1 Ultimate Festival Organizer (UFO)

1.1 Datenmodell

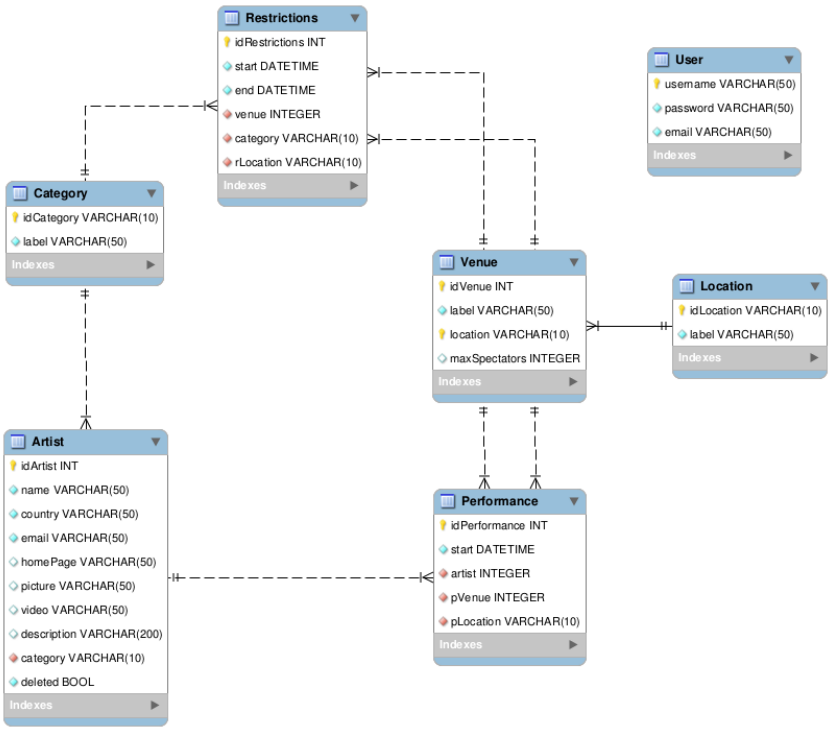

Abbildung 1: OR-Diagramm

Das Datenmodell enthält zusätzlich zu den vorgegebenen Entitäten zwei weitere Entitäten: *Location*-Entität und *Restrictions*-Entität. Die *Location*-Entität enthält als Primärschlüssel eine kurze Bezeichnung einer bestimmter Ort und der vollständige Name des Ortes, z.B. H - Hauptplatz, L - Landstraße. Der Primärschlüssel der *Location*-Entität wird als Fremdschlüssel und gleichzeitig Primärschlüssel in der Spielstätte-Entität verwendet. Die Spielstätte-Entität besitzt einen zusammengesetzten Primärschlüssel aus zwei Attributen: *idVenue* und der Fremdschlüssel *location*. Das Attribut *idVenue* wird nicht automatisch inkrementiert, sondern wird von Programm bestimmt, indem man der Anzahl an aktuell existierende Spielstätte für ein bestimmtes Ort um eins inkrementiert und als nächstes *idVenue* speichert. Dadurch schafft man, dass die *idVenue* für ein bestimmtes Ort, z.B. Hauptplatz, nicht größer wird als der Tatsächlichen Anzahl an Spielstätten für diesen Ort.

Die Entität *Restrictions* kann benutzt werden, um bestimmte Darbietungskategorien einschränken zu können. Zum Beispiel kann eine mögliche Kategorie für Kinder so eingeschränkt, dass die Aufführungen für Kinder nur während einer bestimmten Zeitspanne vorgetragen werden können. Auch Feueraufführungen sollen nur zwischen 21 und 23 Uhr stattfinden, da zu diesem Zeitpunkt das Tageslicht nicht mehr so stakt ist.

Die Entität *Artist* Besitz ein Attribut der Typ Bool *deleted*, der gesetzt wird, falls ein Künstler gelöscht werden soll. Damit wird der Künstler als gelöscht markiert, wird aber

nicht aus der Datenbank gelöscht.

1.2 Datenzugriffsschicht

Um eine möglich gute Abstraktion zu erreichen wurde eine generische *IDao*-Interface implementiert. Jede *Dao* muss diese Interface implementieren.

