HSR
HOCHSCHULE FÜR TECHNIK
RAPPERSWIL

FHO Fachhochschule Ostschweiz

Microsoft .NET Übungen

Mini-Projekt Auto-Reservation, v9.0

Inhaltsverzeichnis

Manuel Bauer
HSR Hochschule für Technik Rapperswil ■ Oberseestrasse 10 ■ CH-8640 Rapperswil
T +41 55 222 41 11 ■ F +41 55 222 44 00 ■ office@hsr.ch ■ www.hsr.ch

1 Aufgabenstellung / Administratives

1.1 Einführung

Sie bekommen als Vorgabe eine Visual Studio Solution, welche einen strukturell einen kompletten Service Stack abbildet. Die Applikation ist aber noch nicht implementiert. Die Server-Applikation implementiert Service-Methoden für die Verwaltung von Auto-Reservationen einer Autoverleih-Firma.

Die Aufgabe wurde mit dem WPF-Teil aus dem Modul «Mobile- and GUI-Engineering» abgestimmt, wo ebenfalls ein Miniprojekt in Form eines WPF User Interface durchgeführt wird. Die beiden Projekte bauen auf einander auf und können nahtlos kombiniert werden.

Ziele dieser Aufgabenstellung:

- Breite Anwendung von Technologien aus der Vorlesung
- Verteilte Applikationen mit Datenbank-Zugriff konzipieren und umsetzen
- Diskussionen über Software-Architektur anregen

Teams:

Das Projekt soll in 2er Teams durchgeführt werden. Einzelarbeiten und 3er Teams sind in Ausnahmefällen möglich. Wir werden in den Übungen eine Einschreibeliste („Gruppeneinteilung Microsoft-Technologien“) auflegen, in welcher sich jede Gruppe einschreiben und den gewünschten Abnahmetermin wählen kann.

1.2 Arbeitspakete

Die Applikation besteht aus mehreren Schichten, welche innerhalb der Gruppe in folgende Deliverables aufgeteilt werden. Es ist den Studierenden grundsätzlich freigestellt, ob die Implementation bottom-up oder top-down erfolgt.

Data Access Layer

1. Implementieren Sie den DAL mit dem Entity Framework Code First
2. Stellen Sie sicher, dass das resultierende Datenbankschema dem der Aufgabenstellung entspricht

Business Layer

1. Implementieren Sie den Business-Layer mit den CRUD-Operationen und diversen Checks. Die Update-Operationen sollen Optimistic-Concurrency unterstützen.
2. Schreiben Sie die geforderten Tests für den Business-Layer.

Service Layer

1. Definieren Sie das Service-Interface und die Datentransferobjekte (DTO’s).
2. Implementieren Sie die Service-Operationen.
   Der Service-Layer ist verantwortlich für das Konvertieren der DTO’s in Entitäten und umgekehrt sowie das Versenden der Fault-Exceptions.
3. Schreiben Sie die geforderten Tests für den Service-Layer.

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. Gehen Sie daher immer das ganze Kapitel durch, bevor Sie mit der Implementation starten.

1.3 Abgabe (letzte Semesterwoche)

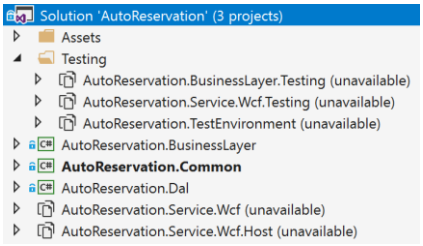
In der letzten Semesterwoche finden die Abgaben gemäss dem Plan „Gruppeneinteilung MsTe Miniprojekt“ statt. Bei der Abgabe des Miniprojektes muss jede Gruppe pünktlich zum Abnahmetermin eine lauffähige Version ihrer Implementation auf einem Übungsrechner oder privaten Notebook bereitstellen können, damit ein reibungsloser Ablauf und die Einhaltung des Terminplans gewährleistet werden kann. Etwas Verzögerung sollte mit eingerechnet werden.

Eine erfolgreiche Bewertung ist die Voraussetzung für die Zulassung zur Modulschlussprüfung. Es wird rechtzeitig eine Check-Liste für die Bewertung der Resultate veröffentlicht.

2 Erläuterungen

2.1 Analyse der Vorgabe

Öffnen Sie die Solution „AutoReservation.sln“ und analysieren Sie den bereits vorhandenen Code und dessen Struktur.



