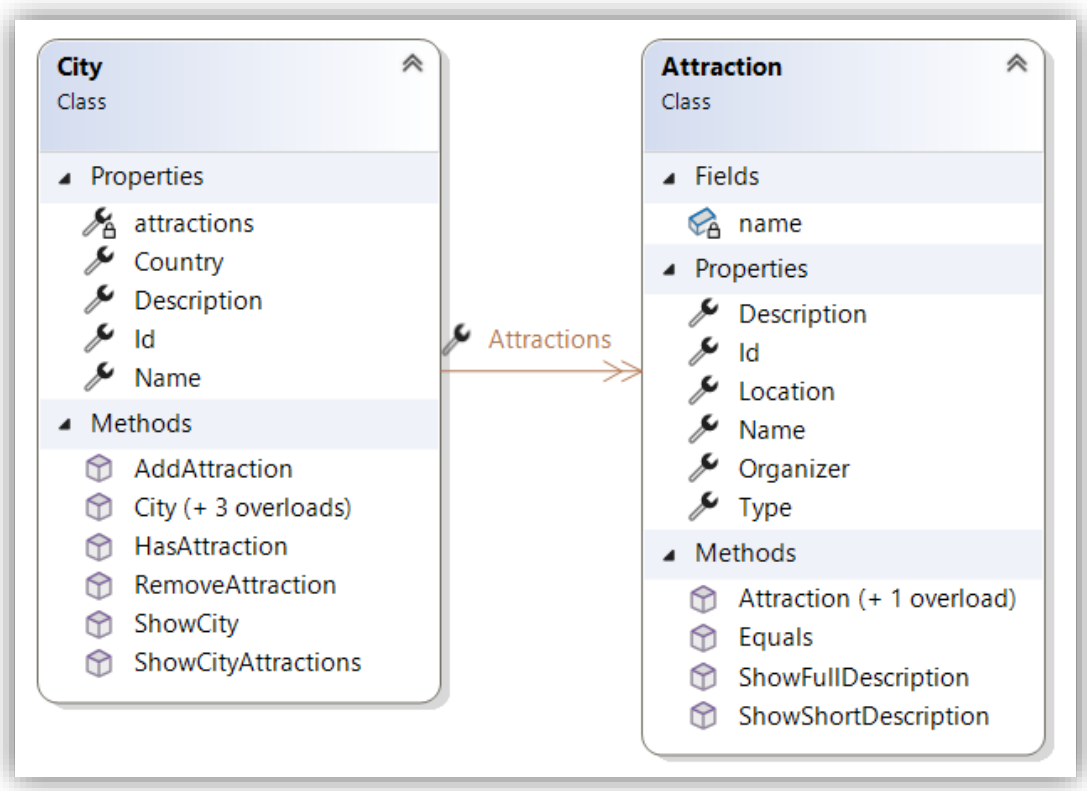
3-Lagen architectuur

We bespreken hier de implementatie van de 3-lagen architectuur aan de hand van een eenvoudig voorbeeld.

Domein model

Een stad (City) heeft een id, die door de databank wordt aangemaakt, een naam (verplicht), een beschrijving en het land waarin de stad is gelegen. Er zijn ook attracties (Attraction) en die hebben eveneens een id die door de databank wordt aangemaakt, een naam (verplicht), een beschrijving, een type, een locatie (verplicht) en een organisator (verplicht). Een attractie moet steeds tot een stad behoren. Om te vermijden dat er twee dezelfde attracties worden toegevoegd kijken we naar de naam en organisator.



```
public class City
{
    public City(string name, string description, string country)...
    public City(string name, string description, string country, List<Attraction> attractions)...
    public City(int id, string name, string description, string country)...
    public City(int id, string name, string description, string country, List<Attraction> attractions)...

    public int? Id { get; set; }
    private string name = "";
    public string Name...
    public string Description { get; set; }
    public string Country { get; set; }
    private List<Attraction> attractions { get; set; } = new();
    public IReadOnlyList<Attraction> Attractions => attractions;
    public bool HasAttraction(Attraction attraction)...
    public void AddAttraction(Attraction attraction)...
    public void RemoveAttraction(Attraction attraction)...
    public string ShowCity()...
    public string ShowCityAttractions()...
}
```

Om de naam van de stad verplicht te maken voorzien we een controle in de setter van de Name property.

```
public string Name
{
    get { return name; }
    set { if (string.IsNullOrWhiteSpace(value)) throw new CityException("name invalid"); name = value; }
}
```

Om te vermijden dat attracties meerdere keren worden toegevoegd voeren we eerst een controle uit. Ook bij het verwijderen doen we een controle of de attractie wel degelijk tot de stad behoort.

```
public bool HasAttraction(Attraction attraction)
{
    return attractions.Contains(attraction);
}
public void AddAttraction(Attraction attraction)
{
    if (!HasAttraction(attraction)) attractions.Add(attraction);
    else throw new CityException("AddAttraction");
}
public void RemoveAttraction(Attraction attraction)
{
    if (!HasAttraction(attraction)) throw new CityException("RemoveAttraction");
    attractions.Remove(attraction);
}
```

Verder zijn er nog een aantal eenvoudige methoden toegevoegd die info geven over de stad.

```
public string ShowCity()
{
    string x = $"Welcome to {Name}, located in {Country}"
        + $"\n\nThis city can be described as {Description}";
    return x;
}
public string ShowCityAttractions()
{
    string x = $"We have to offer {Attractions.Count} attractions";
    foreach(Attraction attraction in Attractions)
    {
        x +="\n"+attraction.ShowShortDescription();
    }
    return x;
}
```

Voor een attractie is het noodzakelijk dat de naam, locatie en organisator zijn ingevuld. Daarom voorzien we ook hier een aantal controles (analoog als bij City) in de setters.

