1 UFO (Ultimate - Festival - Organizer)

Folgende Dokumentation stellt die Gesamtdokumentation der aufbauenden Übung UFO dar, die im Zuge der Realisierung iterativ über die drei Ausbaustufen hinweg erweitert wird.

1.1 Ausbaustufe 1 (ADO.NET)

Folgender Teil dokumentiert die erste Ausbaustufe der aufbauenden Übung UFO. In diesem Teil wird die Persistenz Schicht in .NET unter Hilfenahme von ADO.NET implementiert. Aufgrund der Analyse der Gesamtaufgabenstellung wurde entschieden das vorerst nur die Persistenz Schicht an sich, also INSERT, UPDATE, DELETE der einzelnen Entitäten realisiert wird, da die Geschäftslogik erst bei der Realisierung der Administration und des Webzugriffs endgültig feststehen wird.

Im Zuge der Realisierung des Webzugriffs wird auch der Web-Service implementiert werden müssen, der die Daten der Web Applikation zur Verfügung stellt. Dieser soll die Daten bereits gefiltert und strukturiert zur Verfügung stellen, daher besteht eine gewisse Abhängigkeit zwischen dem Web-Service und der Web Applikation sowie auch der Client Administration.

Daher werden die Web-Service Implementierung und die Administration die eigentliche Geschäftslogik enthalten, die in einer Transaktion abgearbeitet und im wesentlichen aus den logischen Prüfungen gegen die Datenbank bestehen wird sowie der Speicherung und Löschung von Entitäten über die Administration. Die einzelnen Datenabfragen, die benötigt werden können einfach hinzugefügt werden.

1.1.1 Systemaufbau

Folgend ist der Systemaufbau der Persistenz Schicht dokumentiert.

Die folgende Auflistung illustriert die Projektstruktur der Persistenz Schicht:

1. *UFO.Server.Data.Api*
   Dieses Projekt enthält die Spezifikation der Persistenz Schicht in Form von Interfaces, abstrakten Klassen, Exceptions und den Entitäten, die wie bei einem ORM-Mapper DB unabhängig sein sollen (sofern möglich).

2. *UFO.Server.Data.MySql*
   Dieses Projekt enthält die MySql spezifische Implementierung der Persistenz Schicht.

3. *UFO.Server.Test.Data.MySql*
   Dieses Projekt enthält die MySql spezifischen Tests der Persistenz Schicht.

4. *UFO.Common.Util*
   Dieses Projekt enthält die Utlities für die UFO Infrastruktur in .NET, die nicht spezifisch einen Systemteil zuzuordnen sind.

5. *ufo-data-generator*
   Ein kleines Java Projekt welches eine Java Main Klasse enthält und die benötigen Ressourcen um das Testdatenskript zu erstellen. Hierzu wurde *freemarker* verwendet.

Alle .NET Systemteile werden unter dem Namensraum *UFO.** zusammengefasst.

Die folgende Auflistung illustriert die verwendeten Technologien und Frameworks:

1. *MySql*
   Es wird eine MySql Datenbank verwendet, die Open-Source ist und eine Integration in .NET besitzt.

2. *NUNIT*
   Als Test-Framework wird NUNIT verwendet, da es mehr Funktionalität mitbringt als das Standard Test-Framework integriert in .NET.

3. *ADO.NET*
   Als Persistenz Provider wird wie gewünnscht ADO.NET und kein ORM Mapper verwendet.

4. *freemarker*
   Template-Engine in Java mit der das Testdaten SQL Skript erstellt wird.

1.1.2 Datenbank

Es wurde als Datenbank MySql und Moderlierungstool MySql Workbench gewählt, da diese Datenbank erstens Open-Source, zweitens eine gute .NET Integration vorhanden ist sowie drittens bereits Erfahrungen mit dieser Datenbank vorhanden waren.

Es wurden folgende Skripten generiert die einerseits für die Tests und andererseits für die Generierung der Testdaten genutzt werden. Diese Skripten befinden sich im Projekt *UFO.Server.Data.MySql/Resources*.

1. *createDatabase.sql*
   Ein vollständiges Skript für das Anlegen der Datenbank mit allen Constraints und verwendeten Triggern.
2. *deleteDatabase.sql*
   Ein Skript welches alle Einträge in der Datenbank löscht.
3. *dropDatabase.sql*
   Ein Skript für das Droppen der Datenbank UFO.
4. *createTestData.sql*
   Ein Skript welches die Testdaten generiert.

Die Testdaten wurden mit einer Java Applikation mit Hilfe von *Freemarker* erstellt, welches eine Template Engine darstellt. Die Daten werden von **.csv* Dateien zur Verfügung gestellt und anschließend über eine Java Konsolen Applikation verarbeitet. Diese Applikation liest die Daten ein, verpackt diese in Pojos, generiert dynamische Daten, wie z.B.: die Aufführungen mit den Aufführungszeiten und stellt diese Daten einem Template zur Verfügung.

Das folgende ER-Diagramm illustriert das implementierte Datenbank Schema.