Einige Projekte innerhalb der Solution werden «unloaded» abgegeben. Diese sind ausgegraut und mit «(unavailable)» markiert. Diese Projekte können via Rechtsklick > Reload Project wieder geladen werden. Das Entladen eines Projektes führt dazu, dass Sie vom Compiler nicht mehr berücksichtigt werden.

Dies ist genau die Absicht, da die betroffenen Projekte (noch) nicht kompilierfähig sind, zum Beispiel, weil bestimmte Basisfunktionalitäten und Convinience-Features schon mitgeliefert werden.

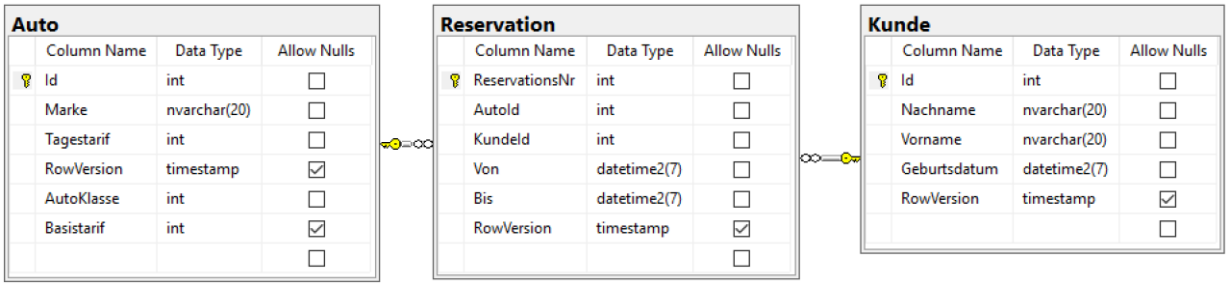
Folgende Projekte sind in der abgegebenen Solution enthalten:

Projekt	**Beschreibung**
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 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.*.Testing	Testprojekt zur jeweiligen Schicht.

2.2 Datenbankstruktur

Die Datenbankstruktur ist relativ einfach gehalten. Es gibt zwei Tabellen mit Stammdaten (Auto, Kunde) sowie eine Transaktionstabelle (Reservation), welche Autos und Kunden in Form einer Reservation verknüpft.

Das Erstellen der Datenbank soll nicht per Hand oder vorgegebenem Skript passieren, sondern automatisch durch Entity Framework Code First passieren (siehe weiter unten). Die resultierende Struktur muss, abgesehen von der Feld-Reihenfolge, genau dem untenstehenden Diagramm entsprechen. Dies umfasst Name, Datentypen, Nullable, Schlüssel, Fremdschlüssel, etc. Abweichungen müssen mit dem Betreuer abgesprochen werden.



Von einem Auto existieren drei Ausprägungen. Diese werden über die Spalte „AutoKlasse“ unterschieden. Das Feld „Tagestarif“ muss bei allen Autos erfasst sein, der „Basistarif“ existiert nur für Wagen der Luxusklasse, ist für diese aber zwingend.

Diese Werte sind für die Unterscheidung vorgesehen:

Wert	Typ
0	Luxusklasse
1	Mittelklasse
2	Standard

2.2.1 Erstellen der Datenbankinstanz

Als Datenbank wird eine lokale Microsoft SQL Server LocalDB Instanz verwendet. Diese wird mit Visual Studio automatisch mit installiert. Damit auf allen Rechnern des Teams der gleiche Connection String verwendet werden kann, führen Sie bitte folgendes Script aus:

Assets\AutoReservation.CreateInstance.bat

Prüfen Sie nach dem Ausführen des Scripts den Output auf der Kommandozeile. In der Liste der lokalen Microsoft SQL Server LocalDB Instanzen sollte nun mindestens „MSSQLLocalDB“ erscheinen.

2.2.2 Konfiguration & 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 die Konfiguration nicht in allen Projekten einzeln gemacht werden muss, ist eine allumfassende App.config-Datei im Solution-Ordner vorhanden. In den einzelnen Projekten wird diese Datei dann als Link referenziert.

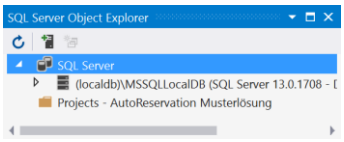
In Bezug auf den Connection String empfiehlt es sich, dass sämtliche Gruppenmitglieder sich auf einen spezifischen Connection String einigen. Ansonsten wird der Austausch des Quellcodes schwierig. Bei Schwierigkeiten mit Suche des richtigen Connection Strings kann folgende Seite konsultiert werden: http://www.connectionstrings.com/.