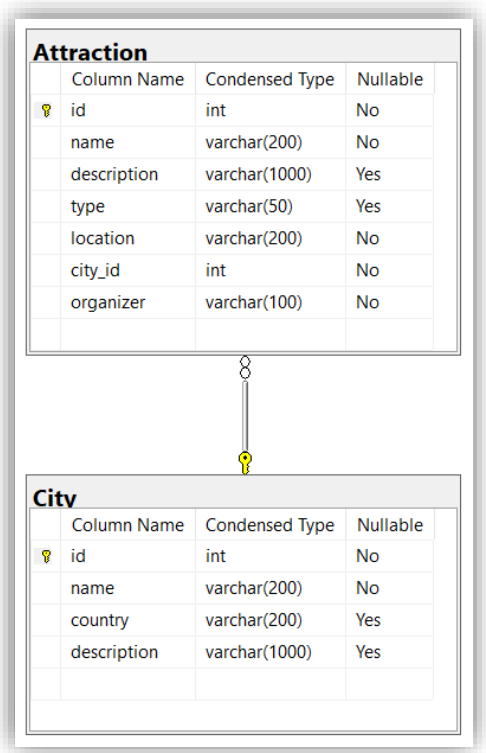
```
public class Attraction
{
    public Attraction(string name, string description, string type, string location,string organizer)...
    public Attraction(int iD, string name, string description, string type, string location,string organizer)...

    public int? Id { get; set; }
    private string name="";
    public string Name...
    public string Description { get; set; }
    public string Type { get; set; }
    private string location;
    public string Location...
    private string organizer;
    public string Organizer...
    public string ShowShortDescription()...
    public string ShowFullDescription()...
    public override bool Equals(object? other)...
}
```

In de City-klasse maken we gebruik van Contains (bij de List van attracties) en om deze methode correct te laten verlopen is het noodzakelijk om de Equals-methode te overschrijven bij Attraction. Als beiden een id hebben (en dus al zijn opgenomen in de databank) dan gebruiken we de id om te controleren of beide attracties aan elkaar gelijk zijn. Indien één van beiden (of beiden) nog geen id hebben, kijken we naar de velden Name en Organizer.

```
public override bool Equals(object? other)
{
    if (other is Attraction)
    {
        Attraction compAttraction = (Attraction)other;
        if (Id.HasValue && compAttraction.Id.HasValue)
        {
            if (Id == compAttraction.Id) return true; else return false;
        }
        else
        {
            return Name == compAttraction.Name &&
           Organizer == compAttraction.Organizer;
        }
    }
    else { return false; }
}
```

Database

Om de gegevens op te slaan maken we gebruik van een SQL databank. Er zijn twee eenvoudige tabellen voorzien.



Attraction

Column Name	Condensed Type	Nullable
id	int	No
name	varchar(200)	No
description	varchar(1000)	Yes
type	varchar(50)	Yes
location	varchar(200)	No
city_id	int	No
organizer	varchar(100)	No

City

Column Name	Condensed Type	Nullable
id	int	No
name	varchar(200)	No
country	varchar(200)	Yes
description	varchar(1000)	Yes

User stories

Voor het beheren van het domein zijn, beperken we ons tot de volgende eenvoudige user stories:

- Voeg een stad toe.
- Verwijder een stad
- Selecteer een stad met zijn attracties.
- Voeg een attractie toe aan een stad.
- Verwijder een attractie bij een stad.

Het beheer zal uitgevoerd worden door de CityManager-klasse, die we voorzien van de volgende methodes :

```
public void AddCity(City city)...
public void AddAttractionToCity(Attraction attraction, City city)...
public City GetCity(int id)...
public void DeleteCity(int id)...
public void RemoveAttractionFromCity(City city, Attraction attraction)...
```

Opmerking : CityManager.AddAttractionToCity is enkel bruikbaar om aan een bestaande City een Attraction toe te voegen. Om aan een nieuwe City een Attraction toe te voegen maken we gebruik van City.AddAttraction

Architectuur

Voor de verdere implementatie is het belangrijk om een woordje uitleg te geven over architectuur. We gaan gebruik maken van een gelaagde architectuur.

Wat zijn de voordelen van een gelaagde architectuur ?

(antwoord van ChatGPT – 13/11/2024)

Een gelaagde architectuur, vaak gebruikt in softwareontwikkeling, heeft verschillende voordelen die bijdragen aan de schaalbaarheid, onderhoudbaarheid en flexibiliteit van een systeem. De gelaagde architectuur verdeelt het systeem in afzonderlijke, gespecialiseerde lagen, zoals de presentatie-, logica- en datalaag, die elk specifieke functies vervullen. Hier zijn enkele belangrijke voordelen van een gelaagde architectuur:

*### 1. **Modulariteit***

- Doordat elke laag zijn eigen verantwoordelijkheden heeft, kan elke laag apart worden ontwikkeld, getest en onderhouden. Dit maakt het eenvoudiger om veranderingen in één laag door te voeren zonder dat dit impact heeft op de andere lagen, wat onderhoud en debugging vergemakkelijkt.

*### 2. **Herbruikbaarheid***

- Componenten binnen een laag kunnen worden hergebruikt in verschillende applicaties. Bijvoorbeeld, de logica-laag kan gedeelde bedrijfslogica bevatten die opnieuw kan worden toegepast in andere projecten zonder dat het aanpassingen vereist in de presentatie- of datalaag.

*### 3. **Schaalbaarheid***

- Elke laag kan apart worden geschaald. Dit betekent dat je bijvoorbeeld de presentatie-laag kunt schalen om meer gebruikers te ondersteunen zonder dat je de logica- of datalaag hoeft aan te passen. Dit maakt het systeem efficiënter en kosteneffectiever.)

*### 4. **Eenvoudig te testen***

- Met een gelaagde architectuur kunnen ontwikkelaars iedere laag afzonderlijk testen, omdat ze losgekoppeld zijn. Een goed ontworpen laag heeft vaak duidelijke interfaces, waardoor unit testing en integratietests makkelijker te implementeren zijn.

*### 5. **Flexibiliteit en aanpasbaarheid***

- Door een duidelijke scheiding van lagen kun je makkelijker nieuwe technologieën en oplossingen implementeren binnen één specifieke laag zonder dat dit impact heeft op de andere lagen. Bijvoorbeeld, je kunt overstappen op een andere database in de datalaag zonder dat dit veranderingen vereist in de presentatie- of logica-laag.

