HSR
HOCHSCHULE FÜR TECHNIK
RAPPERSWIL

FHO Fachhochschule Ostschweiz

Microsoft .NET Übungen

Mini-Projekt Auto-Reservation, v7.3

Inhaltsverzeichnis

Hansjörg Huser ■ Jürg Jucker ■ 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 einfache WPF-Applikation für die Verwaltung von Auto-Reservationen einer Auto-Reservations-Firma.

Es handelt sich hierbei um eine Multi-Tier-Applikation mit einer WCF-Schnittstelle. Als User-Interface existiert ein WPF-Projekt, welches bereits fertig implementiert abgegeben wird.

Ziele:

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

User-Interface (bereits implementiert)

1. Passen Sie das User-Interface so an, dass es mit Ihrem Service-Interface funktionert.
2. Analysieren Sie den Code des User-Interfaces und der ausgelieferten Tests.

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.

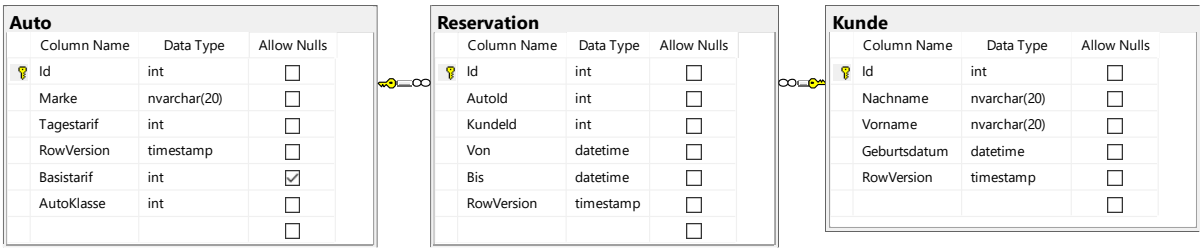
Folgende Projekte sind in der abgegebenen Solution enthalten:

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.Ui	WPF-Applikation Dialogen zum Anzeigen und Einfügen von Autos bzw. Kunden sowie zum Anzeigen, Einfügen und Löschen von Reservationen.

2.2 Datenbankstruktur

Die Datenbankstruktur ist relativ einfach gehalten. Es gibt zwei Tabellen mit Stammdaten (Auto, Kunde) sowie eine Tabelle, 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/.

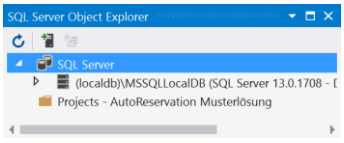
2.2.3 Zugriff auf die Datenbank

Es gibt mehrere Varianten um auf die Datenbank zuzugreifen:

- Microsoft Visual Studio (empfohlen!)
- Microsoft SQL Server Management Studio (Express)
- Weitere Tools

Microsoft Visual Studio

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

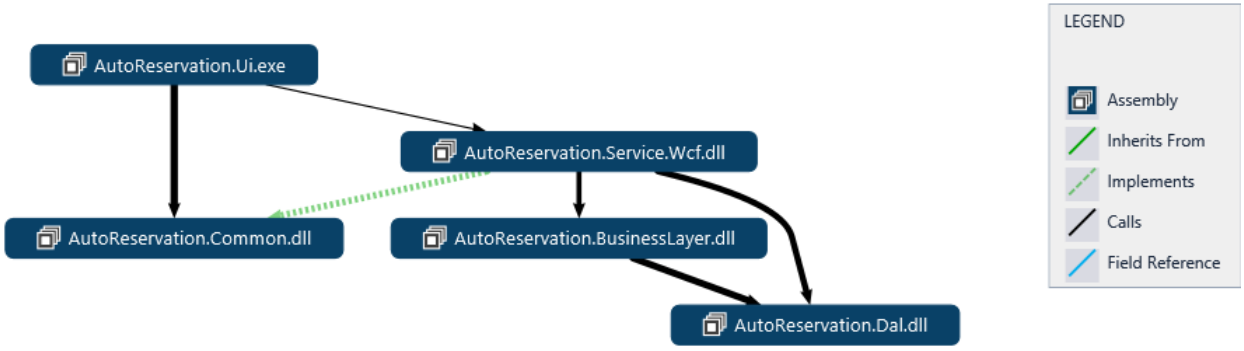


Microsoft SQL Server Management Studio (Express)
Dieses Tool wird mit der Vollversion von Microsoft SQL Server ausgeliefert und unterstützt sämtliche SQL Server Features. Die Express Version ist vom Funktionsumfang her irgendwo zwischen Visual Studio und der Vollversion von Management Studio anzusiedeln.