2.2.3 Zugriff auf die Datenbank

Es gibt mehrere Varianten um auf die Datenbank zuzugreifen und diese zu verwalten:

Microsoft Visual Studio

Über den Menüpunkt View > SQL Server Object Explorer. Unter dem SQL Server sollte nun mindestens (localdb)\MSSQLLocalDB erscheinen. Falls nicht gelistet, müsste dieser über das Plus-Symbol oben hinzugefügt werden.



Microsoft SQL Server Management Studio

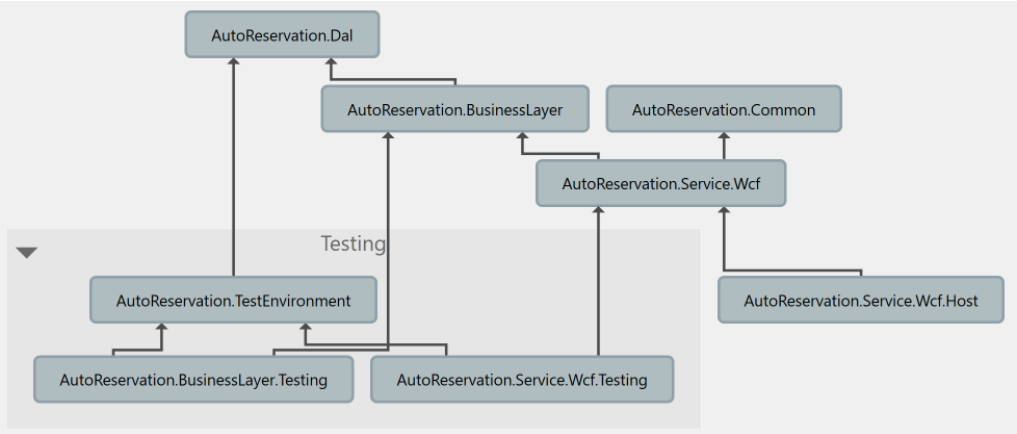
Dieses gehört zur Produktpalette von Microsoft SQL Server und kann in der neuesten Version als stand-alone Installer heruntergeladen werden. Sämtliche SQL Server Features werden von diesem Tool unterstützt.

Weitere Tools

Nebst den offiziellen Tools existiert eine Vielzahl von Drittanbieter-Software. Mit dieser kann selbstverständlich auch gearbeitet werden, Unterstützung kann aber hier nur bedingt gegeben werden.

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 / Projekte der abgegebenen Solution zu finden.



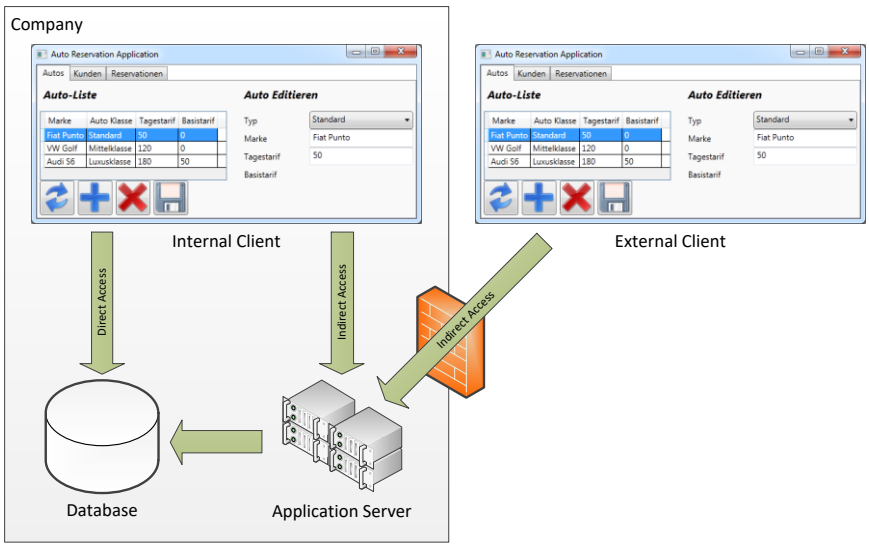
2.3.1 Test-Projekte

Aus obigem Diagramm ist ersichtlich, dass für den Business- und den Service-Layer je ein Test-Projekt existiert. Die in der Aufgabenstellung verlangten Testklassen sind bereits vorgegeben.

