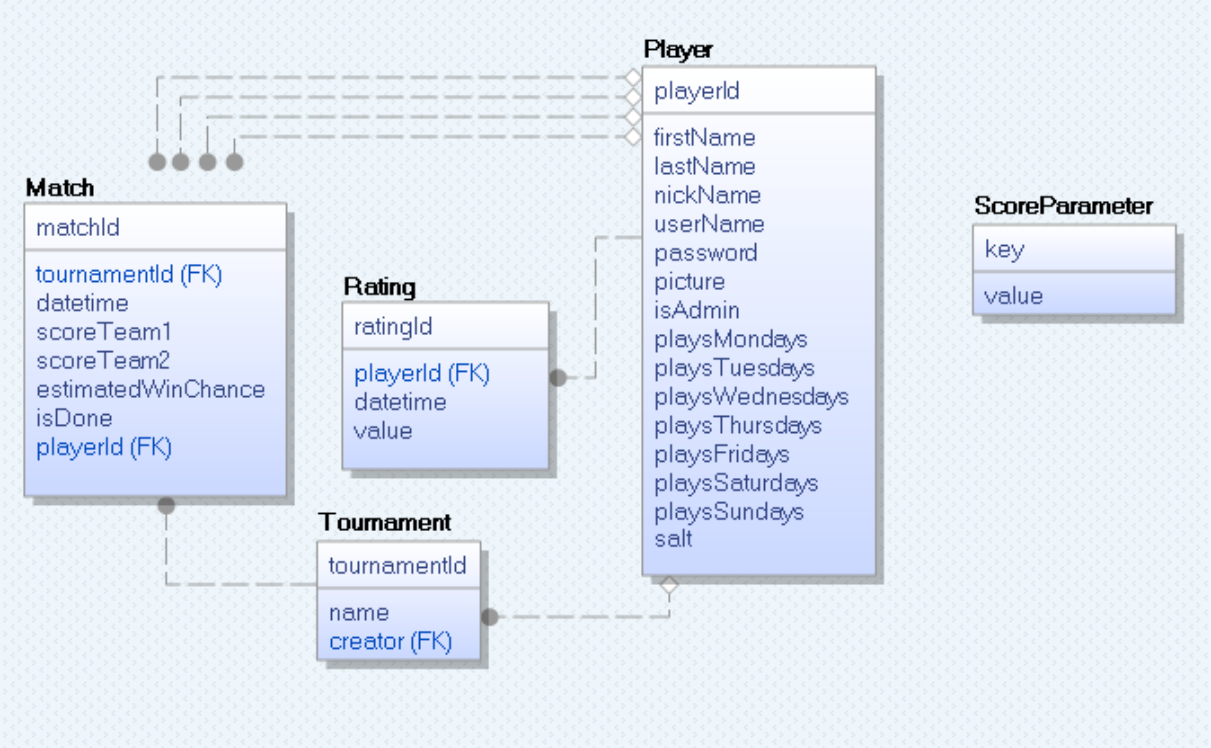
WuHu – WuzzelHub

Erste Ausbaustufe

Datenbank

Modell Diagramm



Beschreibung

Als Datenbank verwende ich ein MySql Localdb File wie in der Übung.

Achtung! Vor Ausführung der Tests sollten die App Settings (App.config in WuHu.Test bzw. WuHu.Server) richtig konfiguriert sein (ConnectionString, DbPath und SqlPath!)

Das Datenmodell ist auf möglichst wenige Entitäten beschränkt, ohne Normalformen zu verletzen. Es deckt dabei trotzdem die Problemstellung ausreichend ab.

Ein *Player* enthält alle Informationen zu einem Spieler. Darunter fallen alle Namen, sein gehashtes Passwort und das zugehörige generierte Salz, um den Passwortstring vor dem Hashes zu salzen. Weiters ist sein Bild binär, sowie sein Adminstatus als boolean gespeichert, und an welchen Tagen er spielt ebenfalls als booleans. Dies ermöglicht ein schnelles Abfragen von Spielern an einem gewissen Tag, ohne aufwendige Joins.

Im *Rating* werden alle Wertungen von Spielern gespeichert. Diese Wertungen sind mit einem Datum versehen, sodass man nacher eine Statistik der Spielerwertungen über einen längeren Zeitraum anzeigen kann. Jede Wertung ist einem gewissen Spieler zugeordnet.

Tournament fasst eine Menge von Spielen zu einem Turnier zusammen. Diesem kann ein Name gegeben werden, und es wird auch der Ersteller des Turniers als creator gespeichert.

Im *Match* werden alle relevanten Informationen eines Spiels gespeichert. Dazu gehören die vier Mitspieler, wobei Spieler1 und Spieler2 als Team1 gegen Spieler3 und Spieler4 als Team2 spielen. scoreTeam1 und scoreTeam2 speichern den derzeitigen Punktestand des Spiels. Dieser kann sich im Verlaufe der Zeit ändern, sollte jedoch fix sein, sobald der *isDone* boolean auf True gesetzt wird. Dieses zeigt an, wenn ein Spiel vorbei ist. *EstimatedWinChance* wird beim Erzeugen des Spiels von der Anwendung berechnet. Es wird aus den aktuellen (also zuletzt eingetragenen) Ratings der Teams kalkuliert. Wenn beide Teams das gleiche durchschnittliche Rating haben, ist die *EstimatedWinChance* gleich null. Desto höher die Differenz vom Team1 gegenüber Team2 zu Gunsten vom Team1 ist, desto höher ist die EstimatedWinChance. Aus dieser Chance kann man sich dann bei Abschluss des Spiels den Punkte-Gewinn bzw. Verlust der zwei Teams errechnen.

Berechnet wird die Winchance mit der Formel

$$E_A = \frac{1}{1 + 10^{(R_B - R_A)/400}}$$

berechnet, wobei Ra und Rb das Durchschnittsrating von Team 1 bzw. Team 2 sind.

Die Punkte, die die Spieler des Team 1 dazu (oder abgezogen) bekommen, errechnet man mit der Formel

$$delta\_points_A = k\text{-}rating \times (Won_A - E_A)),$$

wobei k-rating ein belieber konstanter Faktor (meist ca. 15 – 35) ist, und $Won_A = 0$, wenn Team 1 verloren hat, und 1, wenn sie gewonnen haben.