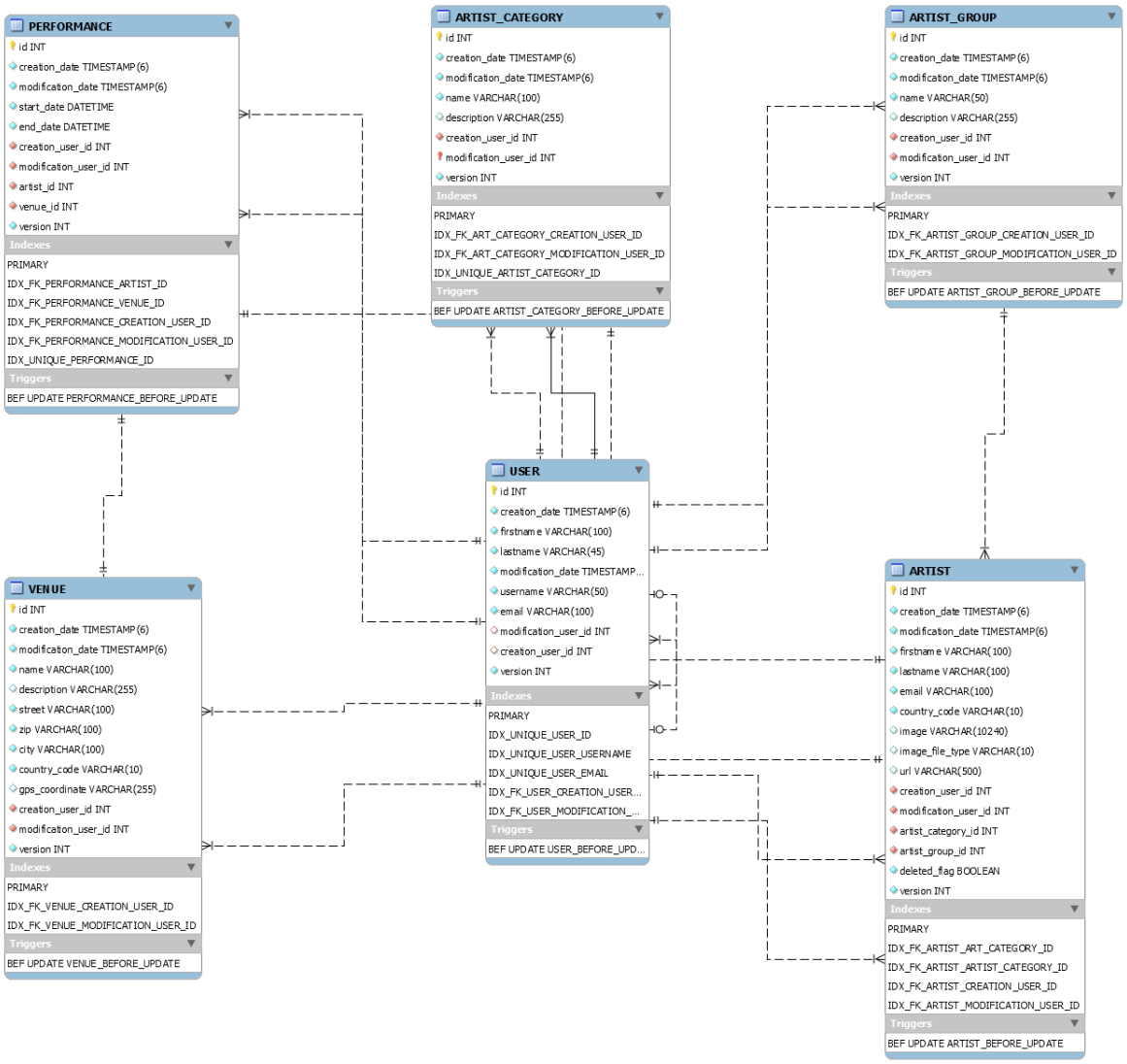

Abbildung 1: ER-Diagramm des Schema 'UFO'

Auf jeder Entität wurde eine Spalte für die Versionierung eingeführt (MySqlDbType.LONG), die über einen Update-Trigger bei jedem Update um eins erhöht wird sowie auch das Modifizierungsdatum. Ein ganzzahliger Datentyp erschient hier mehr sinnvoll, da es hier mit Sicherheit keine Kollisionen geben kann, nicht so wie bei einem Zeit Datentyp.

Ebenso halten alle Entitäten eine Referenz auf den Benutzer der Sie erstellt sowie zuletzt modifiziert hat. Dies dient der Nachverfolgbarkeit von Änderungen, zumindest wer zuletzt eine Änderung vorgenommen hat.

1.1.3 Klassenhierarchien

Folgend sind die Klassenhierarchien der implementierten Klassen und Interfaces dokumentiert.

IDao

Folgendes Klassendiagramm zeigt die Hierarchie des Interfaces *IDao*, das der Basistyp für alle implementierten DAO Interfaces dient, da es bereits alle Basisaktionen auf eine Entität definiert.

