TALLINN UNIVERSITY OF TECHNOLOGY
School of Information Technologies

Piret Gorban 185967IADB

BOOK SWAP

Building Distributed Systems

Project Scope

Supervisor: Andres Käver

Tallinn 2020

Author’s declaration of originality

I hereby certify that I am the sole author of this thesis. All the used materials, references to the literature and the work of others have been referred to. This thesis has not been presented for examination anywhere else.

Author: Piret Gorban

02.03.2020

Table of contents

1 Introduction 5
2 Diagrams... 6
3 Analysis and testing of soft delete and update in SQL.. 7
4 Analysis of Repository Pattern and Data Access Object.. 13
5 Summary... 15
References 16

List of figures

Figure 1. Entity relationship diagram 6

Figure 2. One to many (optional) relationship 7

Figure 3. One to one (optional) relationship.. 7

Figure 4. Tables “Method of delivery” and “Location” before update 9

Figure 5. Tables “Method of delivery” and “Location” after soft update 9

Figure 6. SQL code example of soft updating method of delivery name 10

Figure 7. SQL code example of soft deleting children when parent gets soft deleted ... 10

Figure 8. Code example of soft updating child in one to optional one relationship....... 11

Figure 9. Tables “Book condition” and “Comment” after soft updating one comment .12

Figure 10. Interactions of the repository 13

Figure 11. Relationships for the DAO pattern.. 14

1 Introduction

The goal of this project is to create an application where users can swap books. As a frequent reader author has great sympathy for applications related with books. There is a web page to swap books in Estonia but there is no harm doing it again.

The idea of book swap is that people can exchange books they don´t need any more and collect new books without paying for them. Registered users can see books that are available for offering at the moment and can add their own books. Condition of the book has to be marked so that other users would know what they ordered. Users can choose if their books can be seen by other users or not.

If user can’t find the book they have from existing list they can add new book that will be seen by all users. They can also add new authors, genres and languages.

Books will be exchanged using parcel machines. System supports adding many types of parcel machines (Omniva, SmartPost etc). Payment methods are not included in current application because necessary parcel size is chosen by sender and receiver pays for the package in parcel machine when receiving the order.

Users can follow orders status – receiver can see if sender has posted the package and sender can view if receiver has received the package. Order can be cancelled by the sender.

Users can rate other users who they have exchanged books with. Average rating of all individual ratings is displayed for every user so that people wound know which users to trust to deliver the transactions.

Users who have admin role can see User Manager where they can create and delete users and roles and add or remove roles from users. In ASP .NET application administrators also have full CRUD for every object and separate view for language strings.

Application has support for different languages (currently English and Estonian).

