ins institute for networked solutions

HSR HOCHSCHULE FÜR TECHNIK RAPPERSWIL .NET Competence Center

.NET Einführung

Mini-Projekt Auto-Reservation

Version:	7.0
Autor:	Manuel Bauer
	Hansjörg Huser
Letzte Änderung:	01.10.2013
Status:	Final

Inhaltsverzeichnis

1 Aufgabenstellung / Administratives

1.1 Einführung

Sie bekommen als Vorgabe eine einfache WPF-Applikation für die Verwaltung von Auto-Reservationen für eine Auto-Reservations-Firma.
Es handelt sich hierbei um eine Multi-Tier-Applikation mit einer WCF-Schnittstelle. Als User-Interface existiert bereits ein WPF-Projekt, welches Sie so verwenden können.

Ziele:

- Anwendung von Technologien aus der Vorlesung
- Verteilte Applikationen mit Datenbank-Zugriff konzipieren und umsetzen können

1.2 Teams

Sie können das Projekt als 2er Team bestreiten, Einzelarbeiten und 3er Teams sind auch möglich, sollen aber die Ausnahme bilden.

Wir werden in den Übungen eine Einschreibeliste („Gruppeneinteilung Microsoft-Technologien“) auflegen, in welcher Sie ihr Team und den gewünschten Abnahmetermin eintragen können.

1.3 Arbeitspakete

Paket 1: Data Access Layer und Business Layer (KW 47/48)

1. Implementieren Sie den DAL mit dem Entity Framework.
2. Implementieren Sie den Business-Layer mit den CRUD-Operationen. Die Update-Operationen sollen Optimistic-Concurrency unterstützen.
3. Schreiben Sie die geforderten Unit-Tests für den Business-Layer.

Paket 2: Service Layer (KW 48/49)

1. Definieren Sie das Service-Interface mit den DTO’s.
2. Implementieren Sie die Service-Operationen. Der Service-Layer ist auch verantwortlich für das Konvertieren der DTO‘s in Entities resp. Entities in DTO’s sowie für das Umsetzen der Fault-Exceptions.
3. Schreiben Sie die geforderten Unit-Tests für das Service-Interface.

Paket 3: User-Interface (KW 50/51)

1. Vervollständigen Sie das User-Interface inklusive Factory.
2. Schreiben Sie die geforderten Unit-Tests für die View-Models.

Wichtig:
Lesen Sie die nachfolgenden Kapitel sorgfältig durch. Sie erhalten dort weitere Informationen zu den einzelnen Aufgaben! Die Erläuterungen sind nicht immer in der Reihenfolge, in der Sie die Aufgaben abarbeiten sollten. Lesen Sie daher das ganze Kapitel durch, bevor Sie mit der Implementation starten.

1.4 Alternative

Es steht den Studierenden offen, einen test-driven Ansatz zu verfolgen. Dies ist aber eher für Entwickler zu empfehlen, welche in der Konzeption von Business-Applikationen bereits sattelfest sind.

Wenn Sie diesen Ansatz wählen ist es umso wichtiger, die Aufgabenstellung vollends verstanden zu haben. Ansonsten könnte dies zu erheblichem Mehraufwand führen.

1.5 Abgabe (Wo 51)

In der letzten Semesterwoche finden die Abgaben gemäss dem Plan „Gruppeneinteilung MsTe Miniprojekt“ statt.

Es wird rechtzeitig eine Check-Liste für die Bewertung der Resultate veröffentlicht.

Bei der Abgabe des Miniprojektes muss jede Gruppe pünktlich zum Abnahmetermin eine lauffähige Version ihrer Implementation auf einem Übungsrechner oder privaten Notebook bereitgestellt haben, damit ein reibungsloser Ablauf und die Einhaltung des Terminplans gewährleistet werden kann.

Eine erfolgreiche Bewertung ist die Voraussetzung für die Zulassung zur Modulschlussprüfung.

