Einführung in Entity Framework Core

Untersuchungen zu Assoziation und Vererbung

Dozent: Dr. Thomas Hager, Berlin,
im Auftrag der Firma GFU Cyrus AG
20.10.2025 – 22.10.2025

In diesem Abschnitt werden typische Elemente der objektorientierten Programmierung untersucht, nämlich inwiefern und auf welche Weise sie sich in Datenbankstrukturen überführen lassen.

Hierzu gehört zunächst die **Assoziation**, also die Darstellung von Beziehungen zwischen Objekten. Es werden 1:n-, n:m- und 1:1-Assoziationen untersucht. Es wird gezeigt, wie sich die Assoziationen mittels der Fluid Api in Datenbank-Relationen abbilden lassen. Ein wichtiger Aspekt hierbei ist das Löschverhalten.

Im Weiteren werden **Vererbung**sbeziehungen zwischen Klassen und Varianten ihrer Abbildung in Datenbankstrukturen diskutiert. Hierzu gehören:

- Table splitting
- Table per hierarchy (TPH)
- Table per type (TPT)
- Table per concrete type (TPC)

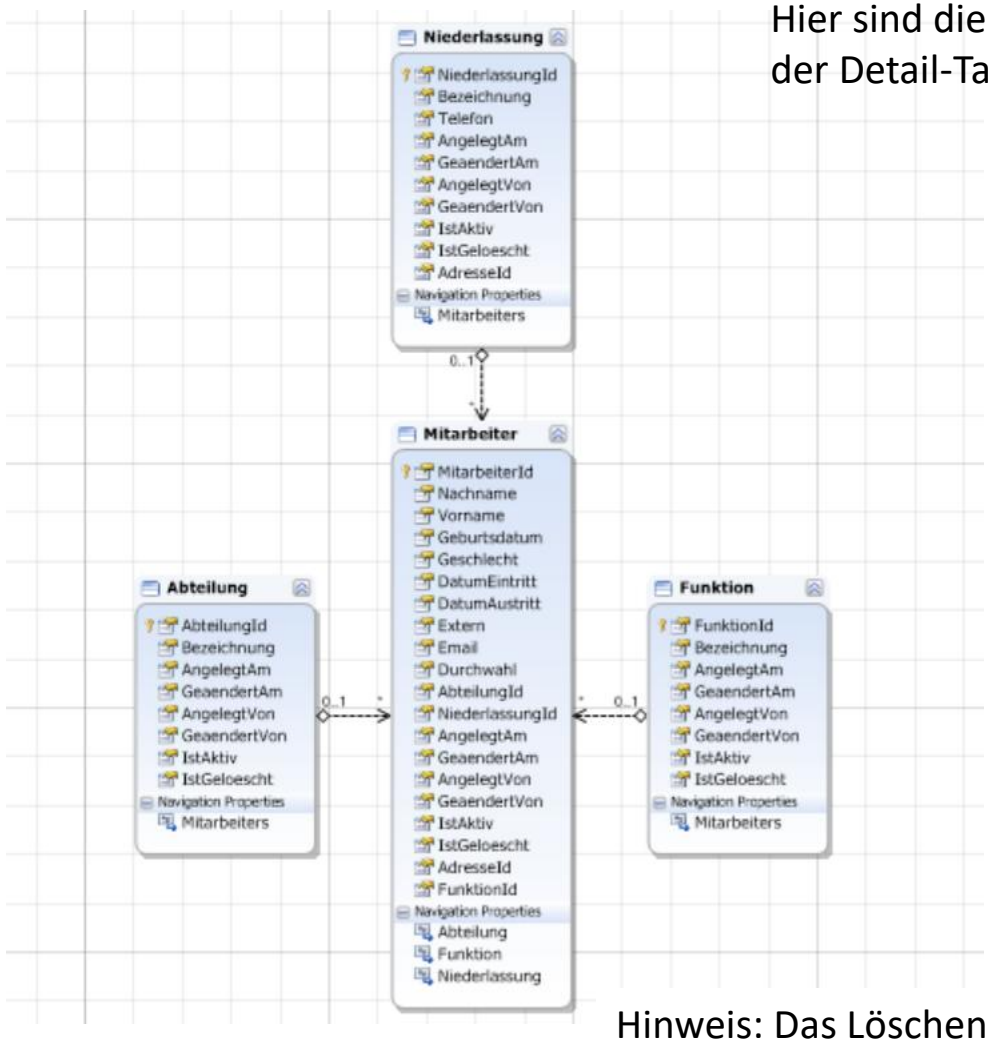
Im Folgenden:

DIE UMSETZUNG DER ASSOZIATIONEN IN RELATIONEN

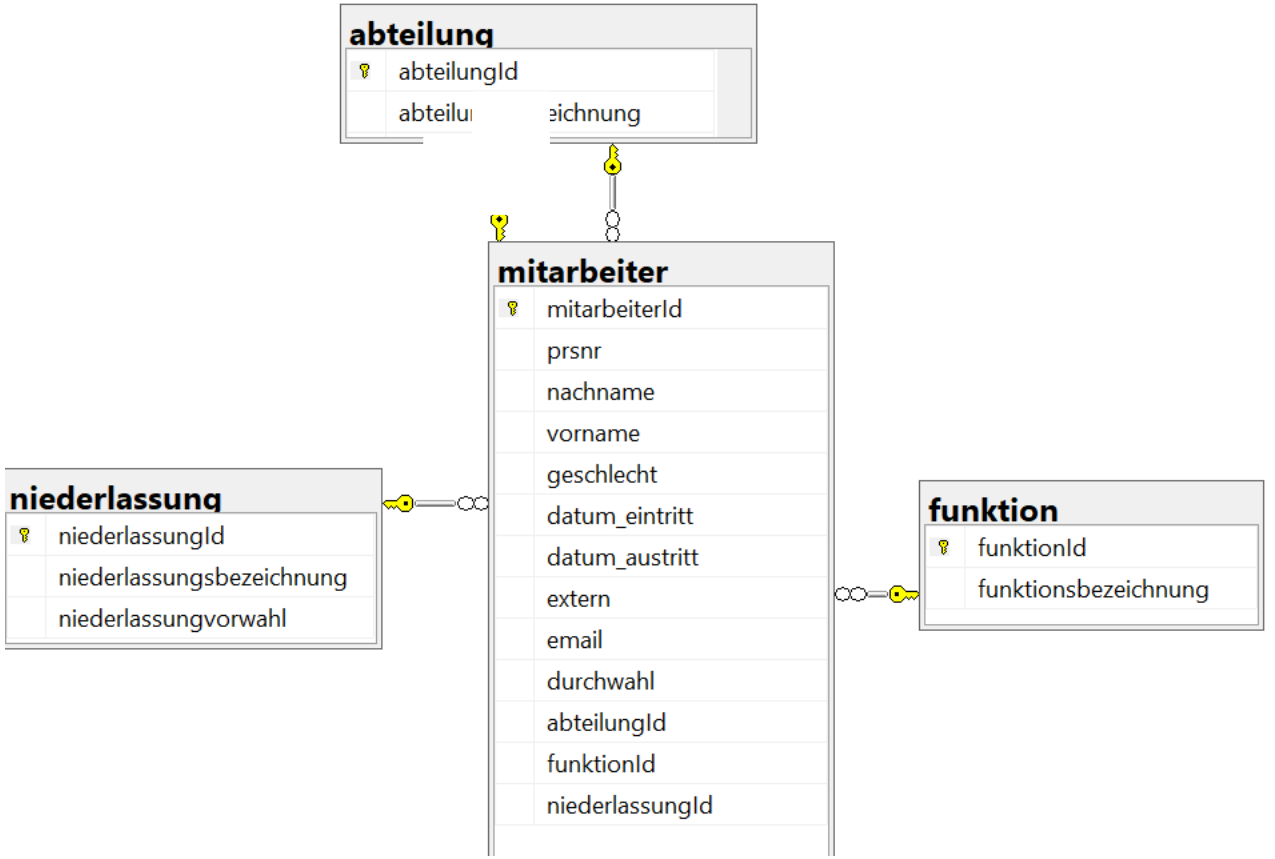
Zur Fluent-Api: Übersicht über DeleteBehavior

Felder3		
Cascade	3	Für Entitäten, die vom Kontext nachverfolgt werden, werden abhängige Entitäten gelöscht, wenn der zugehörige Prinzipal gelöscht wird.
ClientCascade	4	Für Entitäten, die vom Kontext nachverfolgt werden, werden abhängige Entitäten gelöscht, wenn der zugehörige Prinzipal gelöscht wird.
ClientNoAction	6	Hinweis: Es ist ungewöhnlich, diesen Wert zu verwenden. Verwenden Sie stattdessen ClientSetNull , um das Verhalten von EF6 mit deaktivierten kaskadierenden Löschvorgängen abzugleichen.
ClientSetNull	0	Für Entitäten, die vom Kontext nachverfolgt werden, werden die Werte der Fremdschlüsseleigenschaften in abhängigen Entitäten auf NULL festgelegt, wenn der zugehörige Prinzipal gelöscht wird. Dies trägt dazu bei, den Graphen von Entitäten in einem konsistenten Zustand zu halten, während sie nachverfolgt werden, sodass dann ein vollständig konsistentes Diagramm in die Datenbank geschrieben werden kann. Wenn eine Eigenschaft nicht auf NULL festgelegt werden kann, weil es sich nicht um einen Nullable-Typ handelt, wird beim SaveChanges() Aufrufen eine Ausnahme ausgelöst.
NoAction	5	Für Entitäten, die vom Kontext nachverfolgt werden, werden die Werte der Fremdschlüsseleigenschaften in abhängigen Entitäten auf NULL festgelegt, wenn der zugehörige Prinzipal gelöscht wird. Dies trägt dazu bei, den Graphen von Entitäten in einem konsistenten Zustand zu halten, während sie nachverfolgt werden, sodass dann ein vollständig konsistentes Diagramm in die Datenbank geschrieben werden kann. Wenn eine Eigenschaft nicht auf NULL festgelegt werden kann, weil es sich nicht um einen Nullable-Typ handelt, wird beim SaveChanges() Aufrufen eine Ausnahme ausgelöst.
Restrict	1	Für Entitäten, die vom Kontext nachverfolgt werden, werden die Werte der Fremdschlüsseleigenschaften in abhängigen Entitäten auf NULL festgelegt, wenn der zugehörige Prinzipal gelöscht wird. Dies trägt dazu bei, den Graphen von Entitäten in einem konsistenten Zustand zu halten, während sie nachverfolgt werden, sodass dann ein vollständig konsistentes Diagramm in die Datenbank geschrieben werden kann. Wenn eine Eigenschaft nicht auf NULL festgelegt werden kann, weil es sich nicht um einen Nullable-Typ handelt, wird beim SaveChanges() Aufrufen eine Ausnahme ausgelöst.
SetNull	2	Für Entitäten, die vom Kontext nachverfolgt werden, werden die Werte der Fremdschlüsseleigenschaften in abhängigen Entitäten auf NULL festgelegt, wenn der zugehörige Prinzipal gelöscht wird. Dies trägt dazu bei, den Graphen von Entitäten in einem konsistenten Zustand zu halten, während sie nachverfolgt werden, sodass dann ein vollständig konsistentes Diagramm in die Datenbank geschrieben werden kann. Wenn eine Eigenschaft nicht auf NULL festgelegt werden kann, weil es sich nicht um einen Nullable-Typ handelt, wird beim SaveChanges() Aufrufen eine Ausnahme ausgelöst.