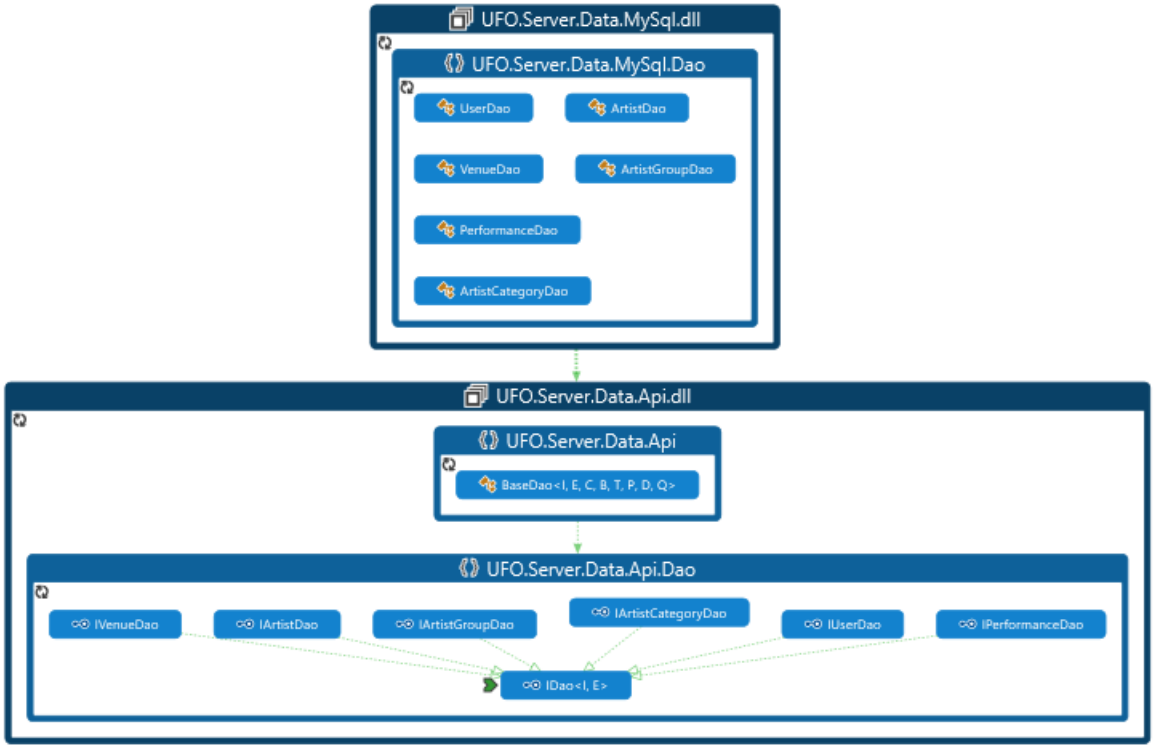

Abbildung 2: Klassenhierarchie IDao Interface

Die Basisklasse *BaseDao* implementiert alle Methoden, die in *IDao* definiert wurden für alle implementieren Entitäten sofern sie *IEntity* implementieren. Dieser generische und abstrakte Ansatz erlaubt es dass die Basisfunktionalität eines DAOs nur einmal für alle Entitätstypen, die *IEntity* implementieren, implementiert werden musste.

Die *DAOs* für die einzelnen Entitäten werden zukünftig Methoden implementieren, die spezifische Datenbankabfragen realisieren, die z.B.: eine komplexe *where caluse* aufweisen, die ihrerseits wieder einen Teil der Geschäftslogik enthält, welche noch nicht vollständig bekannt ist.

IEntity

Folgendes Klassendiagramm illustriert die Klassenhierarchie des Interfaces *IEntity*, welches den Basistyp für alle Entitäten darstellt.

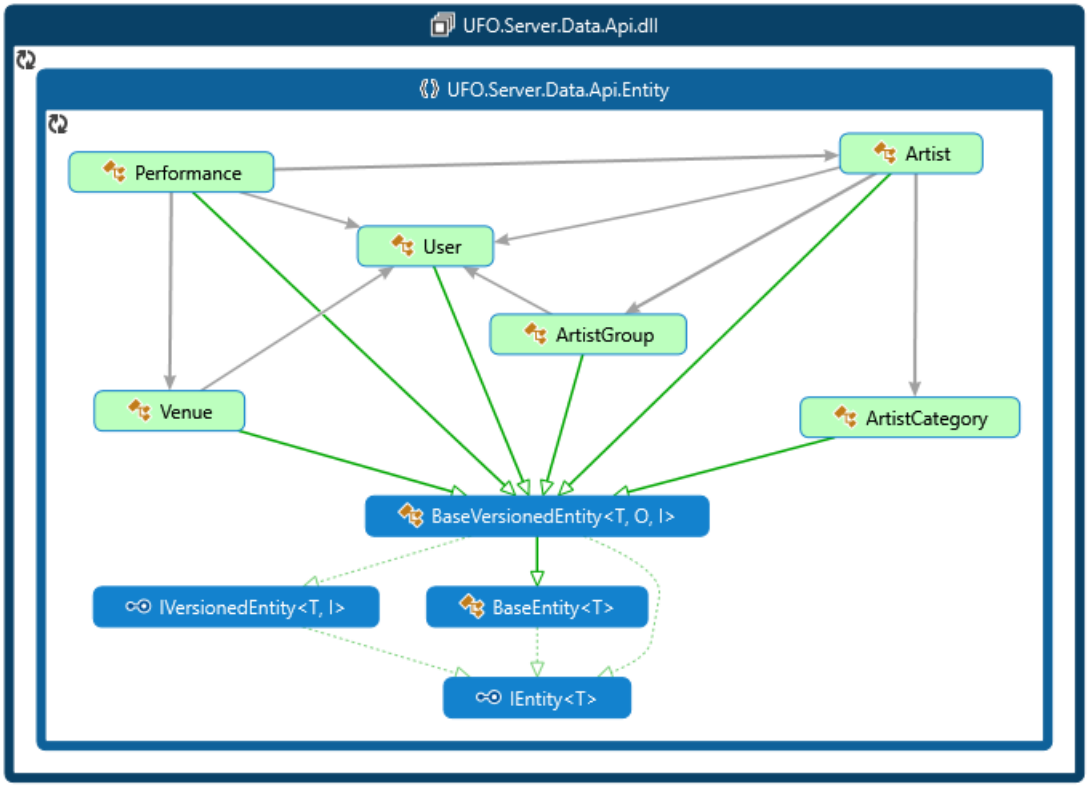

Abbildung 3: Klassenhierarchie IEntity Interface

Es wurden Basisentitäten eingeführt, welche eine Basisstruktur definieren dies sich Entitäten unterwerfen müssen wenn sie von diesen Klassen ableiten. Somit wird eine konsistente Struktur der Entitäten bzw. deren Tabellenrepräsentation gewährleistet.

Die Aufteilung von *IEntity* und *IVersionedEntity* wurde eingeführt, da eine Entität nicht zwangsweise versionierbar sein muss. Ebenso wurde mit den abstrakten Klassen verfahren, die jetzt eine Hierarchie abbilden anstatt die gesamte Funktionalität in eine abstrakte Klasse zu packen.

IEntityHelper

Folgendes Klassendiagramm illustriert die Klassenhierarchie des Interfaces *IEntityHelper* welches die Utility Methoden für das generieren von Entitäten definiert.