2 Diagrams

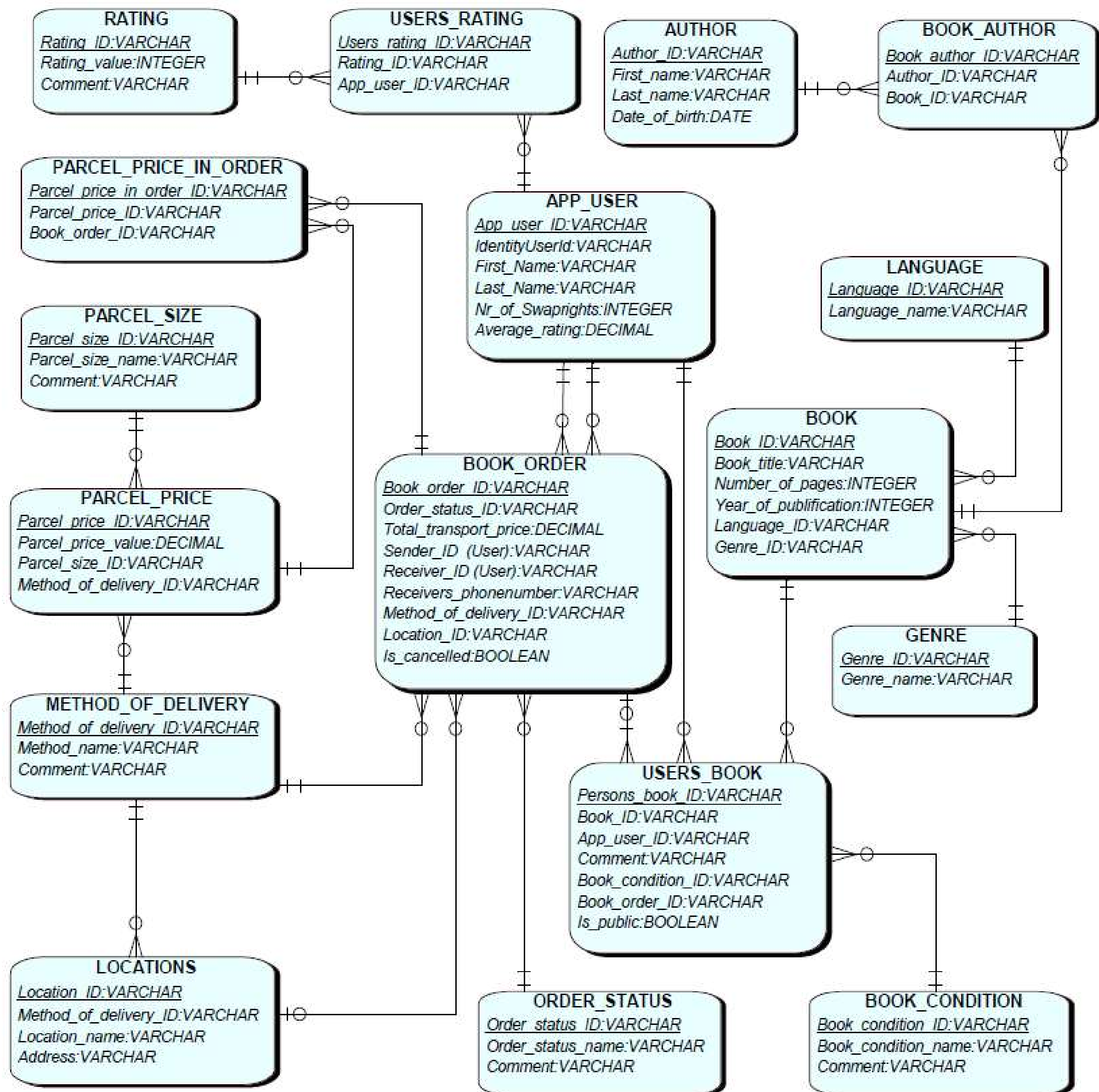

Figure 1. Entity relationship diagram.

3 Analysis and testing of soft delete and update in SQL

The purpose of this chapter is to analyse which methods are usable to implement soft delete and update in SQL by using MS SQL Server. Sample tables to test possibilities are chosen from the project. One to many relationship is presented by "Method of delivery" and "Locations". One method of delivery can have many locations. For example, Itella Smartpost can have parcel machines at Mustamäe Keskus, at Mustika Keskus etc. Described relationship is shown in figure 2.

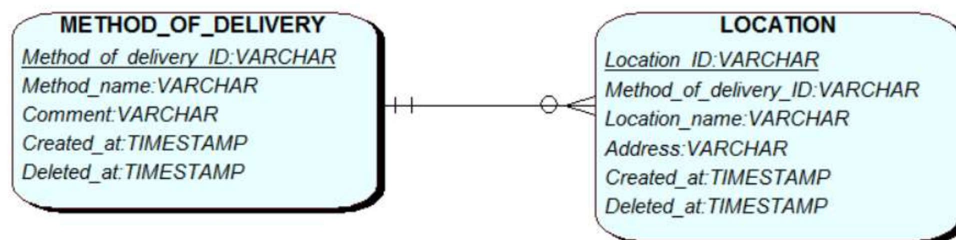

Figure 2. One to many (optional) relationship.

One to one relationship (with one end optional) is not represented in this project so the author will use a fictional table to test that type of relationship. Let's say that book condition can have but must not have one comment that is an entity of comments table. Described relationship is shown in figure 3.

