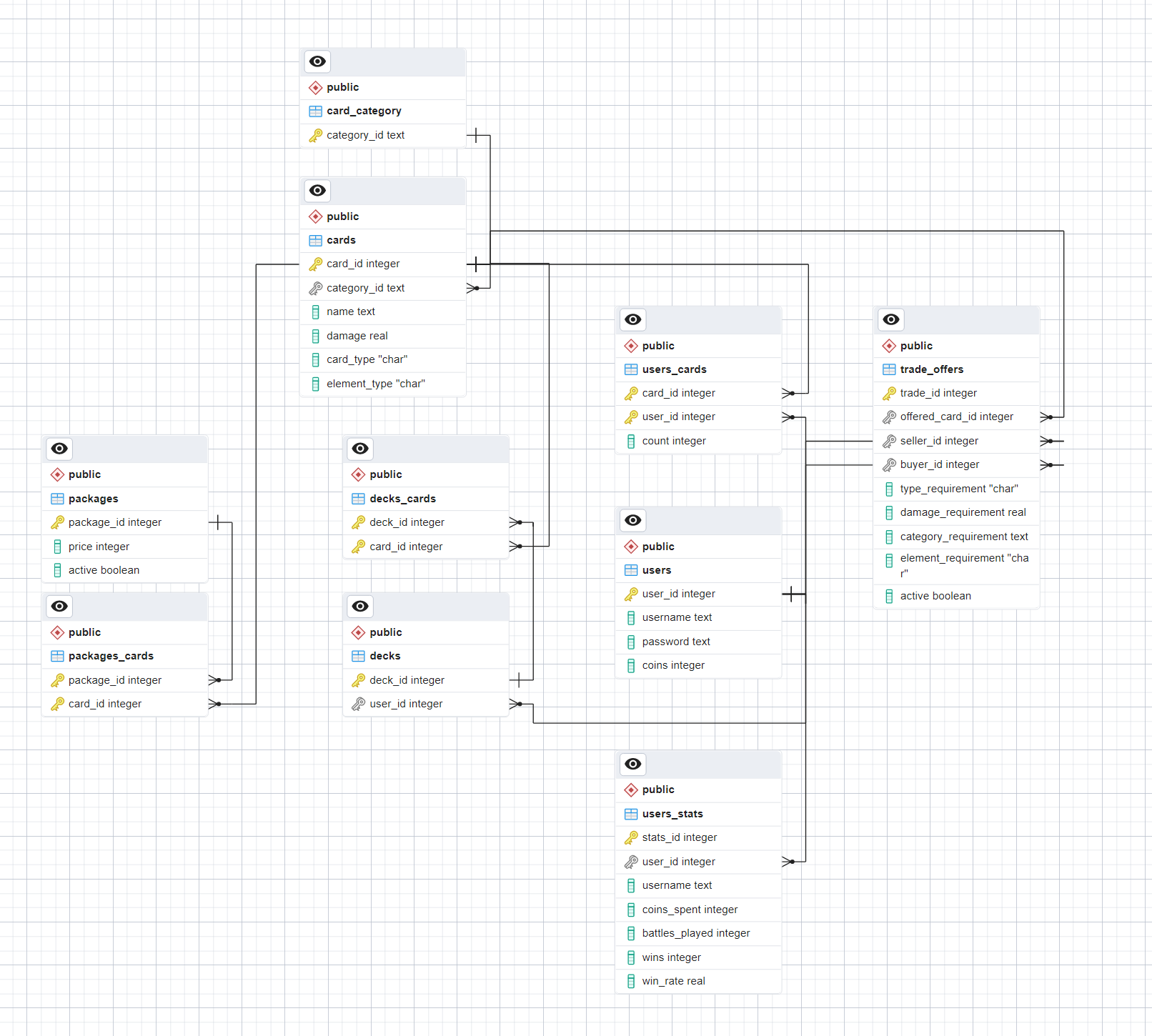
MonsterCradTradingGame – Protocol

Link: https://github.com/Davidhoechtl/SWEN\_Semesterprojekt\_2022.git

# Design

The solution is divided into 4 separate projects. The MonsterTradingCardGame\_Hoechtl.csproj, MTCG.DAL, MTCG.Logic and MTCG.Tests.



## Server Architecture

Ein Bild, das Text, Screenshot, festlegen, mehrere enthält.