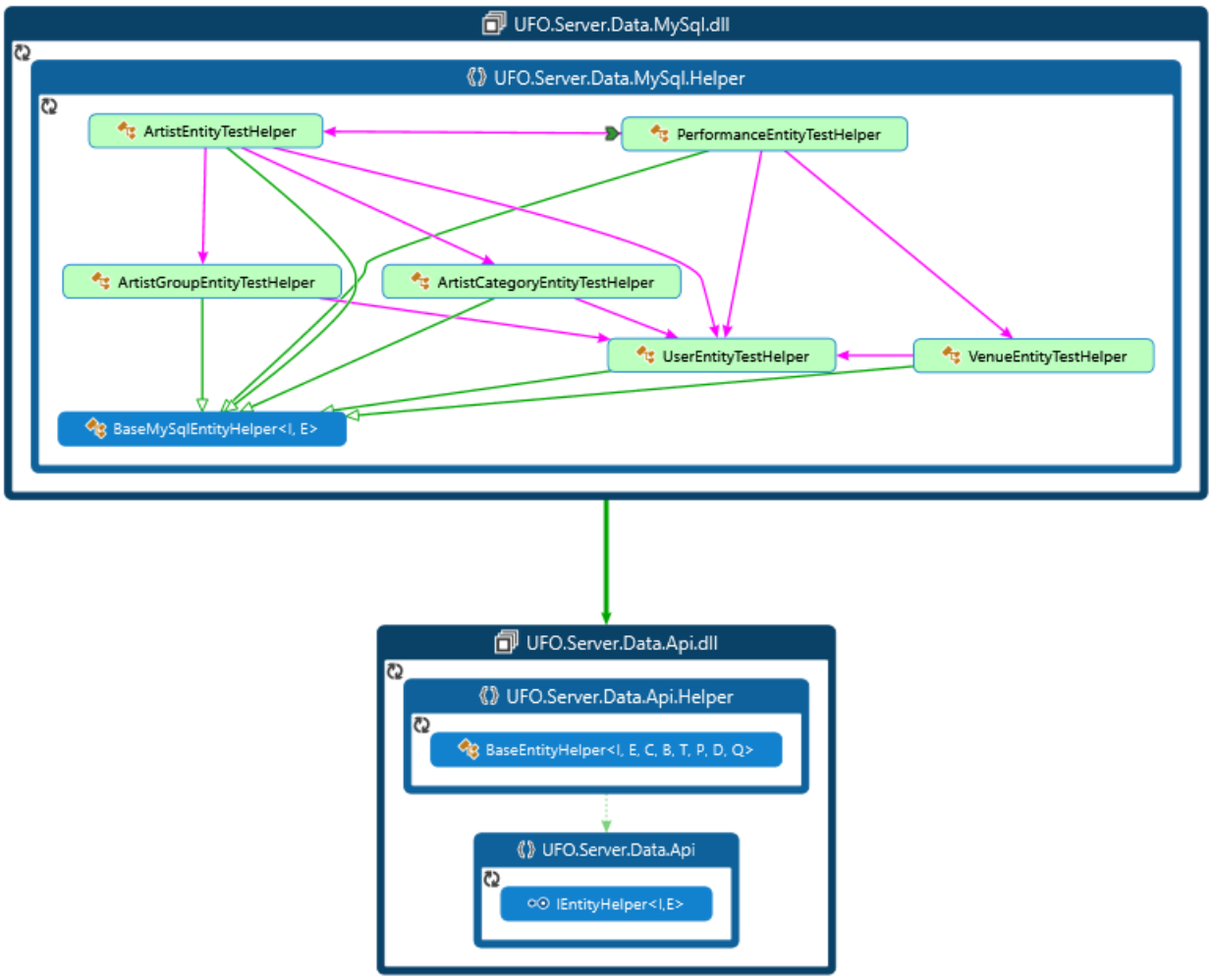

Abbildung 4: Klassenhierarchie IEntityHelper Interface

Dieses Interface und dessen Implementierungen diesen als Hilfestellung für die Test und die Generierung von Testdaten über die Entitäten Modelle. Die abstrakte Basisklasse *BaseEntityHelper* implementiert einige der Interface Methoden und stellt eine Persistenzprovider unabhängige Implementierung für das einfache Speichern von Entitäten zur Verfügung. Diese Hilfsklassen entstanden aufgrund der generischen Testklasse *BaseDaoTest*, die die Entitäten nicht erzeugen kann und daher diese von außen zur Verfügung gestellt werden müssen.

BaseCommandBuilder

Folgendes Klassendiagramm illustriert die abstrakte Klasse *BaseCommandBuilder* die das Handling mit einen *DbCommand* beinhaltet.