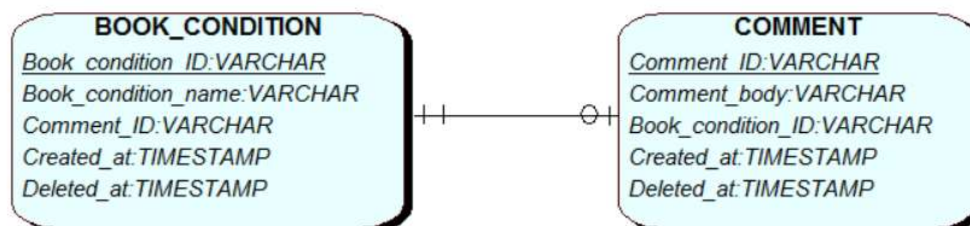

Figure 3. One to one (optional) relationship.

The main idea of soft delete and soft update is to prevent data loss. All changes of all objects can be traced and nothing gets actually deleted from the database. Created at and deleted at columns will provide the information when the record was created and when "deleted". There lays the question if created at should stay the same for all of the versions

of one record or should it represent the moment record was changed. Author of this work tends to support the first option. The time of the changes made with the record can be seen based on the Deleted at field of previous version. Created at field of every version represents the initial creation time of the first version.

There are many possibilities to implement soft delete and update. With single table it's not that complicated. One way is to add an additional field to point to previous version of the record (with null value for first record). When record gets updated or deleted then actually new record with new Id is generated and the old one gets "deleted" – Deleted at field will be filled with current timestamp. Connection with the previous version will be made via additional field. Current version can be distinguished by a null value in deleted at field. To make navigation easier field pointing to next version can also be added. But this solution requires adding additional fields with no other purpose than navigation between different versions.

The other problem with this solution becomes obvious when foreign key – primary key relationships are created between tables. Creating new records with new Id-s taints relationships if not updated. And updating every relationship while making changes or deleting records is a massive amount of work for objects with one to many relationships and big datasets. So that solution is probably not the best for soft delete and update.

Better solution seems to be to use composite primary and foreign keys. That would allow duplicate Id-s for many versions of one record (what is not possible for Id being the sole primary key). Authors first idea was to make a composite key from Id and Deleted at field but that was not an option with Deleted at being null for current record because primary key doesn't allow null values. So the other option was to use Id and Created at value. But that was also problematic. If Created at value is changed when record is updated the relationships stay connected with previous version and we are back in situation described in previous section. If the Created at value is not changed then we don't have unique primary key any more – all the composite primary keys are the same for every version of the record and that goes in conflict with the definition of primary key.

So the solution for composite primary key seems to be to still use the combination of Id and Deleted at field. But instead of leaving Deleted at field null, filling it with some future value (like maximum datetime value in MS SQL – `'12/31/9999 23:59:59.9999'`

[1]). When record is created the Deleted at field is filled with some agreed future value. When record is “deleted”, that future value is replaced with current timestamp. That would break parent – child relationship but that gets fixed by using cascade update (SQL command “ON UPDATE CASCADE” [2]) by changing childs foreign key when parents primary key changes. Example on tables “Method of delivery” and “Location” with filled Deleted at value is shown in figure 4. Tables contain only current records, no updates or “deletes” have been made yet.

Id	Method_name	Comment	Created_at	Deleted_at
1	Itella Smartpost	NULL	2020-03-07 16:02:17.1066667	9999-12-31 23:59:59.9970000
2	Omniva	NULL	2020-03-07 16:02:17.1066667	9999-12-31 23:59:59.9970000

Id	Location_name	Address	Created_at	Deleted_at	Method_name
1	Mustamäe Keskus	A. H. Tammsaare tee, Tallinn…	2020-03-07 16:02:17.1066667	9999-12-31 23:59:59.9970000	Itella Smartpost
2	Mustika Prisma	Karjavälja, Tallinn, Estonia	2020-03-07 16:02:17.1066667	9999-12-31 23:59:59.9970000	Itella Smartpost
3	Tallinna Vilde tee Maxima XX	Vilde tee 77, Tallinn, Eston…	2020-03-07 16:02:17.1066667	9999-12-31 23:59:59.9970000	Itella Smartpost
4	Tallinna Sõpruse Rimi pakiau…	Sõpruse puiestee 174, Tallin…	2020-03-07 16:02:17.1066667	9999-12-31 23:59:59.9970000	Omniva
5	Tallinna Akadeemia Konsumi p…	Akadeemia tee 35, Tallinn, E…	2020-03-07 16:02:17.1066667	9999-12-31 23:59:59.9970000	Omniva
6	Tallinna Sütiste Maxima X pa…	Juhan Sütiste tee 28, Tallin…	2020-03-07 16:02:17.1066667	9999-12-31 23:59:59.9970000	Omniva

Figure 4. Tables “Method of delivery” and “Location” before update.