Die 1:n - Beziehung



Hier sind die 3 Nachschlage- (Lookup-) Tabellen jeweils Master gegenüber der Detail-Tabelle Mitarbeiter.



Hinweis: Das Löschen eines Eintrags aus einer der Master-Tabellen darf nicht dazu führen, dass die verbundenen Mitarbeiter gelöscht werden!

Die 1:n-Beziehung zwischen Abteilung (oder Niederlassung oder Funktion) und Mitarbeiter Vermeidung von kaskadierendem Löschen

- Abteilung, Niederlassung, Funktion sind so genannte Nachschlage- oder Lookup-Tabellen. Sie gehören zu den Merkmalen der Mitarbeiter
- Für die Beziehung zwischen Mastertabellen (Abteilung, Niederlassung, Funktion) und Detail-Tabelle (Mitarbeiter) (auch Prinzipal/Dependent Entities oder Parent/Child) gilt in diesem Fall: Wenn eine Abteilung/Niederlassung/Funktion gelöscht werden sollte, dürfen natürlich die zugehörigen Mitarbeiter nicht gelöscht werden! Sollte eine Abteilung/Niederlassung/Funktion gelöscht werden, muss der entsprechende Fremdschlüssel in den betroffenen Mitarbeitersätzen auf NULL gesetzt werden.
- Per Konvention kann dies dem EFCore mitgeteilt werden, indem der Datentyp int des Fremdschlüssels AbteilungId/NiederlassungId/FunktionId auf int? gesetzt wird
- Eine andere Variante ist die Nutzung der Fluent API:

```
public int? AbteilungId { get; set; }
public int? NiederlassungId { get; set; }
public int? FunktionId { get; set; }
```

```
entity.HasOne(d => d.Abteilung)
  .WithMany(p => p.MitarbeiterListe)
  .HasForeignKey(d => d.AbteilungId)
  .OnDelete(DeleteBehavior.SetNull);

entity.HasOne(d => d.Niederlassung)
   .WithMany(p => p.MitarbeiterListe)
   .HasForeignKey(d => d.NiederlassungId)
   .OnDelete(DeleteBehavior.SetNull);
```

1:n – Beziehung ohne kaskadierendes Löschen

```
modelBuilder.Entity<Abteilung>()
    .HasMany(x => x.MitarbeiterListe)
    .WithOne(op => op.Abteilung)
    .OnDelete(DeleteBehavior.SetNull)
    .HasForeignKey(@"AbteilungId")
    .IsRequired(false);
```

1:n

```
modelBuilder.Entity<Mitarbeiter>()
      .HasOne(x => x.Abteilung)
      .WithMany(op => op.MitarbeiterListe)
      .HasForeignKey(@"AbteilungId")
      .IsRequired(false);
```

```
modelBuilder.Entity<Funktion>()
    .HasMany(x => x.MitarbeiterListe)
    .WithOne(op => op.Funktion)
    .OnDelete(DeleteBehavior.SetNull)
    .HasForeignKey(@"FunktionId")
    .IsRequired(false);
```

1:n

```
modelBuilder.Entity<Mitarbeiter>()
      .HasOne(x => x.Funktion)
      .WithMany(op => op.MitarbeiterListe)
      .HasForeignKey(@"FunktionId")
      .IsRequired(false);
```

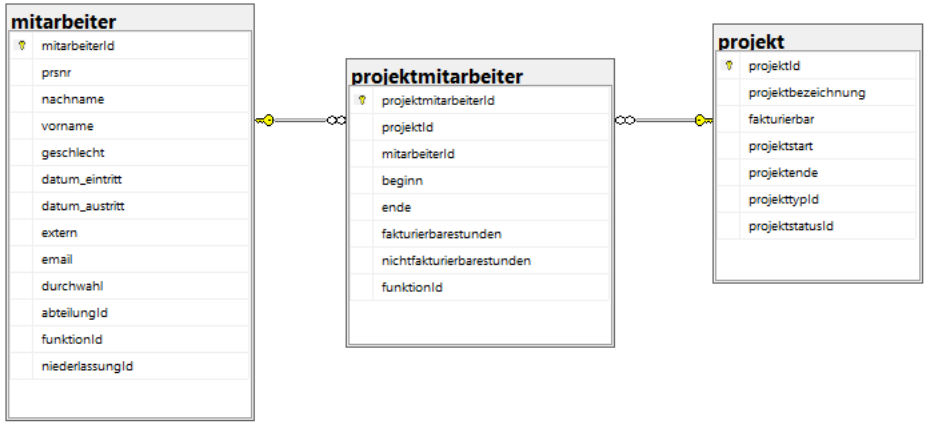
```
modelBuilder.Entity<Niederlassung>()
    .HasMany(x => x.MitarbeiterListe)
    .WithOne(op => op.Niederlassung)
    .OnDelete(DeleteBehavior.SetNull)
    .HasForeignKey(@"NiederlassungId")
    .IsRequired(false);
```

1:n

```
modelBuilder.Entity<Mitarbeiter>()
      .HasOne(x => x.Niederlassung)
      .WithMany(op => op.MitarbeiterListe)
      .HasForeignKey(@"NiederlassungId")
      .IsRequired(false);
```

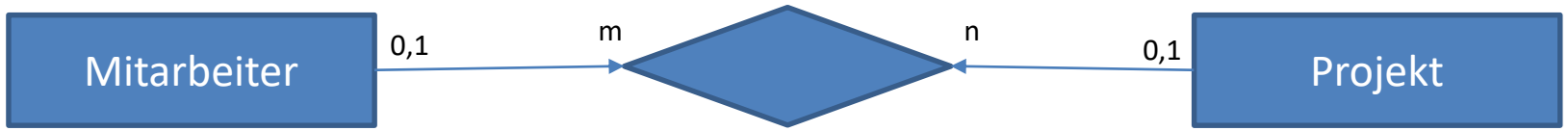
m:n - Beziehungen

- m:n- Beziehungen treten in der Praxis vielfältig auf:
  - Mitarbeiter haben im Allgemeinen mehrere Qualifikationen (Skills). Oft haben mehrere Mitarbeiter eine gleiche Qualifikation
  - Filme haben mehrere Mitwirkende, die jeweils in verschiedenen Filmen mitwirken können
  - In einem Projekt können mehrere Mitarbeiter, ein Mitarbeiter kann in mehreren Projekten beschäftigt sein
- In der relationalen Datenbank werden diese Beziehungen durch eine Vermittlungs- (Zuordnungs-) Tabelle abgebildet (hier: Projektmitarbeiter)



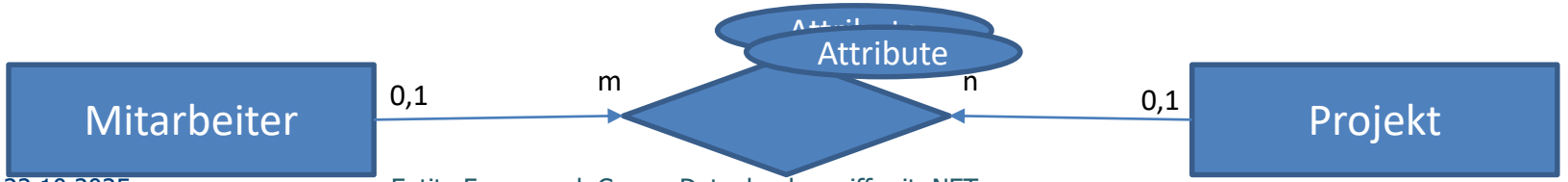
m:n - Beziehungen

- Jetzt gibt es aus der Sicht der Datenbank zwei Varianten
  - Variante 1: Die vermittelnde Tabelle hat nur 2 Felder, nämlich die jeweiligen Primärschlüssel als Fremdschlüssel, die gemeinsam (als Cluster) den Primärschlüssel der vermittelnden Tabelle bilden (MitarbeiterId, ProjektId). Dieses Paar darf also nur jeweils einmal auftreten.



  - Variante 2: Durch die vermittelnde Tabelle werden auch weitere Merkmale des Zusammenwirkens der beiden Master-Entities beschrieben, die sich nicht eindeutig einem der beiden Master zuordnen lassen
    - Im Falle Mitarbeiter-Skills kann das der Grad der Qualifikation sein
    - Im Falle Film-Mitwirkende kann das die Funktion sein (Regisseur, Schauspieler, Drehbuchautor, Komponist, Produzent …). Beispiel: In manchen Filmen ist eine Person sowohl Regisseur als auch Schauspieler
    - Im Falle Mitarbeiter-Projekt kann ein Mitarbeiter nur zeitweise und/oder in unterschiedlichen Funktionen tätig sein

    In diesem Fall können also die Fremdschlüssel-Paare (z.B. MitarbeiterId, ProjektId) mehrfach auftreten. Deshalb muss die Zuordnungstabelle einen eigenständigen Primärschlüssel erhalten!