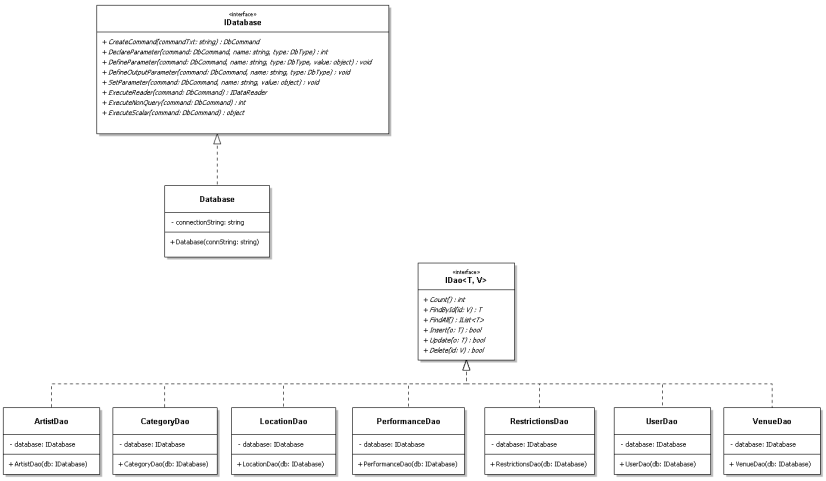

Abbildung 2: Data Access

Um die Daten besser zwischen unterschiedlichen Schichten zu transportieren, wurde für jede Entität eine eigene *Domain*-Klasse implementiert.