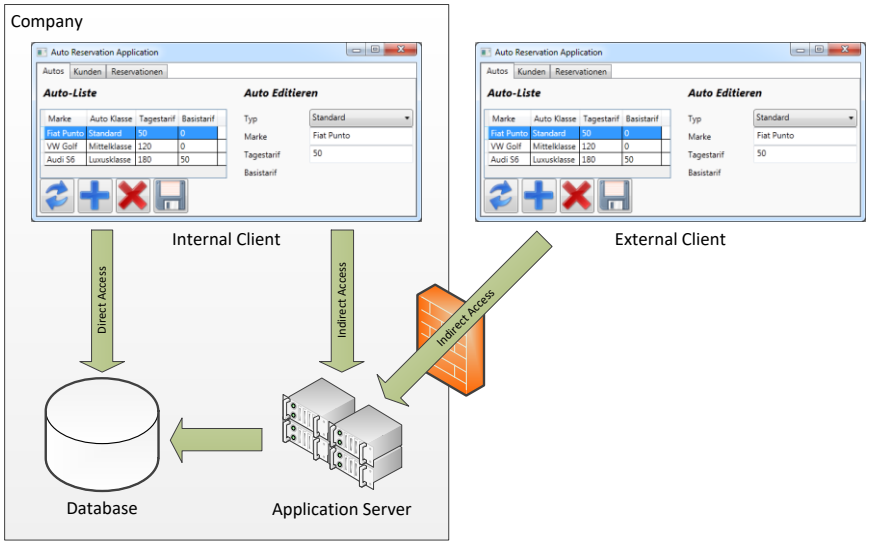
Weitere Tools
Nebst den offiziellen Tools existiert eine Vielzahl von Drittanbieter-Software. Mit dieser kann selbstverständlich auch gearbeitet werden, Unterstützung kann aber hier nur bedingt gegeben werden.

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.

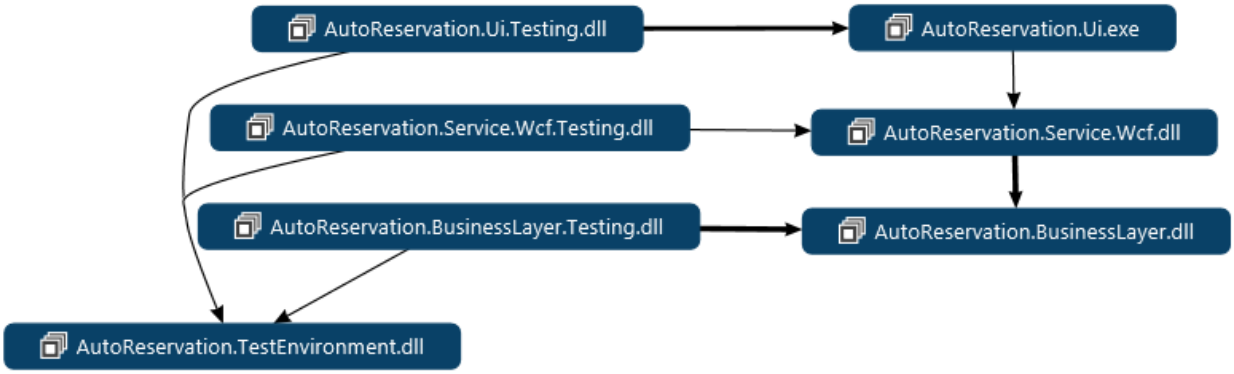


Ein Ziel der Applikation ist es, den Client wahlweise direkt oder indirekt über einen Applikationsserver auf die Datenbank zugreift. Dies dient dem Zweck, den Client für einen Aussendienstmitarbeiter oder einen internen Sachbearbeiter konfigurieren zu können.



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. Weitere Angaben zum Testing sind in Kapitel 4 zu finden.



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.