Dasselbe gilt für Team 2, wobei $E_B = 1 - E_A$ *ist.*

In der letzten Tabelle *ScoreParameter* können alle notwendigen Parameter zur Berechnung gespeichert werden. Diese könnte man auch in einer eigenen Konfigurationsdatei speichern, aber ich habe mich für diese Version entschieden, da Datenbankzugriffe geregelter, sicherer und weniger störanfällig als Filezugriffe sind. So kann ein beliebiger User nicht so einfach auf die Parameter zugreifen oder sie verändern.
Die Paramter werden dabei einfach als key-value Stringpaare gespeichert.

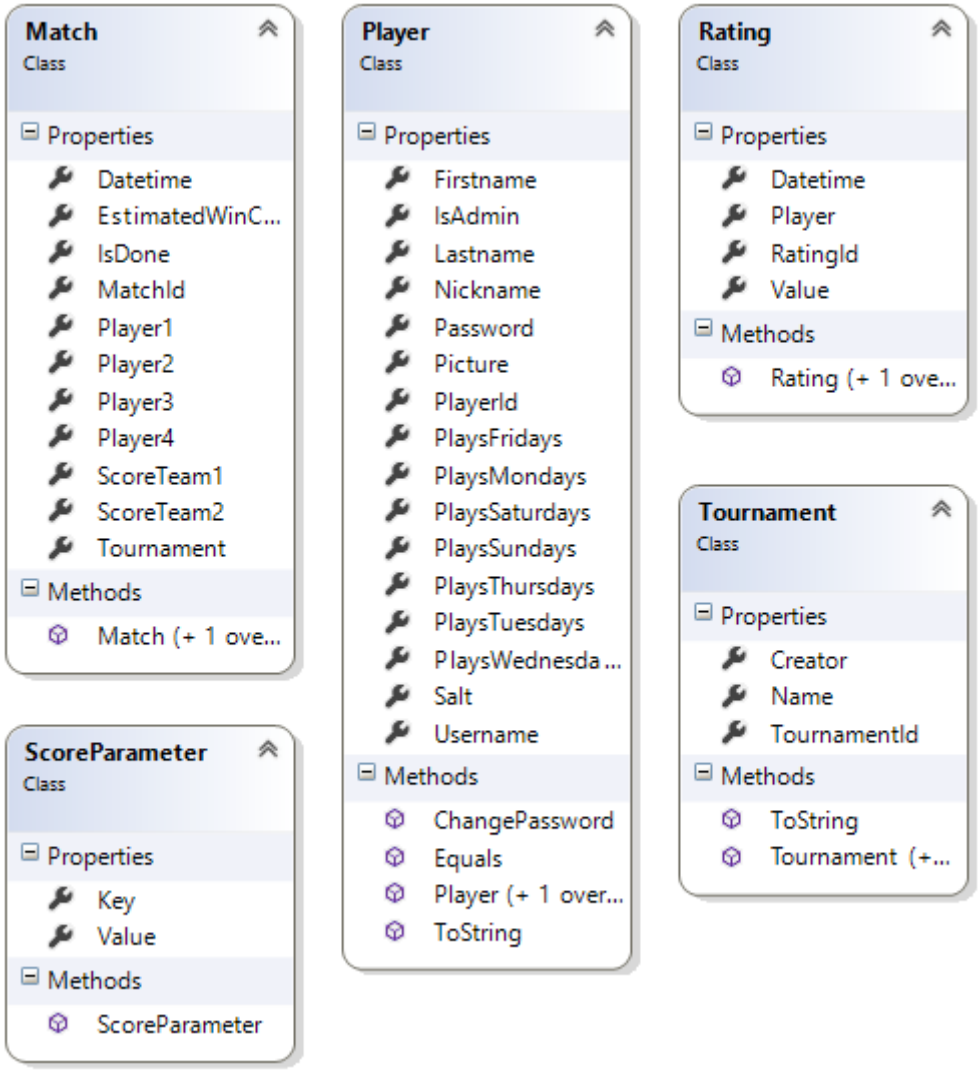
Mögliche Parameter sind:

- *initalscore*: Besagt, wieviele Punkte standardmäßig ein neuer Spieler haben soll.
- *scoredMatches*: Wieviele der letzten Matches zur Punkteberechnung hergenommen werden sollen. Zur Scoreberechnung werden dann nur die letzten n Matches gezählt.
- *halflife:* Damit frühere Spiele weniger in die Wertungsberechnung einfließen als weiter erst vor kurzem gespielte, kann man ein halflife festlegen. Spiele, die so lange zurückliegen, fließen nur mehr halb so stark in die Wertung ein usw.
- *k-rating*: Wird zur Berechnung der delta_points hergenommen. Ein höherer Wert bewirkt stärker fluktuierende Ratings.
- *timepenalty*: Dieser Wert bestimmt, wieviele Punkte einem Spieler (pro Monat/Woche/Tag) abgezogen werden, wenn ein Spieler länger nicht spielt. Seine Score kann jedoch nicht unter *initialscore* sinken.