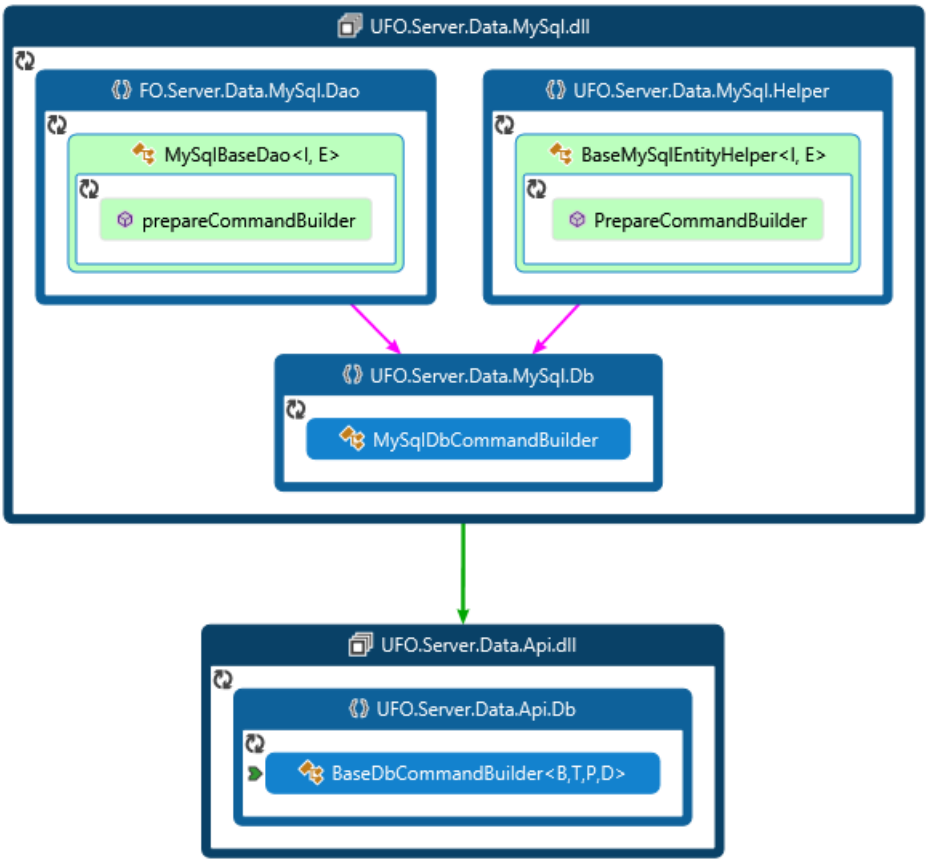

Abbildung 5: Klassenhierarchie der abstrakte Klasse BaseCommandBuilder

Um zu vermeiden sich mit dem Code des Erstellens, Modifizierens und Verwerfens eines *DbCommand*, in unserem Fall ein *MySqlDbCommand*, herumschlagen zu müssen, wurde beschlossen eine Hilfsklasse einzuführen, die uns dieses Handling mit einem *DbCommand* abnimmt. Da sich hier eine Fluent-API gut anwenden lässt, wurde diese Funktionalität in Form eines Builder abgebildet.

IQueryCreator

Folgendes Klassendiagramm illustriert das Interface *IQueryCreator*, die die Datenbank spezifischen Statements enthält.

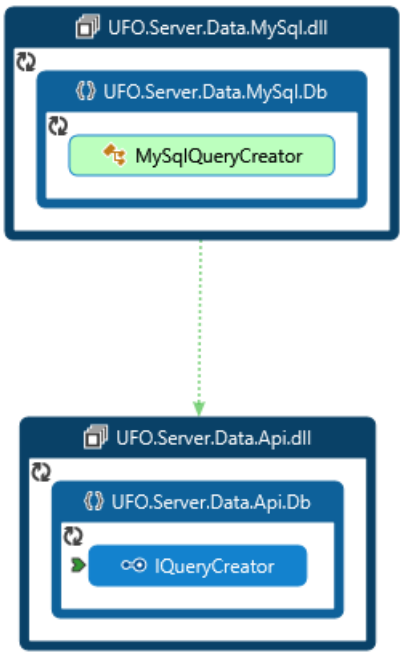

Abbildung 6: Klassenhierarchie des Interface *IQueryCreator*

Diese Interface abstrahiert die Datenbank spezifischen Statements von der Klient Logik. Ebenso erlaubt sie es alle Statements in einem Interface abzubilden und diese an einer Stelle pro Entität zu bündeln.

1.1.4 Hilfsklassen

Im folgenden werden die Hilfsklassen beschrieben, die eingeführt wurden um sich wiederholende und daher immer wiederkehrende Funktionalitäten zu kapseln und zentral zur Verfügung zu stellen.

EntityMetamodel
Diese Klasse löst die Meta-Informationen eines *IEntity* Typs auf und stellt diese aufbereitet nach außen zur Verfügung. Da sich diese Daten zur Laufzeit nicht ändern und daher nur einmalig erzeugt werden sollen, wird eine Factory *EntityMetamodelFactory* eingeführt, die das Caching der *EntityMetamodel* übernimmt.

EntityBuilder
Diese Klasse wurde eingeführt um die Transformation von den Entitäten zur Datenbank und visa versa zu unterstützen, wobei hier einerseits die Werte der Properties, die auf die Datenbank serialisierbar sind, ausgelesen und auf den Property gemapped werden und andererseits die De-Serialisierung vom *IDataReader* zu einer Entität.

IDbTypeResolver
Implementierungen dieses Interface lösen einen C# Typ in einen korrespondierenden Datenbank spezifischen Typ auf.

DbConnectionFactory

Diese Klasse erstellt und verwaltet die verwendeten *DbConnection* Instanzen. Der Typ der zu verwendenden *DbConnection* wird über die *App.config* definiert, sowie der *ConnectionString*.

1.1.5 Tests

Die Tests bestehen aus einer einzigen Testklasse, die das *IDao* Interface bzw. die dessen Ableitungen bzw. dessen Implementierungen, die zurzeit nur aus der Implementierung in *BaseDao* bestehen. Also die Basis Dao Funktionalitäten beinhalten wie.

1. *dao.ById // Throws Exception*
2. *dao.Find // Returns null*
3. *dao.Update*
4. *dao.Persist*
5. *dao.Delete*
6. *dao.CheckIfExists*

Die generische Testklasse *BaseDaoTest* wird mit *NUNIT* Attributen versehen, die die Testklasse mit den zur Verfügung gestellten Typen instanzieren. Danach wird in der Setup Methode die verwendeten Ressourcen über Reflection instanziert und in der Tear-Down Methode disposed. Hier ist die Typinformation zur Laufzeit sehr Hilfreich. In Java als Beispiel würde hier dies Javas Type Erasure unmöglich machen.

```
[TestFixture(typeof(long?), typeof(User), typeof(UserDao), typeof(UserEntityTestHelper))]
[TestFixture(typeof(long?), typeof(ArtistGroup), typeof(ArtistGroupDao), typeof(ArtistGroupEntityTestHelper))]
[TestFixture(typeof(long?), typeof(ArtistCategory), typeof(ArtistCategoryDao), typeof(ArtistCategoryEntityTestHelper))]
[TestFixture(typeof(long?), typeof(Artist), typeof(ArtistDao), typeof(ArtistEntityTestHelper))]
[TestFixture(typeof(long?), typeof(Venue), typeof(VenueDao), typeof(VenueEntityTestHelper))]
[TestFixture(typeof(long?), typeof(Performance), typeof(PerformanceDao), typeof(PerformanceEntityTestHelper))]
[CreateDatabase]
0 references | Thomas Herzog, 21 hours ago | 1 author, 3 changes
public class BaseDaoTest<I, E, D, C> where E : class, IEntity<I>
                                     where D : class, IDao<I, E>
                                     where C : class, IEntityHelper<I, E>
{

    // Here we depend on naming convention of factory method names
    // because here we get the dao from the factory via reflections
    protected D dao;
    protected IEntityHelper<I, E> entityHelper;

    [SetUp]
    0 references | Thomas Herzog, 21 hours ago | 1 author, 2 changes
    public void Init()
    {
        dao = typeof(DaoFactory).GetMethod("Create" + typeof(D).Name).Invoke(null, null) as D;
        entityHelper = Activator.CreateInstance(typeof(C)) as C;
        entityHelper.Init();
        Console.WriteLine("setup called");
    }

    [TearDown]
    0 references | Thomas Herzog, 21 hours ago | 1 author, 2 changes
    public void Dispose()
    {
        Console.WriteLine("tear down called");
        dao?.Dispose();
        entityHelper?.Dispose();
    }
```