Allgemeine Testing-Funktionalität ist im Test-Environment-Projekt vorhanden. Auf den Einsatz eines Mocking-Frameworks wurde der Übersichtlichkeit halber verzichtet. Weitere Angaben zum Testing sind in Kapitel 3.5.2 zu finden.

2.3.2 Topologie

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. In der vorliegenden Aufgabenstellung äussert sich dies darin, dass die zu schreibenden Service-Tests jeweils immer einmal auf einer Objektinstanz und einmal als über einen Service Client Proxy ausgeführt werden sollen.



2.4 Alternativen

In Kapitel 5 Alternativen / Weitere Möglichkeiten werden noch mögliche alternative Ansätze / weitere Möglichkeiten erwähnt, die in Betracht gezogen werden können. Diese sind aber optional und sind für Studenten gedacht, die C# / .NET bereits besser kennen. Es wird empfohlen, dass das Projekt zuerst soweit gelöst wird, dass die Bewertungskriterien erfüllt sind und erst dann Erweiterungen eingebaut werden.

3 Implementation

3.1 Optimistic Concurrency

Die in den nachfolgenden Kapiteln beschriebenen Update-Methoden müssen nach dem Prinzip Optimistic Concurrency[1] implementiert werden. Es existieren verschiedenste Ansätze, um das Problem zu lösen. Die meisten davon erfordern Anpassungen / Erweiterungen im Data Access Layer, einige sogar bis ins GUI. Die drei gängigsten Varianten sind diese:

1. Variante «Timestamp»
   Alle Tabellen erhalten einen Zeitstempel, welcher bei jedem Update automatisch von der Datenbank aktualisiert wird. Beim Speichern wird geprüft, ob der Zeitstempel in der Tabelle noch gleich dem des zu speichernden Objektes ist. Falls nicht, wurde der Datensatz in der Zwischenzeit geändert. Dieser Ansatz wird hier empfohlen.

2. Variante «Original / Modified»
   Das gelesene Objekt wird vor der ersten Änderung geklont. Später beim Speichern werden beide Versionen, die originale und die veränderte, mitgegeben. Durch das originale Objekt kann festgestellt werden, ob der Datensatz in der Zwischenzeit geändert hat. Das Klonen kann auf verschiedenen Layers passieren:
   User Interface, Service Layer oderBusiness Layer

3. Variante «Client-side Change Tracking»
   Änderungen werden clientseitig aufgezeichnet und dem Service-Interface in einer vollständigen Änderungsliste mitgeteilt. Undo-Redo-Funktionalität ist in dieser Variante praktisch ohne Mehraufwand dabei.

Die vorliegende Vorgabe wurde unter Verwendung von Variante 1 gebaut. Die Begründung liegt bei der Einfachheit der Lösung. Hier eine kurze Beurteilung der verschiedenen Ansätze:

Var.	Beurteilung
1.	Relativ geringer Aufwand. Es muss aber berücksichtigt werden, dass der Zeitstempel immer korrekt über die Schichten weitergegeben wird und nie verloren/vergessen geht.
2.	Ebenfalls relativ einfach. Das Klonen gibt aber etwas Aufwand bei der Implementation und auch zur Laufzeit. a. Klonen passiert erst da, wo auch effektiv Änderungen am Objekt passieren. Dafür muss beim Speichern das originale und das veränderte Objekt durchgereicht werden. b. Dem Service muss einen Cache eingebaut werden. Dies kann aber sehr schnell viel Ressourcen verbrauchen. Ausserdem muss der Cache ständig aktualisiert werden. Dies wird sehr schnell sehr kompliziert. c. Siehe b.
3.	Client-seitiges Change Tracking (Self Tracking Entities) sind zwar sehr elegant, aber auch nur mit viel Aufwand umzusetzen.