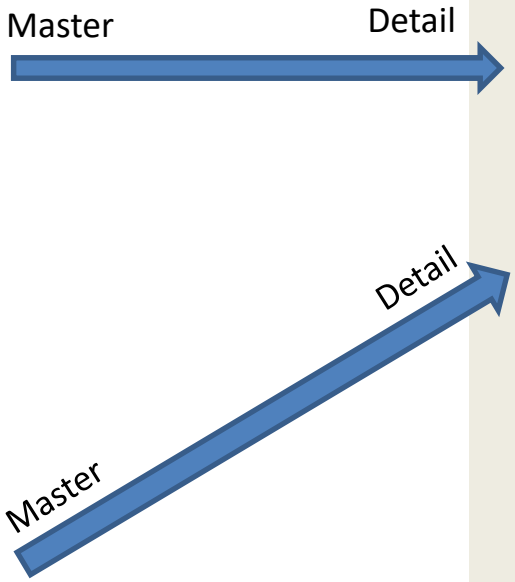
Die m:n-Beziehung zwischen Projekt und Mitarbeiter

- Das Projekt hat (wenn es eröffnet wurde) einen zeitlichen Beginn und ein zeitliches Ende
- Mehrere Mitarbeiter können (zeitlich versetzt, aber auch gleichzeitig) in einem Projekt tätig sein,
- Ein Mitarbeiter kann (zeitlich versetzt, aber auch gleichzeitig) in mehreren Projekten tätig sein
- Innerhalb des Intervalls [Projektbeginn, Projektende] werden Mitarbeiter eingesetzt – manche den ganzen Zeitraum, manche nur teilweise, vielleicht auch mehrfach
- Wenn ein Projekt, dem Mitarbeiter zugeordnet werden, gelöscht wird, müssen automatisch die entsprechenden Zuordnungen, nicht aber die Mitarbeiter gelöscht werden!
- Wenn ein Mitarbeiter, der in einem oder mehreren Projekten mitarbeitet, ausscheidet bzw. gelöscht wird, müssen automatisch die entsprechenden Zuordnungen, nicht aber die Projekte gelöscht werden!
- Diese Komplexität kann man mit Hilfe einer verbindenden Entity abbilden:

Die m:n – Beziehung mit einer Vermittlungsklasse, die spezielle Properties enthält

```
public class Mitarbeiter
{
  public int MitarbeiterId { get; set; }
  public string Nachname { get; set; }
 // ….
  public IList<ProjektMitarbeiter>
              ProjektMitarbeiterListe { get; set; }
}
```

```
public class Projekt
{
  public int ProjektId { get; set; }
  public string Projektbezeichnung { get; set; }
  public string Beschreibung { get; set; }
  public IList<ProjektMitarbeiter>
              ProjektMitarbeiterListe { get; set; }
}
```



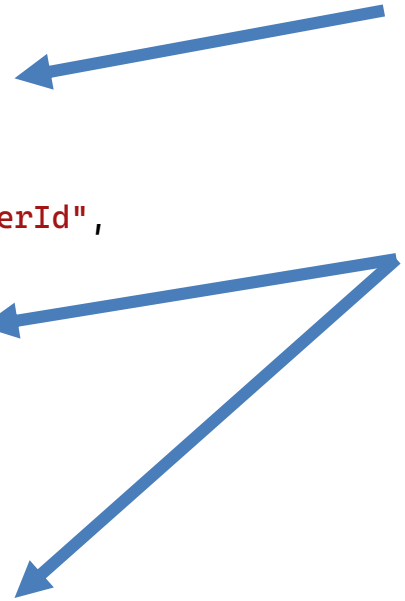

```
public partial class ProjektMitarbeiter:Audit
{
        [Key]
        public int ProjektMitarbeiterId { get; set; }

        [ForeignKey("MitarbeiterId")]
        [Column(Order = 1)]
        [Required]
        public int MitarbeiterId { get; set; }
        public virtual Mitarbeiter Mitarbeiter { get; set; }

        [ForeignKey("ProjektId")]
        [Column(Order = 2)]
        [Required]
        public int ProjektId { get; set; }
        public virtual Projekt Projekt { get; set; }

        [Precision(0)]
        [Column(Order = 3)]
        [Display(Name ="Einsatzbeginn")]
        public virtual DateTime? EinsatzBeginn { get; set; }
        [Display(Name = "Einsatzende")]
        [Precision(0)]
        [Column(Order = 4)]
        public virtual DateTime? EinsatzEnde { get; set; }
        [StringLength(20)]

        [Column(Order = 5)]
        public virtual int? FunktionId { get; set; }
        public virtual Funktion Funktion { get; set; }

}
```

```
public DbSet< Mitarbeiter> MitarbeiterListe { get; set; }
public DbSet<Projekt> ProjektListe { get; set; }
public DbSet<ProjektMitarbeiter> ProjektMitarbeiterListe { get; set; }
```

Diskussion

- In der Migrationsklasse findet man die Anweisung zur Gestaltung der Constraints, wobei uns hier nur der Primär- und die Fremdschlüssel interessieren:

```
constraints: table =>
{
  table.PrimaryKey("PK_ProjektMitarbeiter", x => x.ProjektMitarbeiterId);
  table.ForeignKey(
     name: "FK_ProjektMitarbeiter_Funktion_FunktionId",
     column: x => x.FunktionId,
     principalTable: "Funktion",
     principalColumn: "FunktionId");
  table.ForeignKey(
     name: "FK_ProjektMitarbeiter_Mitarbeiter_MitarbeiterId",
     column: x => x.MitarbeiterId,
     principalTable: "Mitarbeiter",
     principalColumn: "MitarbeiterId",
     onDelete: ReferentialAction.Cascade);
  table.ForeignKey(
     name: "FK_ProjektMitarbeiter_Projekt_ProjektId",
     column: x => x.ProjektId,
     principalTable: "Projekt",
     principalColumn: "ProjektId",
     onDelete: ReferentialAction.Cascade);
});
```

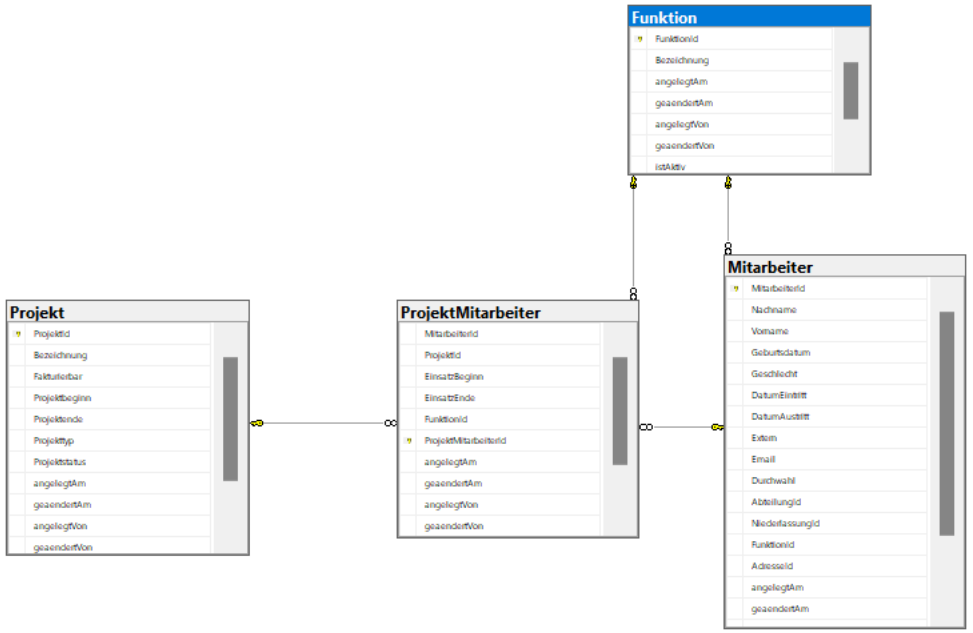


Wie erwartet, führt die Löschung einer Funktion nicht zu einem kaskadierenden Löschen (FunktionId wird auf Null geetzt).

Die Löschung eines Projekts oder eines Mitarbeiters löst dagegen die Entfernung des betroffenen Datensatzes in der Vermittlungstabelle aus.

Hinweis: Vor dem NugetPaket-Manager-Konsolen-Befehl update-database kann die Migrationsklasse inspiziert werden. Man kann dort gegebenenfalls noch Änderungen vornehmen.