2 Erläuterungen

2.1 Analyse der Vorgabe

Öffnen Sie die Solution „AutoReservation.sln“ Die Solution besteht aus folgenden Projekten:

- AutoReservation.BusinessLayer
  Beinhaltet die Implementation der Methoden mit den CRUD-Operationen auf den Business-Entities. Dazu wird das *.DAL Projekt verwendet.
- AutoReservation.Common
  Gemeinsames Projekt von Client und Server, hier werden die in der gesamten Applikation bekannten Artefakte (Service-Interface, DTO's) abgelegt.
- AutoReservation.Dal
  Data Access Layer basierend auf ADO.NET Entity Framework.
- AutoReservation.Service.Wcf
  Projekt für die WCF-Serviceschnittstelle. Implementiert die Service-Operationen, greift dazu auf den Business-Layer zu. Verantwortlich für die Konvertierung von Entities nach DTO's und zurück.
- AutoReservation.Service.Wcf.Host
  Konsolen-Applikation für das Hosting des WCF-Services.
- AutoReservation.Ui
  WPF-Applikation Dialogen zum Anzeigen und Einfügen von Autos bzw. Kunden sowie zum Anzeigen, Einfügen und Löschen von Reservationen.

2.2 Installation Datenbank

Für die Installation der Datenbank steht ein SQL-Script „AutoReservation.Database Create Script.sql“ im Vorgabenverzeichnis zur Verfügung.

Es ist sehr empfehlenswert, dass beim Arbeiten im Team auf allen Rechnern mit der gleichen Datenbank-Instanz gearbeitet wird, da ansonsten die Connection Strings unterschiedlich sind. Diese müssen dann oftmals wieder angepasst werden.

2.2.1 LocalDB

LocalDB ist eine SQL Server Runtime, welche on-demand hochgefahren wird. Die Runtime wird zusammen mit Visual Studio ab Version 2012 automatisch installiert. Es wird empfohlen, mit dieser Version zu arbeiten. Das Script kann via „File > Open > File…“ oder über den „SQL Server Object Explorer“ ausgeführt werden.

Server-/Instanzname: „(localdb)\v11.0“ / „(localdb)\v12.0“ / „(localdb)\Projects“

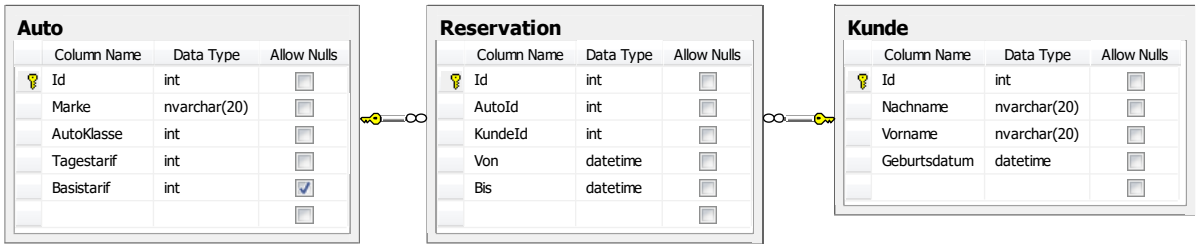
2.2.2 SQL Server / SQL Server Express

Öffnen Sie das Script im Managment Studio (Express) oder wie oben beschrieben via Visual Studio und führen Sie es aus.

Server-/Instanzname: „localhost“ / „localhost\SQLEXPRESS“

2.2.3 Datenbankstruktur

Die SQL Server Datenbank ist einfach gehalten mit folgender Struktur.



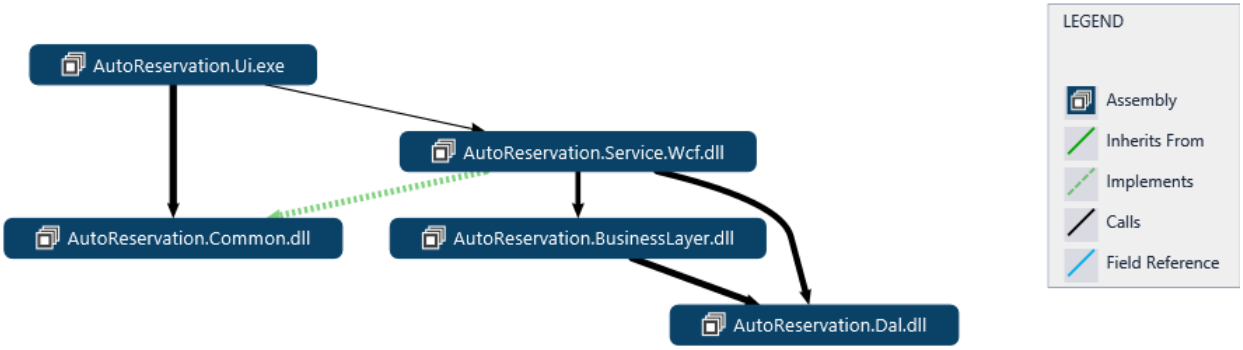
Ein Auto besitzt die Spalte „AutoKlasse“, welche 3 Werte haben kann:

- AutoKlasse = 0 Luxusklasse
- AutoKlasse = 1 Mittelklasse
- AutoKlasse = 2 Standard

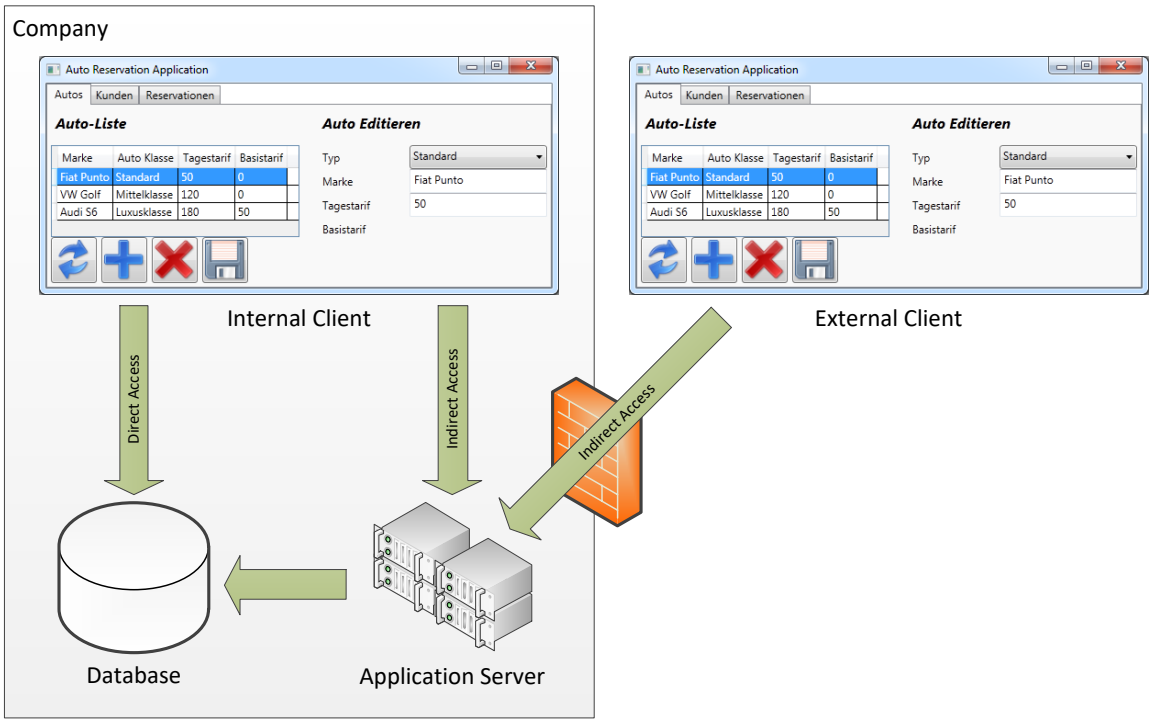
Das Feld „Tagestarif“ muss bei allen Autos erfasst sein, der Basistarif existiert nur für Wagen der Luxusklasse.

2.3 Applikationsarchitektur

Die Architektur der Applikation ist bereits im Groben vorgegeben. Sie bewegen sich in den vorgegebenen Strukturen. In der Abbildung unten sind die vorhandenen Assemblies der abgegebenen Solution zu finden.