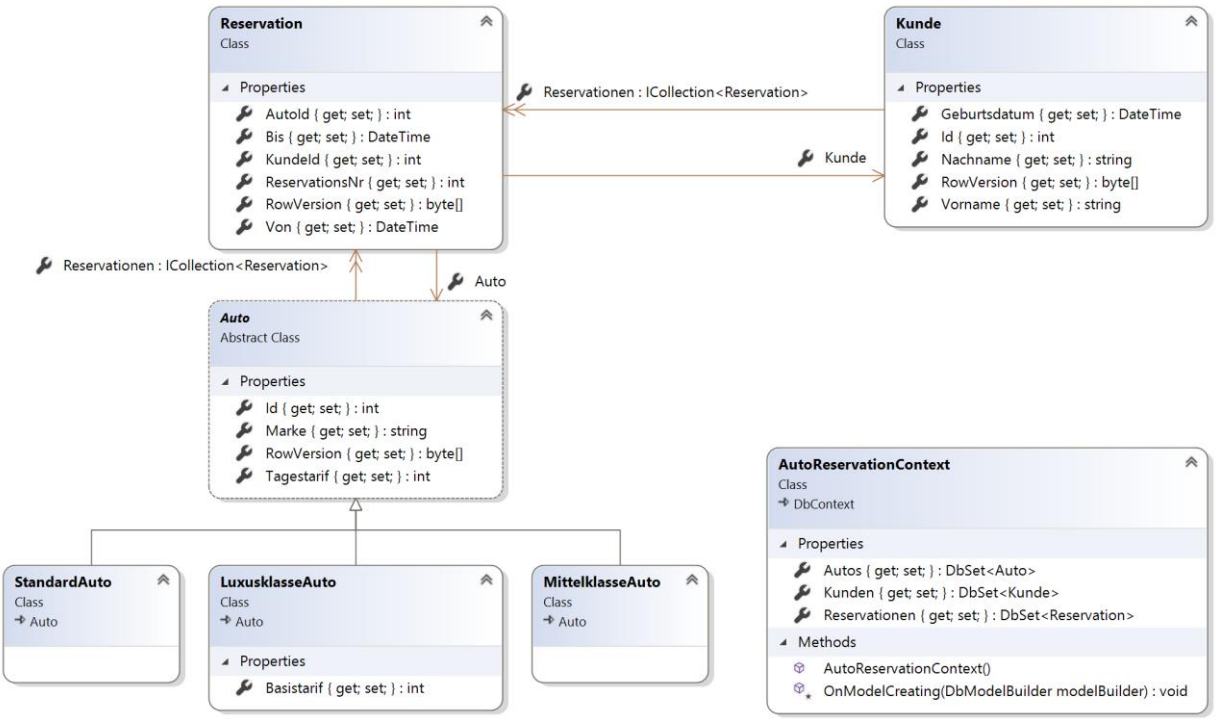
[1] Optimistic Concurrency: https://blogs.msdn.microsoft.com/marcelolr/2010/07/16/optimistic-and-pessimistic-concurrency-a-simple-explanation/

3.2 Data Access Layer (AutoReservation.Dal)

Dieser Layer beinhaltet die Datenzugriffsschicht und sollte mit Entity Framework Code First implementiert werden. Die Idee ist, das Datenmodell sowie den Database Context (Subklasse von DbContext) zu entwickeln und die Datenbank danach basierend auf dem Modell erstellen zu lassen. Vergleichen Sie die erzeugte Struktur mit dem oben vorgegebenen Datenmodell.

Das Datenmodell ist unten bereits dargestellt. Darin ist ersichtlich, dass die Unterscheidung der Auto-Klasse nicht mehr über das Feld `AutoKlasse` erfolgt. Im Modell der Business Entities wird dies durch drei von Auto abgeleitete Klassen realisiert. Die `AutoKlasse` dient implizit im Modell als Diskriminator und verschwindet somit aus der Klasse `Auto`.

Der `DbContext` namens `AutoReservationContext` dient als Zugriffspunkt auf die Datenstruktur.



Wichtige Hinweise:

- Sollten Sie die Klasse `AutoReservationContext` umbenennen, muss dies in den App.config-Dateien (Section `<connectionStrings>`) nachgetragen werden
- Unterscheiden sich die Namensgebung oder Struktur der Entitäten vom Diagramm, müssen die Abweichungen in der Klasse `DtoConverter` angepasst werden

3.3 Business Layer (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 die unten definierte Business Logik implementiert werden. Folgende Operationen müssen für alle drei Entitäten zur Verfügung gestellt werden. Beachten Sie auch nachfolgende Requirements an den Business Layer in den Sub-Kapiteln.

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

Hilfs-Codefragmente für das ADO.NET Entity Framework:

```
// Insert
context.Entry(auto).State = EntityState.Added;
// Update
context.Entry(auto).State = EntityState.Modified;
// Delete
context.Entry(auto).State = EntityState.Deleted;
```

3.3.1 Requirement «Optimistic Concurrency Update»

Jede Update-Methode muss nach dem Optimistic Concurrency-Prinzip implementiert werden. Entity Framework löst bei einer Optimistic Concurrency-Verletzung eine Exception vom Typ `DbUpdateConcurrencyException` aus. Im Falle des Auftretens einer solchen Exception soll eine `OptimisticConcurrencyException` (existiert bereits) geworfen, welche die neuen in der Datenbank vorhandenen Werte beinhaltet.