*### 6. **Ondersteuning van teamontwikkeling**

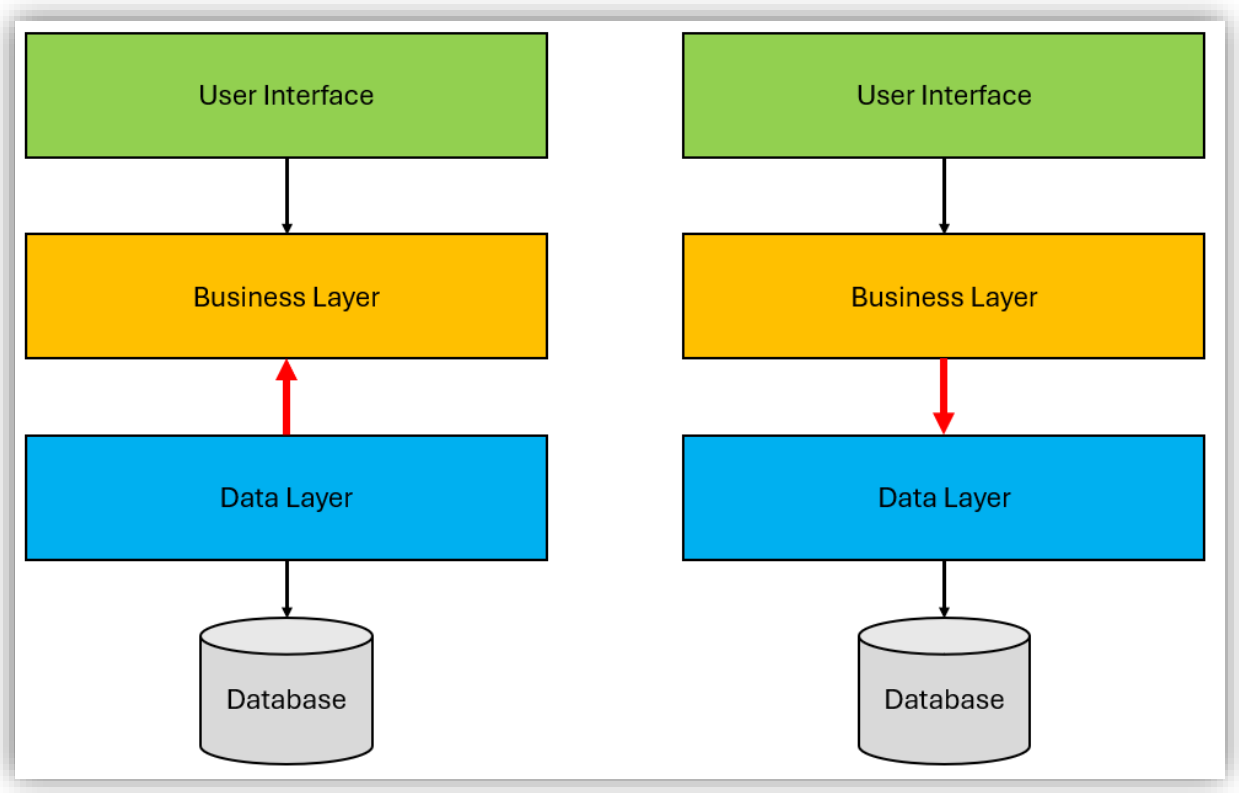
- Teams kunnen zich concentreren op verschillende lagen van de architectuur, waardoor er parallelle ontwikkeling mogelijk is. Dit verbetert de productiviteit en versnelt het ontwikkelproces omdat ontwikkelaars aan verschillende onderdelen kunnen werken zonder elkaar te hinderen.*

*### 7. **Verbeterde beveiliging**

- Een gelaagde architectuur maakt het mogelijk om beveiliging toe te passen op specifieke lagen, bijvoorbeeld door gevoelige data uitsluitend in de datalaag te beheren en toegang te beperken vanuit andere lagen. Hierdoor kan het risico op datalekken worden verminderd.*

Kortom, een gelaagde architectuur biedt een georganiseerde en gestructureerde manier om software te ontwikkelen, die zowel technische als operationele voordelen met zich meebrengt. Dit maakt het een veelgebruikte benadering in complexe en grootschalige softwareprojecten.

Er zijn echter verschillende manieren om deze architectuur te implementeren, wij gaan ons beperken tot de volgende twee:



Optie 1

We maken gebruik van een gelaagde architectuur waarbij de business-laag centraal staat. Zowel de datalaag als de UI-laag kennen de businesslaag.

Aangezien de datalaag de businesslaag kent is het mogelijk om direct met de datalaag te communiceren aan de hand van objecten die gedefinieerd zijn in de businesslaag. We maken gebruik van het repository pattern.

Wat is het repository pattern ?

(antwoord van ChatGPT – 13/11/2024)

*Het *repository pattern* is een ontwerpprincipe in softwareontwikkeling dat zorgt voor een gestructureerde manier om gegevensopslag en -toegang te beheren. Het patroon creëert een tussenlaag tussen de applicatielogica en de gegevensbron (zoals een database), waardoor de applicatie onafhankelijk wordt van de details van de gegevensopslag. Hier is een uitleg van hoe het werkt en waarom het nuttig is:*

Wat is het Repository Pattern?

*Het repository pattern creëert een abstractielaag (repository) die de gegevensopslag en -logica scheidt van de rest van de applicatie. De repository fungeert als een "verzamelplaats" voor de gegevens en biedt methoden om gegevens op te halen, op te slaan en te manipuleren zonder dat de rest van de applicatie weet *hoe* en *waar* deze gegevens precies worden opgeslagen.*

In essentie:

*- **Repository**: Deze laag biedt methoden zoals `GetById`, `Add`, `Remove`, en `Find` waarmee de applicatielogica gegevens kan ophalen, invoegen, updaten en verwijderen. De repository communiceert rechtstreeks met de database, API, of andere gegevensbron.*

*- **Applicatielogica**: Dit deel van de applicatie communiceert met de repository, maar weet niets over de onderliggende gegevensbron (zoals SQL, NoSQL, of een externe API).*

Hoe werkt het Repository Pattern?

Het repository pattern zorgt ervoor dat de applicatielogica de gegevens via een consistent interface kan benaderen, onafhankelijk van de gegevensbron. Dit betekent dat:

*1. **Verbinding met gegevensbronnen**: De repository beheert de verbinding met de database of andere gegevensbronnen en voert SQL-query's of API-aanroepen uit.*

*2. **Methoden bieden voor CRUD-bewerkingen**: De repository biedt methoden voor het uitvoeren van veelgebruikte bewerkingen, zoals ophalen (read), invoegen (create), bijwerken (update), en verwijderen (delete), meestal aangeduid als CRUD-operaties.*

*3. **Data als objecten teruggeven**: Gegevens worden doorgaans geretourneerd als objecten in plaats van database-specifieke rijen of kolommen. Dit vereenvoudigt de toegang tot de gegevens in de rest van de applicatie.*

Voordelen van het Repository Pattern

Het gebruik van een repository patroon heeft verschillende voordelen:

*1. **Scheiding van logica en data-access**:*

- Het patroon helpt de businesslogica en data-access laag van elkaar te scheiden. De applicatielogica communiceert alleen met de repository en weet niet waar of hoe de gegevens

precies worden opgeslagen. Dit maakt het makkelijker om wijzigingen door te voeren in de database zonder aanpassingen in de applicatielogica.

*2. **Testbaarheid**:*

- Repositories maken het eenvoudig om unit tests te schrijven. Omdat de repository een duidelijke interface heeft, kunnen mock-objecten of fake repositories worden gebruikt tijdens het testen, zonder dat een echte database nodig is. Dit verhoogt de snelheid en betrouwbaarheid van tests.