Nach der Generierung erhalten wir das Diagramm:



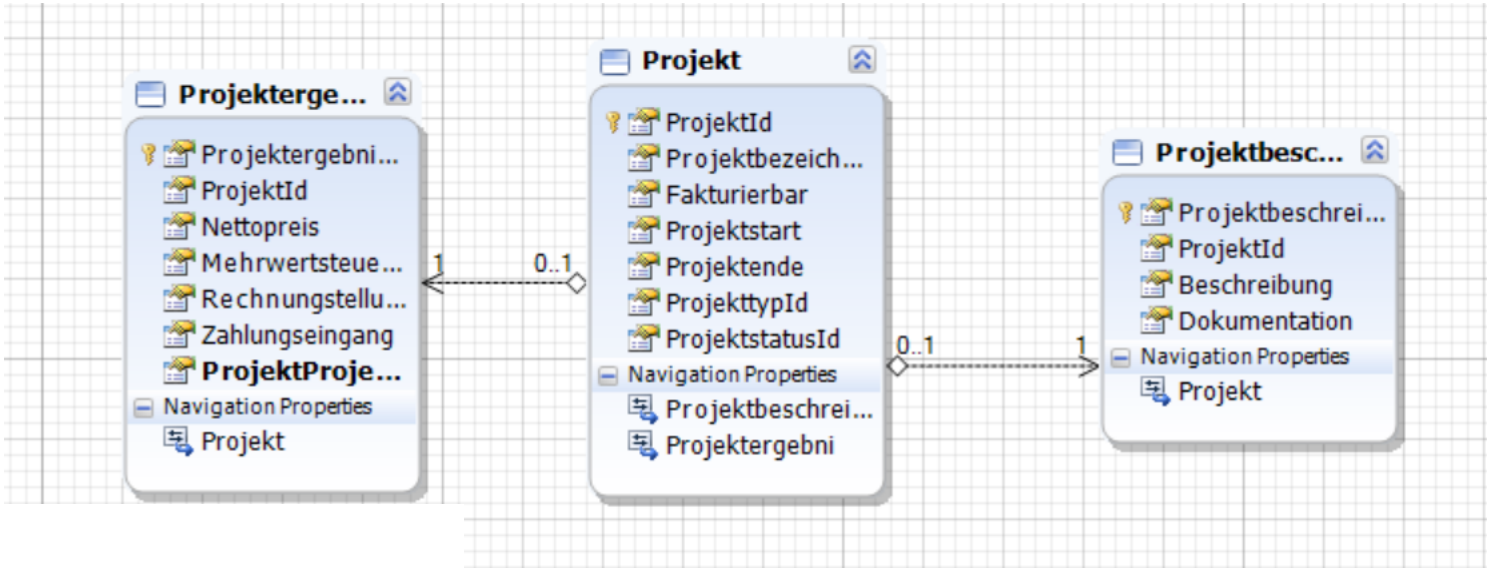

```
CREATE TABLE [dbo].[ProjektMitarbeiter](
[MitarbeiterId] [int] NOT NULL,
[ProjektId] [int] NOT NULL,
[EinsatzBeginn] [datetime2](0) NULL,
[EinsatzEnde] [datetime2](0) NULL,
[FunktionId] [int] NULL,
[ProjektMitarbeiterId] [int] IDENTITY(1,1) NOT NULL,
[angelegtAm] [datetime2](7) NOT NULL,
[geaendertAm] [datetime2](7) NULL,
[angelegtVon] [nvarchar](10) NULL,
[geaendertVon] [nvarchar](10) NULL,
[istAktiv] [bit] NOT NULL,
[istGeloescht] [bit] NOT NULL,
 CONSTRAINT [PK_ProjektMitarbeiter] PRIMARY KEY CLUSTERED
(
[ProjektMitarbeiterId] ASC
)WITH (PAD_INDEX = OFF, STATISTICS_NORECOMPUTE = OFF, IGNORE_DUP_KEY = OFF, ALLOW_ROW_LOCKS = ON,
ALLOW_PAGE_LOCKS = ON, OPTIMIZE_FOR_SEQUENTIAL_KEY = OFF) ON [PRIMARY]
) ON [PRIMARY]
GO

ALTER TABLE [dbo].[ProjektMitarbeiter]  WITH CHECK ADD  CONSTRAINT
[FK_ProjektMitarbeiter_Funktion_FunktionId] FOREIGN KEY([FunktionId])
REFERENCES [dbo].[Funktion] ([FunktionId])
GO

ALTER TABLE [dbo].[ProjektMitarbeiter] CHECK CONSTRAINT
[FK_ProjektMitarbeiter_Funktion_FunktionId]
GO

ALTER TABLE [dbo].[ProjektMitarbeiter]  WITH CHECK ADD  CONSTRAINT
[FK_ProjektMitarbeiter_Mitarbeiter_MitarbeiterId] FOREIGN KEY([MitarbeiterId])
REFERENCES [dbo].[Mitarbeiter] ([MitarbeiterId])
ON DELETE CASCADE
GO

ALTER TABLE [dbo].[ProjektMitarbeiter] CHECK CONSTRAINT
[FK_ProjektMitarbeiter_Mitarbeiter_MitarbeiterId]
GO

ALTER TABLE [dbo].[ProjektMitarbeiter]  WITH CHECK ADD  CONSTRAINT
[FK_ProjektMitarbeiter_Projekt_ProjektId] FOREIGN KEY([ProjektId])
REFERENCES [dbo].[Projekt] ([ProjektId])
ON DELETE CASCADE
GO

ALTER TABLE [dbo].[ProjektMitarbeiter] CHECK CONSTRAINT [FK_ProjektMitarbeiter_Projekt_ProjektId]
GO
```

1:1 - Beziehungen

- Solche Beziehungen treten auf, wenn die Attribute einer Master-Tabelle durch eine weitere Tabelle ergänzt werden sollen. Beispiel: Zu jedem Projekt-Datensatz kann eine ergänzende Aussage „Projektergebnis“ hinzugefügt werden, die z.B. Ergebnis-Aussagen zum jeweiligen Projekt enthält, wenn das Projekt beendet wird
- Diese ergänzende Tabelle steht in einem Detail-Verhältnis zur Master-Tabelle. Wenn das Projekt aus irgendeinem Grund entfernt wird, muss die Detail-Tabelle in diesem Fall natürlich auch entfernt werden.

Projekt mit zwei 1:1-Tabellen

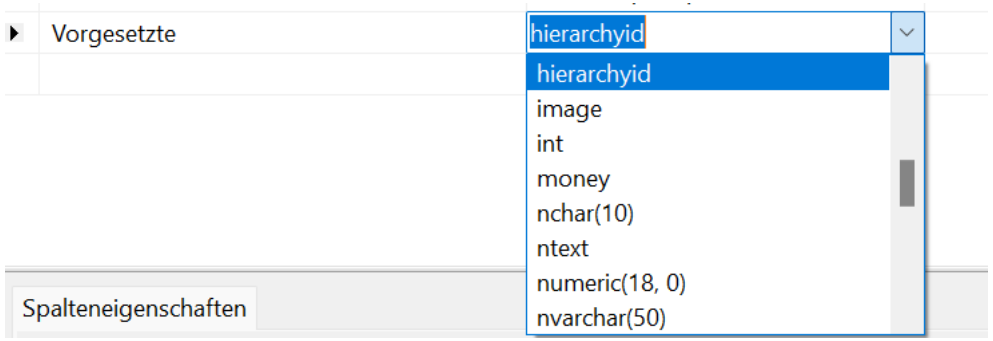
- Die beiden Klassen Projektergebnis und Projektbeschreibung sind jeweils mit einem Projekt über One-To-One verknüpft.
- Wird ein Projekt entfernt, müssen auch die abhängigen Datensätze entfernt werden (kaskadierendes Löschen)




```
modelBuilder.Entity<Projektbeschreibung>()
  .HasOne(x => x.Projekt)
  .WithOne(op => op.Projektbeschreibung)
  .OnDelete(DeleteBehavior.Cascade)
  .HasForeignKey(typeof(Projektbeschreibung), @"ProjektId")
  .IsRequired(true);

modelBuilder.Entity<Projektergebnis>()
  .HasOne(x => x.Projekt)
  .WithOne(op => op.Projektergebnis)
  .OnDelete(DeleteBehavior.Cascade)
  .HasForeignKey(typeof(Projektergebnis), @"ProjektId")
  .IsRequired(true);
```

Rekursive Assoziation – hierarchische Strukturen

- Ein Sepzialfall der 1:n-Assoziation ist die rekursive (selbstreflexive) Assoziation. Hierbei ist nur eine Klasse beteiligt und Objekte dieser Klasse stehen miteinander in Verbindung.
  - Ein Beispiel ist die Beziehung zwischen Mitarbeitern und den jeweiligen Vorgesetzten. Es geht also um hierarchische Strukturen (andere Beispiele: Ein Dateisystem, Aufgaben in einem Projekt, der Graph der Beziehugen zwischen Webseiten …)
- Ab EFC8 besteht die Möglichkeit, die Klasse HierarchyID zu nutzen. Diese korrespondiert mit dem SQL Server Datentyp hierarchyid



- Um diesen Datentyp nutzen zu können, muss das Paket entityframeworkcore.sqlserver.hierarchyid installiert werden. In der Methode OnConfiguring der DBContext-Klasse wird ergänzt:

```
optionsBuilder.UseSqlServer( connectionString, x => x.UseHierarchyId());
```

Hierarchische Strukturen – ein Beispiel

- In der Klasse Mitarbeiter wird das Attribut HierarchiePfad mit dem Datentyp HierarchyId eingefügt:

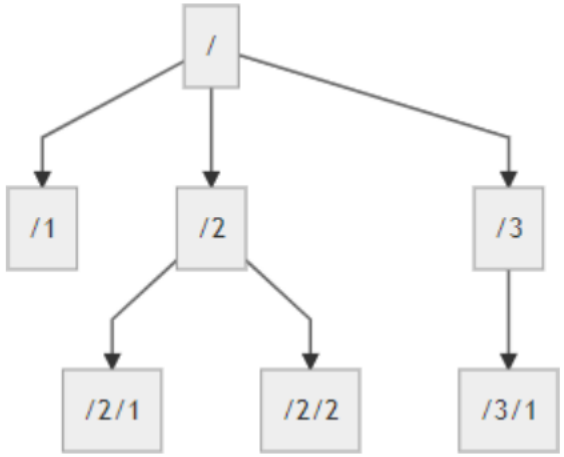
```
public class Mitarbeiter
{
    public int MitarbeiterId { get; set; }

    [Required]
    [StringLength(50)]
    public string Nachname { get; set; }
    [StringLength(50)]
    public string? Vorname { get; set; }
    //Weitere Attribute
    public HierarchyId HierarchiePfad { get; set; }

    public int? AbteilungId { get; set; }
    public virtual Abteilung Abteilung { get; set; }

    public int? FunktionId { get; set; }
    public virtual Funktion Funktion { get; set; }
}
```

- Die Bestimmung des Pfades erfolgt im Beispiel mit

```
switch (m.FunktionId)
{
    case 2:
        m.HierarchiePfad = HierarchyId.Parse("/" + bereichsleiterId.ToString() + "/");
        break;
    case 3:
    case 4:
    case 6:
        m.HierarchiePfad = HierarchyId.Parse("/" + bereichsleiterId.ToString() + "/" + mitarbeiterId.ToString() + "/");
        break;
}
```