Automatisch generierte Beschreibung

The MonsterTradingCardGame\_Hoechtl.csproj is the Server of the Application. It Builds the complete Application via Autofac and starts the HttpListener which listens for user requests.  
The HttpListener requests asynchronize.

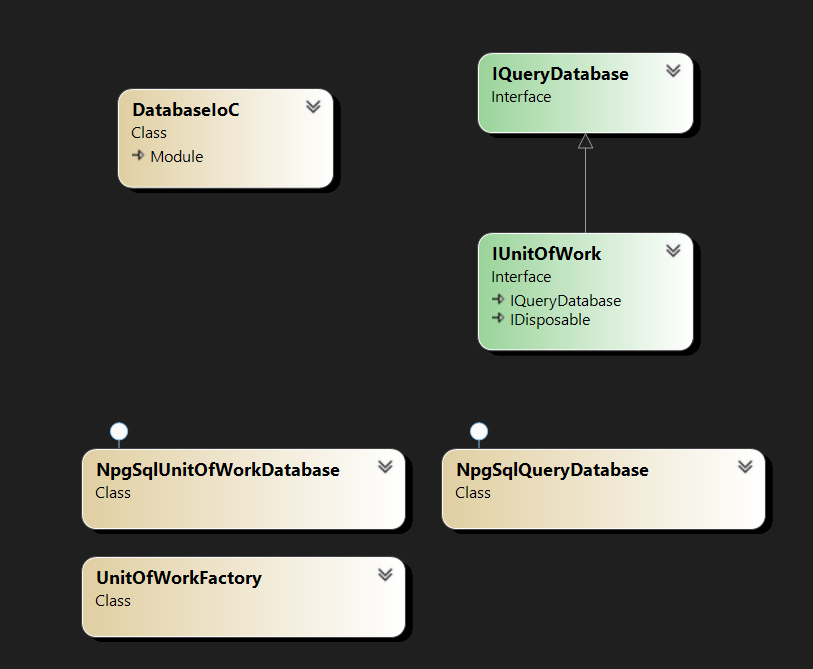
**IHandler** is the interface that is used to create Endpoints of the server. The request is broken down into its individual parts. From the url the module name is extracted and matched with all handler names.

**HandlerMethodResolver** is used to calculate the correct method of the found handler. This happens using HttpMethodAttributes and Reflexion. There can only be two Parameters.

* SessionContext -> Tells which user sent the request,
* object requestPayload -> sent JSON content.

**SessionContext** is used to handle token-based security. Here every user\_id is mapped with a generated GUID, which becomes the API key the user can user after login.

