**Repository Pattern Starting Point**

**Introduction**

The Repository pattern is frequently employed in layered architectures where the domains in the Domain layer expose data access related operations through abstract interfaces. The actual repository layer, e.g. EntityFramework or MongoDb then implement those interfaces.

The primary goal of the Repository pattern is to decouple the domain objects from the physical repository layer. The domain objects and the domain layer are not supposed to know about the actual data access operations such as opening a database connection and querying a database. This is a common approach in [Domain Driven Design](https://dotnetcodr.com/2013/09/12/a-model-net-web-service-based-on-domain-driven-design-part-1-introduction/) (DDD).

We’ll go through the basic concepts of the pattern in this post. If you’d like to see a more complex design then you can go through the series on DDD. Also, there are multiple ways to implement the pattern but the main objective is to keep the technology-specific data access operations out of the domain layer.

**A simple domain**

Let’s start with a simple domain object which resides in the layer called Domains. The layer is a normal C# class library:

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | public class Customer : IDomain  {      public int Id { get; set; }      public string Name { get; set; }      public Address Address { get; set; }  } |

…where Address looks as follows:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | public class Address  {      public string AddressLine1 { get; set; }      public string AddressLine2 { get; set; }      public string City { get; set; }      public string PostalCode { get; set; }  } |

…and IDomain is just an empty interface to mark domain objects:

|  |  |
| --- | --- |
| 1  2  3 | public interface IDomain  {  } |

We’ll come back to this empty interface in part 5 of this series where we discuss the notion of aggregate root.

**Persistence ignorance (PI)**

As we said in the introduction we want to keep the domain layer independent of the actual repository layer, i.e. the one which is responsible for the data access related operations. Our goal is to make the concrete repository layer to depend on the domain layer and not vice versa which is often the case in real-life projects. This repository-independence is central in Domain Driven Design and has a special term for it: persistence ignorance.

The domain objects are kept clean by not polluting them with technology-specific data access operations. This implies that we shouldn’t have e.g. EntityFramework, NHibernate, MongoDb etc. specific code in the domain layer. As soon as we do that we’re implicitly creating a strong coupling between our domain objects and the concrete repository technology.

The result will be an independent domain layer that you can pass around without any tightly coupled dependencies. Also, you’ll be able to switch easily between concrete repositories, e.g. if you want to test MS SQL versus MySQL in the design phase or if your company wants to completely move the data layer to another technology, e.g. MongoDb.