Domainklassen

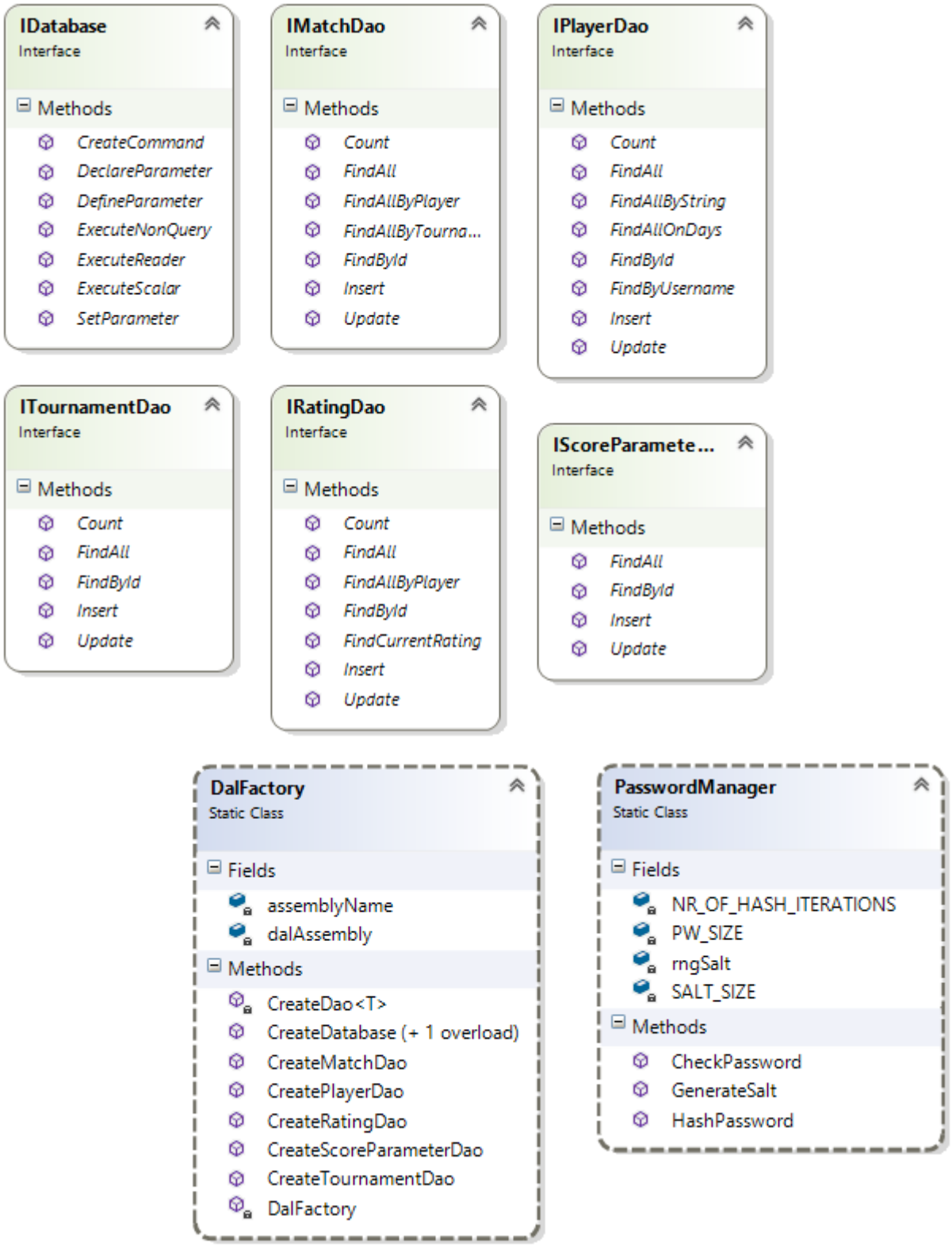
Die Domainklassen bilden die Datenbank-Entitäten im Code ab.

In der Player Klasse erstellt der Konstruktor bzw. die *ChangePassword* Methode automatisch ein Salt und hasht das gegebene Passwort mit diesem, bevor es beides als Property speichert.



Daos

Modell


IDatabase
Interface
Methods
CreateCommand
DeclareParameter
DefineParameter
ExecuteNonQuery
ExecuteReader
ExecuteScalar
SetParameter
IMatchDao
Interface
Methods
Count
FindAll
FindAllByPlayer
FindAllByTourna...
FindById
Insert
Update
IPlayerDao
Interface
Methods
Count
FindAll
FindAllByString
FindAllOnDays
FindById
FindByUsername
Insert
Update
ITournamentDao
Interface
Methods
Count
FindAll
FindById
Insert
Update
IRatingDao
Interface
Methods
Count
FindAll
FindAllByPlayer
FindById
FindCurrentRating
Insert
Update
IScoreParamete...
Interface
Methods
FindAll
FindById
Insert
Update
DalFactory
Static Class
Fields
assemblyName
dalAssembly
Methods
CreateDao<T>
CreateDatabase (+ 1 overload)
CreateMatchDao
CreatePlayerDao
CreateRatingDao
CreateScoreParameterDao
CreateTournamentDao
DalFactory
PasswordManager
Static Class
Fields
NR_OF_HASH_ITERATIONS
PW_SIZE
rngSalt
SALT_SIZE
Methods
CheckPassword
GenerateSalt
HashPassword

DAOs Beschreibung

Die Datenzugriffschicht ist ähnlich wie in der Übung als Interfacesammlung und dazugehörige Implementierungen für die konkrete Datenbank gegliedert. Wie man oben im Klassendiagramm sehen kann, gibt es für jede Entität ein zugehöriges DAO. Diese definieren alle Zugriffe auf die Datenbank über das IDatabase interface. Die konkreten DAOs und die Database selbst wird dann von der DalFactory zur Verfügung gestellt.

Die Methoden sind dabei für alle Klassen größtenteils die gleichen: *Insert*, *Update*, *FindAll*, *FindById* und *Count* werden von fast allen implementiert und sind selbstredend.
Dazu kommt:

Rating implementiert *FindAllByPlayer*, um den Verlauf der Wertung eines bestimmten Players anzeigen zu können, sowie *FindCurrentRating*, um das aktuellste Rating eines Spielers abzufragen.

Match implementiert zwei Methoden, um alle Matches eines Spielers oder eines Turnieres abfragen zu können.

Player implementiert zudem Methoden zur Suche eines Spielers entweder nach seinem Namen (hier wird im Vor-, Nach- und Spitznamen gesucht) sowie eine Methode zur Suche nach seinem Usernamen (für den Login). Außerdem kann man alle Spieler, die an einem (oder mehreren) gewissen Tag(en) spielen, suchen lassen.