Hierarchische Strukturen – ein Beispiel -Ergebnis

- Nach der Migration erhält die Tabelle Mitarbeiter das Feld HierarchiePfad mit dem Datentyp Hyrarchield und es kann eine Abfrage durchgeführt werden:



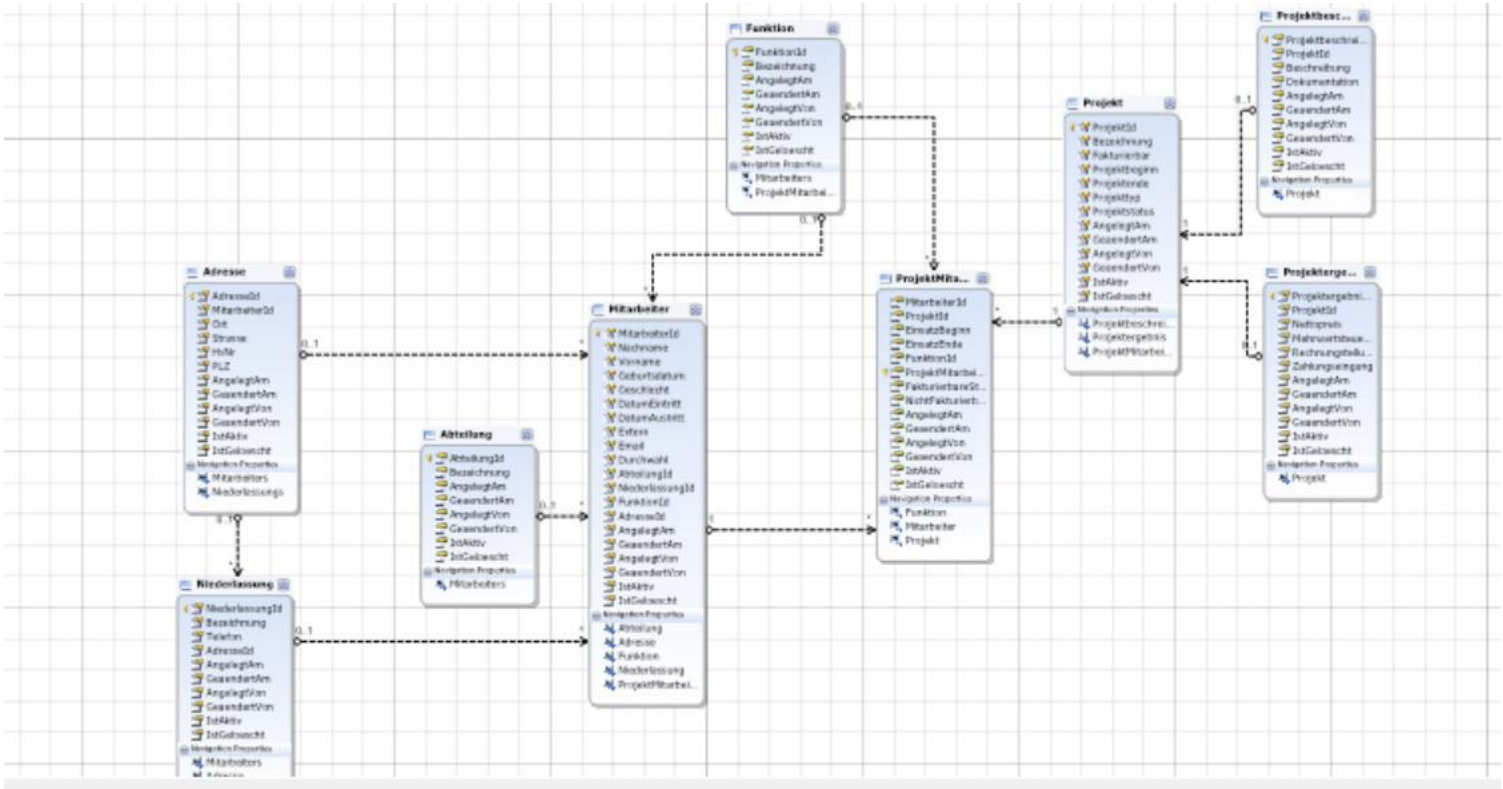

```
select nachname,a.Abteilungsbezeichnung, f.Funktionsbezeichnung ,
hierarchiepfad, hierarchiepfad.ToString() as Position from Mitarbeiter m
inner join Abteilung a on (m.AbteilungId=a.AbteilungId)
inner join Funktion f on (m.FunktionId=f.FunktionId)
order by hierarchiepfad.ToString(), f.Funktionsbezeichnung
```

100 %

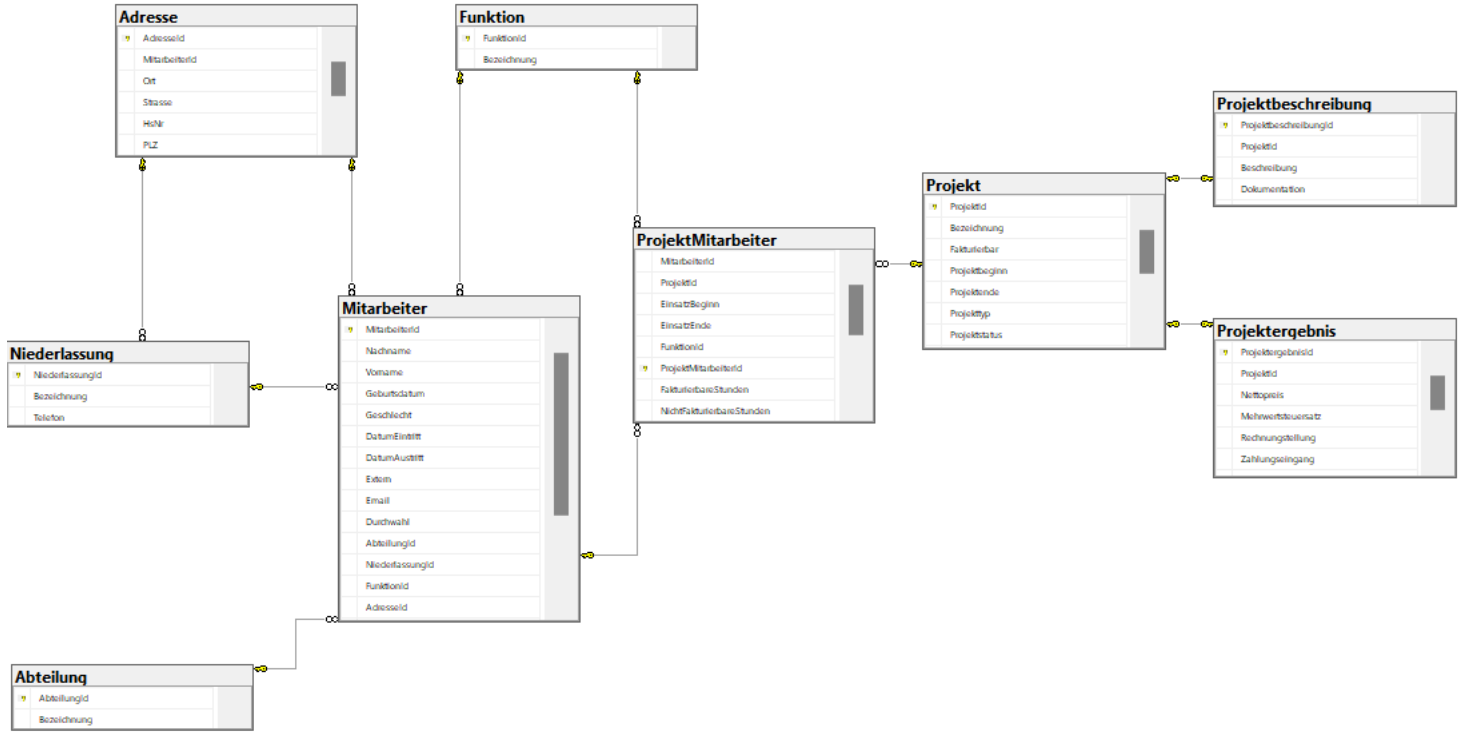
Ergebnisse | Meldungen

	nachname	Abteilungsbezeichnung	Funktionsbezeichnung	hierarchiepfad	Position
1	Paul	Geschäftsleitung/RW	Geschäftsführer/in	0x	/
2	Unterlauf	Geschäftsleitung/RW	Mitarbeiter/in	0x58	/1/
3	Schiefner	Geschäftsleitung/RW	Mitarbeiter/in	0x68	/2/
4	Oltenbürstel	Vertrieb	Bereichsleiter/in	0xC930	/33/
5	Haase	Vertrieb	Freie/r Mitarbeiter/in	0xC93C33	/33/25/
6	Thurow	Vertrieb	Freie/r Mitarbeiter/in	0xC93C35	/33/26/
7	Fischer	Vertrieb	Mitarbeiter/in	0xC93C37	/33/27/
8	Schwienke	Vertrieb	Mitarbeiter/in	0xC93C39	/33/28/
9	Koritz	Infrastruktur	Bereichsleiter/in	0x94	/6/
10	Geist	Infrastruktur	Mitarbeiter/in	0x95E0	/6/3/
11	Wendisch	Infrastruktur	Mitarbeiter/in	0x9610	/6/4/
12	Lorenz	Marketing	Bereichsleiter/in	0x9C	/7/
13	Mühle	Marketing	Mitarbeiter/in	0x9E30	/7/5/
14	Arendt	Produktion	Bereichsleiter/in	0xA6	/9/
15	Oehme	Produktion	Mitarbeiter/in	0xA754	/9/10/
16	Vettin	Produktion	Mitarbeiter/in	0xA75C	/9/11/
17	Kayser	Produktion	Mitarbeiter/in	0xA764	/9/12/