Für das Handling dieser Exception kann die bereits bestehende Methode `ManagerBase.CreateOptimisticConcurrencyException(…)` direkt im catch-Block aufgerufen und weiter geworfen werden.

3.3.2 Requirement «Date Range Check»

Eine Reservation muss 24 Stunden oder mehr dauern. Prüfen Sie dies und stellen Sie auch sicher, dass das «bis» Datum nicht nach dem «von» Datum liegt. Falls nicht muss eine `InvalidDateRangeException` ausgelöst werden (noch nicht implementiert).

Kapseln Sie die Logik in eine separate Methode, damit das Schreiben von Tests (siehe unten) einfacher bleibt.

3.3.3 Requirement «Availability Check»

Beim Erstellen / Updaten einer Reservation muss geprüft werden, ob ein Auto für den gewünschten Zeitraum zur Verfügung steht. Autos können nahtlos (Ende = Start), aber nicht überlappend gebucht werden. Falls ein Auto nicht verfügbar ist, muss eine `AutoUnavailableException` ausgelöst werden (noch nicht implementiert).

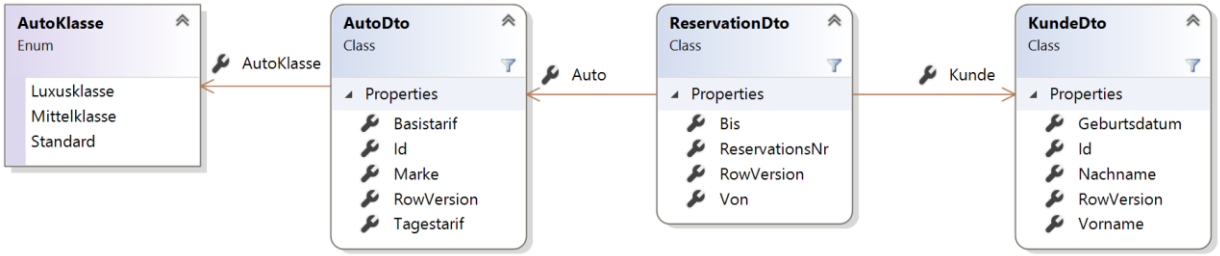
Kapseln Sie die Logik in eine separate Methode, damit das Schreiben von Tests (siehe unten) einfacher bleibt.

3.4 Gemeinsame Komponenten (AutoReservation.Common)

3.4.1 Datentransferobjekte

Um eine saubere Entkopplung der einzelnen Layers zu erhalten, werden so genannte Datentransferobjekte (kurz DTO) eingefügt. Dadurch kann vermieden werden, dass durch die komplette Applikation hindurch eine Abhängigkeit zum DAL besteht. Die Umsetzung erfordert jedoch, dass für jede Entität ein entsprechendes DTO bereitgestellt wird.

Im untenstehenden Modell ist zu beachten, dass ein Enumerator `AutoKlasse` eingeführt wurde. Damit wird die auf dem Data Access Layer definierte Vererbungshierarchie wieder flach gedrückt.



3.4.2 Service-Interface

Das Service-Interface `IAutoReservationService` definiert die Funktionalität für den Service-Layer. Für jede Entität – Auto, Kunde und Reservation – müssen folgende CRUD[2]-Operationen unterstützt werden:

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

Damit die Verfügbarkeit eines Autos auch im Vorfeld abgeklärt werden kann, muss eine zusätzliche Methode auf dem Interface definiert werden. Dadurch lässt sich vermeiden, dass zu viele Fehler provoziert werden.

Der Rückgabewert der definierten Methoden kann `void`, ein Primitivtyp, ein DTO respektive eine Liste davon sein, nie aber eine Entität des Data Access Layers! Anders ausgedrückt, das Interface unterhält keinerlei Referenzen auf den Data Access Layer.