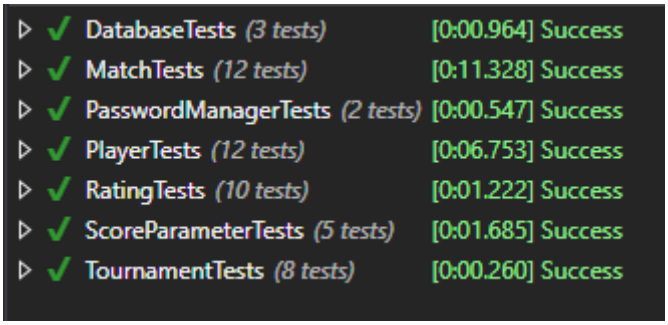
Zuletzt gibt es eine statische Klasse PasswortManager. Diese enthält Funktionen zum Generieren eines kryptografisch sicheren zufälligen „Salz“ mit der Klasse `RNGCryptoServiceProvider`, sowie einen ebenso sicheren Password Hasher (`Rfc2898DeriveBytes`) zum Hashen bzw. Überprüfen eines Passwortes.

Tests

Für die Tests werden normale Microsoft Unit Tests verwendet. Diese decken in 52 einzelnen Tests den kompletten DAO und Domainklassen Code ab. Die Tests sind sinnvoll auf mehrere Testklassen aufgeteilt und testen den Datenbankzugriff (Einfügen, Updaten und Abfragen), die Erzeugung der Domainenobjekte, und die Funktionen des Passwortmanagers.

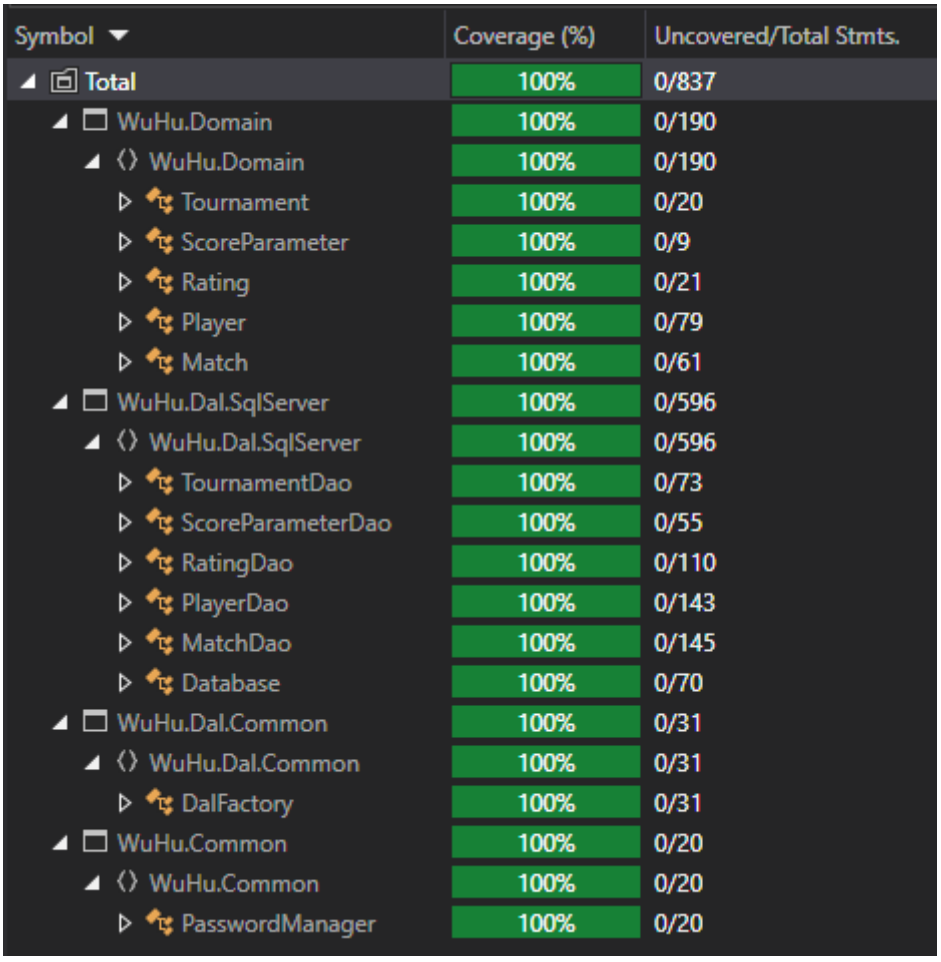
Ausführung

Alle 52 Tests können erfolgreich ausgeführt werden.



Code Coverage

Die oben angeführten Tests decken den gesamten relevanten Code ab.



Symbol	Coverage (%)	Uncovered/Total Stmts.
Total	100%	0/837
WuHu.Domain	100%	0/190
WuHu.Domain	100%	0/190
Tournament	100%	0/20
ScoreParameter	100%	0/9
Rating	100%	0/21
Player	100%	0/79
Match	100%	0/61
WuHu.Dal.SqlServer	100%	0/596
WuHu.Dal.SqlServer	100%	0/596
TournamentDao	100%	0/73
ScoreParameterDao	100%	0/55
RatingDao	100%	0/110
PlayerDao	100%	0/143
MatchDao	100%	0/145
Database	100%	0/70
WuHu.Dal.Common	100%	0/31
WuHu.Dal.Common	100%	0/31
DalFactory	100%	0/31
WuHu.Common	100%	0/20
WuHu.Common	100%	0/20
PasswordManager	100%	0/20

Testdaten

Die Testdaten (2 Jahre normale Nutzung lt. Angabe) werden mit dem Pythonscript „data_generator.py“ generiert. Diese erstellt ein sql script mit Inserts, die direkt ausgeführt werden können. Die statische Klasse *TestHelper* enthält außerdem die Methode *InsertTestData*, die genau auf dieses Sql-Skript zugreift und ausführt. Dieses kann einige Zeit dauern.