Gesamtsicht - Klassenmodell



Gesamtsicht - Datenbankschema

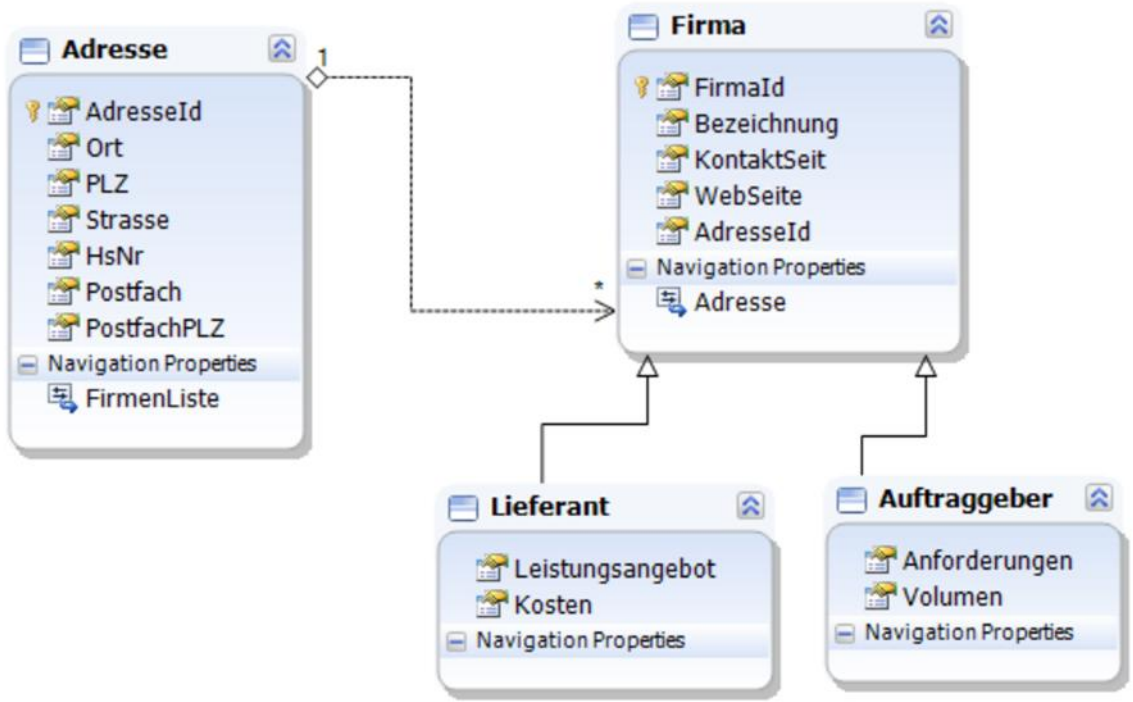


Im Folgenden:

UMSETZUNG VON VERERBUNGS-BEZIEHUNGEN

Die Umsetzung von Vererbungsbeziehungen

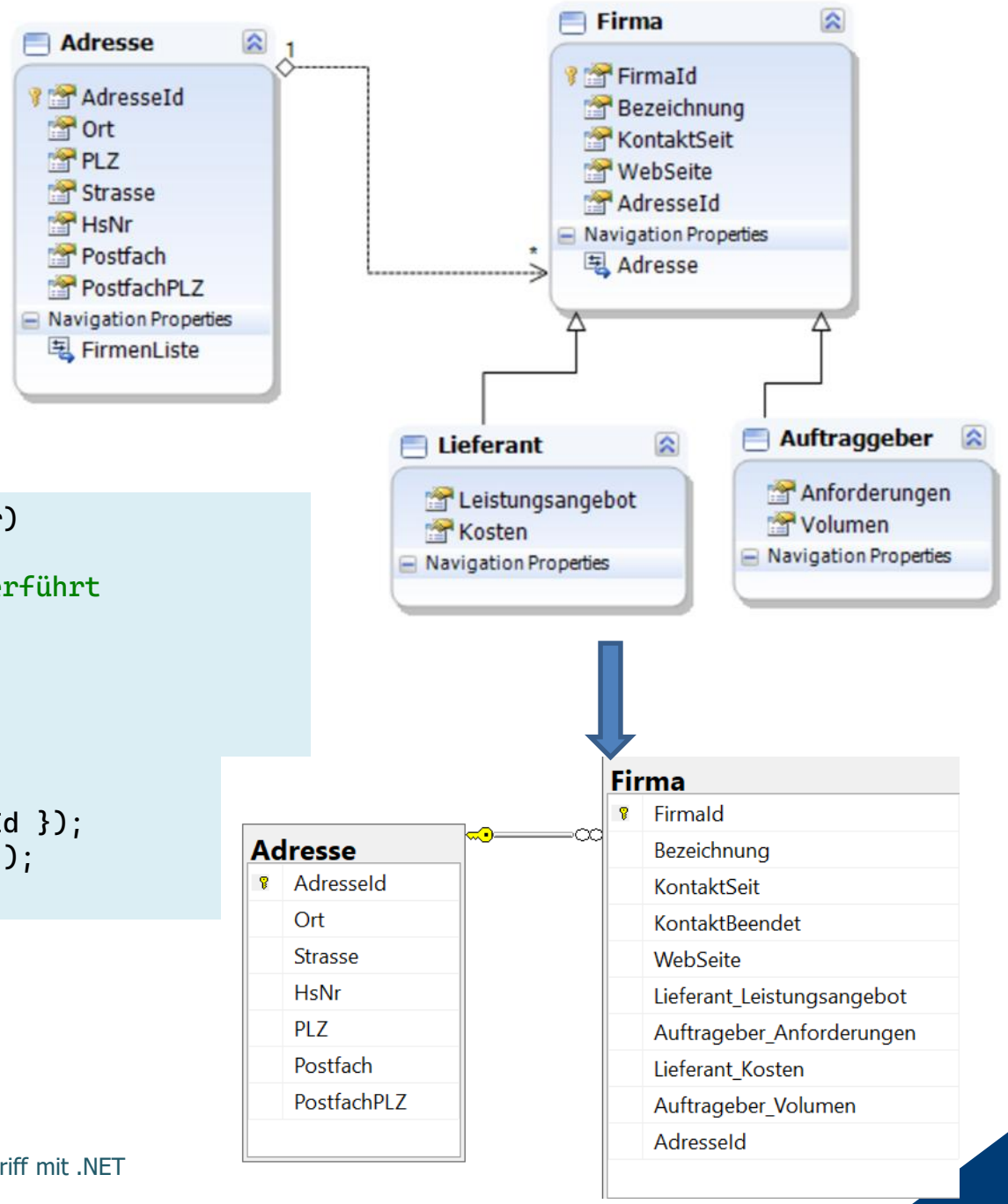
- Im Beispiel ist Firma eine Basisklasse, Lieferant und Auftraggeber erben die Basismerkmale (Properties) von dieser Basiskklasse und erweitern diese Merkmale durch weitere, jeweils spezifische Merkmale
- Es gibt mehrere Möglichkeiten der Umsetzung dieser Vererbungsbeziehungen
  - Table splitting
  - Table per hierarchy (TPH)
  - Table per type (TPT)
  - Table per concrete type (TPC)



Die Umsetzung von Vererbungsbeziehungen

Table splitting: Hierbei werden die drei konkreten Klassen in eine Tabelle abgebildet. Die entstehende Tabelle „Firma“ enthält damit neben den Basismerkmalen auch die zusätzlichen Merkmale aus Lieferant und Auftraggeber

```
protected override void OnModelCreating(ModelBuilder modelBuilder)
{
// Umkehr des Table-Splitting: 3 Entities werden in 1 Tabelle überführt
   modelBuilder.Entity<Firma>().ToTable("Firma");
   modelBuilder.Entity<Auftraggeber>().ToTable("Firma");
   modelBuilder.Entity<Lieferant>().ToTable("Firma");

// Jede der 3 Entities hat den gleichen Primärschlüssel FirmaId
   modelBuilder.Entity<Firma>().HasKey(b => new { b.FirmaId });
   modelBuilder.Entity<Auftraggeber>().HasKey(b => new { b.FirmaId });
   modelBuilder.Entity<Lieferant>().HasKey(b => new { b.FirmaId });
}
```