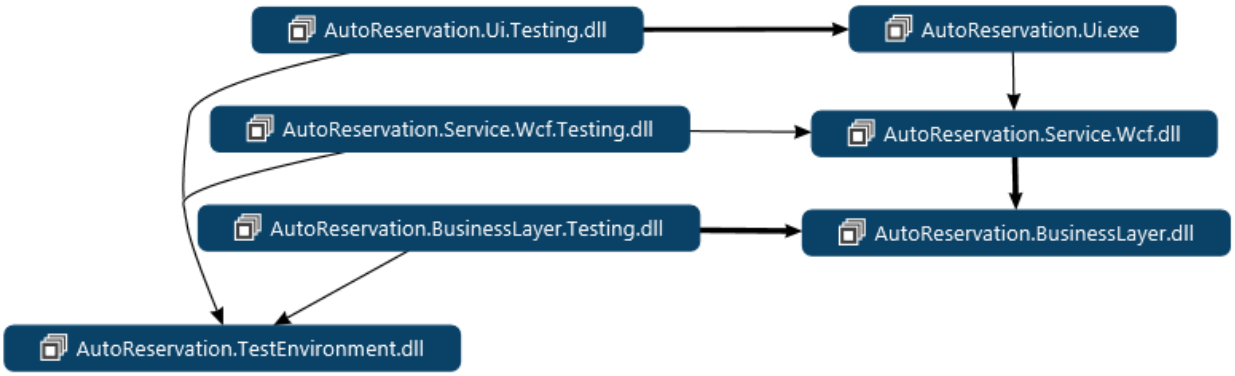


Ein Ziel der Applikation ist es, den Client wahlweise direkt oder indirekt über einen Applikationsserver auf die Datenbank zugreift. Dies dient dem Zweck, den Client für einen Aussendienstmitarbeiter oder einen internen Sachbearbeiter konfigurieren zu können. Diese Anforderung wird im Kapitel 3.5.1 Factory technisch beschrieben.



2.3.1 Test-Projekte

Jede Architekturschicht wird über ein separates Test-Projekt geprüft. Die benötigten Testklassen dafür sind bereits vorgegeben. Allgemeine Testing-Funktionalität ist im Test-Environment-Projekt vorhanden. Auf Mocks wurde der Übersichtlichkeit halber verzichtet.



3 Implementation

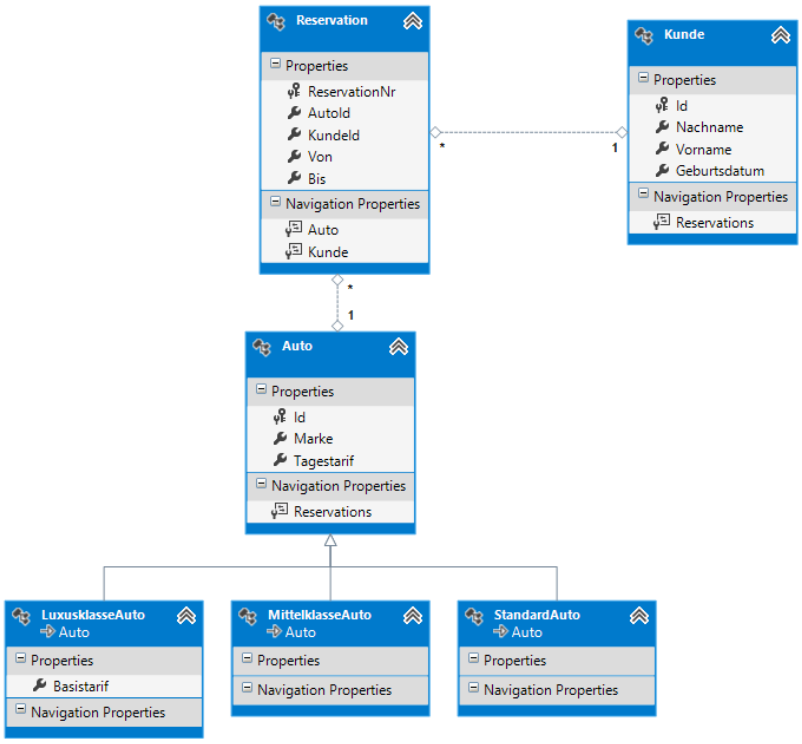
3.1 Data Access Layer

Projekt „AutoReservation.Dal“

Der Data Access Layer wird mit dem ADO.NET Entity Framework 6.0 implementiert. Es muss zwingend der „EF 6.x DbContext Generator“ verwendet werden, dieser wird aber seit Version 6.0 standardmässig verwendet.

Dieser Layer beinhaltet eigentlich nur das *.edmx Datenmodell, welches den Datenzugriff auf die darunterliegende SQL Server Datenbank ermöglicht.

