Generic Repository Component

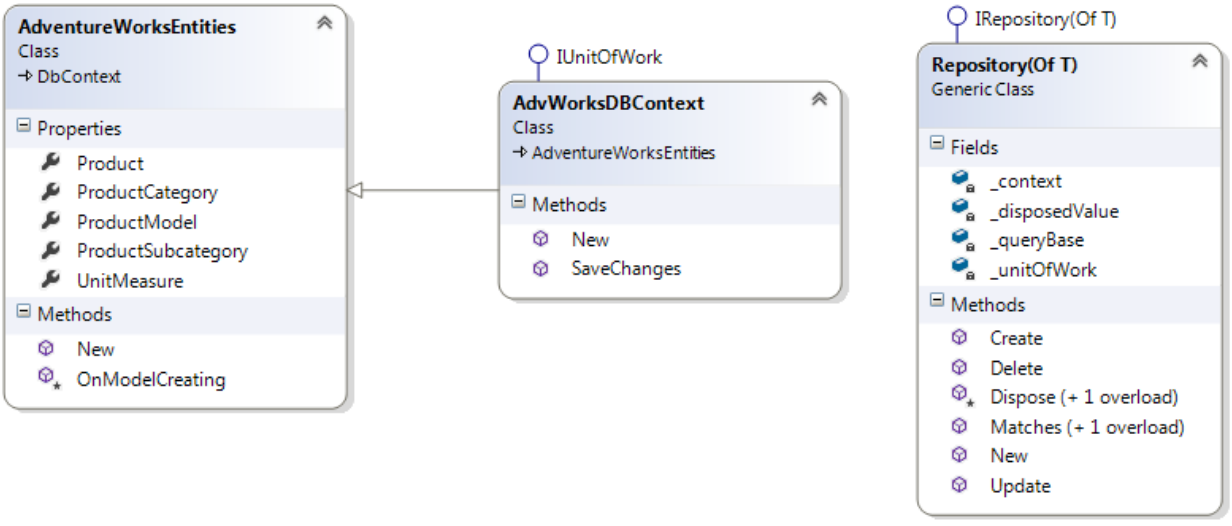
Author: Carlo Banda
Contact: carlos-alfredo.banda.montes@hp.com

Component description

This component is an implementation of the Generic Repository pattern. The main objective of this component is to facilitate the creation of a decoupled and infrastructure agnostic Data Access Layer. This component is made up by a generic class that receives an implementation of a Unit of Work that contains a data context instance and a set of classes that implements the specification and criteria patterns. Specifications and criteria are objects that represent basic expressions used to query data from the repository. They can be combined using logical operators (AND, OR) to build more complex expressions.

The Repository Pattern

The Repository class uses generics to allow reusing common functionality of a repository's implementation while the Criteria and Specifications enable specific behaviors. The Repository class receives a Unit of Work implementation in its constructor. To create the Unit of Work to be used with the Repository you will need to create a new class that inherits from a DBContext and implements the IUnitOfWork interface. For example, in the following diagram there is a DBContext class called AdventureWorksEntities which was created with the Entity Framework. To create the Unit of Work to be used with the Repository you will need to create a new class that inherits from AdventureWorksEntities and implements the IUnitOfWork interface.



The following code is an example to create a Repository instance:

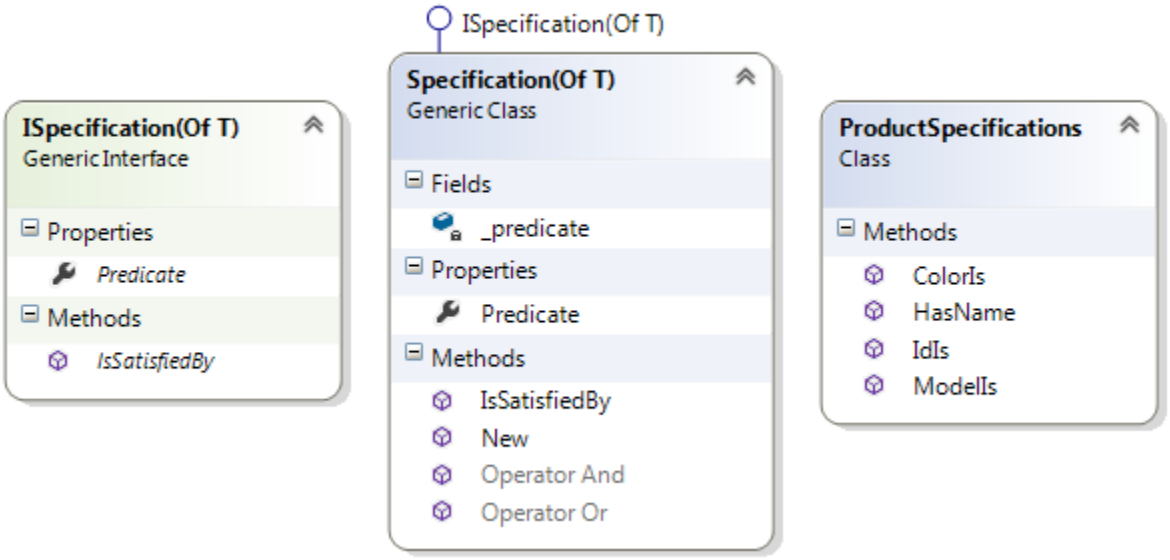
```
AdvWorksDataContext = new AdvWorksDBContext();
ProductRepository = new Repository<Product>(AdvWorksDataContext);
```

The Repository class implements an IRepository interface, in this way the repository can be decoupled from the components that consume it. The repository's consumers should enable dependency injection either by constructor or by properties. For example, consider a "ProductManager" class which uses a Repository of products, in this case the constructor of this class would look something like this:

```
public ProductManager(IRepository<Product> productsRepository){
       this.productsRepository = productsRepository;
}
```

The Repository component implements two approaches to query the entities contained in the Repository: Specifications and Criteria.