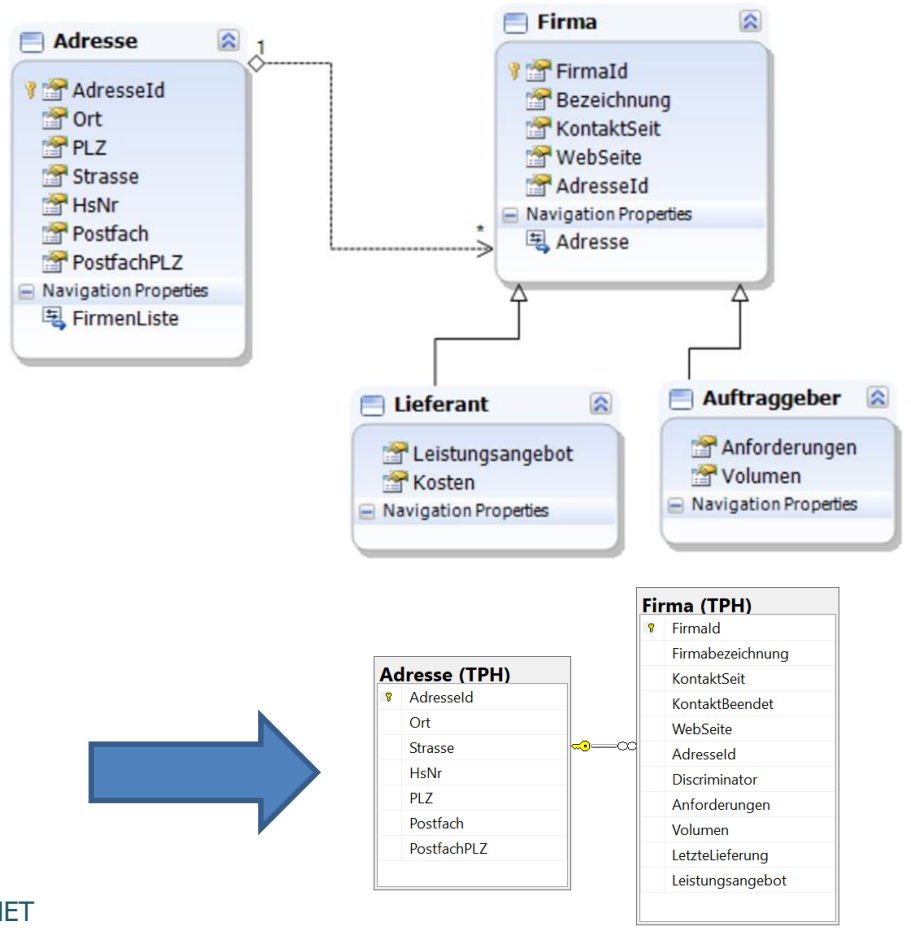
Die Umsetzung von Vererbungsbeziehungen

Table per hierarchy (TPH) : Auch hierbei werden die drei Klassen in eine Tabelle abgebildet.
Die entstehende Tabelle Firma enthält damit neben den Basismerkmalen auch die zusätzlichen Merkmale aus Lieferant und Auftraggeber. Die Vererbung durch abgeleitete Klassen wird durch einen so genannten Diskriminator mit den Ausprägungen „Firma“, „Auftraggeber“ und „Lieferant“ realisiert.

```
public partial class Firma
{
    public virtual int FirmaId { get; set; }
    [StringLength(200)]
    [Required]
    public virtual string Firmabezeichnung { get; set; }
    public virtual DateOnly? KontaktSeit { get; set; }
    public virtual DateOnly? KontaktBeendet { get; set; }
    [StringLength(200)]
    public virtual string WebSeite { get; set; }
    public virtual int? AdresseId { get; set; }
    public virtual Adresse Adresse { get; set; }
}
```

```
public partial class Auftraggeber:Firma
{
    public string Anforderungen { get; set; }
    public double? Volumen { get; set; }
}
```

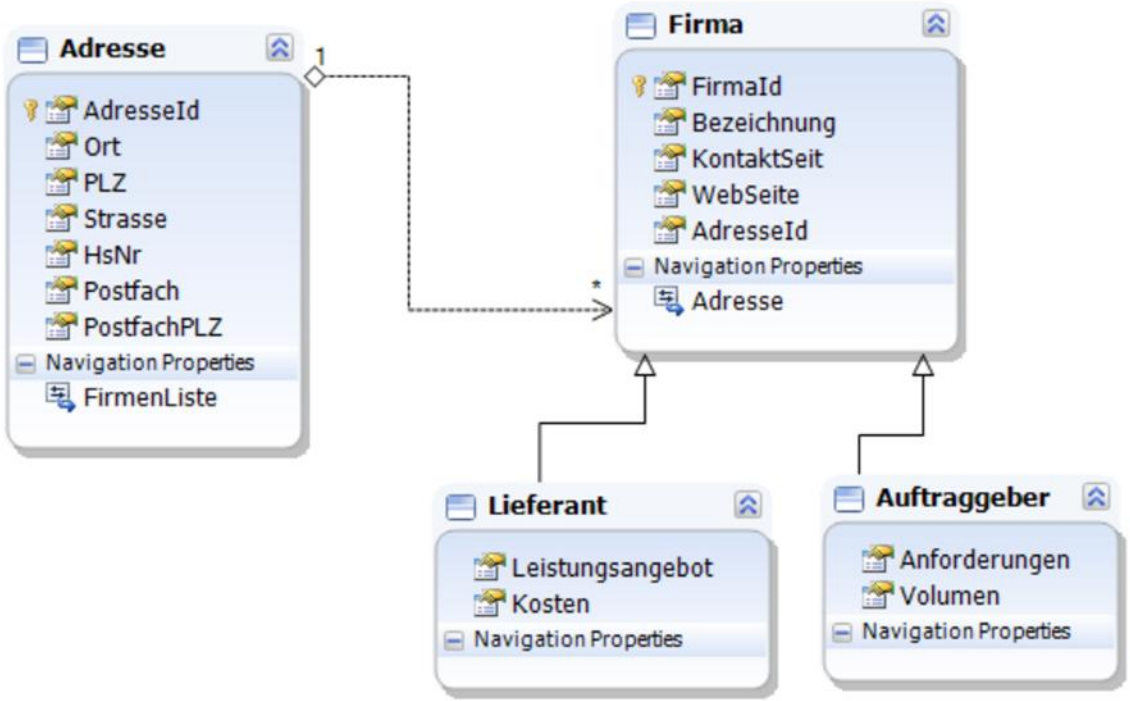
```
public partial class Lieferant :Firma
{
    public DateOnly? LetzteLieferung { get; set; }
    public string Leistungsangebot { get; set; }
}
```



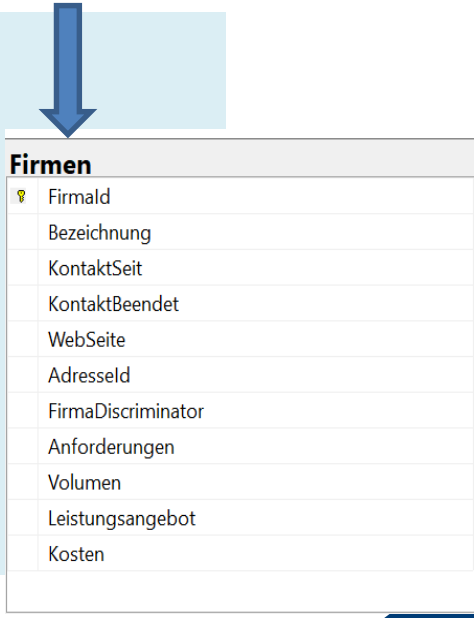
Die Umsetzung von Vererbungsbeziehungen

Table per hierarchy (TPH) :

Es entsteht eine einzige Tabelle, die ein Feld Discriminator enthält. Mit Hilfe der Fluent-API kann dieses Feld spezifiziert werden:




```
private void FirmaMapping(ModelBuilder modelBuilder)
{
  modelBuilder.Entity<Firma>().ToTable(@"Firma", @"dbo");
  modelBuilder.Entity<Firma>()
            .HasDiscriminator(@"FirmaDiscriminator", typeof(string))
                        .HasValue<Firma>(@"Firma")
                        .HasValue<Lieferant>(@"Lieferant")
                        .HasValue<Auftraggeber>(@"Auftraggeber");
  modelBuilder.Entity<Firma>().Property(x => x.FirmaId).HasColumnName(@"FirmaId").IsRequired().ValueGeneratedOnAdd();
  modelBuilder.Entity<Firma>().Property(x => x.Bezeichnung).HasColumnName(@"Bezeichnung").IsRequired().ValueGeneratedNever();
  modelBuilder.Entity<Firma>().Property(x => x.KontaktSeit).HasColumnName(@"KontaktSeit").ValueGeneratedNever();
  modelBuilder.Entity<Firma>().Property(x => x.WebSeite).HasColumnName(@"WebSeite").IsRequired().ValueGeneratedNever();
  modelBuilder.Entity<Firma>().Property(x => x.AdresseId).HasColumnName(@"AdresseId").ValueGeneratedNever();
  modelBuilder.Entity<Firma>().HasKey(@"FirmaId");
  modelBuilder.Entity<Firma>().HasIndex(@"FirmaId").IsUnique(true);
}
```