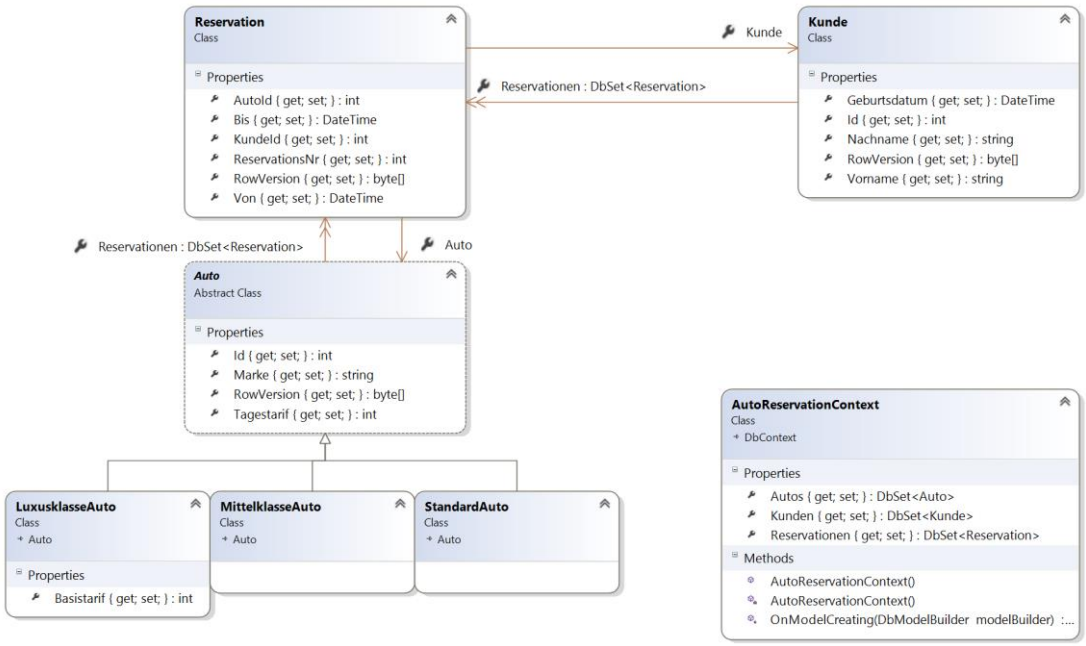
3 Implementation

3.1 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

3.1.1 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:
   a. User Interface
   b. Service Layer
   c. Business 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:

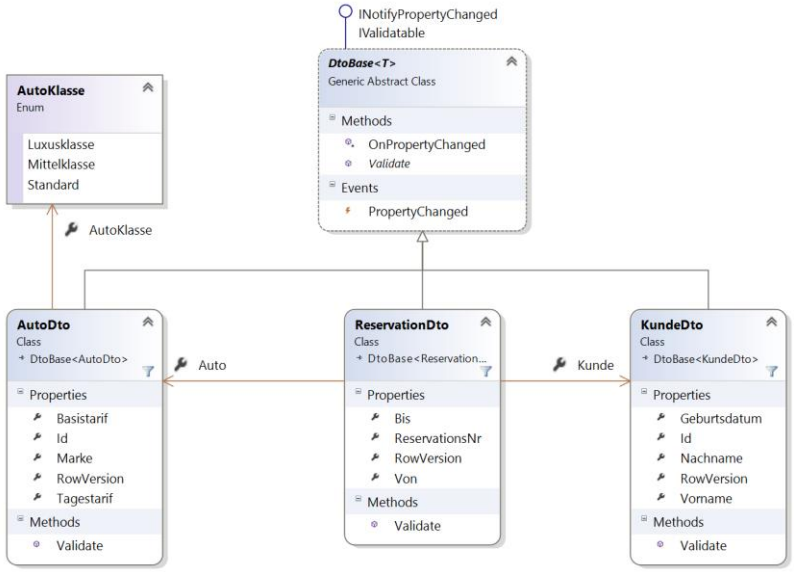
Var.	Beurteilung
1.	Relativ geringer Aufwand. Es muss aber berücksichtigt werden, dass der Zeitstempel immer korrekt über die Schichten weitergegeben wird und nie verloren/vergessen geht.
2.	Ebenfalls relativ einfach. Das Klonen gibt aber etwas Aufwand bei der Implementation und auch zur Laufzeit. a. 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 Gemeinsame Komponenten (AutoReservation.Common)

3.2.1 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.



Die Basisklasse `DtoBase<T>` ist bereits vorgegeben. Sie implementiert zwei Interfaces:

`IValidatable`	Wird im GUI zur Validierung der Integrität des jeweiligen Objekts verwendet.
`INotifyPropertyChanged`	Wird im GUI für das Data Binding benötigt. Alle Property-Setter müssen den `PropertyChanged` Event auslösen. Dazu kann die Methode `OnPropertyChanged` verwendet werden. Hier eine Beispiel-Implementation eines DTO-Properties:

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

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[2]-Operationen unterstützt werden:

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

Der Rückgabewert der definierten Methoden kann `void`, ein Primitivtyp, ein DTO respektive eine Liste davon sein, nie aber eine Entität des Data Access Layers!

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 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 `CreateLocalOptimisticConcurrencyException` (existiert bereits) geworfen, welche die neuen in der Datenbank vorhandenen Werte beinhaltet. Für das Handling dieser Exception kann die bereits bestehende Methode `CreateDbConcurrencyException` direkt im catch-Block aufgerufen und weiter geworfen werden.