For updating the record new copy of old record is made, necessary fields get new values and Deleted at field gets agreed future value. Old version of the record gets current timestamp to Deleted at field. Id-s for old and new versions stay identical. In that case the relationships created before using a composite foreign key stay attached to current version. Example of output after updating method of delivery name from “Itella Smartpost” to “Smartpost” is shown in figure 5. Records in Locations table have connections to method of delivery that isn’t deleted. SQL code example how this result was achieved is shown in figure 6.

Id	Method_name	Comment	Created_at	Deleted_at
1	Itella Smartpost	NULL	2020-03-07 16:02:17.1066667	9999-12-31 23:59:59.9970000
2	Omniva	NULL	2020-03-07 16:02:17.1066667	9999-12-31 23:59:59.9970000

Id	Location_name	Address	Created_at	Deleted_at	Method_name
1	Mustamäe Keskus	A. H. Tammsaare tee, Tallinn…	2020-03-07 16:02:17.1066667	9999-12-31 23:59:59.9970000	Itella Smartpost
2	Mustika Prisma	Karjavälja, Tallinn, Estonia	2020-03-07 16:02:17.1066667	9999-12-31 23:59:59.9970000	Itella Smartpost
3	Tallinna Vilde tee Maxima XX	Vilde tee 77, Tallinn, Eston…	2020-03-07 16:02:17.1066667	9999-12-31 23:59:59.9970000	Itella Smartpost
4	Tallinna Sõpruse Rimi pakiau…	Sõpruse puiestee 174, Tallin…	2020-03-07 16:02:17.1066667	9999-12-31 23:59:59.9970000	Omniva
5	Tallinna Akadeemia Konsumi p…	Akadeemia tee 35, Tallinn, E…	2020-03-07 16:02:17.1066667	9999-12-31 23:59:59.9970000	Omniva
6	Tallinna Sütiste Maxima X pa…	Juhan Sütiste tee 28, Tallin…	2020-03-07 16:02:17.1066667	9999-12-31 23:59:59.9970000	Omniva

Figure 5. Tables “Method of delivery” and “Location” after soft update.

```sql
DECLARE @Time_future DATETIME2
SELECT @Time_future = '9999-12-31 23:59:59.997'

INSERT INTO Method_of_delivery (Id, Method_name, Created_at, Deleted_at)
VALUES (1, 'Itella SmartPost', (SELECT Created_at FROM Method_of_delivery WHERE Id = 1), GETDATE())

UPDATE Method_of_delivery SET Method_name = 'SmartPost' WHERE Id = 1 AND Deleted_at = @Time_future

SELECT * FROM Method_of_delivery

SELECT Location.Id, Location.Location_name,Location.Address, Location.Created_at, Location.Deleted_at,
Method_of_Delivery.Method_name
FROM Location
LEFT JOIN Method_of_delivery ON Location.Method_of_delivery_Id = Method_of_delivery.Id
AND Location.Method_of_delivery_Deleted_at = Method_of_delivery.Deleted_at
```

Figure 6. SQL code example of soft updating method of delivery name.

Question is also if child records should get “deleted” if parent gets “deleted”. That can also depend on business logic but in most cases they should. That can go really deep and complicated with many intersecting relationships and should be thought through for every relationship before implementing it. SQL code example of soft deleting children when parent gets soft deleted is show in figure 7.

```sql
DECLARE @Update_Id INT
SELECT @Update_Id = (SELECT Id FROM Method_of_delivery WHERE Method_name like 'Omniva')

DECLARE @Time2 DATETIME2
SELECT @Time2 = GETDATE()

UPDATE Method_of_delivery SET Deleted_at =  @Time2 WHERE Id = @Update_Id
UPDATE Location SET Deleted_at =  @Time2
WHERE Method_of_delivery_Id = @Update_Id AND Method_of_delivery_Deleted_at =  @Time2
```

Figure 7. SQL code example of soft deleting children when parent gets soft deleted.