[2] CRUD: Create, Read, Update, Delete

3.5 Service Layer (AutoReservation.Service.Wcf)

Der Service Layer ist die eigentliche WCF-Serviceschnittstelle (Klasse `AutoReservationService`) und implementiert das Interface `IAutoReservationService`. Dieser Layer operiert auf den Business Layer Klassen. Der Service-Layer ist in dieser einfachen Applikation nicht viel mehr als ein «Durchlauferhitzer». In grösseren Projekten kann hier aber durchaus noch Funktionalität – z.B. Sicherheitslogik – enthalten sein. Nichts desto trotz muss er hier zwei wichtige Aufgaben erfüllen:

- Das Umwandeln von DTO's (Common) in Entities (DAL) und umgekehrt
- Bekannte Exceptions fangen und in «Faults» verpacken (z.B. Date Range, etc.)

Achtung:
Für den Start eines Service-Host benötigen Sie 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 angestossen werden, damit auf der Konsole des Service immer ausgegeben wird, welche Operationen ausgeführt wurden.

3.5.1 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.ToList();
List<AutoDto> autoDtoList = autoList.ConvertToDtos();
autoList = autoDtoList.ConvertToEntities();
```

3.5.2 Faults

Behandeln Sie sämtliche in den Requirements definieren Spezialfälle. Das heisst, es müssen alle auf dem Business Layer geworfenen Exceptions gefangen und in einen Fault verpackt werden. Damit der Client später den Fehler typsicher unterscheiden kann, soll für jede Art Exception ein unterschiedlicher Fault definiert werden. Auch das Arbeiten mit Fehlercodes wäre eine denkbare alternative.

4 Tests

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

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 Autos, Kunden und Reservationen inklusive immer gleichbleibender Primärschlüssel.

Dies bedeutet, dass die Tests auf die Primärschlüssel aufgebaut werden dürfen und z.B. davon ausgegangen werden kann, dass ein Auto oder ein Kunde mit der ID = 1 existiert.

Jede vorgegebene Testklasse erbt von `TestBase` welche sicherstellt, dass die Initialisierungslogik durchläuft. Die Datenbank wird beim Durchführen des ersten Tests komplett gelöscht und neu erstellt, danach lediglich noch geleert (aus Performance-Gründen).

Für detailliertere Informationen zu dem verwendeten Test-Framework „xUnit“ siehe https://xunit.github.io.

Die Testmethoden wurden bereits erstellt und müssen nur noch implementiert werden.

4.1 Business-Layer (AutoReservation.BusinessLayer.Testing)

Diese Tests sollen relativ früh implementiert werden und eine gewisse Sicherheit geben, dass die Applikation und vor allem die Datenbank-Verbindung und -Abfragen in ihren Grundzügen funktioniert. Mindestens folgende Tests müssen implementiert sein:

- Update Kunde
- Update Auto
- Update Reservation

Ausserdem müssen folgende im Business Layer erwähnten Requirements genau getestet werden.

4.1.1 Requirement «Date Range Check»

Analysieren Sie, welche Wertebereiche zu Problemen führen und schreiben Sie zu jedem einen sinnvollen Test. Dies sollte mindestens fünf Tests ergeben (2 gültige Kombination 3 ungültige Kombinationen von Datumswerten).

4.1.2 Requirement «Availability Check»

Analysieren Sie, was für unterschiedliche Konstellationen möglich sind und schreiben Sie zu jeder einen Test. Dies sollte mindestens fünf Tests ergeben (2+ gültige Kombinationen, 3+ ungültige Kombinationen).

4.2 Service-Layer (AutoReservation.Service.Wcf.Testing)

Studieren Sie als erstes das vorgegebene Konstrukt in der Projektvorgabe:

Klasse	Beschreibung
ServiceTestBase	Abstrakte Basis-Testklasse. Enthält bereits alle Methoden für die zu testende Funktionalität.
ServiceTestLocal	Konkrete Implementation. Testet die Funktionalität des Service anhand einer lokalen Objektinstanz `new AutoReservationService().`
ServiceTestRemote	Konkrete Implementation. Testet die Funktionalität des Service anhand eines WCF-Client-Proxies. `ChannelFactory<IAutoReservationService>`

Dieses Konstrukt scheint auf den ersten Blick überdimensioniert, erfüllt jedoch so folgende 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

Tests
Am genauesten soll der Service Layer getestet werden. Im Mindesten müssen folgende Operationen für alle drei Entitäten (Autos, Kunden, Reservationen) überprüft werden:

- Abfragen einer Liste
- Suche anhand des Primärschlüssels
- Suche anhand eines ungültigen Primärschlüssels
- Einfügen
- Löschen
- Updates
- Abfrage der Verfügbarkeit eines Auto (je einmal true / false als Antwort)

Ausserdem muss das Fault-Handling darauf hin geprüft werden, ob Exceptions sauber gefangen und in Faults verpackt werden.

- Updates mit Optimistic Concurrency Verletzung auf allen 3 Entitäten
- Insert mit ungültigem Date Range (Reservation)
- Update mit ungültigem Date Range (Reservation)
- Insert mit nicht verfügbarem Auto (Reservation)
- Update mit nicht verfügbarem Auto (Reservation)

Dies ergibt in Summe also im Mindesten 27 Tests für den Service-Layer.

5 Alternativen / Weitere Möglichkeiten

Wie zu Beginn erwähnt, können auch alternative Ansätze verfolgt oder das Projekt noch ausgebaut werden. In den Nachfolgenden Kapitel werden einige Punkte aufgegriffen. Es empfiehlt sich, Abweichungen von der Aufgabenstellung mit dem Betreuer zu diskutieren.

5.1 Allgemein

5.1.1 Async / Await

Seit C# 5.0 ist die asynchrone Programmierung mittels async/await vereinfacht worden. Die Methoden können durch das Projekt hindurch auf async/await umgestellt werden.

5.1.2 Dependency Injection

Bei Interesse darf die Verwendung von Dependency Injection über das gesamte Projekt hinweg erweitern. Weit verbreitet und vielfach auch in den Beispielen von Microsoft verwendet sind Unity[3] sowie das in der Schweiz entstandene Ninject[4]. Weitere Infos dazu kann man unter MSDN[5] finden.

5.1.3 Repository Pattern

Die vorliegende Implementation ist nicht unbedingt sehr kompatibel mit Test-Driven Development. Das Problem ist, dass die Komponenten unter Test schlecht gemockt werden können. Damit dies möglich wäre, müsste mehr gegen Interfaces programmiert werden. Das Repository Pattern[6] ist durch seinen Aufbau an dieser Stelle eine ideale Alternative.

5.2 Datenbank

5.2.1 OR-Mapping

Grundsätzlich kann neben dem Code First Ansatz auch ein Database First Ansatz verwendet werden oder ein anderer OR-Mapper eingesetzt werden.

5.2.2 Vererbung

Wie vielleicht bemerkt wurde, kann mit dem aktuellen Design der Datenbank die `AutoKlasse` nicht verändert werden. Das DTO, welches über die Service-Schnittstelle übermittelt wird, lässt dies zwar zu, aber das Entity Framework wirft eine Exception. Dies hat mit der implementierten Vererbung zu tun. Ändert beim DTO die `AutoKlasse`, wird beim Konvertieren vor dem Speichern eine andere Subklasse von Auto instanziert. Entity Framework lässt aber Updates nur dann zu, wenn das gelieferte Objekt noch dem ursprünglichen Typen entspricht (u.A. wegen möglichem Datenverlust).

Dieses Szenario müsste von Hand abgebildet werden, z.B. durch manuelles Löschen und neu erstellen des Objektes oder über das direkte Ausführen von SQL Statements / Stored Procedures auf der Datenbank.

[3] Unity Project: https://github.com/unitycontainer/unity
[4] Ninject: http://www.ninject.org/
[5] Unity on MSDN: https://msdn.microsoft.com/library/dn178463.aspx
[6] Repository Pattern: http://martinfowler.com/eaaCatalog/repository.html

5.3 Service

Anstelle von WCF bietet die .NET Plattform noch andere Services, um Daten zu verarbeiten. Als Alternative könnten auch „ADO.NET Data Services“ (siehe WCF-Unterlagen) oder auch „ASP.NET Web API“ verwendet werden.

5.4 Testing

5.4.1 Mocking

Um nicht gegen eine Datenbank testen zu müssen (unter Puristen verpönt) sollten Mocks eingesetzt werden. Dies verringert die Abhängigkeit zu Systemkomponenten wie z.B. Datenbank, Dateisystem, Netzwerkverbindungen, uva.

Es gibt diverse Mocking-Frameworks in .NET. Weit verbreitet ist Moq[7].

5.4.2 Test-driven Development (TDD)

Es werden in diese Projekt Tests gefordert, damit Sie sich auch in C# / .NET mit dieser Disziplin auseinandergesetzt haben. Wer einen Schritt weitergehen möchte, kann sich auch gerne im Test-Driven Development (TDD) üben und diese Vorgehensweise wählen.

Die geforderten Tests sind lediglich ein Minimum und können noch ausgebaut werden. Der Einsatz von Mocks wäre mit dieser Herangehensweise empfehlenswert.

[7] Moq: https://github.com/Moq/moq4