Das Modell sieht folgendermassen aus:



Hier wird wieder anhand des Feldes „AutoKlasse“ des Autos entschieden, ob es sich um ein Luxusklasse-, Mittelklasse- oder Standardauto handelt. Modellieren Sie Ihren Data Access Layer wie oben dargestellt.

Wichtige Hinweise:

- Wählen Sie beim Importieren „Include Foreign Keys in the Model“
- Der Entity Container Name sollte „AutoReservationEntities“ und der sein, ansonsten müssen die Connection Strings in ALLEN App.config Dateien angepasst werden
  (Rechte Maustaste auf weisse Fläche im *.edmx Designer / Properties / Entity Container Name)
- Ändern Sie die Entity Set Names auf: Autos, Reservationen, bzw. Kunden
- Achten Sie darauf, dass der Primärschlüssel der Reservation „ReservationNr“ ist. Ansonsten wird der `DtoConverter` nicht korrekt funktionieren.
- Setzen Sie jedem Property des Models den Concurrency Mode auf „Fixed“, um Optimistic Concurrency zu aktivieren (ausgenommen Property „Basistarif“).

3.1.1 App.config & Connection Strings

Grundsätzlich ist die .NET Runtime Konfiguration (App.config) so aufgebaut, dass das App.config des ausgeführten Assemblies (*.exe, Unit-Test, etc.) immer alle Konfigurationselemente der referenzierten Assemblies beinhalten muss.

Damit diese Einstellungen nicht in allen Projekten einzeln vorgenommen werden muss, sind, ist eine allumfassende App.config-Datei im Solution-Ordner vorhanden. In den einzelnen Projekten wird diese Datei dann als Link referenziert.

Bei Schwierigkeiten mit Suche des richtigen Connection Strings kann folgende Seite konsultiert werden: http://www.connectionstrings.com/

Es empfiehlt sich, dass sämtliche Gruppenmitglieder sich auf einen spezifischen Connection String einigen. Ansonsten wird der Austausch des Quellcodes schwierig.

3.2 Gemeinsame Komponenten

Projekt „AutoReservation.Common“

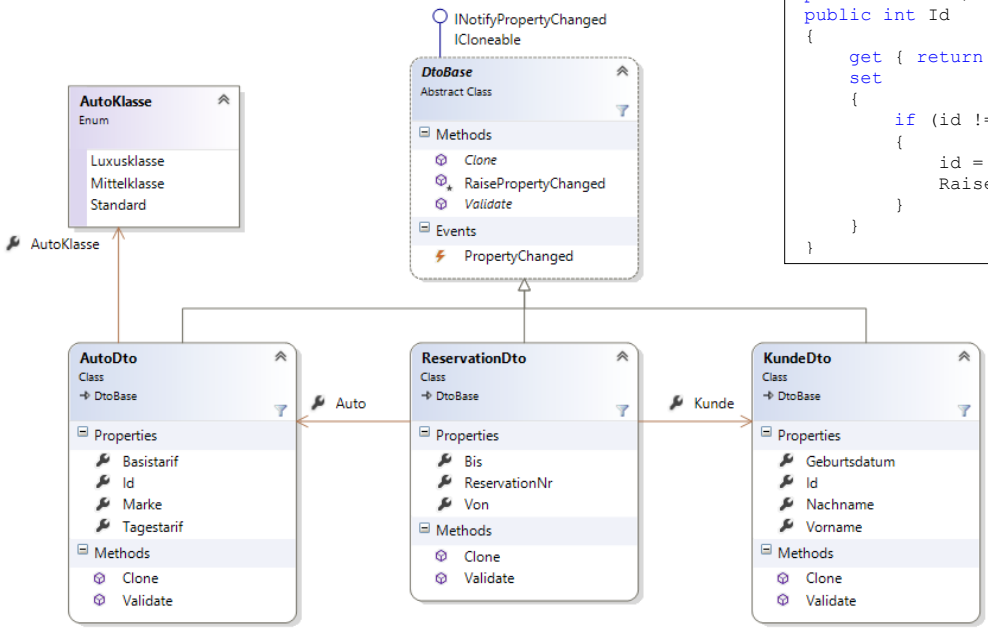
3.2.1 Datentransferobjekte

Im Data Access Layer wird für jede Entität ein DTO zur Übertragung zwischen Client und Server erstellt. Im Folgenden implementieren Sie drei DTO Klassen.