## Data Access Layer

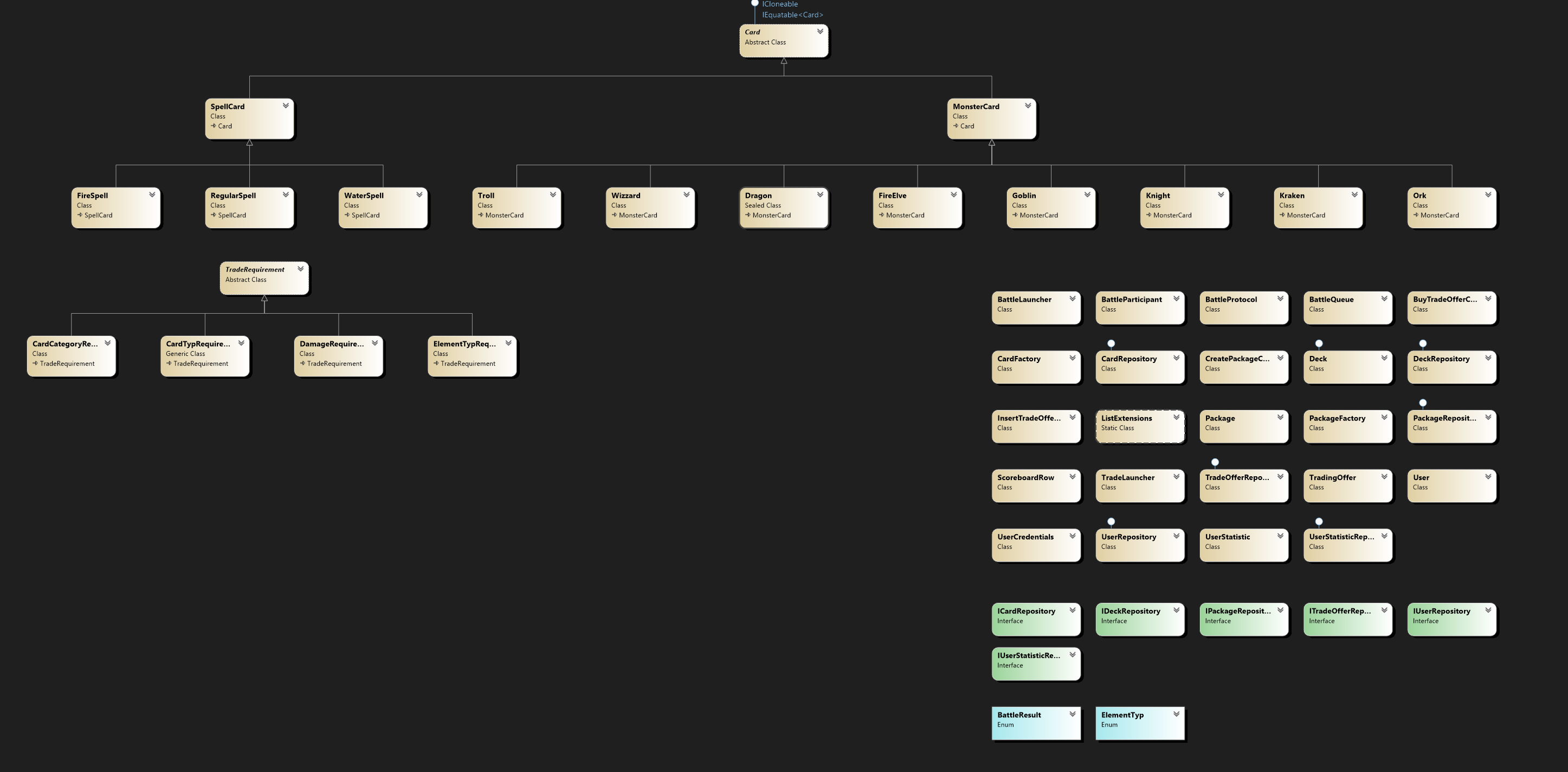


The project uses a Postgres SQL database. Npgsql Extension is used to communicate with it using C#.  
The Abstract Interface for the database implementation is called **IQueryDatabase**. Here methods like GetItem and GetItems are defined in a generic pattern.

**IUnitOfWork** is created via **UnitOfWorkFactory** and is used for transaction-based data access.

NpgSqlParameter and prepared Statements are used for avoiding SQL injection.

## Game Architecture



The Features of the Game are Trading, Battle and acquire cards. The whole game is built around two type of Cards. The SpellCards and the MonsterCards. Most of the cards have their own unique features.

**Battle queuing**: if a user sends a start battle request, he gets queued. A **separate Thread** is constantly checking which players are in the queue and matches them.

**Trading**: Add Trade requirements via abstract class **TradeRequirement**. Currently there are 4 types of requirement: damage, cardtype, category, element type requirements.

**Cards**: the battle logic is inside the sub-classes of the abstract class Card. Cards get instantiate only inside the **CardFactory**.

**Packages**: Packages are created by the **PackageFactory** which produces a given range of card (only admins can perform this action)

# Lessons Learned

## Test driven development is better than I have expected.

Creating unit tests first gives the opportunity to build the application in a isolated manner. For example if I try to implement the unit tests later i need to refactor most of the code, which can be a lot of work.

Also, I noticed i spent much less time with bugfixes and testing when using unit tests first.

## Generic Methods/Classes are great for scalability.

Using an abstract approach to a problem often lead me to the point where i need to distinguish between types. Using generic Methods made my day easier and I spent less time building switch-cases.

## Using Inversion of Control make live better.

With this method I didn’t have to instantiate every class inside the server class and hand it down to the endpoints I could just say “give me an Enumerable of every registered Endpoint” which was a really nice and extensible way of building the architecture of the server.

Also it made the whole application loosely coupled, which is also create for scalability and maintainability of the code.

## Reflection is cool.

In this project I really got to know the purpose and power of reflection. Analysing assembly and working with types and their properties, methods gave me the tools to build a routing system for my console application, which worked out great.

