

.NET Einführung

Mini-Projekt Auto-Reservation

Version:	4.0
Autor:	Hansjörg Huser Manuel Bauer
Letzte Änderung:	26.08.2010
Status:	Final

Inhaltsverzeichnis

1	AUFGABENSTELLUNG / ADMINISTRATIVES	3
1.1	EINFÜHRUNG.....	3
1.2	TEAMS	3
1.3	ARBEITSPAKETE	3
1.4	ALTERNATIVE.....	4
1.5	ABGABE (WO 51)	4
2	ERLÄUTERUNGEN.....	5
2.1	ANALYSE DER VORGABE	5
2.2	INSTALLATION DATENBANK	5
2.3	DATENBANK	5
2.4	ARCHITEKTUR DER APPLIKATION.....	6
2.4.1	KONFIGURATION	7
3	IMPLEMENTATION	8
3.1	DATA ACCESS LAYER	8
3.2	GEMEINSAME KOMPONENTEN	9
3.2.1	DATENTRANSFEROBJEKTE.....	9
3.2.2	SERVICE-INTERFACE.....	10
3.3	BUSINESS LAYER.....	10
3.4	SERVICE LAYER.....	11
3.4.1	DTO CONVERTER.....	12
3.5	GUI LAYER.....	12
3.5.1	FACTORY	12
4	UNIT TESTS.....	13
4.1.1	BUSINESS-LAYER.....	13
4.1.2	SERVICE-LAYER.....	13
4.1.3	GUI LAYER	14

1 Aufgabenstellung / Administratives

1.1 Einführung

Sie bekommen als Vorgabe eine einfache WPF-Applikation für die Verwaltung von Auto-Reservierungen 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 (Wo 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. Erstellen Sie ein Unit-Test-Projekt für den Test des Business-Layers mit den sämtlichen CRUD-Operationen. UpdateReservation, UpdateAuto, UpdateKunde sollen auch mit Optimistic Concurrency getestet werden.

Paket 2: Service Layer (Wo 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 Exceptions in ErrorCodes oder Fault-Exceptions.
3. Erstellen Sie die Unit-Tests fürs Service-Interface

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

1. Vervollständigen Sie das User-Interface
2. Erstellen Sie ein Testprojekt fürs Testen der 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 Microsoft-Technologien“ 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:

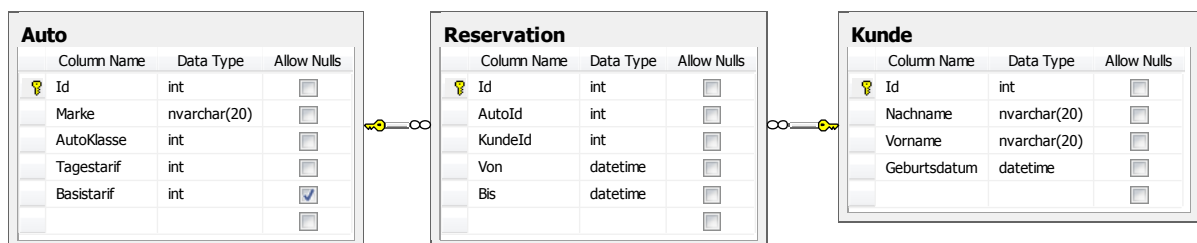
- **AutoReservation.BusinessLayer**
Businesslogik, greift auf die Operationen des DAL zu. Definiert eine Schnittstelle mit den CRUD-Operationen auf den Business-Entities.
- **AutoReservation.Common**
Gemeinsames Projekt von Client und Server, hier werden die in der gesamten Applikation bekannten Artefakte (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 die Business-Layer Schnittstelle zu. Verantwortlich für die Konvertierung von Entities auf DTO's.
- **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

Öffnen Sie im Managment Studio die Datei „AutoReservation.Database Create Script.sql“ aus dem Vorgaben-Verzeichnis und führen Sie es aus. Ihre Datenbank sollte nun erstellt sein.

2.3 Datenbank

Die SQL Server Datenbank ist einfach gehalten mit folgender Struktur.