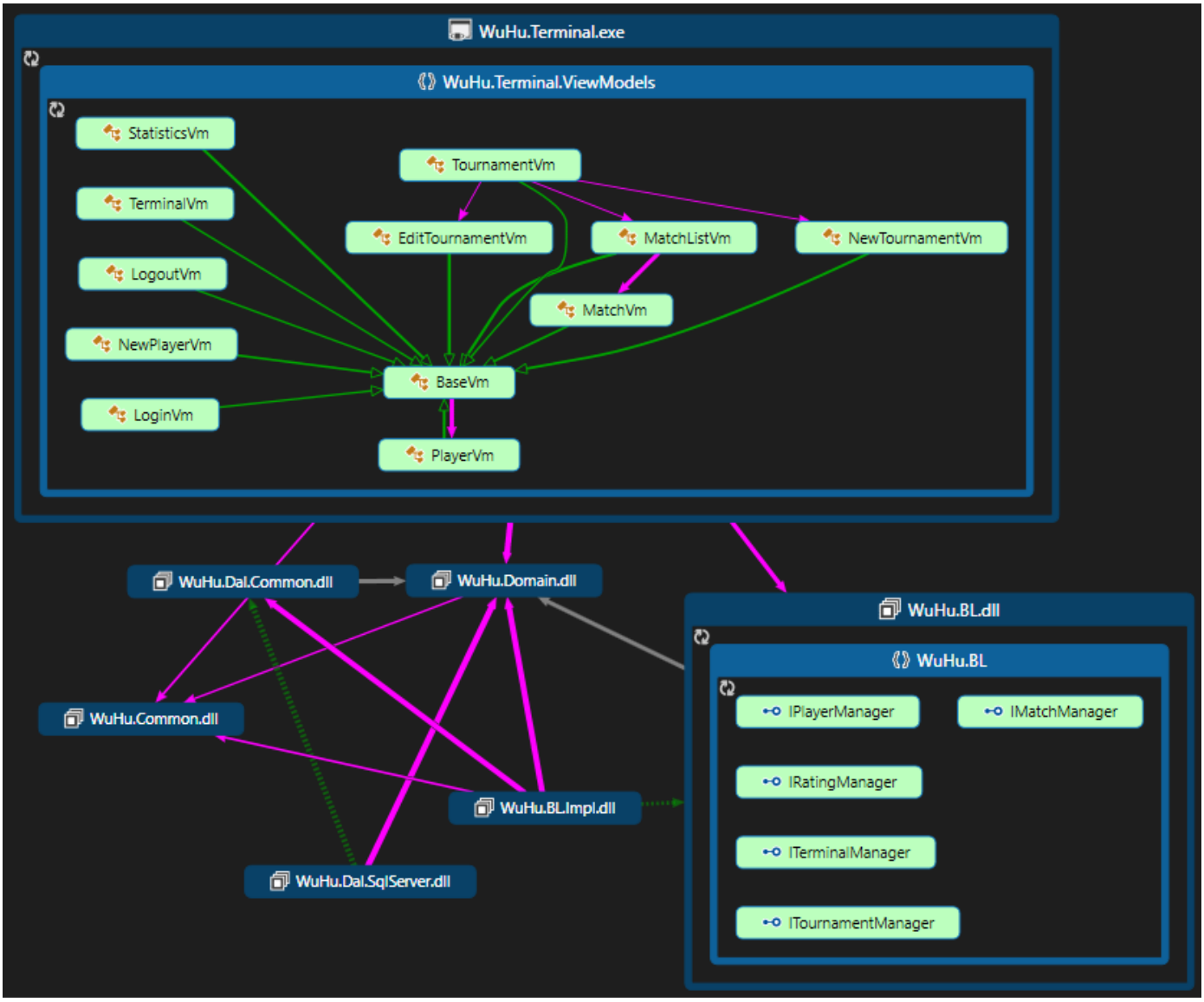
Program.cs im Startupprojekt WuHu.Server ist bereits so konfiguriert, dass es die tables erstellt und die Testdaten einfügt. Dies setzt voraus, dass die App.config Dateien richtig eingestellt sind.

Beim Ausführen der Unit-Tests wird automatisch ein Backup der Datenbank unter dem Namen „WuHuDB.mdf.bak“ bzw. „WuHuDB_log.ldf.bak“ erstellt, falls es noch nicht existiert. Falls das Backup bereits existiert, wird vor Ausführung der Tests die derzeitige Datenbank mit dem Backup überschrieben. So wird eine konsistente Testumgebung gewährleistet.

Ausbaustufe 2

In dieser Ausbaustufe wurde die gesamte Businesslogik sowie das Nutzerinterface als WPF-Applikation implementiert. Aus folgender Codemap kann man sehr gut erkennen, wie die einzelnen Komponenten voneinander abhängen. Die grünen (durchgehenden sowie gestrichelten) Pfeile bezeichnen dabei Ableitungen, die rosanen zeigen Verwendungen. Zum Beispiel leitet TournamentVm von BaseVm ab, aber verwendet MatchListVm, EditTournamentVm und NewTournamentVm in seiner Implementierung. MatchListVm wiederum verwendet und verwaltet MatchVms.