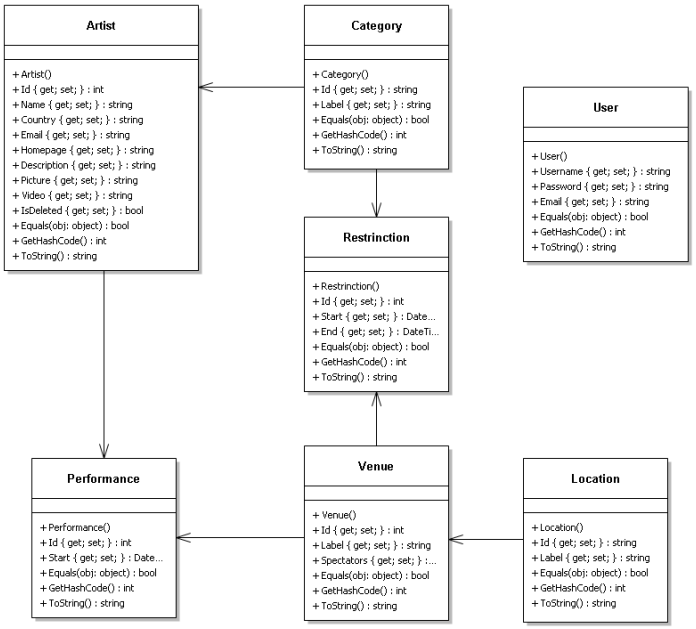

Abbildung 3: Domain Classes

1.3 Unit-Tests

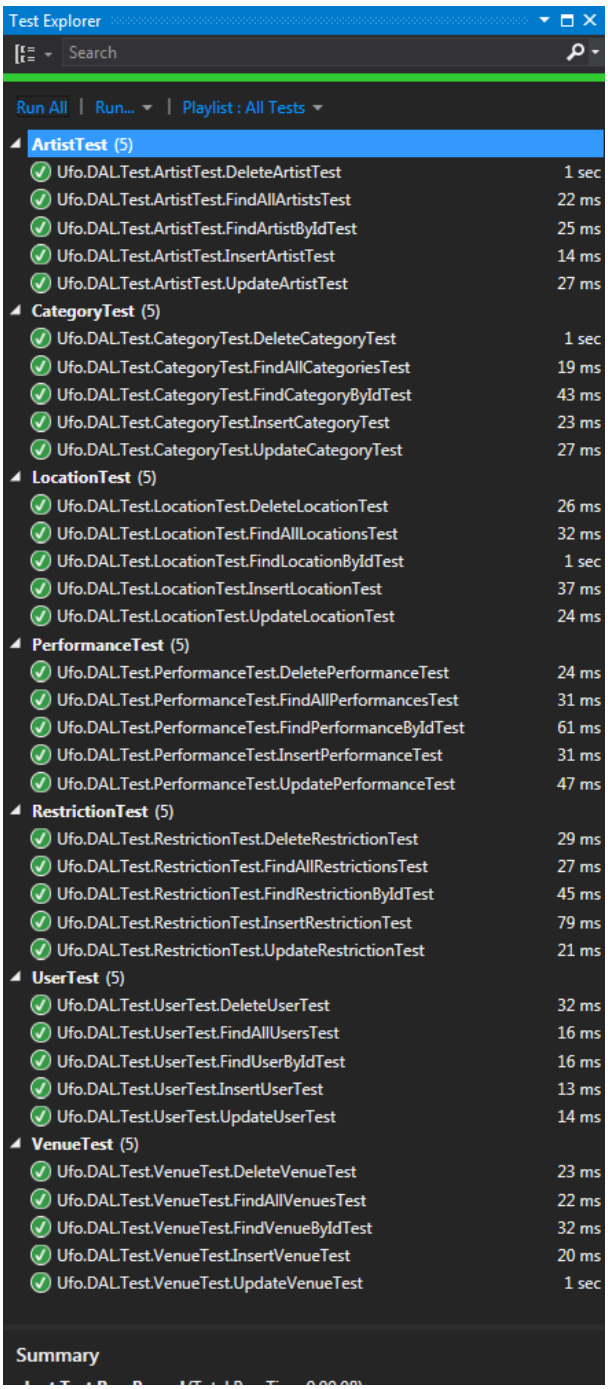

Abbildung 4: Ergebnis Unit-Tests

Für jede implementierte DAO-Methode wurde jeweils ein Unit-Test implementiert. Dafür wurde eine eigene Testdatenbank erstellt. Als Testumgebung wurde XUnit-Framework verwendet. Diesen *Framework* bietet ein *AutoRollback*-Attribut, der von Entwickler implementiert werden muss. Diesen Attribut verwendet Datenbanktransaktionen um die Unit-Test Daten nicht in die Datenbank zu speichern. Damit gibt es keine Abhängigkeit zwischen einzelnen Unit-Tests und jeden Unit-Test ist atomar.

```
using System;
using System.Reflection;
using System.Transactions;
using Xunit.Sdk;

namespace Ufo.DAL.Test
{
    [AttributeUsage(AttributeTargets.Class | AttributeTargets.Method, AllowMultiple = false, Inherited =
        true)]
    class AutoRollbackAttribute : BeforeAfterTestAttribute
    {
        IsolationLevel isolationLevel = IsolationLevel.Unspecified;
        TransactionScope scope;
        TransactionScopeOption scopeOption = TransactionScopeOption.Required;
        long timeoutInMS = -1;

        /// <summary>
        /// Gets or sets the isolation level of the transaction.
        /// Default value is <see cref="IsolationLevel"/>.Unspecified.
        /// </summary>
        public IsolationLevel IsolationLevel
        {
            get { return isolationLevel; }
            set { isolationLevel = value; }
        }

        /// <summary>
        /// Gets or sets the scope option for the transaction.
        /// Default value is <see cref="TransactionScopeOption"/>.Required.
        /// </summary>
        public TransactionScopeOption ScopeOption
        {
            get { return scopeOption; }
            set { scopeOption = value; }
        }

        /// <summary>
        /// Gets or sets the timeout of the transaction, in milliseconds.
        /// By default, the transaction will not timeout.
        /// </summary>
        public long TimeoutInMS
        {
            get { return timeoutInMS; }
            set { timeoutInMS = value; }
        }

        /// <summary>
        /// Rolls back the transaction.
        /// </summary>
        public override void After(MethodInfo methodUnderTest)
        {
            scope.Dispose();
        }

        /// <summary>
        /// Creates the transaction.
        /// </summary>
        public override void Before(MethodInfo methodUnderTest)
        {
            TransactionOptions options = new TransactionOptions();
            options.IsolationLevel = isolationLevel;
            if (timeoutInMS > 0)
                options.Timeout = new TimeSpan(timeoutInMS * 10);
            scope = new TransactionScope(scopeOption, options);
        }
    }
}
```