**The Repository Pattern**

**Context**

In many applications, the business logic accesses data from data stores such as databases, SharePoint lists, or Web services. Directly accessing the data can result in the following:

* Duplicated code
* A higher potential for programming errors
* Weak typing of the business data
* Difficulty in centralizing data-related policies such as caching
* An inability to easily test the business logic in isolation from external dependencies

**Objectives**

Use the Repository pattern to achieve one or more of the following objectives:

* You want to maximize the amount of code that can be tested with automation and to isolate the data layer to support unit testing.
* You access the data source from many locations and want to apply centrally managed, consistent access rules and logic.
* You want to implement and centralize a caching strategy for the data source.
* You want to improve the code's maintainability and readability by separating business logic from data or service access logic.
* You want to use business entities that are strongly typed so that you can identify problems at compile time instead of at run time.
* You want to associate a behavior with the related data. For example, you want to calculate fields or enforce complex relationships or business rules between the data elements within an entity.
* You want to apply a domain model to simplify complex business logic.

**Solution**

Use a repository to separate the logic that retrieves the data and maps it to the entity model from the business logic that acts on the model. The business logic should be agnostic to the type of data that comprises the data source layer. For example, the data source layer can be a database, a SharePoint list, or a Web service.

The repository mediates between the data source layer and the business layers of the application. It queries the data source for the data, maps the data from the data source to a business entity, and persists changes in the business entity to the data source. A repository separates the business logic from the interactions with the underlying data source or Web service. The separation between the data and business tiers has three benefits:

* It centralizes the data logic or Web service access logic.
* It provides a substitution point for the unit tests.
* It provides a flexible architecture that can be adapted as the overall design of the application evolves.

There are two ways that the repository can query business entities. It can submit a query object to the client's business logic or it can use methods that specify the business criteria. In the latter case, the repository forms the query on the client's behalf. The repository returns a matching set of entities that satisfy the query. The following diagram shows the interactions of the repository with the client and the data source.

Interactions of the repository



The client submits new or changed entities to the repository for persistence. In more complex situations, the client business logic can use the Unit of Work pattern. This pattern demonstrates how to encapsulate several related operations that should be consistent with each other or that have related dependencies. The encapsulated items are sent to the repository for update or delete actions. This guidance does not include an example of the Unit of Work pattern. For more information, see [Unit of Work](http://martinfowler.com/eaaCatalog/unitOfWork.html) on [Martin Fowler](http://www.martinfowler.com/)'s Web site.

Repositories are bridges between data and operations that are in different domains. A common case is mapping from a domain where data is weakly typed, such as a database or SharePoint list, into a domain where objects are strongly typed, such as a domain entity model. One example is a database that uses **IDbCommand** objects to execute queries and returns **IDataReader** objects. Another example is SharePoint, which uses **SPQuery** objects to return **SPListItem** collections. A repository issues the appropriate queries to the data source, and then it maps the result sets to the externally exposed business entities. Repositories often use the [Data Mapper](http://martinfowler.com/eaaCatalog/dataMapper.html) pattern to translate between representations. Repositories remove dependencies that the calling clients have on specific technologies. For example, if a client calls a catalog repository to retrieve some product data, it only needs to use the catalog repository interface. For example, the client does not need to know if the product information is retrieved with SQL queries to a database or Collaborative Application Markup Language (CAML) queries to a SharePoint list. Isolating these types of dependences provides flexibility to evolve implementations.