*3. **Onderhoudbaarheid en herbruikbaarheid**:*

- Een repository kan in meerdere delen van de applicatie worden gebruikt. Dit verhoogt de herbruikbaarheid en zorgt ervoor dat code voor gegevensbeheer centraal wordt beheerd en makkelijker kan worden aangepast of geoptimaliseerd.

*4. **Schakelbaarheid van gegevensbronnen**:*

- Omdat de rest van de applicatie niet afhankelijk is van specifieke database query's of structuren, kan het veranderen van de gegevensbron (bijv. van SQL naar NoSQL) relatief eenvoudig zijn door alleen de implementatie van de repository aan te passen.

*5. **Schaalbaarheid**:*

- Door de repository als bemiddelende laag te gebruiken, kun je de complexiteit in gegevensbeheer verhogen zonder dat de rest van de applicatie daar last van heeft. Geavanceerde caching, performance-optimalisaties, of extra gegevensbronnen kunnen worden toegevoegd binnen de repository.

Voorbeeldimplementatie van het Repository Pattern

Stel dat we een `UserRepository` hebben voor een applicatie met gebruikersgegevens. Hier is een eenvoudige implementatie in pseudo-code:

```python
class UserRepository:
    def __init__(self, database_connection):
        self.db = database_connection

    def get_user_by_id(self, user_id):
        # SQL-query naar de database om een gebruiker op te halen
        query = "SELECT * FROM users WHERE id = ?"
        result = self.db.execute(query, user_id)
        return User(result)  # Retourneert een User-object

    def add_user(self, user):
        # Voeg een nieuwe gebruiker toe aan de database
        query = "INSERT INTO users (name, email) VALUES (?, ?)"
```

self.db.execute(query, user.name, user.email)

def delete_user(self, user_id):

 # Verwijdert een gebruiker uit de database

query = "DELETE FROM users WHERE id = ?"

self.db.execute(query, user_id)

```

*In de rest van de applicatie hoeft de logica niets te weten over de SQL-query's of databaseverbindingen, omdat deze volledig worden beheerd binnen de `UserRepository`.*

*### Samenvatting*

*Het repository pattern biedt een abstractielaag tussen de applicatielogica en de gegevensbron, waardoor de code beter onderhoudbaar, testbaar, en schaalbaar wordt. Het maakt de applicatie flexibel, omdat de repository interface kan blijven bestaan terwijl de onderliggende implementatie van gegevensopslag eenvoudig kan worden aangepast of uitgebreid.*

Laten we nu eens kijken op welke manier we dit idee kunnen implementeren. De beheerder van het domein, in ons geval de klasse CityManager, zal nu gebruik maken van een repository om te communiceren met de databank. Aangezien de businesslaag centraal staat, moeten we ook in de businesslaag definiëren wat we precies verwachten van de repository. Dit kunnen we doen aan de hand van een interface ICityRepository waar we alle methoden definiëren die we nodig hebben.

```
public void AddCity(City city)
{
    try
    {
        cityRepository.AddCity(city);
    }
    catch (Exception ex) {throw new CityException("AddCity",ex); }
}
public City GetCity(int id)
{
    try
    {
        return cityRepository.GetCity(id);
    }
    catch (Exception ex) { throw new CityException("GetCity", ex); }
}
```

Het wegschrijven en ophalen van een City-object is nu heel eenvoudig aangezien de repository de business-taal spreekt. Voor het verwijderen van een City-object moeten we eerst nog een controle doen omdat we geen City-objecten wensen te verwwijderen als er nog attracties aan verbonden zijn. Dit is een business-regel dus deze hoort dan ook in de businesslaag.
```

```
public void DeleteCity(int id)
{
    try
    {
        if (cityRepository.HasAttractions(id)) {
            throw new CityException("City has attractions - no delete");
        }
        cityRepository.DeleteCity(id);
    }
    catch (Exception ex) { throw new CityException("DeleteCity", ex); }
}
```

Opmerking : een alternatief zou zijn om een City-object op te halen en aan dit object te vragen og het attracties heeft, maar dat is een stuk minder efficiënt.

Voor het toevoegen (of verwijderen) van attracties bij een stad kunnen we gebruik maken van de volgende code. We maken eerst gebruik van de business-regels die reeds geïmplementeerd zijn in het domein-object, namelijk dat er geen dubbels mogen voorkomen. Als aan de regels is voldaan dan geven we opdracht aan de repository om de aanpassingen door te voeren.

```
public void AddAttractionToCity(City city,Attraction attraction)
{
    try
    {
        city.AddAttraction(attraction);
        cityRepository.AddAttraction((int)city.Id, attraction);
    }
    catch (Exception ex) { throw new CityException("AddAttractionToCity", ex); }
}
public void RemoveAttractionFromCity(City city, Attraction attraction)
{
    try
    {
        if (city.HasAttraction(attraction))
        {
            city.RemoveAttraction(attraction);
            cityRepository.RemoveAttraction((int)attraction.Id);
        }
    }
    catch (Exception ex) { throw new CityException("RemoveAttractionFromCity", ex); }
}
```

Op basis van de geïmplementeerde user stories krijgen we dan de volgende interface.

```
public interface ICityRepository
{
    void AddAttraction(int id, Attraction attraction);
    void AddCity(City city);
    void DeleteCity(int id);
    City GetCity(int id);
    bool HasAttractions(int id);
    void RemoveAttraction(int id);
}
```

De specifieke implementatie van deze interface kan dan in de datalaag plaatsvinden. De CitManager-klasse hoeft niet te weten op welke manier dit precies wordt afgehandeld, elke klasse die de interface implementeerd voldoet. Aangezien de CityManager-klasse gebruikt maakt van "een" implementatie van de ICityRepository interface voorzien we ook een variabele van dit type en gebruiken we dependency injection via de constructor om aan een concreet object te geraken.

```
public class CityManager
{
    private ICityRepository cityRepository;

    public CityManager(ICityRepository cityRepository)
    {
        this.cityRepository = cityRepository;
    }
```

In de datalaag maken we nu een implementatie van de ICityRepository interface. Er is gekozen om gebruik te maken van een SQL-databank en ADO.NET. Omdat we in de businesslaag enkel de interface gebruiken, kan de datalaag eenvoudig worden vervangen met een andere implementatie, voorbeeld entity framework, zonder dat er in de businesslaag aanpassingen nodig zijn.

```
public class CityRepository : ICityRepository
{
    private string connectionString;

    public CityRepository(string connectionString)
    {
        this.connectionString = connectionString;
    }

    public void AddAttraction(int id, Attraction attraction)...
    public bool HasAttractions(int id)...
    public void AddCity(City city)...
    public void DeleteCity(int id)...
    public City GetCity(int id)...
    public void RemoveAttraction(int id)...
}
```