Die Basisklasse `DtoBase` ist bereits vorgegeben. Sie implementiert unter Anderem das Interface `INotifyPropertyChanged`, welches von WPF dann für das Data-Binding gebraucht wird. Beim Implementieren der Properties müssen Sie darauf achten, dass nach dem Verändern des Wertes der `PropertyChanged` Event gefeuert wird. Verwenden Sie dazu die Methode `RaisePropertyChanged`.

Unten finden Sie eine Beispiel-Implementation eines solchen Properties.

```
private int id;
public int Id
{
    get { return id; }
    set
    {
        if (id != value)
        {
            id = value;
            RaisePropertyChanged();
        }
    }
}
```



Hinweis:
Beachten Sie auch den Enumerator `AutoKlasse`.

3.2.2 Service-Interface

Das Service-Interface `IAutoReservationService` definiert die Funktionalität für den Service-Layer. Für jede Entität – Auto, Kunde und Reservation – müssen folgende CRUD[1]-Operationen unterstützt werden:

- Alle Entitäten lesen
- Eine Entität anhand des Primärschlüssels lesen
- Einfügen
- Update
- Löschen

Der Rückgabewert der beschriebenen Methoden - wenn vorhanden – ist immer ein DTO respektive eine Liste davon, nie eine Entität des Data Access Layers.

Bedenken Sie hier auch, dass sämtliche Update-Methoden nach dem Prinzip Optimistic Concurrency funktionieren müssen. Grundsätzlich existieren zwei praktikable Ansätze, diese Problematik zu bewältigen:

1. Die Service-Implementation kennt den Original-Status des gelesenen Objektes und kann so anhand des modifizierten Objektes auf die Änderungen schliessen. Auch hier gibt es wieder zwei Varianten:
   a. Beide Objekte werden vom Client mitgegeben. Das Service-Interface hat die Signatur:
      `UpdateAuto(AutoDto modified, AutoDto original);`
   b. Die Service-Instanz hält die Liste der gelesenen Original-Entitäten und holt sich beim Update den Original-Status aus dieser Liste. Hier wäre die Signatur:
      `UpdateAuto(AutoDto modified);`
2. Änderungen müssen clientseitig aufgezeichnet und dem Service-Interface mitgeteilt werden.

Der einfachste Ansatz für diesen Zweck dürfte wohl Variante 1a darstellen. Ansatz 1b führt zu einer Caching-Problematik und Ansatz 2 erhöht den Implementationsaufwand massiv.

3.3 Business Layer

Projekt „AutoReservation.BusinessLayer“

Im Business Layer findet der Zugriff auch den Data Access Layer (DAL) statt, sprich hier werden die Daten vom DAL geladen und verändert. Im Business Layer existiert eine Klasse `AutoReservationBusinessComponent`, in welcher die Business-Operationen implementiert werden sollen, die für die Implementation des Service-Interface benötigt werden.

Das Handling der `DbUpdateConcurrencyException`– eine Exception welche beim Auftreten einer Optimistic Concurrency-Verletzung geworfen wird – soll bei den Update-Methoden gehandhabt werden. Im Falle des Auftretens einer solchen Exception wird eine `LocalOptimisticConcurrencyException` (existiert bereits) geworfen, welche die neuen in der Datenbank vorhandenen Werte beinhaltet.
Für das Handling dieser Exception kann die bereits bestehende Methode `HandleDbConcurrencyException` direkt im catch-Block aufgerufen werden.

[1] CRUD = Create, Read, Update, Delete

Falls Sie den Update-Methoden das modifizierte und das Original-Objekt mitgeben, können Sie diese nach dem Konvertieren wieder dem verwendeten OR-Mapper angehängt werden.

Codefragmente für das ADO.NET Entity Framework:

```
// Insert
context.Autos.Add(auto);

// Update
context.Autos.Attach(original);
context.Entry(original).CurrentValues.SetValues(modified);

// Delete
context.Autos.Attach(auto);
context.Autos.Remove(auto);
```

Hinweis:
Die Businesslogik wird nur sehr rudimentär implementiert. Die Verfügbarkeit der Fahrzeuge zum Beispiel wird nicht überprüft.