Abbildung 7: Ausschnitt aus der Testklasse *BaseDaoTest*

1.2 Ausbaustufe 2 (Commander)

Folgender Teil dokumentiert die zweite Ausbaustufe des Projektes *UFO* in dem die Administration mit WPF implementiert werden sollte.

Folgende Projekte wurden der Solution hinzugefügt.

1. *UFO.Commander.ServiceApi*
   Diese Projekt enthält die Schnittstellen Spezifikation für den Service Layer.
2. *UFO.Commander.Service.Impl*
   Dieses Projekt enthält die Implementierungen der Service Schnittstellen Spezifikationen
3. *UFO.Commander.Wpf.Administration*
   Dieses Projekt enthält die WPF Anwendung, die die Administration abbildet.

Des Weiteren wurde der Wurzelnamensraum auf *UFO* beschränkt unter dem jetzt alle Projekte liegen.

1.2.1 Klassenhierarchien

Folgender Teil der Dokumentation dokumentiert die definierten Klassenhierarchien der WPF Anwendung und des Service Layers.

IService

Folgendes Klassendiagramm zeigt die Hierarchie des Interfaces *IService*, welche die Wurzel aller Service Interfaces darstellt und *IDisposable* erweitert. Somit ist jeder abgeleiteter Service dazu gezwungen die Methode *Dispose* zu implementieren um dort seine gebundenen Ressourcen freizugeben. Des Weiteren wurde eine Basisklasse namens *BaseService* eingeführt, welche die Datenbankverbindung und Transaktionsmethoden implementiert, sodass die abgeleitet Klassen diese Nutzen können.

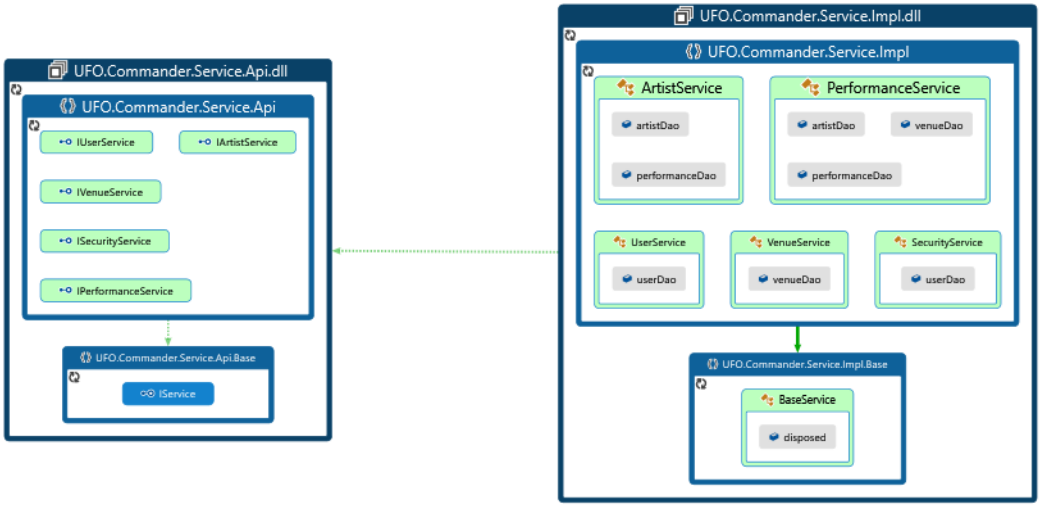

Abbildung 8: Basis Interface für Service Interfaces und Implementierungen

Alle Services bekommen in Ihrem Konstruktor eine *DbConnection* Instanz übergeben, da alle verwendeten *DAO* Implementierungen dieselbe Datenbank Verbindung nutzen müssen, damit alle sich in derselben Transaktion bewegen.

ITabModel

Folgendes Klassendiagramm zeigt die Hierarchien des Interfaces *ITabModel*, die Operationen definiert, die von der Klasse *TabController* verwendet werden. Die Klasse *TabController* wurde eingeführt um *ITabModel* Instanzen zu initialisieren und beim Wechseln eines Tab aufzuräumen.

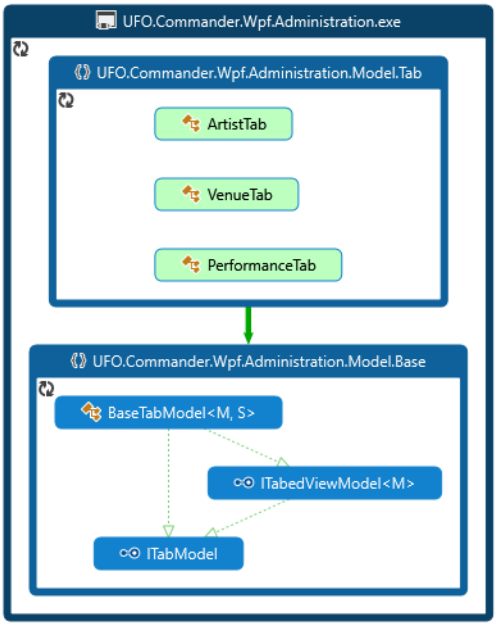

Abbildung 9: Basis Interface für Tab-Model Implementierungen

Das Interface *ITabedViewModel* definiert die Struktur der Implementierten Tab-Klassen, somit verhält sich jeder Tab gleichermaßen und kann beliebig erweitert werden, je nach seinem View-Content.