Het opvragen van een City-object kan er dan als volgt uit zien. We kunnen alle gegevens met één query opvragen en dan het City-object samenstellen.

```
public City GetCity(int id)
{
    City city = null;
    string SQL = "SELECT t1.id cityid,t1.name cityname,t1.country,t1.description citydescription,t2.*
      FROM [City] t1 left join Attraction t2 on t1.id=t2.city_id WHERE t1.id=@id";
    using (SqlConnection conn = new SqlConnection(connectionString))
    using (SqlCommand cmd = conn.CreateCommand())
    {
        try
        {
            conn.Open();
            cmd.CommandText = SQL;
            cmd.Parameters.AddWithValue("@id",id);
            IDataReader reader = cmd.ExecuteReader();
            while (reader.Read())
            {
                if (city==null) city=new City(id, (string)reader["cityname"], (string)reader
                  ["citydescription"], (string)reader["country"]);
                if (!reader.IsDBNull(reader.GetOrdinal("id")))
                {
                    Attraction attraction = new Attraction((int)reader["id"], (string)reader["name"],
                    (string)reader["description"], (string)reader["type"], (string)reader["location"],
                    (string)reader["organizer"]);
                    city.AddAttraction(attraction);
                }
            }
            return city;
        }
        catch (Exception ex)
        {
            throw new Exception("Get City", ex);
        }
    }
}
```

Voor het wegschrijven is er ook één methode voorzien. Deze methode kan een city-object eventueel met een lijst van attracties in één transactie wegschrijven.

```
public void AddCity(City city)
{
    string SQLcity = "INSERT INTO city(name,description,country) output INSERTED.ID VALUES
      (@name,@description,@country)";
    string SQLattraction = "INSERT INTO attraction(name,description,type,location,city_id) output
      INSERTED.ID VALUES(@name,@description,@type,@location,@city_id,@organizer)";
    using (SqlConnection conn = new SqlConnection(connectionString))
    using (SqlCommand cmd = conn.CreateCommand())
    {
        try...
    }
}
```

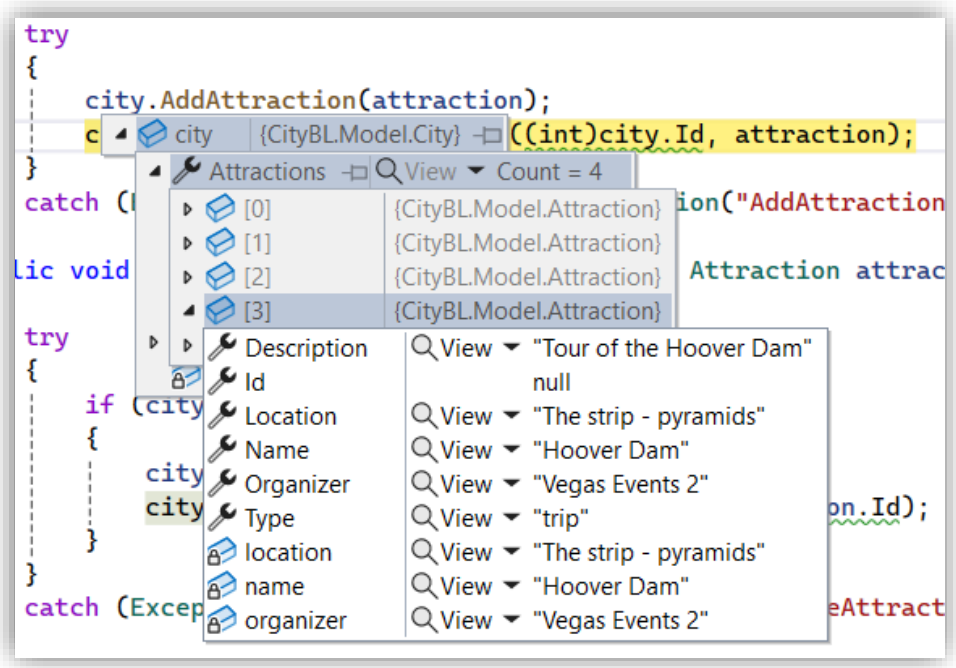
```
try
{
    cmd.Transaction = conn.BeginTransaction();
    conn.Open();
    cmd.CommandText = SQLcity;
    cmd.Parameters.AddWithValue("@name", city.Name);
    cmd.Parameters.AddWithValue("@description", city.Description);
    cmd.Parameters.AddWithValue("@country", city.Country);
    int id = (int)cmd.ExecuteScalar();
    city.Id = id;
    cmd.CommandText = SQLattraction;
    cmd.Parameters.Add(new SqlParameter("@name",SqlDbType.Int));
    cmd.Parameters.Add(new SqlParameter("@description", SqlDbType.VarChar));
    cmd.Parameters.Add(new SqlParameter("@type", SqlDbType.VarChar));
    cmd.Parameters.Add(new SqlParameter("@location", SqlDbType.VarChar));
    cmd.Parameters.Add(new SqlParameter("@city_id", SqlDbType.Int));
    foreach (Attraction attraction in city.Attractions)
    {

        cmd.Parameters["@name"].Value=attraction.Name;
        cmd.Parameters["@description"].Value=attraction.Description;
        cmd.Parameters["@type"].Value=attraction.Type;
        cmd.Parameters["@location"].Value=attraction.Location;
        cmd.Parameters["@city_id"].Value=city.Id;
        cmd.Parameters["@organizer"].Value = attraction.Organizer;
        int aid = (int)cmd.ExecuteScalar();
        attraction.Id = aid;
    }
    cmd.Transaction.Commit();
}
catch (Exception ex)
{
    cmd.Transaction.Rollback();
    throw new Exception("City Save", ex);
}
```

Hoe zit het feitelijk met de “auto generated” ids ?