With this approach children records (on one to many relationships many side and one to one relationships optional side) will have connection to current records and lose connection with “deleted” records. That could be enough for some cases but might be problem in other cases. There is no easy way to get the current state of the database at the certain moment of time. Queries and joins must be made separately to get the valid data at that moment.

One option to relieve that problem is to create duplicate records in database that stay connected to versions that get “deleted” while updating data. That means that every time parent record gets updated so do the children. And its children etc. Updating child doesn’t have that problem because parent can have connections to both current and historical records. “Deleted” children can be distinguished by Deleted at value. That will create a lots of new records in database and many changes of children are caused only by parent

record changes. But the state of database can be distinguished with ease for every moment of time by filtering Created at and Deleted at fields.

One to one relationship acts as a special case of one to many relationship. Difference is that parent can have only one child. That could be implemented on database level by using unique composite foreign key of parents Id and parents Deleted at field (SQL command `CONSTRAINT UNIQUE` [3]). If parent record gets “deleted” so should the child. If child gets “deleted” there is two options. One is to leave “deleted” child connected to parent if no new child needs to be added. But that only postpones the problem that occurs when new child needs to be added.

The other possibility is to create “deleted” version of parent that will stay connected to “deleted” child and current parent will be disconnected from “deleted” child. That will prevent problems when new child needs to be added. Updating is the same for parent and child. New versions of both – parent and child is needed to keep the relationships. Figure 8 shows SQL code example for soft updating child in one to one (optional) relationship. First copy of book condition is created which gets soft deleted. Then new version of comment is created and old version gets soft deleted. New version of comment stays connected to current book condition, soft deleted versions get connected to each other. Result of both tables is shown in figure 9.

```sql
DECLARE @Time4 DATETIME2
SELECT @Time4 = '2011-03-03 23:59:59.997'

DECLARE @Updated_condition_Id INT
SELECT @Updated_condition_Id = (SELECT Book_condition_Id FROM Comment WHERE Id = 2)

SET IDENTITY_INSERT Book_condition ON

INSERT INTO Book_condition(Id, Book_condition_name, Created_at, Deleted_at)
VALUES (@Updated_condition_Id,
    (SELECT Book_condition_name FROM Book_condition WHERE Id = @Updated_condition_Id),
    (SELECT Created_at FROM Book_condition WHERE Id = @Updated_condition_Id),
    @Time4)

SET IDENTITY_INSERT Book_condition OFF
SET IDENTITY_INSERT Comment ON

INSERT INTO Comment (Id, Comment_body, Created_at, Deleted_at, Book_condition_Id, Book_condition_Deleted_at)
VALUES (2, 'Some pages missing', (SELECT Created_at FROM Comment WHERE Id = 2), @Time4, @Updated_condition_Id, @Time4)

UPDATE Comment SET Comment_body = 'Few pages missing or damaged' WHERE Id = 2 AND Deleted_at = @Time_future
```

Figure 8. Code example of soft updating child in one to optional one relationship.

Id	Book_condition_name	Created_at	Deleted_at
1	New	2001-01-01 00:00:00.0000000	9999-12-31 23:59:59.9970000
2	Used but like new	2001-01-01 00:00:00.0000000	9999-12-31 23:59:59.9970000
3	Worn out	2001-01-01 00:00:00.0000000	2008-12-31 23:59:59.9970000
4	Shabby	2001-01-01 00:00:00.0000000	2011-03-03 23:59:59.9970000
4	Shabby	2001-01-01 00:00:00.0000000	9999-12-31 23:59:59.9970000

Id	Comment_body	Created_at	Deleted_at	Book_condition_name	Deleted_at
1	All pages still attached	2001-01-01 00:00:00.0000000	2008-12-31 23:59:59.9970000	Worn out	2008-12-31 23:59:59.9970000
2	Some pages missing	2001-01-01 00:00:00.0000000	2011-03-03 23:59:59.9970000	Shabby	2011-03-03 23:59:59.9970000
2	Few pages missing or damaged	2001-01-01 00:00:00.0000000	9999-12-31 23:59:59.9970000	Shabby	9999-12-31 23:59:59.9970000

Figure 9. Tables “Book condition” and “Comment” after soft updating one comment.