BasePropertyChangeViewModel

Folgendes Klassendiagramm zeigt die Hierarchien der Basisklasse *BasePropertyChangeViewModel*, die die Wurzelklasse aller implementierten *ViewModels* darstellt, da es immer mindesten einen Property gibt der diesen Event benötigt. Des Weiteren wurde die Klasse *BaseValidationViewModel* eingeführt die die erste Ableitung von *BasePropertyChangeViewModel* darstellt und die Logik für die Validierung über *System.ComponentModel.DataAnnotations* realisiert. Von dieser Klasse erbt die Basisklasse *BaseEntityViewModel*, die als Wrapper für ViewModels verwendet wird, die eine *IEntity* Instanz für die View wrappen.

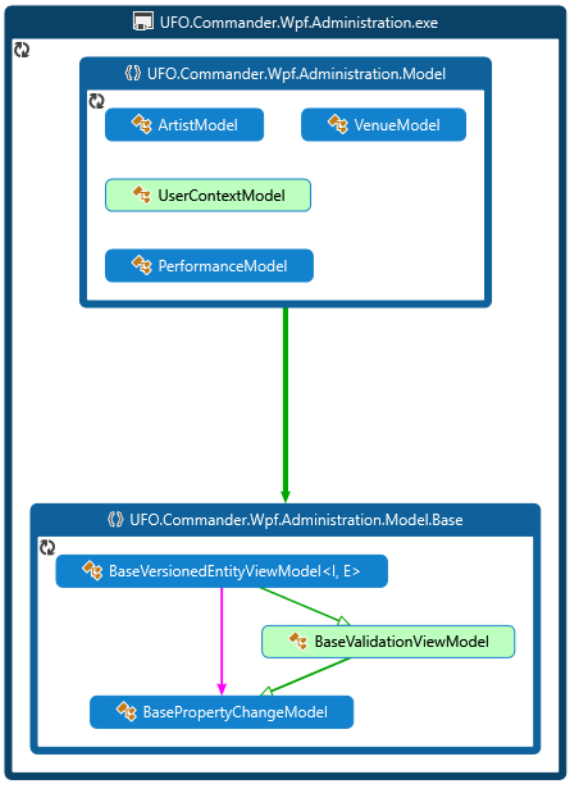

Abbildung 10: Basisklasse für ViewModels

Die Klasse *UserContextModel* wurde eingeführt um einen eingeloggten Benutzer zu repräsentieren und wird als statischer Property der Klasse *App* definiert, da es nur einen eingeloggten Benutzer pro gestarteter Anwendung geben kann. Alle Klassen, die auf den UserContext angewiesen sind müssen die Instanz der Klasse *App* verwenden.

SimpleObjectModel

Folgendes Klassendiagramm zeigt die Hierarchien der Klasse *SimpleObjectModel*, die eingeführt wurde um in Listen, Comboboxen und dergleichen Entitäten für die Darstellung zu halten. Die Controls verwenden Konverter, in denen aus der String Repräsentation wieder in die Entität konvertiert wird. Diese Klasse hält hierbei eine object Instanz (z.B.: Artist) und den anzuzeigenden Label. In den Konvertern wird eine Instanz *SimpleObjectModel* aus dem zu verwaltenden Objekt erzeugt und visa versa.

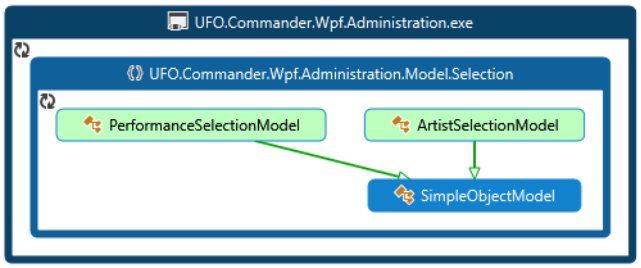

Abbildung 11: Klasse für Controls mit Konverter

Die beiden abgeleitet Klassen erweitern hierbei die Klasse *SimpleObjectModel* um spezifische Attribute, die in dern Controls verwendet werden.

IConverter

Folgendes Klassendiagramm zeigt die Hierarchien der Klasse *IConverter*.