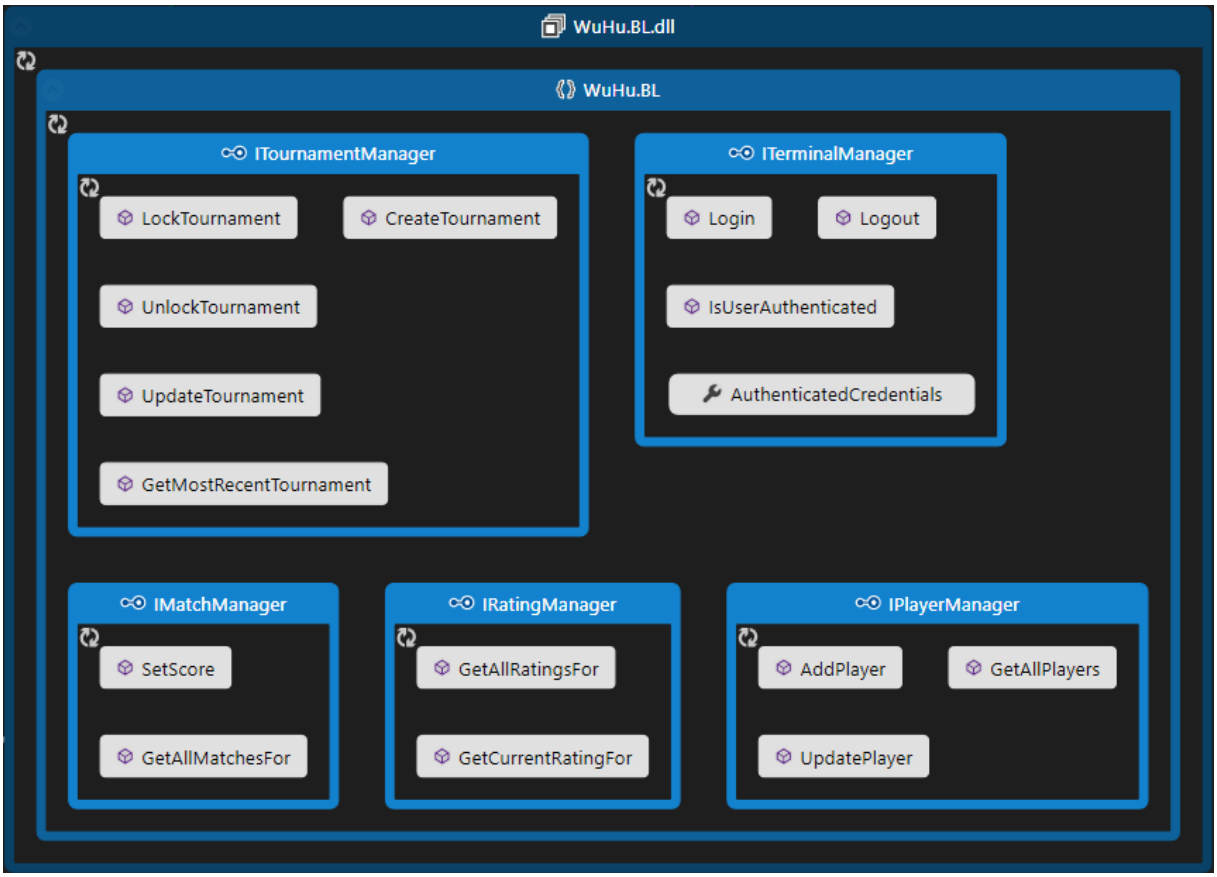
Gesamt Architektur



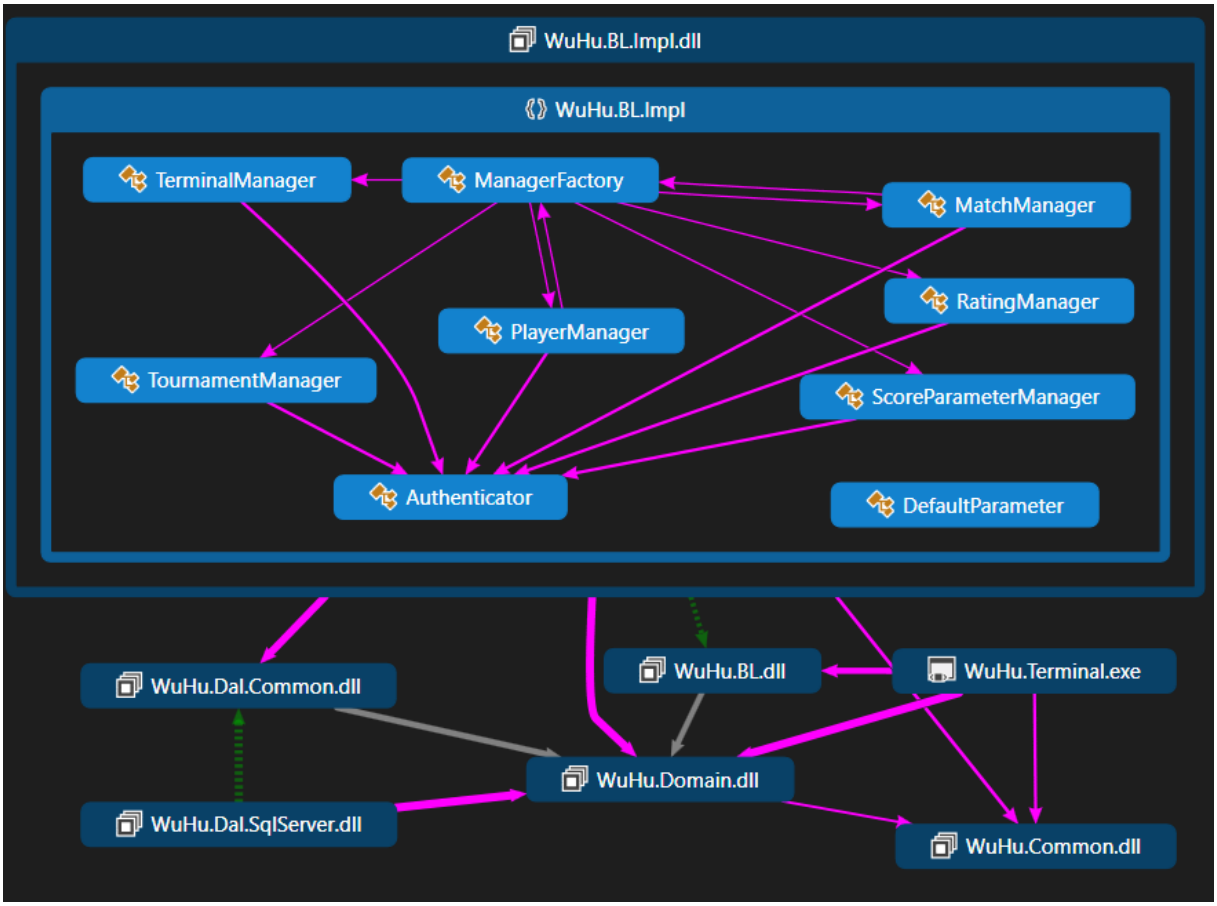
Auf dieser Map erkennt man, dass die Domainklassen eine zentrale Rolle spielen. Sie werden von allen Komponenten verwendet.

