-

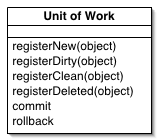
-

-

**MARTİN FOWLER:**

*Maintains a list of objects affected by a business transaction and coordinates the writing out of changes and the resolution of concurrency problems.*

For a full description see [P of EAA](https://martinfowler.com/books/eaa.html) page **184**



When you're pulling data in and out of a database, it's important to keep track of what you've changed; otherwise, that data won't be written back into the database. Similarly you have to insert new objects you create and remove any objects you delete.

**You can change the database with each change to your object model, but this can lead to lots of very small database calls, which ends up being very slow.**

**Furthermore it requires you to have a transaction open for the whole interaction, which is impractical if you have a business transaction that spans multiple requests**. The **situation is even worse if you need to keep track of the objects you've read so you can avoid inconsistent reads.**

**A Unit of Work keeps track of everything you do during a business transaction that can affect the database. When you're done, it figures out everything that needs to be done to alter the database as a result of your work.**

Unit of Work is the **concept** related to the effective implementation of the Repository Pattern. To understand this concept in better it is important to understand the concept of the Repository Pattern. We will not get into the details of the Repository Pattern in this discussion. But a small idea of this concept is necessary to proceed further.

## **The Repository Pattern**

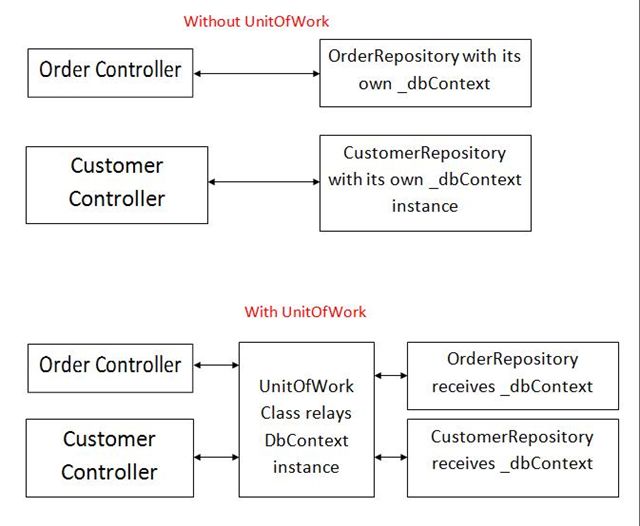
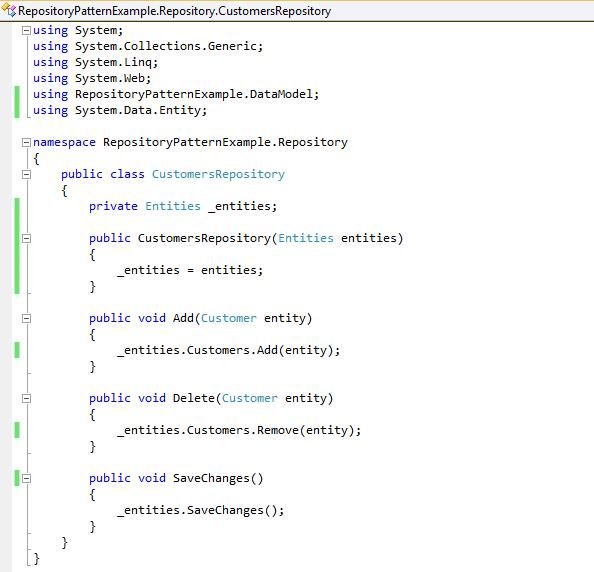
A repository is nothing but a class defined for an entity, with all the operations possible on that specific entity. For example, a repository for an entity Customer, will have basic CRUD operations and any other possible operations related to it. A Repository Pattern can be implemented in Following ways:

* **One repository per entity (non-generic) :**This type of implementation involves the use of **one repository class for each entity**. For example, if you have two entities Order and Customer, each entity will have its own repository.
* **Generic repository:** A generic repository is the one that can be used for all the entities, in other words it **can be either used for Order or Customer or any other entity.**

## **Unit of Work in the Repository Pattern**

Unit of Work is referred to as a **single transaction** that involves multiple operations of insert/update/delete and so on kinds. To say it in simple words, it means that for a specific user action (say registration on a website), **all the transactions like insert/update/delete and so on are done in one single transaction, rather then doing multiple database transactions**. This means, one unit of work here involves insert/update/delete operations, **all in one single transaction**.

To understand this concept, consider the following implementation of the Repository Pattern using a **non-generic** repository, for a Customer entity.  
  
  
  
The code above seems to be fine. The issue arises when we add a repository for another entity, say Order. In that case, both repositories will **generate and maintain their own instance of the DbContext**. This may lead to issues in the future, since each DbContext will have its own in-memory list of changes of the records, of the entities, that are being added/updated/modified, in a single transaction/operation. **In such a case, if the SaveChanges of one of the repository fails and other one succeeds, it will result in database in-consistency.** This is where the concept of UnitOfWork is relevant.  
  
To avoid this, we will add another layer or intermediate **between** the controller and the Customer repository. This layer will act as a centralized store for all the repositories to receive the instance of the DbContext. This will ensure that, for a **unit of transaction**, that spans across multiple repositories, **should either complete for all entities or should fail entirely, as all of them will share the same instance of the DbContext**. In our above example, while adding data for the Order and Customer entities, in a single transaction, both will use the same DbContext instance. This situation, without and with Unit of work, can be represented as in the following:

  
  
In the above representation, during a single operation, that involves Customer and Order entities, **both of them use the same DbContext instance**. This will ensure that even if one of them breaks, the other one is also not saved, **thus maintaining the database consistency**. So when SaveChanges is executed, it will be done for both of the repositories.  
  
Let us implement this concept in our example. **We add a new class called UnitOfWork and this class will receive the instance of the DbContext.** The **same class will further generate the required repository instances, in other words repository instances for Order and Customer and pass the same DbContext to both the repositories**. So our UnitOfWork will be like the following:  
  
  
  
And, our Customer Repository will be changed, to receive the instance of DbContext from the unit of work class. See the code below:  
  
  
  
Similarly, we can have the code for the Order repository. Finally, our controller code will be like the following :  
  
  
  
Here, both the Order and Customer repository **use the same instance of DbContext** and **we are executing the save changes using the instance unit of work class.** So the changes of a **single transaction** are **either done for both or none.** Run the code and see the results.  
  
So this was about the concept of unit of work in the Repository Pattern. Any suggestions are welcome.