Het aanmaken van de ids (zowel voor city- als attraction-objecten) gebeurt door de databank. Initieel worden de objecten dus aangemaakt zonder id, maar nadat ze zijn opgeslagen in de databank, moeten de ids wel worden ingevuld in de business-objecten. Laten we als voorbeeld het geval nemen waarbij we een attractie toevoegen aan een stad.

```
public void AddAttractionToCity(City city,Attraction attraction)
{
    try
    {
        city.AddAttraction(attraction);
        cityRepository.AddAttraction((int)city.Id, attraction);
    }
    catch (Exception ex) { throw new CityException("AddAttractionToCity", ex); }
}
```

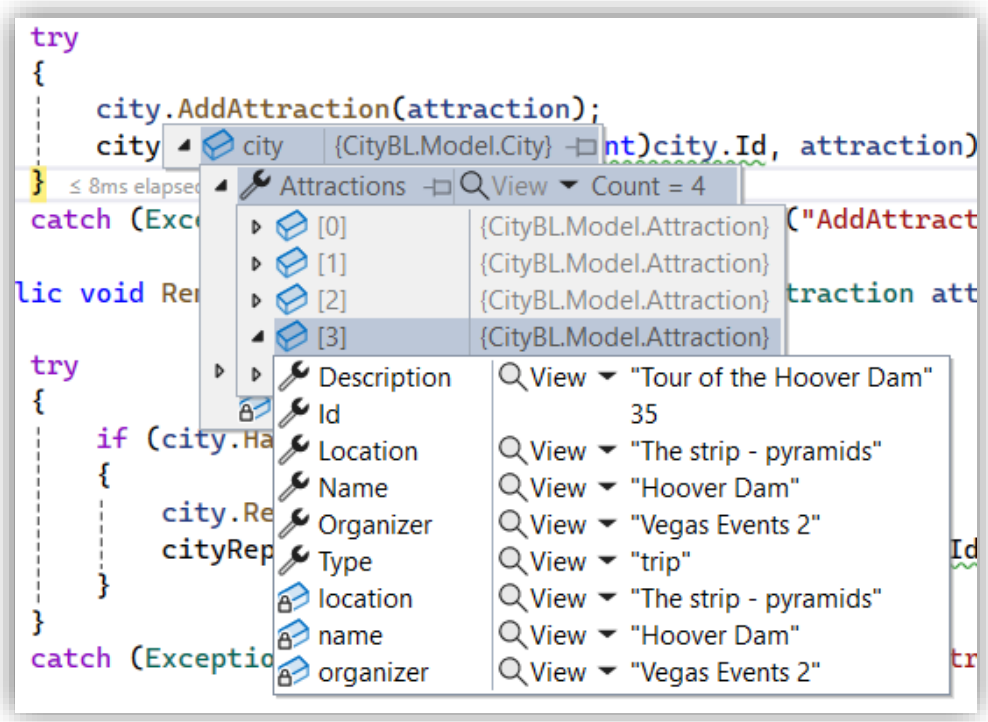
In de CityManager-klasse voegen we een attraction toe aan het city-object. Op dit moment heeft de attraction nog geen id, zoals we kunnen zien in de onderstaande figuur.



Wanneer we de repository-methode AddAttraction uitvoeren, geven we een business-object attraction mee en stellen we het id in van zodra dit beschikbaar is na het uitvoeren van het INSERT-statement.

```
public void AddAttraction(int id, Attraction attraction)
{
    string SQL = "INSERT INTO attraction(name,description,type,location,city_id,organizer) output
      INSERTED.ID VALUES(@name,@description,@type,@location,@city_id,@organizer)";
    using (SqlConnection conn = new SqlConnection(connectionString))
    using (SqlCommand cmd = conn.CreateCommand())
    {
        try
        {
            conn.Open();
            cmd.CommandText = SQL;
            cmd.Parameters.AddWithValue("@organizer",attraction.Organizer);
            cmd.Parameters.AddWithValue("@name",attraction.Name);
            cmd.Parameters.AddWithValue("@description",attraction.Description);
            cmd.Parameters.AddWithValue("@type",attraction.Type);
            cmd.Parameters.AddWithValue("@location",attraction.Location);
            cmd.Parameters.AddWithValue("@city_id",id);
            int aid = (int)cmd.ExecuteScalar();
            attraction.Id = aid;
        }
        catch (Exception ex)
        {
            throw new Exception("AddAttraction", ex);
        }
    }
}
```

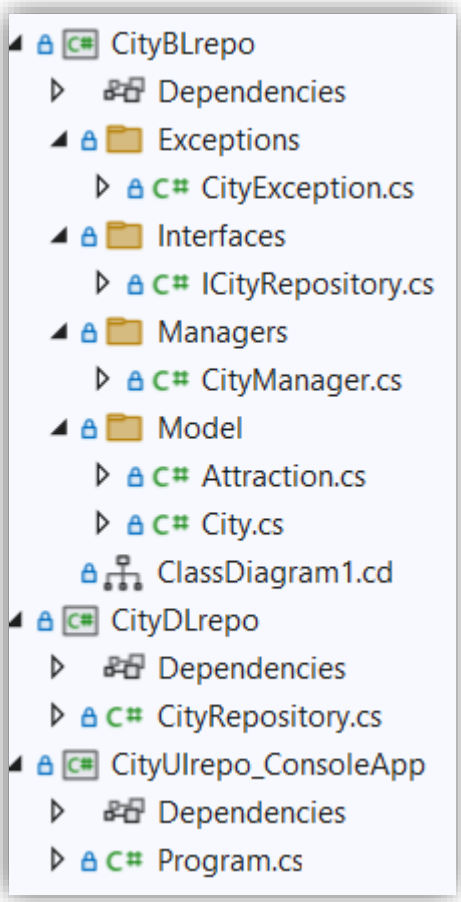
Aangezien het attraction-object dat we meegeven, hetzelfde object is waarnaar in het city-object wordt verwezen, zal elke aanpassing daar dan ook meteen zichtbaar zijn.



Rest ons nog het bekijken van de UI-laag. In de UI-laag (in dit geval een console-app) vinden we dan de volgende code terug. Er wordt een specifieke implementatie van de ICityRepository meegegeven met de constructor van de CityManager. Voor het verdere verloop zullen acties in de UI steeds vertaald worden naar het oproepen van methodes van de CityManager-klasse.

```
ICityRepository repo=new CityRepository(conn);
CityManager cm = new CityManager(repo);
City c = cm.GetCity(7);
```

De opbouw van het project ziet er dan als volgt uit.



Optie 2

Hier stellen we dat de UI-laag de businesslaag kent en de businesslaag weet hoe de datalaag in elkaar zit. Een éénduidige oplossing is hier niet voor handen en we kiezen er voor om gebruik te maken van DAO's (Data Access Objecten). En zelfs bij deze keuze zijn er verschillende implementaties (en interpretaties) mogelijk. De volgende informatie over DAO's is terug te vinden op de wikipedia-pagina.

In software, a **data access object** (**DAO**) is a pattern that provides an abstract interface to some type of database or other persistence mechanism. By mapping application calls to the persistence layer, the DAO provides data operations without exposing database details. This isolation supports the single responsibility principle. It separates the data access the application needs, in terms of domain-specific objects and data types (the DAO's public interface), from how these needs can be satisfied with a specific DBMS (the implementation of the DAO).