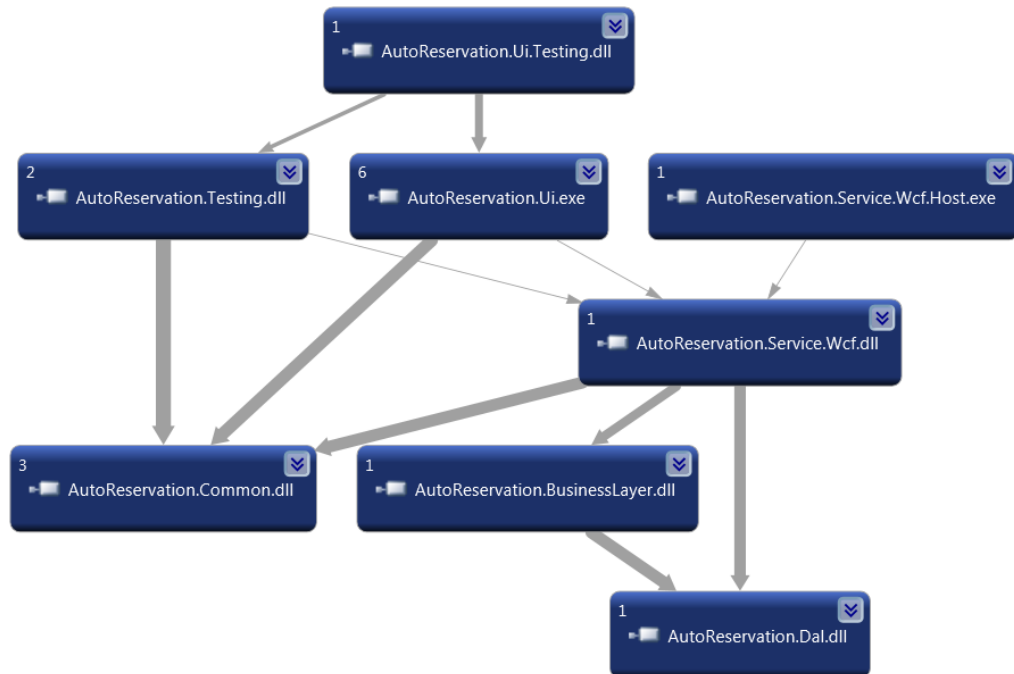
Ein Auto besitzt die Spalte „AutoKlasse“, welche 3 Werte haben kann:

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

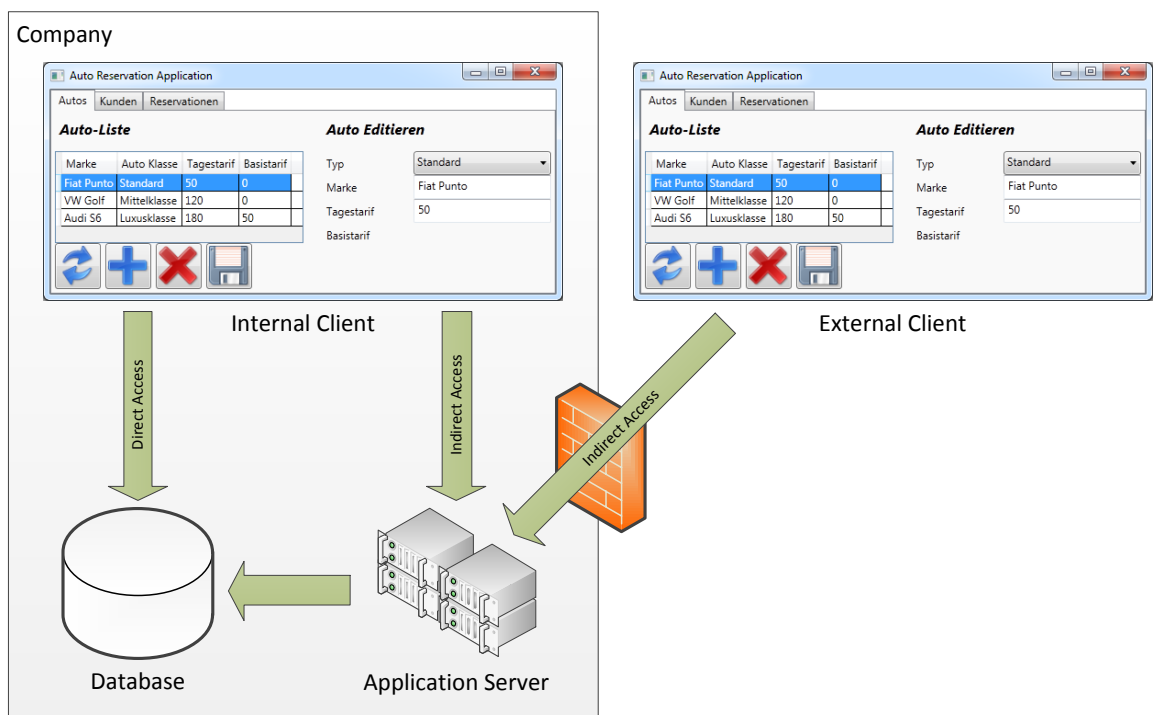
Das Feld „Tagestarif“ muss bei allen Autos erfasst sein, der Basistarif existiert nur für Wagen der Luxusklasse.

