeamento através do método *ForMember*, que recebe o nome da propriedade destino e uma função que extrai um valor do objeto fonte. No desenvolvimento deste método gerou-se a questão de que se fosse adicionado uma restrição a um nome de uma propriedade e de seguida fosse invocada esta função sobre este nome qual é o que prevaleceria? E.g

```
Mapper<StudentForMember, PersonForMember> mapper = AutoMapper
                                        .Build<StudentForMember, PersonForMember>()
                                        .IgnoreMember("Id")
                                        .ForMember("Id", src => src.Nr.ToString())
                                        .CreateMapper();
```

Foi-nos indicado que a ultima operação era dominante, ou seja, no exemplo anterior era aplicado a função de transformação sobre Nr e o resultado seria introduzido em Id.
Para garantir este comportamento foi aplicado a seguinte logica: o *ForMember* adiciona sempre a propriedade à lista de propriedades validas. O *IgnoreMember* verifica se houve alguma propriedade adiciona pelo *ForMember* que deverá ser ignorada, se isto se verificar a propriedade é removida.

Como a obtenção do valor para afetar a propriedade destino poderia vir de dois modos (ou através da função indicada pelo ForMember ou através da invocação do método Get da propriedade fonte), e para minimizar de certo modo a utilização de reflexão, construiu-se um objeto, *PropertyAssociation*, que contém a propriedade destino e a função que extrai o valor para ser aplicado a esta propriedade.