3.4 Service Layer

Projekt „AutoReservation.Service.Wcf“

Der Service Layer ist die eigentliche WCF-Serviceschnittstelle (Klasse `AutoReservationService`) und implementiert das Interface `IAutoReservationService`.

Im Normalfall müsste es hier genügen, eine Instanz der `AutoReservationBusinessComponent` zu halten und die eingehenden Calls mehr oder weniger direkt an diese weiterzureichen. Der Service-Layer ist in dieser einfachen Applikation also nicht viel mehr als ein „Durchlauferhitzer“. Die wichtigste Aufgabe ist das Konvertieren von DTO’s in Objekte des Business-Layers sowie das Mapping von Exceptions auf WCF-FaultExceptions. In grösseren Projekten kann hier aber durchaus noch Funktionalität – z.B. Sicherheitslogik – enthalten sein.

Wie Sie das Hosting des WCF Services handhaben ist Ihnen überlassen. Empfohlen ist jedoch, die Projekte AutoReservation.Service.Wcf.Host und AutoReservation.Ui als Startprojekte zu definieren, so umgehen Sie allfällige Probleme mit dem Generieren von Service-Referenzen und dem Autohosting Feature im Visual Studio.

Achtung:
Sie benötigen für diesen Schritt Admin-Rechte auf dem Entwicklungsrechner.

Hinweis:
Die Klasse `AutoReservationService` ist bereits vorhanden und beinhaltet eine statische Methode `WriteActualMethod`, welche den Namen der aufrufenden Methode auf die Konsole ausgibt. Diese sollte bei jedem Service-Aufruf ausgeführt werden, so wird auf der Konsole des Services immer ausgegeben, was gerade passiert.

3.4.1 DTO Converter

Die im Projekt vorhandene Klasse `DtoConverter` bietet diverse Erweiterungsmethoden an, um DTO's in Entitäten und umgekehrt zu konvertieren. Die gleiche Funktionalität steht auch für Listen von DTO's respektive Listen von Entitäten zur Verfügung.
Hier ein Beispiel für die Anwendung:

```
// Entität konvertieren
Auto auto = db.Autos.First();
AutoDto autoDto = auto.ConvertToDto();
auto = autoDto.ConvertToEntity();

// Liste konvertieren
List<Auto> autoList = db.Autos;
List<AutoDto> autoDtoList = autoList.ConvertToDtos();
autoList = autoDtoList.ConvertToEntities();
```

3.5 GUI Layer

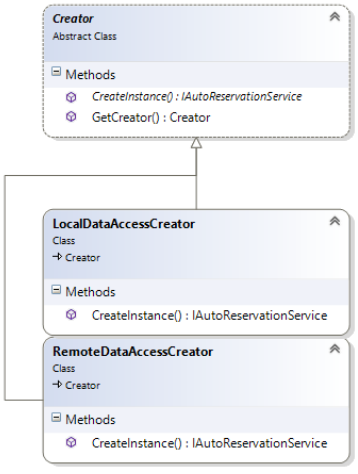
Projekt „AutoReservation.Ui"

Das GUI setzt auf Standard WPF-Komponenten und ist nach dem MVVM-Prinzip[2] aufgebaut. Durch den Einsatz von WPF, Data Binding und MVVM ist es möglich, das UI ohne C# Code zu implementieren. Die gesamte Logik ist im ViewModel angesiedelt.

Der Einfachheit halber wurden die Verwaltungsseiten für Autos und Kunden bereits implementiert. Benutzen Sie diese als Vorlage und implementieren Sie die gleiche Funktionalität für die Reservationen.

3.5.1 Factory

Damit der GUI Layer seine Daten via WCF-Service oder von einer lokalen Objektinstanz holen kann, soll eine Factory implementiert werden, welche für die Instantiierung des jeweiligen Layers zuständig ist. Der Rückgabewert der Factory-Metode ist `IAutoReservationService`.