Weiterhin sieht man, dass das Terminal-Projekt (also die WPF-Applikation) in keinem Fall direkt auf die DAL zugreift. Sie verwendet ausschließlich die Business Logic, die Domainenklasse, sowie in einem Fall einen CryptoService im „Common“ Projekt. Nur die Business Logik selbst greift auf die Datenbank zu. Dies geschieht auch nur indirekt: Die WPF App verwendet nur die BL Interfaces, und die Implementierung (WuHu.BL.Impl) greift dann auf die Interfaces in der WuHu.Dal.Common zu. Diese wiederum werden von WuHu.Dal.SqlServer implementiert.

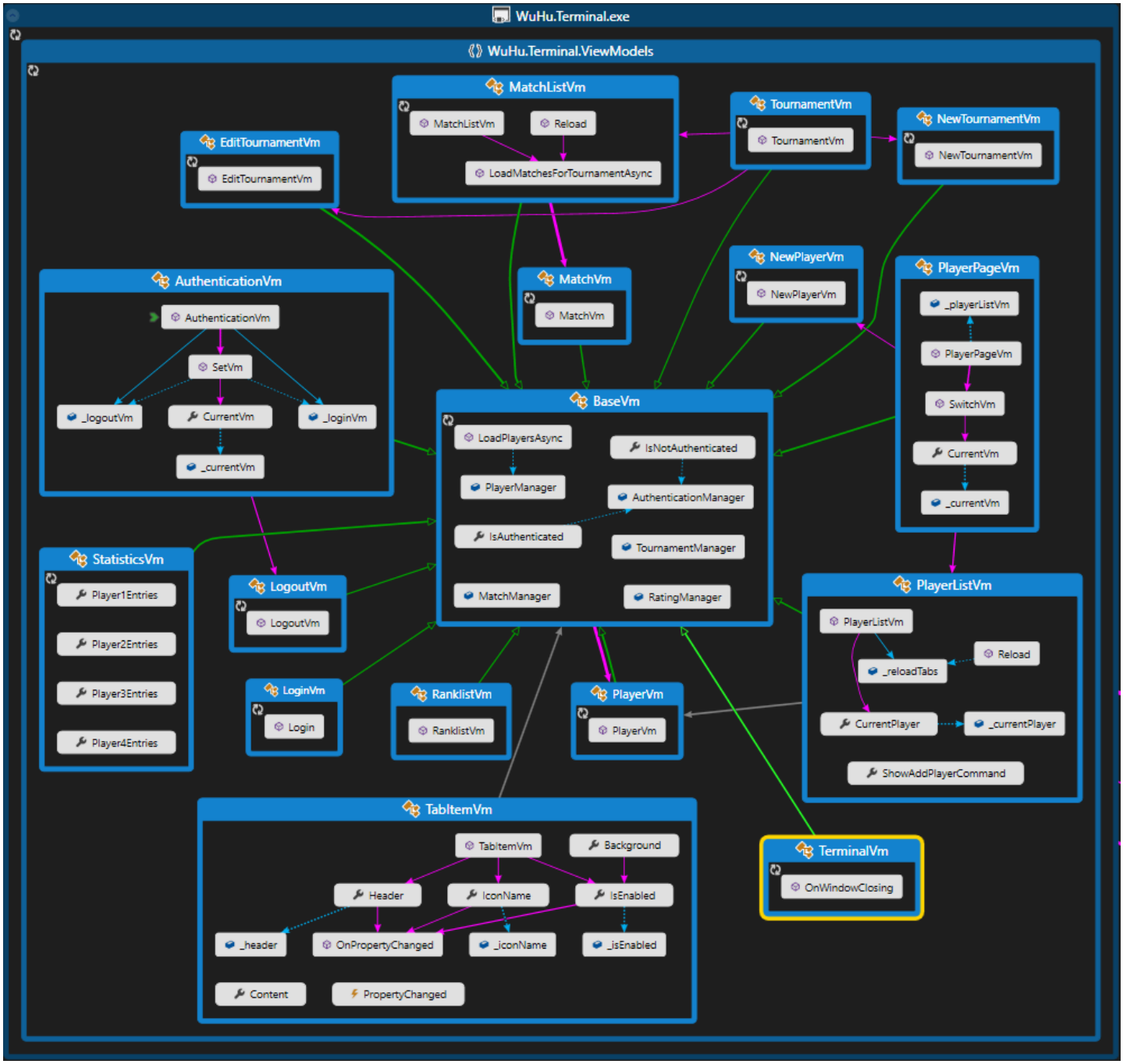
Business Logik



Hier sieht man die gesamte Funktionalität der Business Logik anhand der Interfaces. Sie decken die Funktion der DAL ab und sind ausreichend, um die Funktion der gesamten Applikation abdecken zu können.



Hier sieht man Details zur BL Implementation. Die Manager sind in logische Gruppen aufgeteilt. Sie werden von der ManagerFactory verwaltet und können so (als Interfaces) von der WPF App verwendet werden.

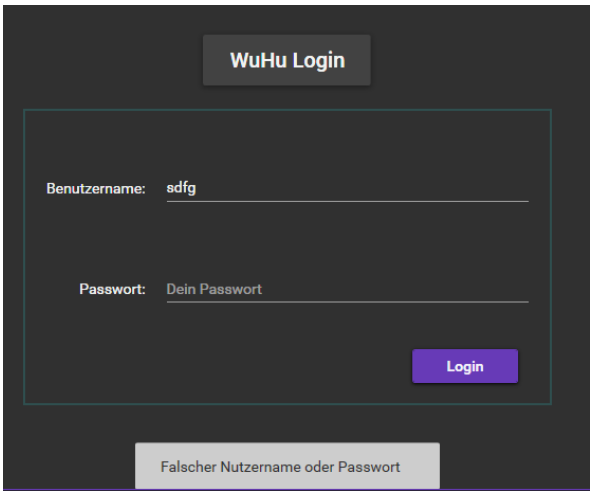


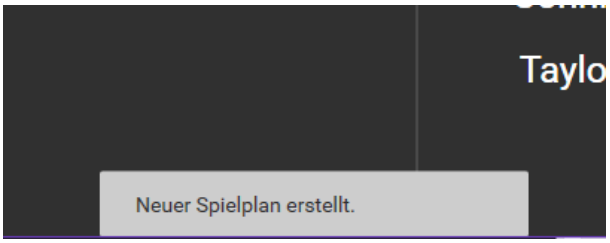
Hier sieht man den Aufbau der Implementierung des WPF Klienten im Detail. Es gibt ein Base-Viewmodel, von dem alle anderen ableiten. In diesem werden die Manager erstellt und verwaltet, sowie eine Methode zum Laden der aktuellen Spielerliste, die in einigen Klassen verwendet wird implementiert.