2.4 Architektur der Applikation

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 zuzugreifen. 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.



2.4.1 Konfiguration

Das Microsoft .NET Framework verfolgt den Ansatz, dass die Konfiguration (App.config) einer Applikation über die ausgeführte Datei (*.exe / *.dll) vorzunehmen ist. Das bedeutet, ein Connection-String für den Datenbankzugriff muss im vorliegenden Fall in verschiedenen Projekten richtig gesetzt sein.

- AutoReservation.Ui
Wenn das UI auf einer lokalen Objekt-Instanz arbeitet
- AutoReservation.Service.Wcf.Host
Der Server läuft als separater Prozess, muss also auch separat konfiguriert werden
- AutoReservation.Testing / AutoReservation.Ui.Testing
Das Testing-Framework stützt sich auf die Konfiguration der entsprechenden dll. Ausserdem muss in der Klasse `TestEnvironmentHelper` ein separater Connection-String angegeben werden.

Hinweis:

Arbeiten Sie mit SQL-Express, welches standardmässig mit Visual Studio installiert wird, müssen Sie die Connection-Strings zwingend anpassen (um „\SQLEXPRESS“ erweitern).

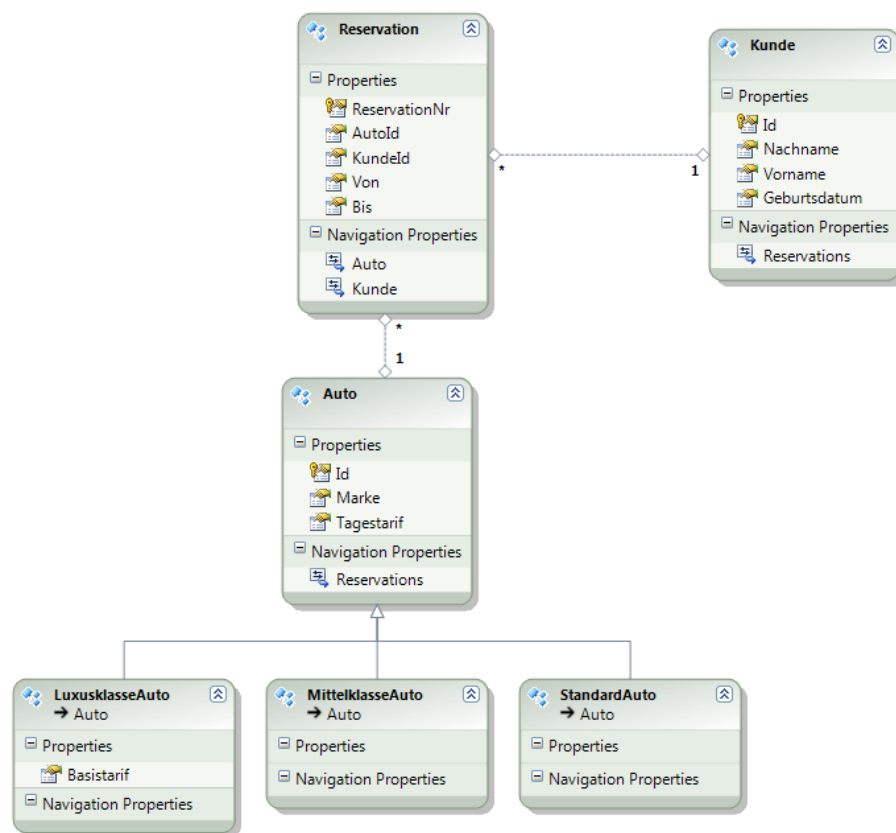
3 Implementation

3.1 Data Access Layer

Projekt „AutoReservation.Dal“

Der Data Access Layer wird mit dem ADO.NET Entity Framework implementiert. Dieser Layer beinhaltet eigentlich nur das *.edmx Datenmodell (z.B. AutoReservation.edmx), 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 (siehe Kapitel Datenbank). Modellieren Sie Ihren Data Access Layer wie oben dargestellt.