There are various ways in which this object can be implemented:

- One DAO for each table.
- One DAO for all the tables for a particular DBMS.
- Where the SELECT query is limited only to its target table and cannot incorporate JOINS, UNIONS, subqueries and Common Table Expressions (CTEs)
- Where the SELECT query can contain anything that the DBMS allows.

Advantages [edit]

Using data access objects (DAOs) offers a clear advantage: it separates two parts of an application that don't need to know about each other. This separation allows them to evolve independently. If business logic changes, it can rely on a consistent DAO interface. Meanwhile, modifications to persistence logic won't affect DAO clients.[2][3]

All details of storage are hidden from the rest of the application (see information hiding). Unit testing code is facilitated by substituting a test double for the DAO in the test, thereby making the tests independent of the persistence layer.

In the context of the Java programming language, DAO can be implemented in various ways. This can range from a fairly simple interface that separates data access from the application logic, to frameworks and commercial products.

Disadvantages [edit]

Potential disadvantages of using DAO include leaky abstraction,[*citation needed*] code duplication, and abstraction inversion. In particular, the abstraction of the DAO as a regular Java object can obscure the high cost of each database access. Developers may inadvertently make multiple database queries to retrieve information that could be returned in a single operation. If an application requires multiple DAOs, the same create, read, update, and delete code may have to be written for each DAO.[5]

Note that these disadvantages only appear when you have a separate DAO for each table and the SELECT query is prevented from accessing anything other than the target table.

(https://en.wikipedia.org/wiki/Data_access_object)

Aangezien de business-laag nu wel de datalaag kent, en niet omgekeerd, is het niet mogelijk voor de methodes in de datalaag om business-objecten terug te geven. Het is nu noodzakelijk om in de datalaag objecten te definiëren die de business-laag kan opvragen. Er is hier gekozen om DAO's te maken voor elke tabel en dus maken we ook voor elke record uit de tabel een object aan. Deze objecten dienen nu enkel om gegevens tussen de datalaag en de businesslaag uit te wisselen. Zulke objecten worden transfer objecten (TO) of data transfer objecten (DTO) genoemd en bevatten enkel gegevens, geen methodes, business-regels, ...

```
public class AttractionTO
{
    public AttractionTO(int? iD, string name, string description, string type, string location, int
    ¦ city_Id,string organizer)...

    public int? ID { get; set; }
    public string Name { get; set; }
    public string Description { get; set; }
    public string Type { get; set; }
    public string Location { get; set; }
    public string Organizer { get; set; }
    public int City_Id { get; set; }
}
```

Het transfer-object voor attracties bevat dus alle kolommen uit de databanktabel, inclusief het city_id. Opmerking een object in de businesslaag zou nooit het city_id als property hebben, maar een City-object.

Het City-transfer-object bevat in tegenstelling tot het City-object in de businesslaag geen lijst van attraction-objecten omdat we er voor gekozen hebben dat er voor elke tabel een aparte DAO zal zijn.

```
public class CityTO
{
    public CityTO(int? id, string name, string description, string country)...

    public int? Id { get; set; }
    public string Name { get; set; }
    public string Description { get; set; }
    public string Country { get; set; }
}
```

De DAO's zelf bevatten de klassieke CRUD-operaties, aangevuld met een aantal specifieke query-verzoeken.

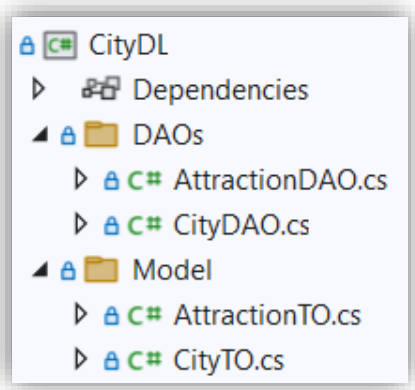
```
public class AttractionDAO
{
    private string connectionString;

    public AttractionDAO(string connectionString)...
    public AttractionTO Save(AttractionTO attraction)...
    public void Delete(int id)...
    public List<AttractionTO> GetByCity(int cityId)...
    public AttractionTO GetById(int id)...
    public bool CityHasAttractions(int id)...
}
```

```
public class CityDAO
{
    private string connectionString;

    public CityDAO(string connectionString)...
    public CityTO Save(CityTO city)...
    public void Delete(int id)...
    public List<CityTO> GetAll()...
    public CityTO GetById(int id) ...
}
```

De methodes zelf zijn eenvouder dan bij het gebruik van de repository omdat ze slechts één tabel bevragen of bewerken.

```
public CityTO GetById(int id)
{
    string SQL = "SELECT * FROM city WHERE id=@id";
    using (SqlConnection conn = new SqlConnection(connectionString))
    using (SqlCommand cmd = conn.CreateCommand())
    {
        try
        {
            conn.Open();
            cmd.CommandText = SQL;
            cmd.Parameters.AddWithValue("@id", id);
            IDataReader reader = cmd.ExecuteReader();
            reader.Read();
            return new CityTO(id, (string)reader["name"], (string)reader["Description"], (string)reader
              ["Country"]);
        }
        catch (Exception ex)
        {
            throw new Exception("City GetById", ex);
        }
    }
}
```

De datalaag zelf ziet er dan als volgt uit.



In de businesslaag bevinden zich de domein-entiteiten City en Attraction, welke identiek zijn aan deze met de architectuur waarbij de businesslaag centraal staat. Voor het beheren van het domein wordt er weer gebruik gemaakt van een CityManager-klasse die in essentie dezelfde methodes bevat als bij de vorige optie. In plaats van gebruik te maken van een ICityRepository implementatie-object, worden er nu DAO-klassen gebruikt. Er is ook nog een extra klasse, CityMapper die we nodig hebben om de transfer-objecten om te zetten naar business-objecten en omgekeerd.

```
public class CityManager
{
    private CityDAO cityDAO;
    private AttractionDAO attractionDAO;

    public CityManager(CityDAO cityDAO, AttractionDAO attractionDAO)...
    private CityMapper cityMapper;

    public void AddCity(City city)...
    public void AddAttractionToCity(Attraction attraction, City city)...
    public City GetCity(int id)...
    public void DeleteCity(int id)...
    public void RemoveAttractionFromCity(City city, Attraction attraction)...
}
```