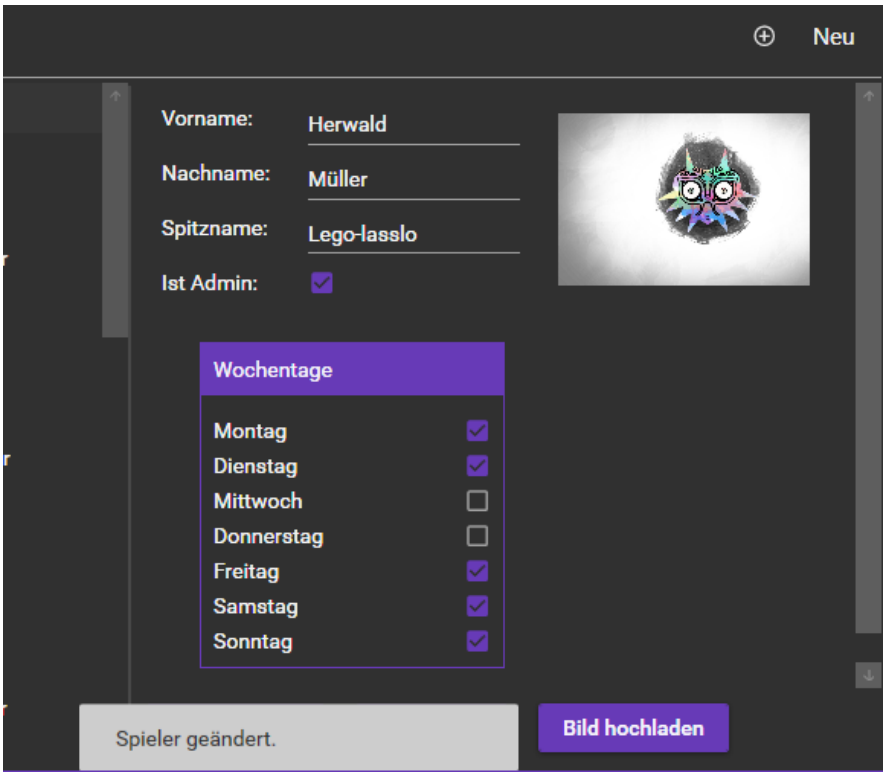
MessageQueue

Meine Implementierung beinhaltet ein relativ kompliziertes Messaging System. Nachrichten werden über Delegates von der auslösenden View über Delegates Stück für Stück nach oben propagiert und am Ende vom Fester selbst als „Snackbar“ angezeigt. Diese benachrichtigen den User über wichtige Ereignisse, zum Beispiel ob der Login fehlgeschlagen ist oder nicht, ob ein Spieler oder ein Spielplan angelegt worden konnte oder nicht, und, falls dem so ist, dass ein Turnier gerade in Bearbeitung ist.

Beispiele:





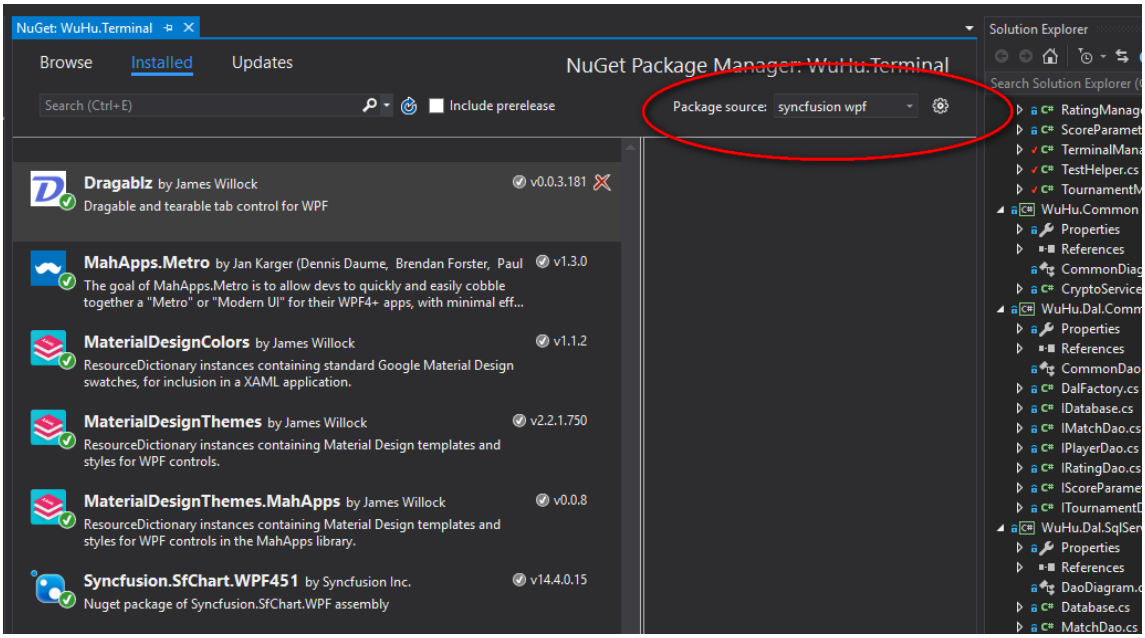


Externe Ressourcen

Ich habe in dieser Ausbaustufen ein paar Packages verwendet. Folgende können ganz einfach von Nuget geladen werden:

- MahApps.Metro
- Dragablz
- MaterialDesignColors
- MaterialDesignThemes
- MaterialDesignThemes.MahApps

Folgendes Package kann nur über eine alternative „Package source“ geladen werden. Dazu muss man sie zunächst im Nuget Package Manager rechts oben umstellen.



Danach kann über den „Browse“-Tab folgendes Package geladen und installiert werden:

- Syncfusion.Sfchart.Wpf451