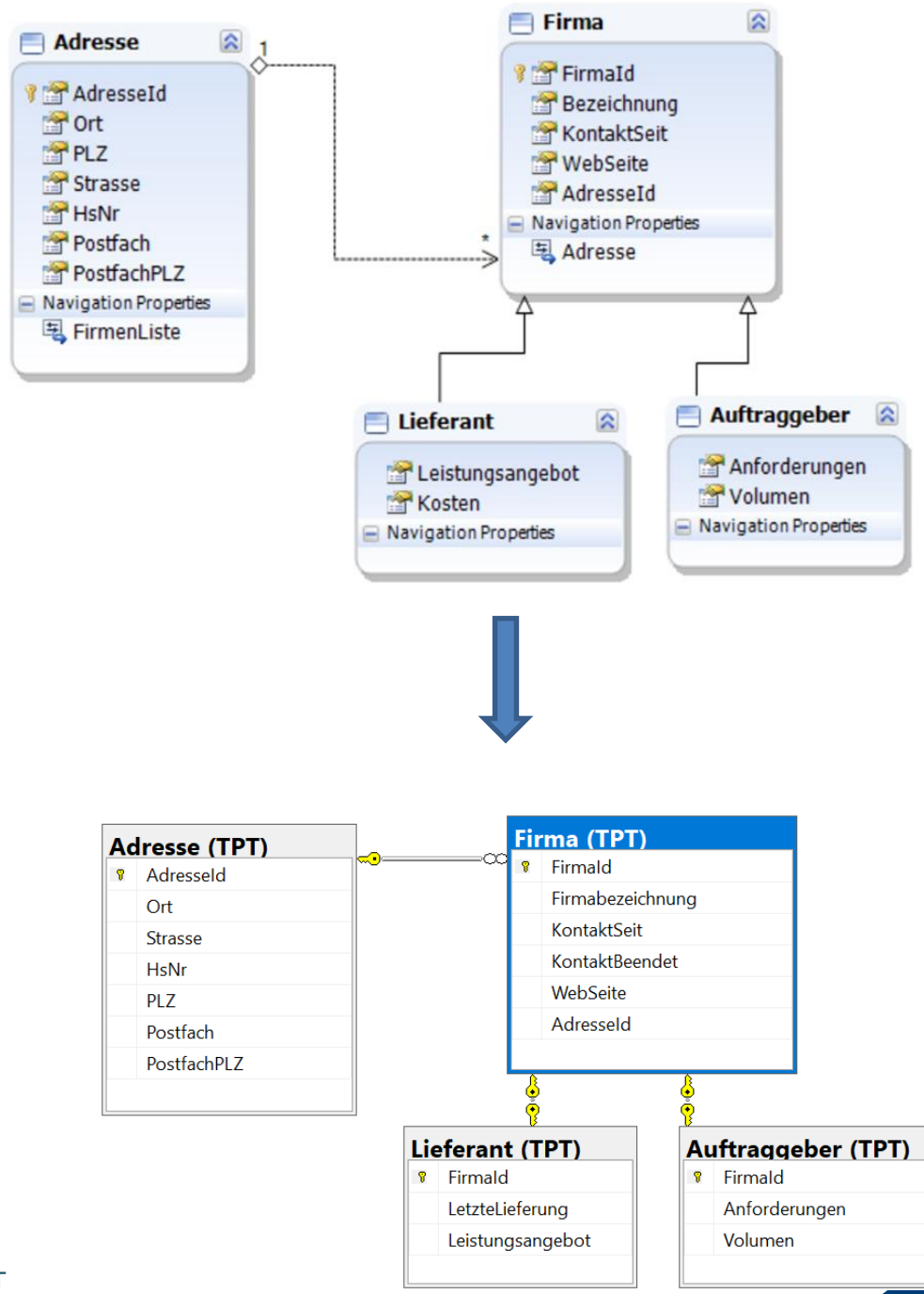


Die Umsetzung von Vererbungsbeziehungen

Table per type (TPT) : Hierbei werden die drei Klassen in drei Tabellen abgebildet. Lieferant und Auftraggeber sind nachgeordnet, dort tritt die in der Tabelle Firma generierte FirmaId zugleich als Primär- und als Fremdschlüssel auf.

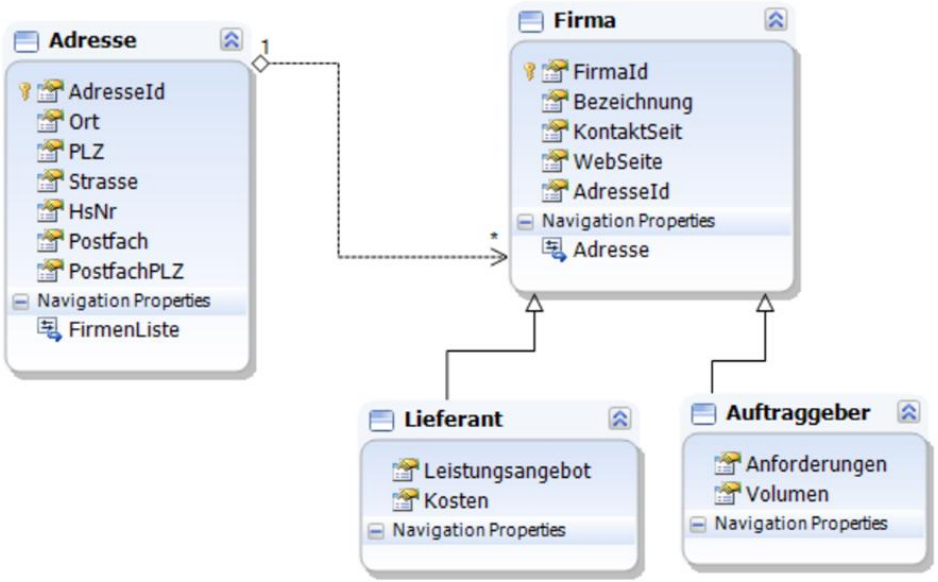
```
[Table("Firma", Schema = "TPT")]
public partial class Firma
{
    public virtual int FirmaId { get; set; }
    [StringLength(200)]
    [Required]
    public virtual string Firmabezeichnung { get; set; }
    public virtual DateOnly? KontaktSeit { get; set; }
    public virtual DateOnly? KontaktBeendet { get; set; }
    [StringLength(200)]
    public virtual string WebSeite { get; set; }
    public virtual int? AdresseId { get; set; }
    public
}
```

```
[Table("Auftraggeber", Schema = "TPT")]
public partial class Auftraggeber:Firma
{
    public string Anforderungen { get; set; }
    public double? Volumen { get; set; }
}

[Table("Lieferant", Schema = "TPT")]
public partial class Lieferant :Firma
{
    public DateOnly? LetzteLieferung { get; set;
}
    public string Leistungsangebot { get; set; }
}
```



Die Umsetzung von Vererbungsbeziehungen

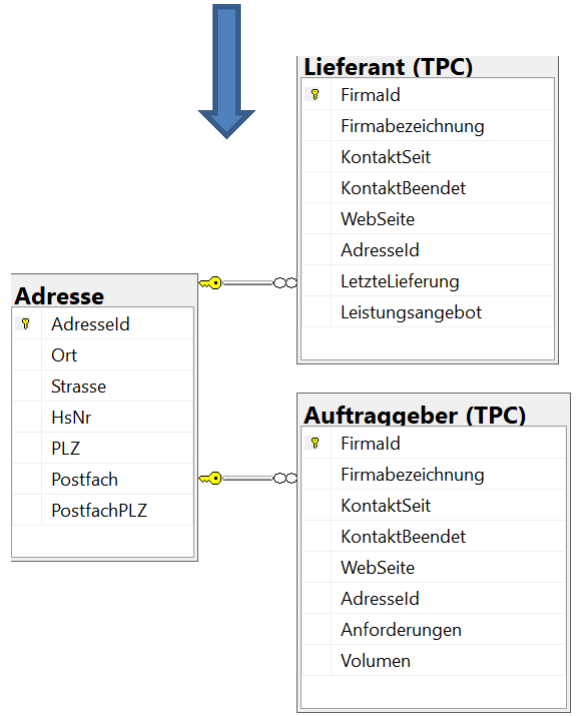
Table per concrete type (TPC) - ab EFC 7.0 - Die TPC-Strategie ähnelt der TPT-Strategie, mit der Ausnahme, dass nur für jeden *konkreten* Typ in der Hierarchie eine andere Tabelle erstellt wird, aber Tabellen **nicht** für *abstrakte* Typen erstellt werden - daher der Name "Table-per-concrete-type". Wie bei TPT gibt die Tabelle selbst den Typ des gespeicherten Objekts an. Im Gegensatz zur TPT-Zuordnung enthält jede Tabelle jedoch Spalten für jede Eigenschaft im konkreten Typ und ihre Basistypen. TPC-Datenbankschematas werden denormalisiert.




```
public abstract class Firma
{
    public virtual int FirmaId { get; set; }
    [StringLength(200)]
    [Required]
    public virtual string Firmabezeichnung { get; set; }
    public virtual DateOnly? KontaktSei
    public virtual DateOnly? KontaktBee
    [StringLength(200)]
    public virtual string WebSeite { ge
    public virtual int? AdresseId { get
    public virtual Adresse Adresse { ge
}
```

```
[Table("Lieferant", Schema = "TPC")]
public partial class Lieferant :Firma
{
    public DateOnly? LetzteLieferung { get; set; }
    public string Leistungsangebot { get; set; }
}
```

```
[Table("Auftraggeber", Schema = "TPT")]
public partial class Auftraggeber:Firma
{
    public string Anforderungen { get; set; }
    public double? Volumen { get; set; }
}
```



Vor- und Nachteile der verschiedenen Strategien

- TPH
  - Pros:
    - Abfragen zur Basisklasse sind sehr schnell, da alles in einer Tabelle, **INSERT, UPDATE** und **DELETE** sind ebenfalls schnell.
  - Cons:
    - Tabelle wird evtl. unübersichtlich wegen der zusätzlichen Spalten. Es treten viele NULL-Werte auf
- TPT
  - Pros:
    - Sehr stark normalisiertes Modell, weniger Speicheraufwand als **TPH** und **TPC**.
  - Cons:
    - Abfragen evtl. verlangsamt wegen mehrerer **JOINs**. **INSERTs**, **UPDATEs** and **DELETEs** ebenfalls langsamer als TPH.
- TPC
  - Pros:
    - Geschwindigkeit für CRUD-Operationen zwischen TPH und TPC
  - Cons:
    - Requires the usage of a specific identifier generation strategy, such as **Hi-Lo**.
- Allgemein:
  - **TPC** lohnt sich, wenn die Abfragen sich auf konkrete Typen beziehen;
  - **TPH** ist auch sinnvoll, vor allem wegen der günstigen Auswirkungen auf die Performance;
  - **TPT** ist nur in Ausnahmefällen zu empfehlen.

Zusammenfassung

- "Table per Type" (TPT): Es gibt für jede Klasse in der Vererbungshierarchie genau eine Datenbanktabelle. Für alle abgeleiteten Klassen sind beim Laden von Datensätzen daher Joins notwendig, was aufwändig werden kann, insbesondere wenn die Vererbungshierarchie über mehrere Ebenen geht.
- "Table per Concrete Type" (TPC alias TPCT): Es gibt nur für jede **konkrete** Klasse in der Vererbungshierarchie genau eine Datenbanktabelle. Es sind beim Laden der Instanzen der abgeleiteten Klassen folglich keine Joins notwendig. Konkrete Klassen sind Klassen, die nicht mit "abstract" deklariert sind.
- "Table per Hierarchy" (TPH): Es gibt für alle Klassen in der Vererbungshierarchie nur eine Datenbanktabelle mit einer sogenannten "Diskriminator-Spalte", die festlegt, welcher Objekttyp in dem jeweiligen Datensatz gespeichert ist. Es sind beim Laden von Objekten keine Joins notwendig, sondern nur eine zusätzliche Auswahl (WHERE Discriminator = 'Klassenname') über die Diskriminatorspalte.
- Zusammenfassend lässt sich feststellen, dass TPH in der Regel für die meisten Anwendungen in Ordnung und ein guter Standardwert für eine Vielzahl von Szenarien ist, so dass Sie die Komplexität von TPC nicht unnötigerweise hinzufügen müssen. Wenn Ihr Code hauptsächlich Entitäten vieler Typen abfragt, z. B. Abfragen für den Basistyp, dann sollten Sie TPH gegenüber TPC bevorzugen.