De CityMapper-klasse heeft dus 4 methodes, 2 om de transfer-objecten om te zetten naar business-objecten en 2 methodes voor de omzetting in de andere richting.

```
public class CityMapper
{
    private AttractionDAO attractionDAO;

    public CityMapper(AttractionDAO attractionDAO)...
    public Attraction MapAttractionFromDL(AttractionTO attractionDM)...
    public City MapCityFromDL(CityTO cityDM)...
    public AttractionTO MapAttractionFromBL(Attraction attraction,City city)...
    public CityTO MapCityFromBL(City city)...
}
```

De omzettingen van business-objecten naar transfer-objecten is relatief eenvoudig en omvat enkel het oproepen van de constructor. Alle informatie om de objecten aan te maken is aanwezig in de transfer-objecten zodat dit vlot kan verlopen.

```
public AttractionTO MapAttractionFromBL(Attraction attraction,City city)
{
    return new AttractionTO(attraction.Id, attraction.Name, attraction.Description, attraction.Type,
      attraction.Location,(int)city.Id,attraction.Organizer);
}
public CityTO MapCityFromBL(City city)
{
    CityTO cityDM = new CityTO(city.Id, city.Name, city.Description, city.Country);
    return cityDM;
}
```

De omzetting in de andere richting is complexer voor het City-object omdat deze ook een lijst van Attraction-objecten bevat en die zijn niet aanwezig in het transfer-object. Daarom is het noodzakelijk om een attractionDAO-object te voorzien in de CityMapper om de nodige Attraction-transfer-objecten op te halen en om te zetten.

```
public Attraction MapAttractionFromDL(AttractionTO attractionDM)
{
    return new Attraction(attractionDM.ID, attractionDM.Name, attractionDM.Description,
      attractionDM.Type, attractionDM.Location,attractionDM.Organizer);
}
public City MapCityFromDL(CityTO cityDM)
{
    List<Attraction> attractions = new List<Attraction>();
    foreach(AttractionTO attractionDM in attractionDAO.GetByCity((int)cityDM.Id))
    {
        attractions.Add(MapAttractionFromDL(attractionDM));
    }
    City city= new City((int)cityDM.Id, cityDM.Name, cityDM.Description, cityDM.Country,attractions);
    return city;
}
```

De methodes in de CityManager-klasse zijn gelijkaardig, maar de implementatie daarentegen is wel verschillend aangezien we met DAO's zullen werken in plaats van een repository. Wanneer we een City-object wensen weg te schrijven, moeten we nu eerst de cityDAO gebruiken voor het schrijven naar de city-tabel en daarna de attractionDAO voor het wegschrijven in de attraction-tabel.

```
public void AddCity(City city)
{
    try
    {
        city.Id = cityDAO.Save(cityMapper.MapCityFromBL(city)).Id;
        foreach (Attraction attraction in city.Attractions)
        {
            attraction.Id = attractionDAO.Save(cityMapper.MapAttractionFromBL(attraction, city)).ID;
        }
    }
    catch (Exception e)
    {
        throw new CityException("AddCity");
    }
}
```

Het opvragen van een City-object is wel gelijkaardig, maar dat is enkel omdat we het samenstellen van het object hebben opgenomen in de CityMapper-klasse. Het opvragen van een City-object vraagt nu twee aparte queries in plaats van één query zoals bij de repository implementatie.

```
public City GetCity(int id)
{
    try
    {
        return cityMapper.MapCityFromDL(cityDAO.GetById(id));
    }
    catch (Exception e) { throw new CityException("GetCity"); }
}
```

En hoe zit het met de "auto-generated" ids hier ?

We bekijken opnieuw de situatie waarbij we een attractie toevoegen aan een stad. Eerst checken we opnieuw de business-regel aangaande duplicates en voegen de attractie dus toe aan het City-

object. Daarna mappen we het attraction-object naar het overeenkomende transfer-object en slaan het object op in de databank. Het opslaan van het transfer-object (zie verder) bezorgt ons de nieuwe id die bij de Save-methode is toegekend aan het transfer-object. Daarna moeten we de id ook nog toekennen aan het business-object. Om dit te laten werken is het noodzakelijk dat we een referentie hebben naar het attraction-transfer-object.

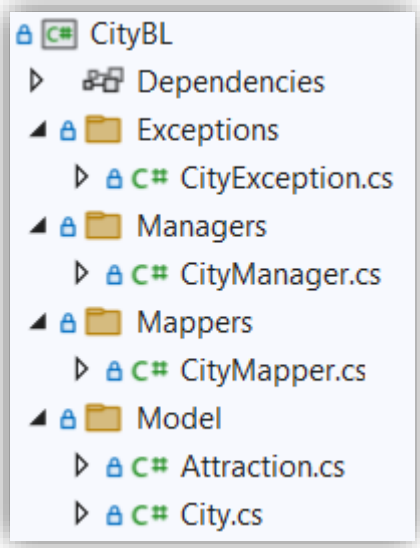
```
public void AddAttractionToCity(Attraction attraction, City city)
{
    try
    {
        city.AddAttraction(attraction);
        AttractionTO aTO = cityMapper.MapAttractionFromBL(attraction, city);
        attractionDAO.Save(aTO);
        attraction.Id=aTO.ID;
    }
    catch (Exception e)
    {
        throw new CityException("AddAttractionToCity");
    }
}
```

Na het uitvoeren van het INSERT-statement plaatsen we de nieuwe id in het attraction-transfer-object.

```
public AttractionTO Save(AttractionTO attraction)
{
    string SQL = "INSERT INTO attraction(name,description,type,location,city_id,organizer) output
      INSERTED.ID VALUES(@name,@description,@type,@location,@city_id,@organizer)";
    using (SqlConnection conn = new SqlConnection(connectionString))
    using (SqlCommand cmd = conn.CreateCommand())
    {
        try
        {
            conn.Open();
            cmd.CommandText = SQL;
            cmd.Parameters.AddWithValue("@name", attraction.Name);
            cmd.Parameters.AddWithValue("@description", attraction.Description);
            cmd.Parameters.AddWithValue("@type", attraction.Type);
            cmd.Parameters.AddWithValue("@location", attraction.Location);
            cmd.Parameters.AddWithValue("@organizer", attraction.Organizer);
            cmd.Parameters.AddWithValue("@city_id", attraction.City_Id);
            int id = (int)cmd.ExecuteScalar();
            attraction.ID = id;
            return attraction;
        }
        catch (Exception ex)
        {
            throw new Exception("attraction Save", ex);
        }
    }
}
```

Ook bij het opslaan van een City-object moeten we extra aandacht besteden aan het toekennen van de aangemaakt ids.

De businesslaag ziet er dan als volgt uit.


CityBL
Dependencies
Exceptions
CityException.cs
Managers
CityManager.cs
Mappers
CityMapper.cs
Model
Attraction.cs
City.cs