Tipp 1:

Wählen Sie beim Importieren “Include Foreign Keys in the Model“

Tipp 2:

Achten Sie darauf, dass Ihr Entity Container Name (Rechte Maustaste auf weisse Fläche im *.edmx Designer / Properties / Entity Container Name) so lautet:
AutoReservationEntities

Tipp 3:

Ändern Sie die Entity Set Names auf: Autos, bzw. Reservationen, bzw. Kunden

Tipp 4:

Achten Sie darauf, dass der Primärschlüssel der Reservation „ReservationNr“ ist. Ansonsten wird der DtoConverter nicht korrekt funktionieren.

Tipp 5:

Setzen Sie jedem Property des Models den Concurrency Mode auf „Fixed“, um Optimistic Concurrency zu aktivieren (ausgenommen Property „Basistarif“).

3.2 Gemeinsame Komponenten

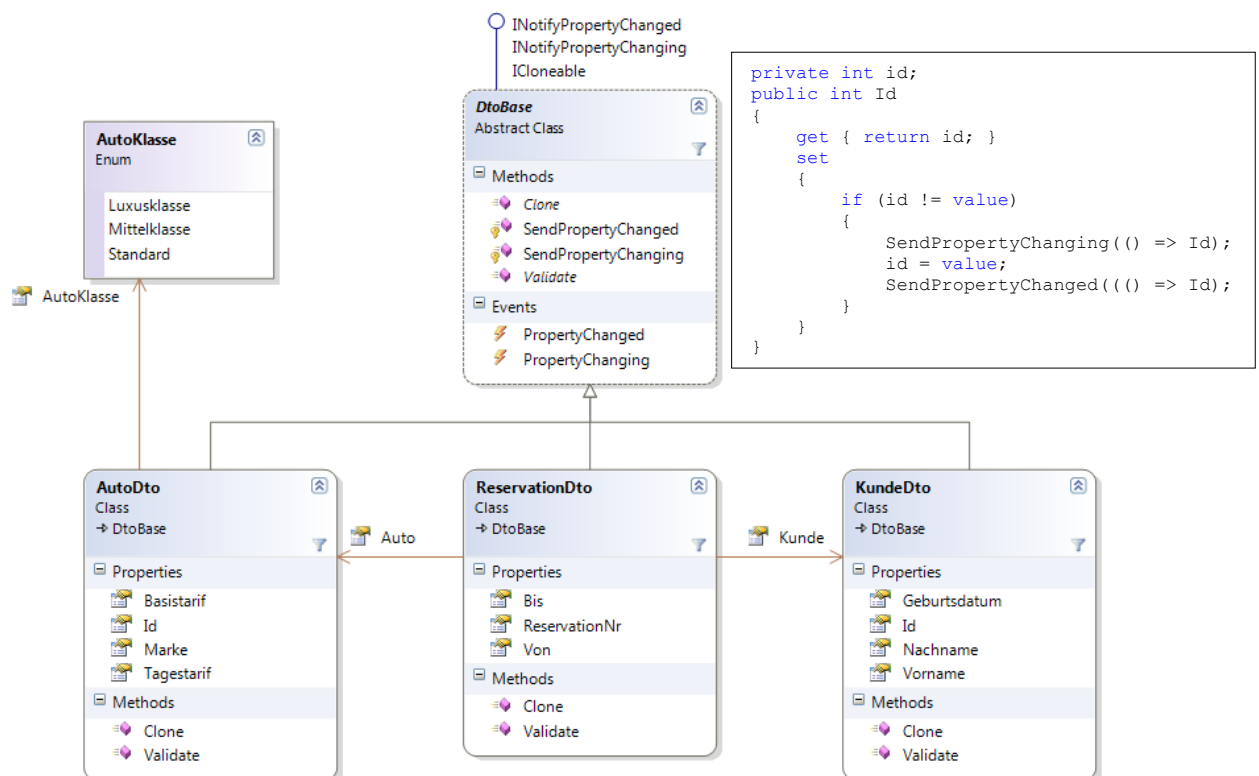
Projekt „AutoReservation.Common“

3.2.1 Datentransferobjekte

Für jede Entität im Data Access Layer wird ein DTO für die Übertragung zwischen Client und Server erstellt. Erstellen Sie eine Implementation, welches dem unten dargestellten Modell entspricht.