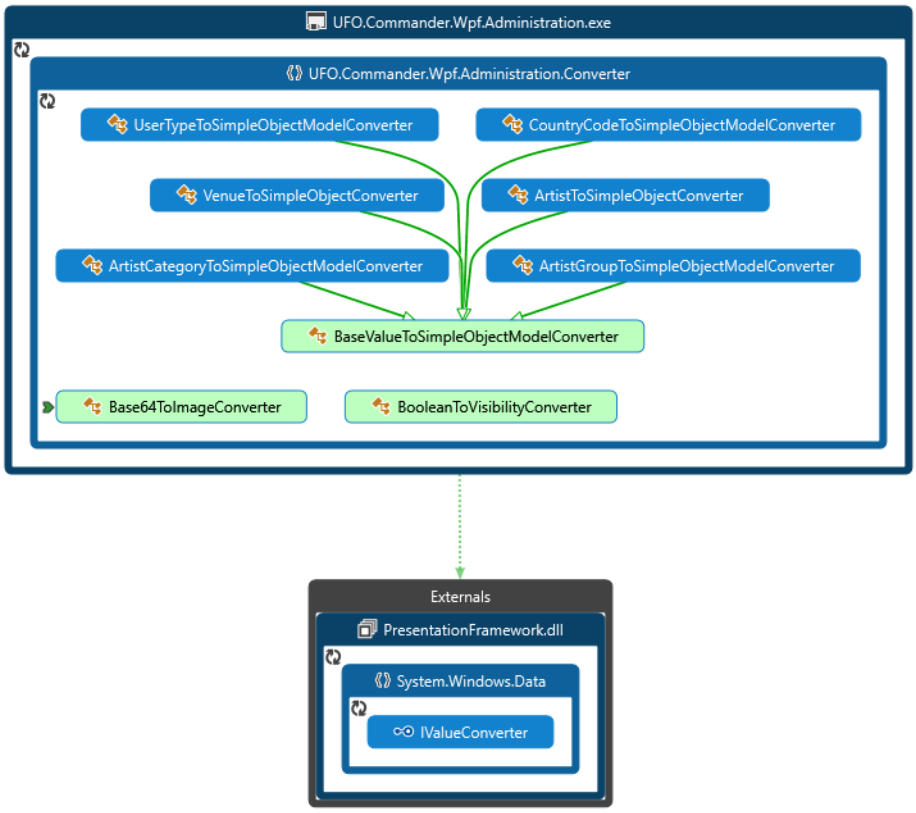

Abbildung 12: Konverter Hierarchie

Die Klasse *BaseValueToSimpleObjectConverter* stellt die Basisklasse aller Konverter dar, die Instanzen von *SimpleObjectModel* konvertieren. In diese Klasse wurden alle gemeinsamen Funktionalitäten wie

1. *Typprüfung*
   ES wird geprüft ob der Typ des übergebenen Value dem erwarteten Typ entspricht
2. *ConvertBack*
   Da hier nur Instanzen von *SimpleObjectModel* konvertiert werden kann diese Methoden in einer Basisklasse implementiert werden, da hier nur auf den Property *Data* zugegriffen wird.

Es wurden auch Konverter für die Visibility (true=Visibility.VISIBILE, false=Visibility.HIDDEN) und zum dekodieren von Base64 Strings in Image Instanzen eingeführt.

1.2.2 WPF Views

Folgende Abbildung zeigt wie die Views innerhalb des Projektes strukturiert wurden. *IConverter*.

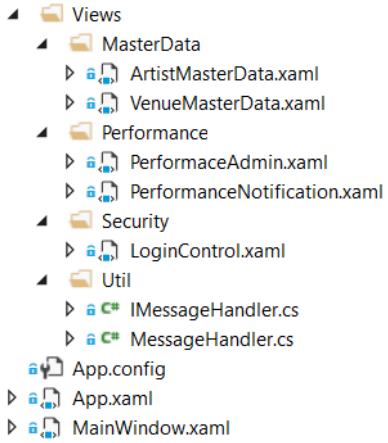

Abbildung 13: Verzeichnisstruktur der Views innerhalb des Projektes

Bis auf die *App.xaml* und *MainView.xaml* wurden alle Views innerhalb eines Verzeichnis names *Views* gebündelt wobei hierbei eine Trennung zwischen den einzelnen Typen der Views duchgeführt wurde.

1. *MasterData*
   Alle Views für dier Verwaltung der Stammdaten

2. *Performance*
   Alle Views für die Verwaltung des Festivalprogramms

3. *Security*
   Die Login-View

4. *Util*
   Utility Klassen um innerhalb von View-Models mit der View zu interagieren ohne Referenzen auf View Namespaces verwenden zu müssen.

Alle diese Views sind als UserControls implementiert worden und werden innerhalb von *MainView.xaml* verwendet (außer PerformanceNotification.xaml), die diese UserControls in einem Tab-Control als DataTemplate für die verschiedene Typen der Tab-Models definiert.

Folgende Abbildung zeigt die eingeführte Verzeichnisstruktur um die WPF-Ressourcen innerhalb diese Projektes zu strukturieren.

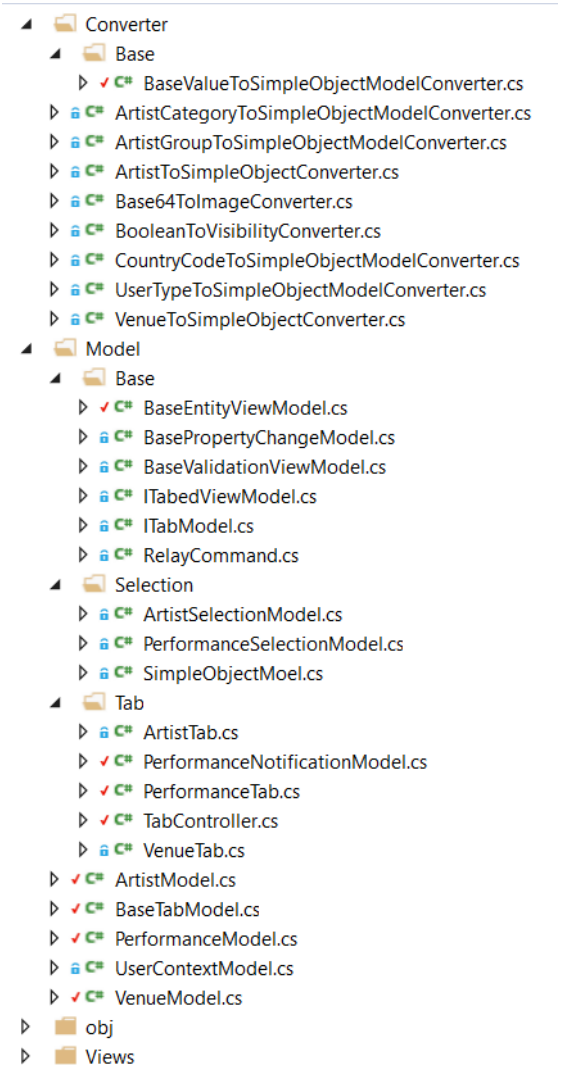

Abbildung 14: Verzeichnisstruktur der WPF-Ressourcen

Die jeweiligen Base-Verzeichisse beinhalten die eingeführten Basisklassen für den jeweiligen Kontext (z.B.: Converter, Models, ...).