## Understanding of the concept of UnitOfWork.

In work I have faced the interface IUnitOfWork a lot. Unit now it was a big mystery for me how it works and what the benefits of it are. Through this project I have learned what it is and how to build it myself.

# Unit-Tests Decisions

The 3 big parts of the game are battle, trade and quire packages. I tried to build the unit tests around those features. For battle logic it was straight forward to build the unit-tests (I also used test driven development here).

For Trading it was a little bit more difficult. I wrote the logic first tested it and it seemed to work after testing it manually. I then wanted to implement some unit-tests for it and I noticed that i need to isolate the code more. So, I started refactoring a lot.

# Unique Features (Architectural)

## Method routing to the server Endpoint

I started with implementing custom attributes to the game. One for each Http-Method that will be provided by the API. I then created the class Handler Method resolver which takes in a path and some other information and then executes the right method of the handler using reflection.

Ein Bild, das Text enthält.

Automatisch generierte Beschreibung

## IUnitOfWork and IQueryDatabase abstract implementation of Database access

I wanted one interface that I can use to make calls to the postgres sql server. This interface is **IQueryDatabase.**

Ein Bild, das Text enthält.

Automatisch generierte Beschreibung

It provides all sorts of useful and generic methods to acquire and write data to the database. So if I wanted to add another Database like SQL-Lite i could do so by implementing this interface and register it to the IoC-Container.

I then noticed that I need to make transaction-based calls for the critical parts like updating multiple elements. Without them the database would in case of an error end up in an invalid state. Here the IUnitOfWork Interface comes into place.

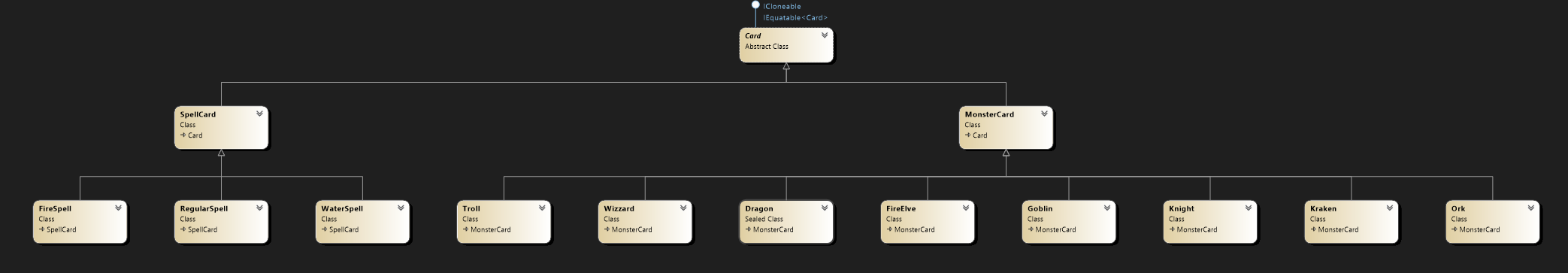
Ein Bild, das Text enthält.

Automatisch generierte Beschreibung

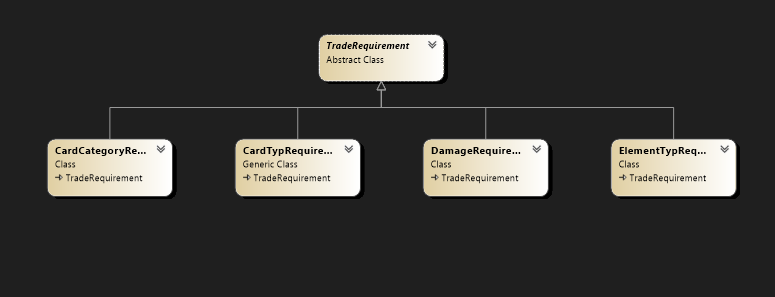
It implements the IQueryDatabase interface and add the methods BeginTransaction() => start of a new transaction and Commit => the end of the transaction.

## Abstract approaches

I tried to keep the game as extensible and maintainable as possible. For Example, I use abstract approaches for cards and trade requirements.



If you want to add a new Card to the game you just need to add the category to the table card\_category and create a new sub class of MonsterCard or Spellcard.



For the Trade requirements its also easy to add new ones. Add another field in the trade\_offers table and implement a new subclass fo TradRequirement.

# Tracked Time

Ein Bild, das Tisch enthält.

Automatisch generierte Beschreibung