Die Factory wird konfiguriert über die Settings.settings Datei im UI-Projekt. Das Setting „ServiceLayerType" entspricht dem Assembly Qualified Name[3] des konkreten Creators:

AutoReservation.Ui.Factory

.LocalDataAccessCreator.cs, …
(Konfiguration für direkten Datenzugriff ohne WCF)

.RemoteDataAccessCreator, …
(Konfiguration für indirekten Datenzugriff via WCF; der WCF Service muss zuvor gestartet werden)

Um eine Instanz des Creators aus diesem String zu erzeugen können Sie dieses Code-Fragment in der Methode „GetCreator" benützen.

```
Type serviceLayerType = Type.GetType(Settings.Default.ServiceLayerType);
if (serviceLayerType == null) { return new LocalDataAccessCreator();}
return (Creator)Activator.CreateInstance(serviceLayerType);
```

[2] MVVM ist die Abkürzung für Model-View-ViewModel (http://msdn.microsoft.com/en-us/magazine/dd419663.aspx)
[3] Assembly Qualified Name: http://msdn.microsoft.com/en-us/library/system.type.assemblyqualifiedname.aspx

4 Unit Tests

Schreiben Sie Unit-Tests, welche die Business-Layer-Schnittstelle testen. Verwenden Sie die im Visual Studio integrierte Testbench um die Tests zu schreiben.

Tipp:
Im Projekt „AutoReservation.TestEnvironment“ existiert schon eine Klasse `TestEnvironmentHelper` (Methode `InitializeTestData`), welche Ihnen die Initialisierung der Testumgebung abnimmt. Diese löscht den gesamten Datenbankinhalt und erstellt jeweils die gleichen drei Autos, vier Kunden und eine Reservation (inklusive Primärschlüssel).

4.1.1 Business-Layer

Projekt „AutoReservation.BusinessLayer.Testing“

Diese Tests sollen relativ früh implementiert werden und eine gewisse Sicherheit geben, dass die Applikation – vor allem die Datenbank-Verbindung und –Abfragen – in ihren Grundzügen funktioniert.
Im Mindesten sollen folgende Operationen für alle drei Entitäts-Typen (Autos, Kunden, Reservationen) mit Tests überprüft werden:

- Update Kunde
- Update Auto
- Update Reservation

4.1.2 Service-Layer

Projekt „AutoReservation.Service.Wcf.Testing“

Studieren Sie das vorgegebene Konstrukt in der Projektvorgabe:

- ServiceTestBase
  Abstrakte Basis-Testklasse. Enthält bereits alle Methoden für die zu testende Funktionalität.

- ServiceTestLocal
  Konkrete Implementation. Testet die Funktionalität des Services anhand einer lokalen Objektinstanz (`new AutoReservationService()`)

- ServiceTestRemote
  Konkrete Implementation. Testet die Funktionalität des Services anhand eines WCF-Client-Proxies (via `ChannelFactory<T>`).

Dieses Konstrukt scheint auf den ersten Blick überdimensioniert, erfüllt jedoch so mehrere Aspekte, die beim Testing wichtig sind.

- Die Testlogik muss nur einmal in der abstrakten Basisklasse implementiert werden.
- Jede Implementation von `IAutoReservationService` kann in einer abgeleiteten Klasse praktisch ohne Mehraufwand getestet werden.
- Es wird so auch sichergestellt, dass Serialisierungs-Mechanismen und Exception-Handling ebenfalls getestet werden.
  (Die Methoden können sich lokal / via WCF jeweils anders verhalten)

Im Mindesten sollen folgende Operationen für alle drei Entitäts-Typen (Autos, Kunden, Reservationen) mit Tests überprüft werden:

- Abfragen einer Liste
- Suche anhand des Primärschlüssels
- Einfügen
- Updaten
- Löschen
- Updates
- Updates mit Optimistic Concurrency Verletzung

4.1.3 GUI Layer

Projekt „AutoReservation.Ui.Testing“

Ziel dieser Teilaufgabe ist es aufzuzeigen, dass mit dem MVVM-Prinzip relativ einfach GUI-nahe Tests implementiert werden können.
Sie müssen für die ViewModels lediglich folgende Tests in der Klasse „ViewModelTest“ implementieren:

- AutoViewModel Load / CanLoad
- KundeViewModel Load / CanLoad
- ReservationViewModel Load / CanLoad

Prüfen Sie, ob die CanLoad-Methode den erwarteten Wert liefert und ob nach dem Ausführen der Load-Methode die Daten in das ViewModel geladen wurden.

Theoretisch würde natürlich alle weiteren Commands (Save, Delete) ebenfalls noch getestet. Da dies jedoch sehr repetitiv ist wird an diesem Punkt vereinfacht.