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;
```

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

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

3.4 Service Layer (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 angestossen werden, damit auf der Konsole des Service immer ausgegeben wird, welche Operationen ausgeführt wurden.

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

3.5 GUI Layer (AutoReservation.Ui)

Das User Interface wird vollständig implementiert abgegeben. Dieses war früher Teil der Aufgabenstellung und wird nun noch als Veranschaulichungs-Beispiel abgegeben. Eine detaillierte Beschreibung zum GUI finden Sie im Anhang.

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 Autos, Kunden und Reservationen inklusive immer gleichbleibender Primärschlüssel.

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.

Es sollen folgende Operationen für alle drei Entitäts-Typen (Autos, Kunden, Reservationen) mit Tests abgedeckt werden:

- Update Kunde
- Update Auto
- Update Reservation

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

Am genauesten soll der Service Layer getestet werden. Im Mindesten müssen die unten gelisteten Operationen für alle drei Entitäts-Typen (Autos, Kunden, Reservationen) mit Tests überprüft werden:

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

Dies ergibt in Summe also im Mindesten 21 Tests für den Service-Layer.

Studieren Sie 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

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. Für die UI-Factories wird bereits Ninject eingesetzt. Es steht Ihnen aber frei, ein anderes DI-Framework zu verwenden. Weit verbreitet, und vielfach auch in den Beispielen von Microsoft verwendet, ist Unity[3]. Weitere Infos dazu kann man unter MSDN[4] 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[5] 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] Unity on MSDN: https://msdn.microsoft.com/library/dn178463.aspx
[5] 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[6].

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.

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

6 Anhang

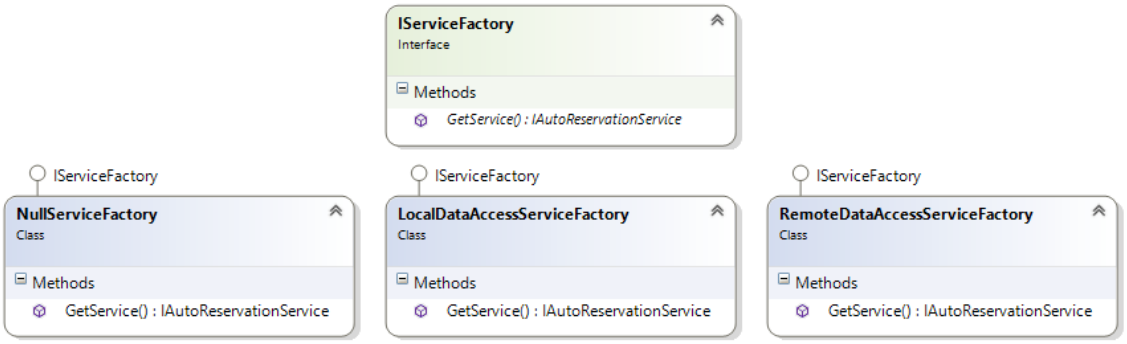
6.1 GUI Layer (AutoReservation.Ui)

Das GUI setzt auf Standard WPF-Komponenten und ist nach dem MVVM-Prinzip[7] 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 Reservationen bereits implementiert. Da WPF nicht mehr Bestandteil dieser Vorlesung ist, wird die View für Kunden sowie das dazugehörige ViewModel bereitgestellt. Das ViewModel muss lediglich analog zum AutoViewModel implementiert werden.

6.1.1 Factory

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



Für die Instanziierung einer Factory wird Dependency Injection (DI) verwendet. Als gängiges Framework in .NET kommt hier Ninject[8] zum Einsatz. Welche Factory ausgewählt werden soll, kann über einen Eintrag in der Dependencies.Ninject.xml-Datei ausgewählt werden. Der einfachheithalber wurden alle Mappings bereits definiert. Es fehlen nur die konkreten Implementationen zu den Factories. Zudem wird bereits eine `NullServiceFactory` Klasse bereitgestellt, die eine leere Implementation einer Factory darstellt.

Die beiden zu implementierenden Factories heissen wie folgt:

AutoReservation.Ui.Factory

.LocalDataAccessServiceFactory.cs
(Konfiguration für direkten Datenzugriff ohne WCF)

.RemoteDataAccessServiceFactory.cs
(Konfiguration für indirekten Datenzugriff via WCF; der WCF Service muss zuvor gestartet werden)

[7] Model-View-ViewModel (MVVM): http://msdn.microsoft.com/en-us/magazine/dd419663.aspx
[8] Ninject: http://www.ninject.org

6.2 GUI Layer Tests (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:

Klasse	**Tests**
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 dies nicht verlangt.