Repository Linq Specifications



Specification objects contain Lambda expressions that can be used as predicates. Specifications can be combined using logical operators (AND and OR) to create more complex expressions. There is a base Specification class which implements an ISpecification interface. The ISpecification interface has a property called "Predicate" that contains the expression used by the specification. Specification instances can be grouped in different class that will contain all the specifications for a specific entity. For example the following class groups all the specifications for the "Product" entity:

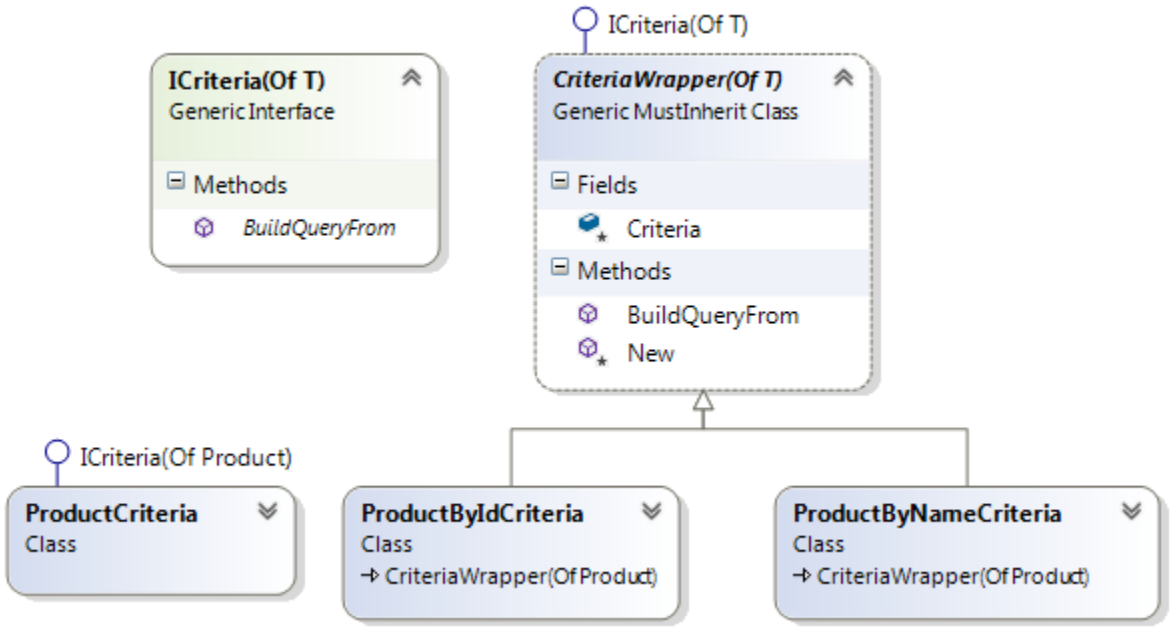
```
    public class ProductSpecifications
    {
        public static Specification<Product> NameIs(string productName) {
            return new Specification<Product>(P => P.Name.Contains(productName));
        }

        public static Specification<Product> IdIs(int id)
```

```
        {
            return new Specification<Product>(P => P.ProductID == id);
        }

        public static Specification<Product> ModelNameIs(string modelName)
        {
            return new Specification<Product>(P =>
string.Compare(P.ProductModel.Name,modelName) == 0);
        }

        public static Specification<Product> ColorIs(string color)
        {
            return new Specification<Product>(P => P.Color.Contains(color));
        }
    }
```

You can query the Repository using the specifications defined in the “ProductSpecifications” as follows:

```
var products = ProductRepository.Matches(ProductSpecifications.IdIs(316) ||
ProductSpecifications.HasName("Crankarm")).ToList();
```

Repository Criteria



Repository criteria are implementations of the Decorator design patter to enable the creation of composed criteria to query the entities contained in the repository. It is just an alternative to the Specifications patter described above.

The ICriteria interface has a definition for a method called “BuildQueryFrom” that receives a Set of Entities which will be the base of the filter. The “BuildQueryFrom” method will be called recursively using the decorator patter to narrow down the filter.

The CriteriaWrapper class is the base class to build the specific criteria. All the specific criteria classes will inherits from the CriteriaWarpper class. For example, if you want to create a criteria class to filter Products by ID, you would need to create a class called “ProductById” that inherits from the CriteriaWrapper class as follows:

```
public class ProductById : CriteriaWrapper<Product>{
    private int id;

    public ProductById (ICriteria<Of Product> criteria, int id) : base(criteria){
        this.id = id
    }
    public override IQueryable<Product> BuildQueryFrom(System.Data.Entity.DbSet<Of
Product> entitySet){
        return base.BuildQueryFrom(entitySet).where(Product p => p.ProductID == id)
    }

}
```

The following code demonstrates how to use the Criteria pattern to query a Repository of Product:

```
var products = ProductRepository.Matches(new ProductById(new ProductCriteria(), 316), _
                                                    new ProductByIdCriteria(new
ProductCriteria(), 328)).ToList();
```

Software Requirements

- .NET Framework 4.0 or higher
- Microsoft Entity Framework 5.0 or higher
- LinqKit

Implementation steps

To implement the Generic Repository component in a specific application you need to follow these steps:

1. Create DBContext class
2. Implement IUnitOfWork interface
3. Create Specifications
4. Create Criteria