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 `SendPropertyChanged`. Der Vollständigkeit halber soll auch vor der Änderung der Event `PropertyChanged` ausgelöst werden.

Unten finden Sie eine Beispiel-Implementation eines solchen Properties.



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 CRUD¹-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 soll eine Klasse `AutoReservationBusinessComponent` erstellt werden, welche die Business-Operationen implementiert, die für die Implementation des Service-Interface benötigt werden.

Das Handling der `OptimisticConcurrencyException` – 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.

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.

Codebeispiel ADO.NET Entity Framework:

```
// Insert
context.AddToAuto(autoInsert);

// Update
context.Autos.Attach(autoOriginal);
context.Autos.ApplyCurrentValues(autoModified);

// Delete
context.Autos.Attach(autoDelete);
context.Autos.DeleteObject(autoDelete);
```

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“ (standardmässig deaktiviert)

Das GUI setzt auf Standard WPF-Komponenten und ist nach dem MVVM-Prinzip² aufgebaut. Durch den Einsatz von WPF mit Data Binding und MVVM ist es möglich, das UI praktisch ohne C# Code zu implementieren. Die gesamte Logik ist schlussendlich 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-Methode ist `IAutoReservationService`.

Die Factory wird konfiguriert über die `Settings.settings` Datei im UI-Projekt. Das Setting „BusinessLayerType“ entspricht dem Assembly Qualified Name³ des konkreten Creators:

AutoReservation.Ui.Factory

.LocalDataAccessCreator.cs, ...

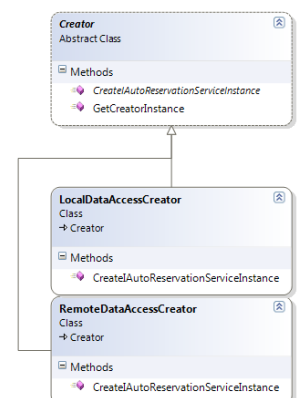
(Konfiguration für direkten Datenzugriff ohne WCF)

.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 „GetCreatorInstance“ benutzen.

```
Type businessLayerType = Type.GetType(Settings.Default.BusinessLayerType);
return (Creator) Activator.CreateInstance(businessLayerType);
```



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 vorgegebenen Projekt existiert schon eine Klasse `TestEnvironmentHelper`, welche Ihnen die Initialisierung der Testumgebung abnimmt. Es werden in jeder Testmethode die gleichen drei Autos, vier Kunden und eine Reservation erstellt.

Achtung:

Je nach Datenbank-Installation muss der Connection-String in der Klasse `TestEnvironmentHelper` noch angepasst werden.

4.1.1 Business-Layer

Projekt „AutoReservation.Testing“ / Klasse „BusinessLayerTest“

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.Testing“ / Klassen „ServiceTest____“, „ServiceTestBase“

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

Die Funktionalität muss einmal als Test auf eine lokale Klasseninstanz (`new AutoReservationService()`) und einmal als Test auf einen WCF-Service (via `ChannelFactory<T>`) getestet werden. So kann sichergestellt werden, dass das Verhalten der Serialisierungsmachanismen von WCF abgedeckt sind.

Tipp:

Verwenden Sie dazu die bereits bestehende Klasse `ServiceTestBase`. Die erforderlichen Testmethoden sind bereits angegeben.

4.1.3 GUI Layer

Projekt „AutoReservation.Ui.Testing“ / Klasse „ViewModelTest“

Ausserdem müssen Sie für die ViewModels folgende Tests implementieren:

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

Hier gilt zu testen, ob die CanLoad-Methode den erwarteten Wert liefert und ob nach dem Ausführen der Load-Methode die Daten in das ViewModel geladen wurden.