When trying to get the current state of tables records at certain moment of time we hit next problem. Since the created at time shows the creation time of the first version and is the same for all records we can’t filter records by creation time. If we want records from comments table that were active in the beginning of 2010 and we filter comments by condition `Comment.Deleted_at > '2010-01-01 00:00:00.000'` we get two results – one that was active at that moment and the other that is active now but was actually created later.

One way to solve that problem is that Created at field should always change when new version is created. Then records can be filtered with conditions `Comment.Created_at <= '2010-01-01 00:00:00.000' AND Comment.Deleted_at > '2010-01-01 00:00:00.000'`. But that creates another problem – initial creation time can’t be seen from current record if there are older versions. But it’s possible to query creation time from first version and display that on newer versions.

The other possibility is to add another field to records – Modified at. Then Created at will show initial creation time of first version and Modified at will get the value of the moment the record was actually created. Then every record will have straight access to all values necessary and active records can be queried with conditions `Comment.Modified_at <= '2010-01-01 00:00:00.000' AND Comment.Deleted_at > '2010-01-01 00:00:00.000'`. Which solution is better probably depends of the project.

In conclusion it can be said that soft delete and soft update are complicated subjects that have no single correct answer. All possibilities must be thought through according to business logic of every project to find the best solution.

4 Analysis of Repository Pattern and Data Access Object

By definition Repository Pattern mediates between the domain and data mapping layers using a collection-like interface for accessing domain objects [4]. It allows to separate business logic from database operations like saving and retrieving data and remove duplicate code for same actions of different objects. Controllers should not deal with database operations directly but to delegate them to repositories. That ensures that business logic is designed only based on real needs not considering the actions needed to be done to handle database operations. Client side doesn't have to know how to access data and can use provided methods from the interfaces. The Repository Pattern also allows to test applications more easily with unit tests by allowing to use test data from some other source than applications real database. The interactions of the repository are shown in figure 10. Business logic layer communicates with repository via business entities and repository communicates with data source. Data source can be switched by modifying only the repository and no changes are necessary in business logic.

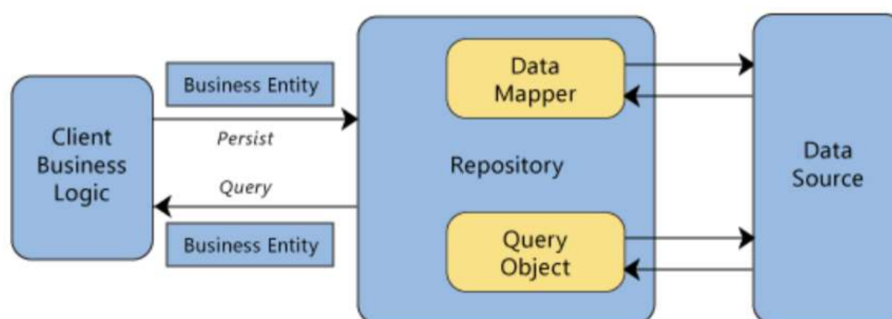

Figure 10. Interactions of the repository. [5]

The idea of the Data Access Object (DAO) pattern is the same as the Repository Pattern. Relationships for the DAO pattern are shown in figure 11. DAO is an intermediate object between business logic and data source. Transfer object represents the business entity of repository model. Database actions are implemented in DAO without business object knowing the specifics how it is done.

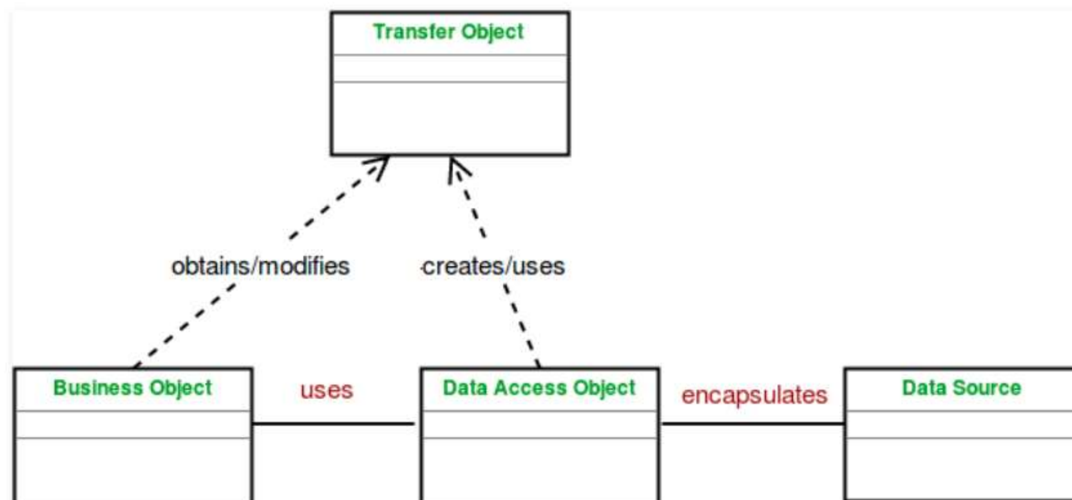

Figure 11. Relationships for the DAO pattern. [6]

But the difference between repository and DAO patterns is that DAO is closer to database and handles more data specific operations. Repositories should contain methods like find and add an entity or get collection of entities. For more specific actions like update entity repositories usually use separate objects like unit of work (UOW). UOW coordinates the work of multiple repositories by creating a single database context class shared by all of them [7]. DAO contains the logic of all methods (add, find, update etc.) in itself. Repositories can be the layer on top of the DAO but not the other way around.

In current BookSwap project the repository pattern has been implemented. For removing duplicate code there is two layers of repository interfaces – base repository for all common methods and specific interfaces to add domain object specific methods. Specific interfaces inherit from base repository and actual repositories implement those interfaces. In addition to repository pattern UOW is implemented. From database operations UOW is currently only handling saving data (changes) to database. Add, find, remove and update operations are the responsibility of repositories which somewhat goes in controversy with division of responsibilities described above. That should be taken into consideration designing the further changes in project structure.

5 Summary

Three levels on mappings to map objects from one Data Transfer Object (DTO) to another were used in this project. Domain objects were mapped to Data Access Layer (DAL) DTOs in repositories. DAL DTOs were mapped to Business Logic Layer (BLL) DTOs in services. ASP .NET controllers use BLL DTOs in View Models. REST controllers use Public Api DTOs, mapping to those happens in REST controllers. AutoMapper is used to map flat objects (for example in add and update). More complex object mappings with many layers of nested objects use custom mappers for every level.

Base projects not specific to current application were pushed to NuGet Gallery as packages and can also be used in further projects. Swagger was used to describe API structure.

References

[1] Microsoft SQL Docs. datetime (Transact-SQL). https://docs.microsoft.com/en-us/sql/t-sql/data-types/datetime-transact-sql?view=sql-server-ver15 (7.03.2020)

[2] SQL Server Tutorial. SQL Server FOREIGN KEY. https://www.sqlservertutorial.net/ sql-server-basics/sql-server-foreign-key/ (7.03.2020)

[3] W3Schools tutorials site. SQL UNIQUE Constraint. https://www.w3schools.com/ sql/sql_unique.asp (7.03.2020)

[4] E. Hieatt, R. Mee, Repository. https://martinfowler.com/eaaCatalog/repository.html (23.03.2020)

[5] The Repository Pattern. Microsoft Docs. https://docs.microsoft.com/en-us/previous-versions/msp-n-p/ff649690(v=pandp.10) (23.03.2020)

[6] Core J2EE Patterns - Data Access Object. Oracle Technology Network. https://www.oracle.com/technetwork/java/dataaccessobject-138824.html (23.03.2020)

[7] Implementing the Repository and Unit of Work Patterns. Microsoft Docs. https://docs.microsoft.com/en-us/aspnet/mvc/overview/older-versions/getting-started-with-ef-5-using-mvc-4/implementing-the-repository-and-unit-of-work-patterns-in-an-asp-net-mvc-application (23.03.2020)